Universität Leipzig
Institut für Informatik
Abeilung Datenbanken

UNIVERSITÄT LEIPZIG

Seminararbeit

Schema-Versionierung

Seminar: Schema Evolution
(WS 2005/2006)

Autor: Christian Lehmann (Mat.-Nr. 9638048)
Studiengang: Informatik - Master (3. Semester)

Betreuer: David Aumüller

Eingereicht am: 13.01.2006

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

____

Leipzig, den 13. Januar 2006

Inhaltsverzeichnis

Erklärung 2

Inhaltsverzeichnis 3

Abkürzungsverzeichnis 5

1 Einleitung und Motivation 6

2 Grundlagen der Schema Versionierung 7

2.1 Allgemeine Anforderungen 7
2.1.1 Versionierung mehrerer Generationen 7
2.1.2 Autorenfreundlichkeit 7
2.1.3 Benutzung von Schemata und Instanzen 7

2.2 Kompatibilität bei Versionsänderungen 8

2.3 Unterscheidung zwischen Versionierung und Erweiterbarkeit 9

2.4 Abgrenzung zum Begriff Schema Evolution 10

3 Ausgewählte Versionierungstechniken 11

3.1 Class Versioning 11
3.1.1 Modell einer Objektorientierten Datenbank 11
3.1.2 Konvertierung vs. Kompatibilität 12
3.1.3 Managen von verschiedenen Versionen 12
3.1.4 Physische Repräsentation von verschiedenen Versionen 14
3.1.5 Modifikation der Klassenhierarchie 14
3.1.6 Bewertung des Systems 15

3.2 Branching and Temporal Versioning 16
3.2.1 Datenmodell 16
3.2.2 Versionsmanagement 18
3.2.3 Semantik der Schemaänderungen 19
3.2.4 Bewertung des Systems 20

3.3 XML Schema Versionierung 21
3.3.1 XML – Eine Einführung 21
3.3.2 XML-Schema 23
3.3.3 Versionierungsmöglichkeiten 25
3.3.4 Best Practices 29

3.4 Zusammenfassung und Ausblick 31

Abbildungsverzeichnis 32

Tabellenverzeichnis 32

Listingverzeichnis 32

Literaturverzeichnis 33

Abkürzungsverzeichnis

CDB	Curent Database
CV	Current Version
DBU	Database Update
DTD	Document Type Definition
HTML	Hypertext Markup Language
SGML	Standard Generalized Markup Language
TV	Temporal Version
TVU	Temporal Version Update
W3C	The World Wide Web Consortium
XML	Extensible Markup Language
XSL	Extensible Style Language

1 Einleitung und Motivation

Datenbank- und Softwaresysteme unterliegen im Allgemeinen Systemmodifikationen, die meistens nachhaltig eingepflegt werden müssen. Die dazu notwendigen Änderungsprozesse an Schemata ziehen somit oft Probleme nach sich, die vornehmlich durch Schema Evolutionsstrategien gelöst werden sollen. Ansätze der Schema Evolution ermöglichen es dabei, Schemaänderungen an befüllten Datenbanken vorzunehmen, ohne existierende Informationen zu verlieren.

Neben der Schema Evolution existiert das komplexere und stärkere Paradigma der Schema Versionierung, um Schemainformationen einer ganzen Versionslinie zu archivieren und zu verwalten. Dadurch wird es möglich, bestehende Anwendungen Zugriff auf alte Schemata zu geben, ohne sie anpassen zu müssen.

In dieser Arbeit werden verschiedene Ansätze und Techniken zur Versionierung von Schemata diskutiert und bewertet.

2 Grundlagen der Schema Versionierung

Dieses Kapitel zeigt zunächst die Grundlagen der Schemaversionierung. Dabei werden ausgehend von den allgemeinen Anforderungen an einen Versionierungsmechanismus die Kompatibilitätsaspekte erläutert und eine Abgrenzung zum Begriff der Schema Evolution durchgeführt.

2.1 Allgemeine Anforderungen

In diesem Abschnitt soll zunächst eine kurze Diskussion stattfinden, welche allgemeinen Bedingungen und Anforderungen zu einer Lösung der Versionierungsproblematik führen. Hierzu werden zunächst einige allgemeine Anforderungen für versionierbaren Code nach [Hoylen05] betrachtet. Diese Anforderungen wurden ursprünglich auf die Versionierung von XML-Schemata angewandt, können jedoch als allgemeingültige Aussage für jegliche zu versionierende Schemata betrachtet werden:

2.1.1 Versionierung mehrerer Generationen

Der Versionierungsmechanismus muss die Versionierung mehrerer Versionen erlauben. Dadurch muss eine Skalierbarkeit gewährleistet sein, um zu verschiedenen Zeitpunkten die Erstellung neuer Versionen aus bereits versionierten Schemata zu ermöglichen.

2.1.2 Autorenfreundlichkeit

Der Versionierungsmechanismus muss in der Lage sein, Instanzdokumente zu erstellen, die manuell editierbar und Menschenlesbar sind. Dieser Effekt muss während dem gesamten Versionierungsprozess sämtlicher Generationen anhalten und darf sich in späteren Versionen nicht verschlechtern.

2.1.3 Benutzung von Schemata und Instanzen

Der Versionierungsmechanismus muss aus dem Framework aus Instanzdokument und zugehörigem Schema bestehen.

Diese sehr allgemein gehaltenen Anforderungen sollen nun spezialisiert und konkretisiert werden. Hierzu sollen nun eher pragmatische Gründe der Versionierung nach [Roddick95] herangezogen werden.

Die durch den Datenbankadministrator auszuführenden Schemaänderungen, sollten einen minimalen Grad des Eingriffs in das Schema erfordern.

Der Grad des Eingriffs in eine vorhandene Schemaversion sollte demnach möglichst Minimal gehalten werden und angemessen in Bezug auf die auszuführenden Änderungen sein.

2.2 Kompatibilität bei Versionsänderungen

Zunächst einmal sollten Schemata gleichzeitig Vorwärts- und Rückwärtskompatibel sein. Rückwärtskompatibel insofern, dass das Format einer neuen Version dennoch eine valide Instanz einer älteren Version ist, das heißt eine Anwendung kann nach einer Schemaänderung noch immer auf Daten zugreifen, die vor der Änderung erzeugt wurden. Rückwärtskompatibel sind alle Änderungsoperationen, die subtraktiv sind. Additive Schemaänderungen hingegen sind nicht-Rückwärtskompatibel, da ältere Instanzen nicht auf die hinzugefügten Daten zugreifen können. Vorwärtskompatibilität (auch als Programmkompatibilität bezeichnet) wird erreicht, wenn alte Versionen auch noch bei neuen Versionen valide sind, das heißt Benutzer einer alten Version können noch immer mit dem Format der neuen Version arbeiten. Alle additiven Schemaänderungen sind somit Vorwärtskompatibel, da lediglich neue Informationen hinzugefügt werden, aber alte bestehen bleiben. [Obasanjo04]

Das wesentliche Problem der Versionierung besteht nun im Ausgleich zwischen Vorwärts- und Rückwärtskompatibilität, da eine Schemaänderung meist sowohl additive als auch subtraktive Modifikationen mit sich bringt.

2.3 Unterscheidung zwischen Versionierung und Erweiterbarkeit

Nach [Obsanjo04] gibt es vier grundlegende Klassen an Änderungen, die bei einer Versionsänderung auftreten können:

1. Neue Konzepte werden hinzugefügt (beispielsweise neue Elemente oder Attribute)
2. existierende Konzepte werden geändert (neue Interpretationen existierender Elemente beziehungsweise Attribute oder die Semantik der Elternelemente ändert sich)
3. existierende Konzepte werden abgelehnt (die Verwendung von existierenden Elementen oder Attributen durch eine Anwendung sollte Warnungen ausgeben)
4. existierende Konzepte werden gelöscht (die Verwendung existierender Elemente oder Attribute wird nicht länger unterstützt)

Die Klassen zwei bis vier sind nach Kapitel 2.2 Rückwärtskompatibel, da entweder die Semantik von bestehenden Konzepten geändert wird oder bestehende Elemente bzw. Attribute gelöscht werden. Klasse 1 hingegen ist Vorwärtskompatibel.

Da Versionierungsprobleme immer Vorwärts- als auch Rückwärtskompatibilität betrifft, kann eine Abgrenzung zu einer Erweiterung nach [Obsanjo04] durch zwei deutliche Merkmale identifiziert werden. Der wesentliche Unterschied ist, dass ein Versionierungsmechanismus Änderungen linear unterstützen muss, während Erweiterbarkeit dies nebenläufig bewerkstelligen muss. Versionierung muss also einen Mechanismus bereitstellen, damit Version 2 Rückwärtskompatibel zu Version 1 ist und eine Version 3 Rückwärtskompatibel zu Version 2 ist.

Erweiterungen werden typischerweise durch Dritte hinzugefügt. Ein Erweiterungsmechanismus muss also in irgendeiner Weise erlauben, neue Daten nebenläufig beziehungsweise nebeneinander hinzuzufügen, ohne die eigentlichen Kerndaten zu beeinflussen.

2.4 Abgrenzung zum Begriff Schema Evolution

Das komplexe Gebiet der Schema Evolution befasst sich mit der dynamischen Änderung der Zugehörigkeit von Objekten zu Klassen sowie der Änderung der Klassendefinition selbst [Heuer97]. Bei Datenbanksystemen beschreibt es beispielsweise die Fähigkeit, Schemaänderungen ohne Informationsverlust durchzuführen. Dies geschieht beispielsweise indem eine alte Schemaversion lediglich durch eine neue ersetzt wird. Dadurch wird die alte Schemaversion ungültig und gleichzeitig sämtliche Applikationen, die auf das alte Schema zurückgreifen unbrauchbar.

Abgrenzend zum Evolutionsprozess existiert der Begriff der Schema Versionierung, der sich indes mit der Verwaltung von verschiedenen Schemata beschäftigt, wodurch Applikationen weiterhin auf ältere Versionen zurückgreifen können. Datenbanksysteme müssen also die Fähigkeit besitzen, sowohl das Betrachten als auch das Ändern von Instanzen unter jeder Schemaversion zu ermöglichen.

3 Ausgewählte Versionierungstechniken

In diesem Kapitel sollen Vorgehensweisen spezifiziert werden, um versionierbare Schemata zu erstellen. In dieser Arbeit werden drei Ansätze vorgestellt. Zunächst wird das Modell nach [Clamen92] behandelt, in welchem das Ziel der Klassenversionierung verfolgt wird. Anschließend wird das Modell nach [Grandi00] vorgestellt, welches Ansätze von abzweigenden und temporalen Versionen beinhaltet. Abschließend erfolgt eine praktische Herangehensweise zur Erstellung von versionierbaren XML-Schemata nach [Costello05] und [Obsanjo04].

3.1 Class Versioning

Das Modell nach [Clamen92] bietet Programmkompatibilität durch einen Klassenversionierungsmechanismus für objektorientierte Datenbanken. Es nutzt wesentliche Eigenschaften des in [Kim88] spezifizierten Datenbankverwaltungssystems ORION und soll effizientere Evolutionsstrategien als die pure Emulation von alten Versionen im Datenbanksystem „Encore“ bieten. Dennoch ist „Class Versioning“ als abstraktes Framework zu betrachten und ist bis zum heutigen Zeigpunkt noch nicht praktisch umgesetzt.

Die Grundidee der Klassenversionierung ist die Existenz einer endlichen, nichtleeren Menge unterschiedlicher Versionen einer Klasse. Diese verschiedenen Versionen beschreiben quasi die Änderungshistorie einer Klasse. Für das Verständnis des Versionierungsmechanismus wird jedoch zunächst ein vereinfachtes Modell eines Objektorientierten Datenbanksystems nach [Clamen92] betrachtet.

3.1.1 Modell einer Objektorientierten Datenbank

Alle Objekte einer Datenbank sind Instanzen von Klassen. Eine Klasse ist weiter ein Datensatz mit Attributen und Methoden. Attribute definieren den internen Objektzustand und können nur über Zugriffs- und Änderungsmethoden angesprochen werden. Methoden stellen die Schnittstelle zur externen Welt dar und sind Operationen zur Beschreibung des Verhaltens.

Der Zugriff auf eine Klasse erfolgt über ihr Interface, welches über die öffentlichen Methoden der Klasse definiert wird. Das Klassengerüst ist in einer Typhierarchie als azyklischer, gerichteter Graph angeordnet, wodurch ein Klasseninterface eine Generalisierung seiner Superklassen und Supertypen ist.

Das Prinzip der Substituierbarkeit besagt, dass Instanzen einer Subklasse B in jedem Kontext verwendet werden können, in dem Instanzen der Superklasse A möglich sind – allerdings nicht umgekehrt [Kemper04]. Parallel dazu kann eine Klasse B, welche als Subtyp einer Klasse A deklariert wurde, in derselben Form referenziert werden, als wäre Klasse B Instanz der Superklasse A.

Die Gesamtheit aller Klassen stellt das Datenbankschema dar, in welchem jede Klasse eine eindeutige ID zugewiesen bekommt. Weiter erhält jedes Objekt der Datenbank eine eindeutige ObjektID und ist mit der KlassenID der zugehörigen Klasse verknüpft.

3.1.2 Konvertierung vs. Kompatibilität

Werden bei System wie ORION noch direkte Änderungen an einem einzigen logischen Schema vorgenommen, indem sämtliche Instanzen konvertiert werden, existieren bei einem Klassenversionierungsmechanismus viele verschiede Schnittstellen zu einer Klassen, und zwar jeweils eine pro Version.

Der Nachteil von Konvertierungsmechanismen liegt eindeutig bei der Programmkompatibilität. Dadurch sind Applikationen, welche sich auf ein altes Schema berufen, nicht mehr lauffähig. Klassenversionierung hingegen kann vollständige Programmkompatibilität bieten, da eine Applikation genau auf die Version zugreifen kann, die sie benötigt. Die Problematik besteht darin, dass das Datenbanksystem den Konflikt der von einer Applikation erwarteten Versionsinstanz lösen muss, welche im anschließenden Kapitel behandelt wird.

3.1.3 Managen von verschiedenen Versionen

Eine Klasse kann wie beschrieben verschiedene Versionen besitzen, wodurch nach Clamen jede Instanz eine Zusammensetzung von multiplen Facetten ist. Eine Facette stellt somit den Instanzzustand eines bestimmten Interfaces beziehungsweise einer Version dar. Während bei anderen Ansätzen wie Encore für jede Version ein eigenständiges Interface mit eigenem Speicherbereich

angelegt wird und die verschiedenen Versionen untereinander gewrapped werden, bildet Clamen Beziehungen zwischen den Objektrepräsentationen unterschiedlicher Versionen. Die Umsetzung von bereits existenten Attributen einer Objektrepräsentation in eine andere Version kann mittels Funktionen geschehen. Hierfür teilt Clamen die Attribute in vier verschiedene Gruppen ein:

- Geteilt: ein Attribut ist in beiden Versionen enthalten
- Unabhängig: ein Attribut steht zu keiner Beziehung zu irgendeinem anderen Attribut der anderen Version
- Abhängig: Der Wert eines Attributs ist abhängig von dem Wert eines Attributs der anderen Version, kann jedoch nicht automatisch aus diesem Wert berechnet werden
- Abgeleitet: Ein Attributwert kann direkt aus einem Attributwert der anderen Version abgeleitet beziehungsweise berechnet werden

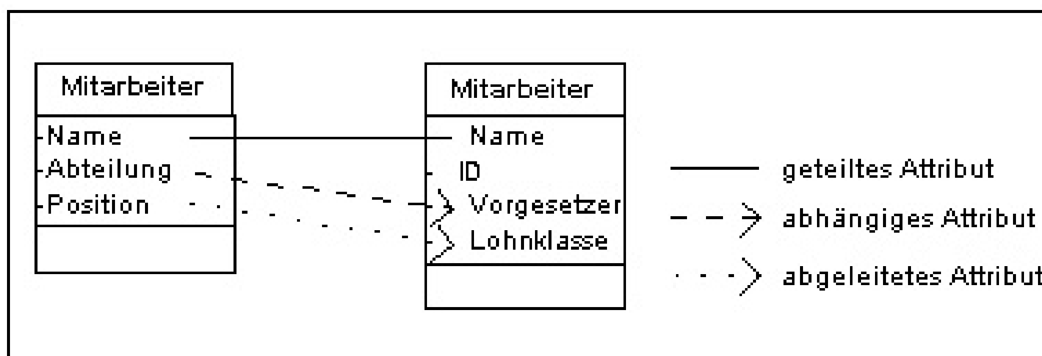

Abbildung 1: Repräsentation unterschiedlicher Versionen einer Klasse Mitarbeiter

Die vier Attributgruppen sollen nun anhand eines Beispiels verdeutlicht werden, welches auf Abbildung 1 dargestellt ist. Im dargestellten Beispiel der Klasse *Mitarbeiter* existiert in beiden Versionen das unveränderte Attribut *Name* und ist somit ein geteiltes Attribut. Die *ID* hingegen ist ein unabhängiges Attribut, da sie neu hinzugefügt worden ist und keine Beziehung zu einer anderen Version existiert. Das Attribut *Vorgesetzter* ist abhängig von der *Abteilung*. Ändert sich die Abteilung eines Mitarbeiters, ändert sich zwangsläufig ebenfalls der Vorgesetzte, aber nicht unbedingt umgekehrt. Ändert sich der Vorgesetzte kann dies mit einem Abteilungswechsel zusammenhängen, oder aber auch durch eine Neubesetzung

der Vorgesetztenstelle geschehen. Die Abhängigkeit muss allerdings intern mit einer zugehörigen Funktion definiert werden, in welcher die Zuordnungen zwischen beiden Versionen vorgenommen werden. *Lohnklasse* ist ein abgeleitetes Attribut aus *Position*, das heißt anhand einer Position lässt sich mit Hilfe einer Ableitungsfunktion die zugehörige Lohnklasse bestimmen. Eine einfache Ableitungsfunktion ordnet beispielsweise jeder Position eine bestimmte Lohnklasse zu:

$$Lohnklasse = \begin{cases} if\ \ Position = "Junior"\ Lohnklasse = 1 \\ if\ \ Position = "Senior"\ Lohnklasse = 3 \\ if\ \ Position = "Abteilungsleiter"\ Lohnklasse = 4 \end{cases}$$

3.1.4 Physische Repräsentation von verschiedenen Versionen

Während sich das vorherige Kapitel mit der semantischen Schemabeschreibung befasste, wird an dieser Stelle eine mögliche physikalische Umsetzung in einem Datenbanksystem beschrieben.

Eine Applikation greift immer nur auf ein Interface, also eine Version zu und Attributsänderungen im aktuellen beziehungsweise sichtbaren Interface werden sofort an alle andern Versionen propagiert. Dieser Mechanismus sollte nach Clamen der üblichen Trigger-Funktionalität der Relationalen Datenbanken entsprechen.

Die Erstellung einer neuer Facette, also die Aktualisierung eines Objektes einer Instanz, kann hingegen verzögert ausgeführt werden. Dadurch können Speicherkosten reduziert werden und die Objekte nur bei einem Zugriff aktualisiert werden. Ein Attribut in einer anderen Version wird somit nur aktualisiert, wenn darauf zugegriffen wird. Die Aktualität soll intern durch ein Flag geprüft werden, das bei jedem Zugriff einer Applikation ausgelesen wird. Der Nachteil dieser Methode ist die verzögerte Zugriffzeit auf ein Objekt, wenn es bei Zugriff zunächst überprüft werden muss und es möglicherweise zuvor zu einer Resynchronisation kommen kann.

3.1.5 Modifikation der Klassenhierarchie

Die zuvor diskutierten Aspekte betreffen lediglich Änderungen in einer einzelnen Klasse. Wie werden jedoch Modifikationen der verschiedenen Klassen

untereinander propagiert und wie machen sich Änderungen der Klassenhierarchie bemerkbar? Diese Gesichtspunkte sollen fortführend erläutert werden.

Änderungen der Klassenhierarchie werden in objektorientierten Datenbanksystemen durch Subtypenbeziehungen vorgenommen. Während die Modifikation eines Attributes beziehungsweise Methode des Supertyps noch durch einfache Anpassung erfolgen kann, gibt es Probleme bei evolutionären Anpassungen des Supertyps. Änderungen in einer Superklasse werden unabhängig zu ihren Subklassen entwickelt, wodurch Modifikationen nicht an Subklassen propagiert werden. Das Hinzufügen einer neuen Methode in einem Supertyp führt zu einer Verletzung der Vererbungsinvariante, welche besagt, dass jede Subklasse alle Methoden ihrer Superklasse beinhaltet. Das Löschen einer Methode im Supertyp hingegen verletzt die Vererbungsinvariante nicht.

Andere Systeme wie ORION verbieten einfach die Änderung des Typgraphen [Bannerjee87]. Clamen schlägt hingegen eine Typhierarchieversionierung vor. Das Prinzip der Klassenversionierung wird somit auf einer höheren Hierarchieebene angewendet. Die Zugehörigkeit eines bestimmten Typs ist dadurch immer abhängig von der Version des zugehörigen Supertyps. Eine konkrete Vorgehensweise der Hierarchieproblematik findet bei Clamen jedoch nicht statt.

3.1.6 Bewertung des Systems

Das Modell nach Clamen unterstützt Programmkompatibilität durch einen Klassenversionierungsmechanismus. Dies wird erreicht indem für jedes Objekt einer Klasse genau eine Version pro Klassenversion existiert. Dadurch ist sowohl das Lesen als auch Ändern von Objekten anderer Klassenversionen möglich.

Das Modell bietet keine vollständige Schemaversionierung, da der Fokus immer nur auf einer Version einer Klasse liegt, die unabhängig voneinander versioniert werden. Aus der Konklusion der Klassenversionen einer Klasse in Zusammenhang mit der Klassenhierarchie könnte jedoch eine vollständige Schemaversionierung stattfinden.

Die Versionierung der Attribute verschiedener Versionen wird mittels Abhängigkeitsfunktionen adaptiert. Durch diese Vorgehensweise entstehen allerdings Probleme bei der Konvertierung einer Objektrepräsentation in ein

beliebiges Format. Das Konvertierungsprinzip ist hier nur manuell zu lösen, indem der Mechanismus bei unmöglichen Adaptionsvorschriften Default-Werte benutzt und diese später durch den Administrator manuell eingepflegt werden müssen.

Eine Implementierung des Klassenversionierungsmechanismus fand aus den sich ergebenen Problemen, vor allem aber durch die Klassenhierarchieproblematik bis dato noch nicht statt.

3.2 Branching and Temporal Versioning

[Grandi00] stellt ein Framework von abzweigenden und temporalen Versionen (engl. branching and temporal versioning) vor, welches durch eine nachvollziehbare Versionslinie Zwischenversionen in objektorientierten Datenbanken zulässt. Branching findet ursprünglich vor allem in CAD/CAM-Bereichen Einsatz, um im dynamischen Designprozess unterschiedliche Szenarien durchspielen zu können. [Katz90]

Der temporale Aspekt ist notwendig, um den Objektevolutionsprozess innerhalb einer Version dokumentieren und archivieren zu können, das heißt die temporale Versionierung repräsentiert den Verlauf der Objektstruktur anhand einer definierten validen Realzeit und/oder der Transaktionszeit. Durch den Zeitstempel ist es dann möglich einen beliebigen Zustand innerhalb der Version zu rekonstruieren. Somit ist es bei diesem Modell möglich verschiedene konsolidierte Versionen zu integrieren, die in einer bestimmten Beziehung zueinander stehen und zusätzlich zu jeder Version temporale Versionen existieren können.

3.2.1 Datenmodell

Das Modell wird nach [Grandi00] als gerichteter, azyklischer Graph dargestellt und ist auf Abbildung 2 abgebildet. Es ermöglicht durch konsolidierte und temporale Versionen sowohl die Schemaentwicklung als auch die Administration dieser. Im dargestellten Szenario existieren vier konsolidierte Versionen CV_1 - CV_4 mit der Möglichkeit aus einer dieser Versionen eine neue Version zu generieren. Die Datenbank wird als Zusammenstellung von Knoten und Kanten dargestellt. Ein Knoten entspricht einer konsolidierten Version. Eine Kante ist die Beziehung einer konsolidierten Version zu einer korrespondierenden oder abgeleiteten Version. Die zuerst eingeführte konsolidierte Version CV_1 entspricht weiter dem Wurzelelement

des Graphen. Jede konsolidierte Version erlaubt die Einbringung von so genannten bitemporalen Versionen, die in der Abbildung mit TV_n gekennzeichnet sind. Unter Bitemporal versteht [Grandi00] die Definition einer temporalen Version anhand der zwei Dimensionen Echtzeit und Transaktionszeit, wodurch sowohl aktuelle, in der Vergangenheit liegende oder in der Zukunft liegende Versionen generiert werden können.

Die eindeutige Identifikation einer Version ist durch das Tupel {Versionsname, Echtzeit, Transaktionszeit) definiert. Durch die Angabe eines Zeitintervalls können weiter alle Versionen erfasst werden, die in einem bestimmten Zeitrahmen entstanden sind.

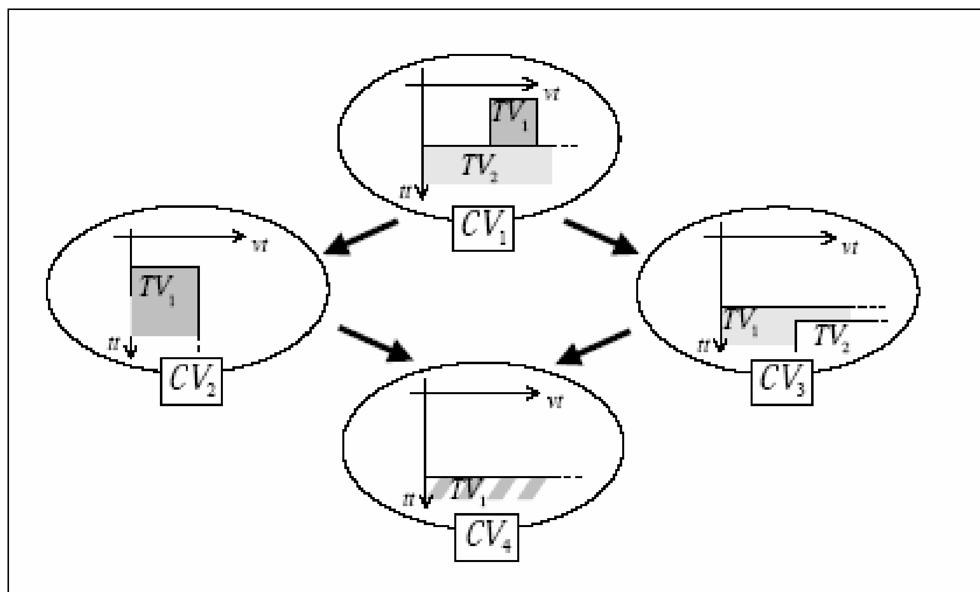

Abbildung 2: Beispielgraph nach [Grandi00]

Konsolidierte Versionen können in einem Designprozess stabile Stationen entsprechen, in welchen Zwischenversionen durch temporale Versionen umsetzbar sind. In einer Datenbankumgebung ist es aber auch möglich eine konsolidierte Version als eine Schemaoption zu betrachten, in welcher andere konsolidierte Versionen grundlegend verschiedene Schemaimplementierungen entsprechen könnten. Temporale Versionen entsprächen dann den Schemaänderungsverlauf eines fest definierten Schemas.

3.2.2 Versionsmanagement

Für die Umsetzung des Branching erweitert [Grandi00] die elementaren Operationen für Schemaänderungen um Graphenoperationen, die fortführend näher betrachtet werden und in einer Übersicht in Tabelle 1 dargestellt sind.

Schemaänderung auf Knotenebene
Attribut/Methode/Klasse hinzufügen
Attribut/Methode/Klasse löschen
Attribut-/Methoden-/Klassennamen ändern
Attributtyp/Methodencode ändern
Superklasse hinzufügen
Superklasse löschen
Vereinigung von Schemamerkmalen
Attribut aus Klasse übernehmen
Klasse übernehmen
Methode übernehmen
Version übernehmen
Graphenänderung
Neuen Knoten erzeugen
Neue Kante erzeugen

Tabelle 1: Übersicht einfacher Schemaänderungen

Schemaänderungen auf Knotenebene umfassen sämtliche herkömmlichen Operationen, die auf objektorientierte Datenbanken ausführbar sind.

Die Vereinigung von Schemamerkmalen erlaubt die Integration von Elementen oder einer kompletten temporalen Version in eine konsolidierte Version. Dabei werden neben den Objekteigenschaften zusätzlich die enthaltenen Daten übernommen.

Durch die Graphenänderungoperationen wird die Ableitung einer existenten Version (neue Kante) oder die Einbringung einer neuen konsolidierten Version (neuer Knoten) gewährleistet.

3.2.3 Semantik der Schemaänderungen

In diesem Abschnitt findet eine Beschreibung der Semantik für die in Tabelle 1 beschriebenen Schemaänderungen statt. Primitive Schemaänderungen werden durch eine Database Update Funktion (DBU) wie in Abbildung 3 gezeigt gemanaged.

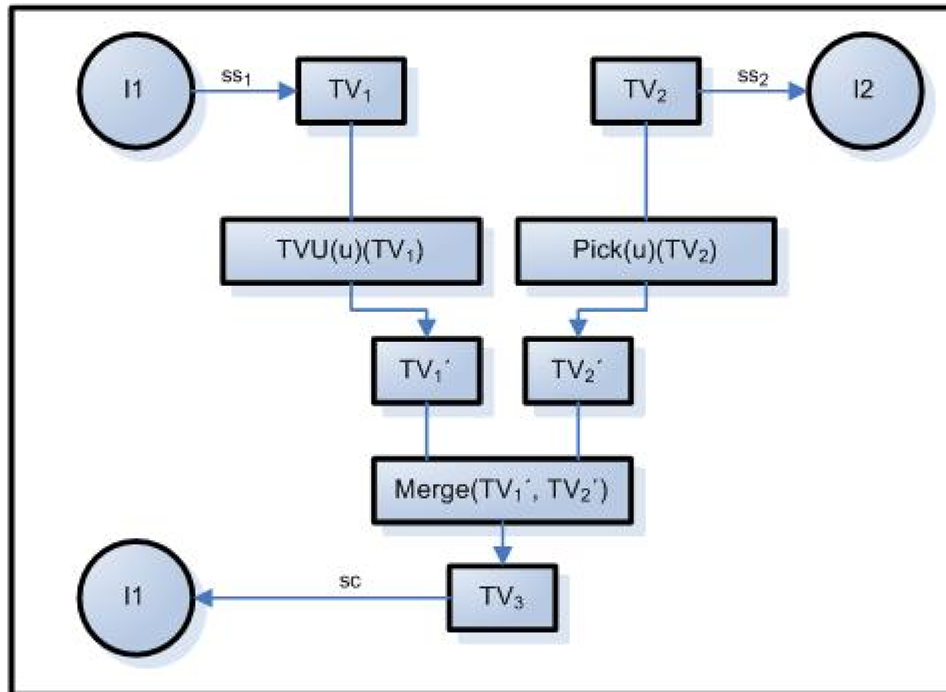

Abbildung 3: Graphenverhalten der Schemaänderung auf eine Kante

Jede Schemaänderung betrifft im Allgemeinen zwei konsolidierte Versionen I1 und I2. Beide können Ausgangspunkt für einen Änderungsprozess sein. Eine Auswahl an Änderungsoperationen soll nachfolgend beschrieben werden:

1. Die Erstellung einer neuen konsolidierten Version L´ geschieht durch die Updateoperation auf einen bereits existenten Knoten I1 durch eine Schemaänderung u und wird mittels der Funktion $L' = L\ u\ \{l1\}$ beschrieben.

2. Eine neue Kante E´ wird mittels der Funktion $E' = \begin{cases} E\ u\ \{(l2, l1)\} & wenn\ l2 \neq null \\ E & sonst \end{cases}$ erstellt.

3. Jede konsolidierte Ausgangsversion zeigt auf eine ausgewählte temporale Version, die durch die zwei Zeit-Parameter ss_n (schema selection validities) definiert sind. Eine selektierte temporale Version TV1 wird mittels der

Funktion $TV_1 = CDB(l_1, ss_1)$ beschreiben, wobei CDB die Current Database Funktion darstellt.

4. Eine temporale Version wird durch Updateoperation TVU (Abk. Temporal Version Update) eine Schemaänderung u auf eine selektierte temporale Version erzeugt: $TV_1' = TVU(u)(TV_1)$

5. Der Verbundsoperator (engl. Merge) verbindet zwei existente temporale Versionen zu einer neuen Version und kann durch die Funktion $TV_3 = Merge(TV_1', TV_2')$ beschrieben werden.

6. Die neue temporale Version TV_3 wird abschließend als neue temporale Version von l1 hinzugefügt.

Die ausgewählten semantischen Änderungen eines Schemas werden durch eine *DBU* formalisiert, die abstrakt durch die Funktion $DBU(Operation)(Parameter)$ abgebildet werden kann. Alle möglichen Änderungsoperationen können auf Graphenebene abgebildet werden. Eine Datenbankgraph (DBG) wird nach [Grandi00] durch die Funktion $DBG' = (L', E', DB') = DBU(u)(DBG, l_1, ss_1, l_2, ss_2, sc)$ definiert, wobei *sc* die schema change validation darstellt, die durch die Parameter Echtzeit und Transaktionszeit gekennzeichnet ist.

Auf Graphenebene wäre die Einbringung eines neuen Knoten beispielsweise durch eine Updateoperation mittels der Funktion $DBG' = DBU(NewNode)(DBG, l_1, null, null, null, null)$ umsetzbar. Eine neue Kante hingegen kann mittels $DBG' = DBU(NewEdge)(DBG, l_1, null, l_2, null, null)$ erzeugt werden.

3.2.4 Bewertung des Systems

[Grandi00] bietet einen abstrakten Ansatz zur verzweigenden (branching) und temporalen Versionierung von Schemata, welcher durch einen Graphen modelliert wird. In dem weit reichenden definierten Rahmenwerk werden umfassende Schemaänderungsoperationen definiert, die mit Hilfe von Graphenoperationen für das Branching erweitert werden.

Das vorgestellte Modell ist für die Erstellung von unterschiedlichen Szenarien mit eigenen Versionshistorien geeignet. Die Spezifikation bietet zwar eine Vielzahl an Änderungsoperationen am Graphen, jedoch gehen die Autoren nicht näher auf die Bereitstellung der Instanzen ein, das heißt es werden keine Angaben über Umsetzungsmechanismen gegeben und wurde ebenfalls bis heute noch nicht implementiert.

3.3 XML Schema Versionierung

Einleitend mit einer Übersicht von XML (Extensible Markup Language) und vor allem von XML Schema, zeigt dieses Kapitel einige Methoden zur Erstellung von versionierbarem XML-Code, welches anschließend mit einem Best-Practice-Beispiel abgeschlossen wird.

3.3.1 XML – Eine Einführung

Die XML ist eine Auszeichnungssprache (im Folgenden als Markup-Sprache bezeichnet), welche die Vorteile von SGML (Standard Generalized Markup Language) und HTML (Hypertext Markup Language) miteinander vereint.

Für die Anwendung im Web erwies sich der Einsatz von SGML nach [Trähnert05] als grundsätzlich zu teuer und zu kompliziert. Sämtliche semantischen Erweiterungen in HTML erwiesen sich hingegen als permanent inadäquat. Zwar wurde durch Standardisierung versucht, semantische Erweitungen in HTML einzubringen, jedoch besteht dabei die Gefahr, die Elementmenge auf eine unbestimmte Größe zu erweitern. Diese Entwicklungen führten zur Entwicklung von XML, welches die wichtigsten Funktionen von SGML zur Verfügung stellt und die bekannten Dinge aus HTML nutzt.

Eine Übersicht der XML-Standards sowie einiger damit verbundener Anwendungen und Werkzeuge sind auf Abbildung 4 dargestellt.

Für die Anzeige sowie das Editieren von XML-Dokumenten können schon einfache Texteditoren oder Browser herangezogen werden. Jedoch gibt es auch umfangreiche Werkzeuge, wie beispielsweise XMLSpy [XMLSpy05], die einen komfortablen Zugriff auf XML-Dokumente ermöglichen.

Weiterhin ist XML nicht ein einzelner Standard, sondern vielmehr als Familie von Techniken zu verstehen, die miteinander kombiniert werden können. Einzelne Bestandteile, die auf Abbildung 4 abgebildet sind, werden im Folgenden genannt:

- XLink als Standardmethode, um Hyperlinks in XML-Dateien einzufügen [XLink01]
- XPointer sind Syntaxen, um auf Teile eines XML-Dokuments zu verweisen [XPointer01]
- XSLT (XSL Transformation) ist eine Transformationssprache, die XML-Dokumente in bestimmte Zielformate (wie PDF oder HTML) überführt [XSLT99]
- XSL (Extensible Style Language) ist eine Weiterentwicklung, um Style Sheets zu erstellen und basiert auf XSLT [XSL05]
- DOM (Document Object Model) ist eine Standardmenge von Funktionsaufrufen für Programmiersprachen [DOM05]
- XML Schema 1 und 2 unterstützten Entwickler bei der präzisen Definition eigener XML-Formate [XSD00]

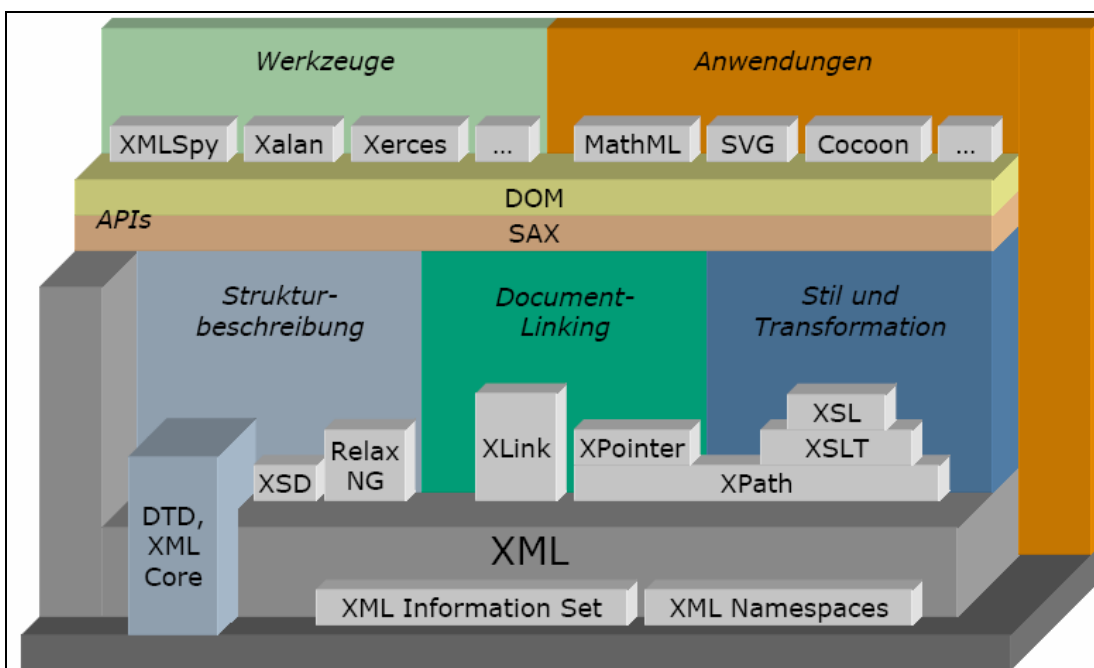

Abbildung 4: XML Übersicht nach [Trähnert05]

Mit XML ist es möglich eigene Markup-Sprachen zu entwickeln, in der beispielsweise eigene Tags für bestimmte Elemente definiert werden können, denen bestimmte logische Bedeutungen zugewiesen werden können.

Die Verarbeitung von XML-Dokumenten erfolgt mit XML-Parsern, welche eine XML-Instanz anhand eines zugrunde liegenden Schemas auf Validität prüfen. Ein Schema definiert die Struktur, den Inhalt und die Semantik eines XML-Dokuments [XSD00]. Für die Umsetzung von Schemata eignen sich eine DTD (Document Type Definition), XSD (XML Schema Definition) oder RelaxNG. Eine DTD dient lediglich der Dokumentbeschreibung und ist nicht mächtig genug, komplexe Schemadefinitionen umzusetzen. RelaxNG hingegen ist ebenfalls sehr mächtig, bietet eine intuitive Syntax und steht in direkter Konkurrenz zu XML Schema. Die vorliegende Arbeit behandelt jedoch in Anlehnung an die W3C-Spezifikation die Schema-Definition mittels XSD.

3.3.2 XML-Schema

Eine vom World Wide Web Consortium (W3C) spezifizierte Sprache zur Definition von XML-Vokabularen ist XML-Schema, welche auch als XML Schema Description Language (kurz XSD) bekannt ist. XML-Schema ist eine mächtige Beschreibungssprache und nutzt selbst die Syntax von XML, während DTDs XML-Dokumente lediglich in einer einfachen und vor allem eigenen Syntax beschreiben.

XML-Schema stellt bereits eine Vielzahl an vordefinierten Datentypen und bietet die Möglichkeit, eigene komplexe Datentypen zu definieren. Eine Auswahl der Datentypen wird fortführend aufgezeigt:

- xs:string Zeichenketten und Text
- xs:integer ganzzahliger Integerbereich
- xs:decimal Fließkommazahlen
- xs:boolean boolsche Were
- xs:date Datum im Format Jahr-Monat-Tag

Weiterhin soll ein einfaches Beispiel die Anwendung von XML-Schema im Vergleich zu einer DTD veranschaulichen.

artikel.xml

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <!DOCTYPE artikel SYSTEM "artikel.dtd">
3  <artikel>
4     <titel>Der Titel</titel>
5     <teaser>Der Teaser</teaser>
6     <inhalt>Der Inhalt</inhalt>
7  </artikel>
```

artikel.dtd

```
1  <!ELEMENT artikel (titel, teaser, inhalt )>
2  <!ELEMENT titel (#PCDATA)>
3  <!ELEMENT teaser (#PCDATA)>
4  <!ELEMENT inhalt (#PCDATA)>
```

artikel.xsd

```
1  <?xml version="1.0"encoding="ISO-8859-1"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3    <xs:element name="artikel">
4     <xs:complexType>
         <xs:sequence>
5          <xs:element name="titel" type="xs:string" use="required"/>
6          <xs:element name="teaser" type="xs:string"/>
7          <xs:element name="inhalt" type="xs:string"/>
8        </xs:sequence>
9      </xs:complexType>
10   </xs:element>
11 </xs:schema>
```

In der ersten Zeile des XML-Schema-Dokuments werden zunächst XML-Version und Zeichensatzkodierungen spezifiziert. Anschließend wird der Namespace definiert, welcher das Wurzelelement bildet und das gesamte Dokument umschließt. Er ist wichtig, um Konflikte bei der doppelten Verwendung von etwaigen Elementnamen zu verhindern und wird in der Form „xmlns:“ verwendet. In Zeile 3 wird wird das Wurzelelement eines XML-Instanz-Dokuments definiert. Der Name eines Elements wird hierbei als Attribute „name=“ angegeben. Die Angabe von „<xs:complexType> spezifiziert, dass das Wurzelelement weitere Unterelemente beinhaltet. Weiterhin werden drei einfache Elemente definiert, die nicht weiter verschachtelt sind und den Datentyp „xs:string“ haben. Das Element „name=“titel““ wird allerdings mit dem Attribut „use=“required““ definiert, wodurch

definiert wird, dass das Element unbedingt im Instanzdokument angegeben werden muss.

In dem Beispiel lässt die DTD keine Aussage über etwaige Einschränkungen zu, sondern lediglich ob es sich um ein Element oder Attribut handelt und um welchen einfachen Datentyp es sich handelt.

XML-Schema bietet komplexe Datentypen, constraints und Einschränkungen, wodurch es beispielsweise möglich ist, Aussagen über die Länge eines Strings oder den Wertebereich zu definieren.

3.3.3 Versionierungsmöglichkeiten

Für die Versionierung von XML Schemata gibt es keine optimale Vorgehensweise, an die sich ein Entwickler zu halten hat. Viel mehr sind die hier aufgeführten Varianten Vorschläge, die im besten Fall kombiniert werden können, da alle Varianten sowohl Nach- als auch Vorteile aufweisen. Einige Vorschläge der XML-Schema Versionierung, die sich nach [Costello05] als Best Practice-Fälle herausgestellt haben, werden fortführend näher betrachtet.

Prinzipiell können in XML-Schemata zwei verschiedene Arten an Änderungsoperationen eintreten:

1. Die Interpretation einiger Elemente ändert sich, das heißt betroffene Elemente sind durch etwaige veränderte Datentypen in einer neuen Version nicht mehr valide.

2. Der Schema-Namensraum wird erweitert. Instanzdokumente, die gegen eine alte Schema-Version valide waren, behalten ihre Gültigkeit, da Namensraum-Erweiterungen bestehende Elemente beziehungsweise Attribute belassen und neue hinzufügen.

3.3.3.1 Ändern von Namen oder Speicherort

Eine ziemlich einfache und zunächst intuitiv erscheinende Möglichkeit der Versionierung ist das Ändern von Lokation und Namen von XML-Schema Dokumenten. Der Autor versieht sein Schema anhand von eigenen Konventionen

mit einem repräsentativen Namen und einer jeweiligen Versionsnummer oder Erstellungsdatum.

Diese Variante zieht allerdings nur Nachteile mit sich. Zunächst besteht bereits das Problem der Konventionsfindung. Allein die Wahl eines einfachen Zeitstempels, wie beispielsweise Monat-Jahr, kann Konflikte verursachen, falls im selben Monat eine neue Version erscheint.

Ein weiteres Problem entsteht, da das Schema intern nicht auf Validität geprüft werden kann. Das Abprüfen der Versionsnummer ist so nur durch externe Applikationen durchführbar. Zwar ist somit eine Prüfung durchführbar, ob eine neue oder veränderte Version vorliegt, allerdings erhält eine Applikation keine Mitteilung, welche Elemente und Attribute geändert worden sind.

Weiterhin müssen sämtliche Instanzdokumente immer wieder mit der Referenz auf das Schema angepasst werden, selbst wenn die Schemaänderungen in einer Instanz keine Auswirkungen haben. Ähnliche Änderungsoperationen müssen in Schemata vorgenommen, welche das geänderte Schema importieren.

Der weitaus größte Nachteil ergibt sich allerdings aus der Tatsache, dass das schemaLocation Attribut eines Instanzdokuments optional ist. Es dient lediglich als Hilfe für den Prozessor, um ein passendes Schema zu finden.

Wird die Masse an Nachteilen dieser Methode betrachtet, wird deutlich, dass dies kein gutes Versionierungsverfahren ist.

3.3.3.2 Ändern der internen Schema-Versions-Attribute

Eine weitere einfache Möglichkeit bietet das optionale Versions-Attribut, welches einfach geändert werden kann und bereits in der Schemaspezifikation beschrieben wird. Listing 1 zeigt eine Beispielimplementierung der aufgeführten Variante:

```
12 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
13           elementFormDefault="qualified"
14           attributeFormDefault="unqualified"
15           Version="1.0">
```

Listing 1: Schema-Versions-Attribut

Neben der einfachen Anwendung brauchen Instanzdokumente bei einer Namensraumerweiterung nicht angepasst werden und bleiben valide. Weiterhin erhalten Applikationen anhand des Versions-Attributes einen Hinweis, dass es sich um eine neue Version handelt und können darauf reagieren. Allerdings wird das Versionsattribute bei der Validitätsprüfung nicht einbezogen.

3.3.3.3 Erstellen eines Schema-Versions-Attributes im Wurzelelement

Das Erzeugen eines eigenen Versions-Attributes mit abprüfbaren Attributeigenschaften am Wurzelelement stellt eine Erweiterung von 3.3.3.2 dar und bietet zwei verschiedene Einsatzmöglichkeiten.

1. Es wird ein neues Attribut erstellt und mit den Flags „required“ und „fixed value“ bestückt. Somit muss jedes Instanzdokument, welches das zugrunde liegende Schema verwendet, das Versionsattribut setzen und wird ebenfalls durch den Validationstest überprüft. Das folgende Beispielschema in Listing 2 besitzt das Versionsattribut „schemaVersion“ und wird auf Version „1.0“ gesetzt:

```
1  <xs:schema xmlns:xs=http://www.w3.org/2001/XMLSchema
2     elementFormDefault="qualified"
3     attributeFormDefault="unqualified">
4      <xs:element name="Beispiel">
5        <xs:complexType>
6          < ... >
7          <xs:attribute name"schemaVersion"
8              type="xs:decimal" use="required" fixed="1.0"/>
9        </xs:complexType>
10     </xs:element>
```

Listing 2: festes Versions-Attribut im Wurzelelement

   Durch die Validitätsüberprüfung wird das Versionsattribut zu einer bindenden Bedingung. Allerdings sind nur Instanzdokumente valide, welche die im Schema ausgewiesene Versionsnummer besitzen. Allein durch eine Namensraumerweiterung, wäre eine Instanz, die eigentlich noch gültig ist nicht mehr valide.

2. Eine weitere Möglichkeit des Einsatzes eines Schema-Versions-Attributes wird dadurch erreicht, indem angegeben wird, mit welcher Schemaversion ein Instanzdokument kompatibel ist. Hier ist das Versions-Attribut gerade nicht fixiert, sondern ist eine Liste oder eine definierte Konvention, die

aussagt, mit welchen Schemata ein Dokument valide ist. Beispielsweise kann eine Konvention angegeben werden, die aussagt, dass ein Instanzdokument mit einer Schemaversion 1.6 und früheren Versionen kompatibel ist. In der Umsetzung muss es einen Mechanismus geben, also eine externe Applikation, der prüft zu welchen Versionen ein Dokument valide ist.

```
1. <xs:schema xmlns:xs=”http://www.w3.org/2001/XMLSchema”
2.     elementFormDefault=“qualified“
3.     attributeFormDefault=“unqualified“
4.     version=“1.6“>
5.
6.     <xs:element name=“Beispiel“>
7.       <xs:complexType>
8.         < ... >
9.     <xs:attribute name“schemaVersion“
10.              type=“xs:decimal“ use=“required“/>
11.         </xs:complexType>
12.       </xs:element>
```

Listing 3: Beispielschema der Version 1.6

```
1. <Beispiel schemaVersion=“1.2“
2.      xmlns=http://www.beispiel
3.      xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
4.      xsi:schemaLocation=”http://www.beispiel
5.      Beispiel.xsd”>
```

Listing 4: Beispielinstanzdokument der Version 1.2

In Listing 3 ist ein Beispielschemadokument dargestellt, in welchem eine Schema-Version 1.6 deklariert wird. Das Instanzdokument in Listing 4 weist die Version 1.2 aus. Dies bedeutet, das Instanzdokument ist ohne Konventionsprüfung mit einer Schemaversion 1.2 kompatibel. Durch die Einbringung einer Applikation kann jedoch eine Konvention erstellt werden, die beispielsweise aussagt, dass das Instanzdokument mit einer Schemaversion 1.2 und abwärts kompatibel ist.

Der große Nachteil des vorher betrachteten Falls wird durch diese Lösung behoben, indem Instanzdokumente nicht angepasst werden müssen, wenn sie weiterhin gegen neue Schemaversionen valide sind. Wie in 3.3.3.2 erhält eine externe Applikation ebenfalls eine Nachricht, wenn sich ein Schema geändert hat. Durch die Einbringung einer Applikation, welche die Konvention prüft, wäre es denkbar, dass einem Instanzdokument zusätzlich die korrekte *SchemaLocation* bekannt gegeben wird, falls die Version nicht mehr gültig ist.

Gerade durch die Einbringung einer externen Applikation müssen aber neben der internen Validitätsprüfung weitere Prozesse abgewickelt werden. Neben dem eigentlich internen Parservorgang wird noch eine externes parsen vorangestellt, welcher die Konventionsprüfung übernimmt.

3.3.3.4 Anpassung des Schema-Namensraums

Weiterhin besteht die Möglichkeit, die Schemaversion im targetNamespace des Schemadokuments zu verankern, wie in dem Beispiel in Listing 5 dargestellt wird:

```
1. xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
2.     xmlns="http://www.SchemaV1.0"
3.     targetNamespace="http:www.SchemaV1.0"
4.     elementFormDefault="qualified"
5.     attributeFormDefault="unqualified">
```

Listing 5: Schemaversion im targetNamespace

Ein Instanzdokument wird durch diese Option bei einer Schemaänderung benachrichtigt, da das Dokument den Namenraum nicht mehr finden kann. Ein Instanzdokument muss somit immer angepasst werden, wenn sich der Namensraum ändert. Einerseits werden somit Kompatibilitätsprobleme minimiert, jedoch müssen sämtliche Instanzdokumente, die auf den alten Namensraum verweisen abgeglichen werden. Neben der Anpassung sämtlicher zugreifender Instanzdokumente müssen zusätzlich sämtliche Schemata, die das versionierte Schema einbinden ebenfalls angepasst werden.

3.3.4 Best Practices

In diesem Abschnitt soll eine mögliche Vorgehensweise zur Versionierung von XML-Schemata vorgestellt werden. Hierbei handelt es sich nach [Costello05] um prinzipielle Gedanken, die bei der Erstellung und Modifikation eingebracht werden sollten.

1. Identifikation der Schema-Versionsnummer, welche im XML-Schema spezifiziert wurde

2. Identifikation mit welchen Schemaversionen das Instanzdokument kompatibel ist

3. ältere Schemaversionen verfügbar machen, um Applikation den Zugriff auf ältere Versionen möglich zu machen und somit die Möglichkeit schaffen, ältere Versionen auf neue Versionen zu migrieren.

4. Bei einer Schemaerweiterung, wie dem Hinzufügen neuer Elemente oder Attribute, sollten existierende Instanzdokumente valide bleiben. Die kann geschehen, indem neue Elemente/Attribute, lediglich optionalen Charakter haben.

5. Bei Änderung der Interpretation von Elementen/Attributen eines existenten Schemas, sollte ein neuer Namensraum eingeführt werden, um Namenskonflikte zu verhindern

Wie die beispielhafte Vorgehensweise aufzeigt, liegt es bisher noch vollkommen an den Fertigkeiten des Entwicklers, in welcher Form die XML-Schemata versionierbar sind. Eine umfassende Validitätsprüfung ist nur durch manuellen Eingriff durchführbar und erfordert den Einsatz von externen Applikationen, die ebenfalls selbst entwickelt werden müssen. Ein Framework, welches zumindest einige Versionierungsrichtlinien umsetzt wäre somit wünschenswert.

3.4 Zusammenfassung und Ausblick

In dieser Seminararbeit wurden Konzepte und signifikante Techniken zur Versionierung von Schemata vorgestellt. Dazu wurden ausgehend von einigen Grundlagen bezüglich Schema Evolution und Kompatibilität drei verschiedene Konzepte diskutiert und bewertet.

Class Versioning nach [Clamen92] bietet einen abstrakten Klassenversionierungsmechanismus für objektorientierte Datenbanksysteme indem für jedes Objekt einer Klasse eine Version pro Klassenversion existiert.

Der Ansatz vom „Branching and Temporal Versioning“ nach [Grandi00] beschreibt ein Modell von abzweigenden und temporalen Versionen für objektorientierte Datenbanksysteme. Das Modell ermöglicht eine nachvollziehbare Versionslinie mit konsolidierten Versionen und temporalen Zwischenversionen.

Beide Modelle sind abstrakte Vorgehensweisen und wurden bisher noch nicht implementiert. Sie beschreiben zwar ein abstraktes Framework, gehen jedoch nicht auf die Bereitstellung der Schnittstellen für den Zugriff auf die einzelnen Instanzen ein. Weiterhin bietet bis heute kein kommerzielles Datenbanksystem einen komplexen Versionierungsmechanismus.

Im dritten vorgestellten Konzept werden vielmehr praktische Möglichkeiten zur Erstellung von versionierbaren XML beschrieben. Auch in diesem Bereich gibt es noch kein umfassendes Framework, welches eine automatische Verwaltung von verschiedenen Schema-Versionen bereitstellt oder eine rudimentäre Versionsüberprüfung durchführt.

Es scheint fasst paradox, dass sich bisher noch keiner der großen Datenbankhersteller umfassend über die Umsetzung von Versionierungstechniken Gedanken gemacht hat und das obwohl erste Ansätze bereits seit Mitte der achtziger Jahre existieren. Da jedoch bereits die Implementierung von Schema-Evolutionsstrategien in Datenbanksystem im Wesentlichen noch unausgereift sind, lassen Versionierungstechniken wohl noch weiterhin auf sich warten.

Abbildungsverzeichnis

Abbildung 1: Repräsentation unterschiedlicher Versionen einer Klasse Mitarbeiter 13
Abbildung 2: Beispielgraph nach [Grandi00] 17
Abbildung 3: Graphenverhalten der Schemaänderung auf eine Kante 19
Abbildung 4: XML Übersicht nach [Trähnert05] 22

Tabellenverzeichnis

Tabelle 1: Übersicht einfacher Schemaänderungen 18

Listingverzeichnis

Listing 1: Schema-Versions-Attribut 26
Listing 2: festes Versions-Attribut im Wurzelelement 27
Listing 3: Beispielschema der Version 1.6 28
Listing 4: Beispielinstanzdokument der Version 1.2 28
Listing 5: Schemaversion im targetNamespace 29

Literaturverzeichnis

[Banerjee87] **Jay Bannerjee, Won Kim, Hyoung-Joo Kim, Henry F. Korth**: *Semantics and Implementation of Schema Evolution in Object-Oriented Databases*. In: Proc. Of the SIGMOD International Conference on Management of Data. San Fransisco, CA, 1987.

[Clamen92] **Steward M. Clamen**: *Type Evolution and Instance Adaption*, Technical Report CMU-CS-92-133, Carnegie Mellon University, June 1992

[Costello05] **Roger L. Costello**: *XML Schema Versioning*. XML Schemas – Best Practices. Abrufdatum: 15.11.2005 http://www.xfront.com/Versioning.pdf

[DOM05] **W3C**: *Document Object Model (DOM)*, Web Reference, Erscheinungsdatum: 19.01.2005, Abrufdatum: 01.06.2005, http://www.w3.org/DOM/

[Gabriel04] **Jim Gabriel**: *How to version schemas*. XML 2004 November 15-19, Washington DC

[Grandi00] **Fabrio Grandi; Federica Mandreoli; Maria Rita Scalas:** *A Generalized Modelling Framework for Schema Versioning Support*, Database Conference, 2000. ADC 2000. Proc. 11th Australasian, 33-40

[Heuer97] **Andreas Heuer**: *Objektorientierte Datenbanksysteme – Konzepte, Modelle, Standards und Systeme*. Bonn: Addison Wesley Longman Verlag GmbH 1997, 2. Auflage

[Hoylen05] **Sue Hoylen**: *XML Schema Versioning Use Cases*, W3C XML Schema Working Group, Draft, June 2005 http://www.w3.org/XML/2005/xsd-versioning-use-cases/

[Katz90] **Randy H. Katz**: *Toward a Unified Framework for Version Modelling in Engineering Databases*. ACM Computing Surveys, 1990

[Kemper04] **Alfons Kemper, André Eickler**: *Datenbanksysteme. Eine Einführung*. 5., aktualisierte und erweiterte Auflage, Oldenbourg Verlag, 2004.

[Kim88] **Kim, W. and Chou, H.**: 1988. *Versions of Schema for Object-Oriented Databases.* In Proc. 14th int.Conf. on Very Large Data Bases, 148-159.

[Obasanjo04] **Dare Obasanjo**: *Designing XML Formats: Versioning vs. Extensibilty*. XML 2004 Proceedings by SchemaSoft http://www.idealliance.org/proceedings/xml04/abstracts/paper46.html

[RelaxNG03] **James Clark**: *Relax NG Homepage*, Web Reference, Erscheinungsdatum: 24.09.2003, Abrufdatum: 15.11.2005, http://www.relaxng.org/

[Roddick95] **J.F. Roddick**: *A Survey of Schema Versioning Issues for Database Systems*. Information and Software Technology, 3(7):383-393, 1995. http://citeseer.ist.psu.edu/roddick95survey.html

[Trähnert05] **Maik Trähnert**: *Einführung in XML*. Vorlesungsskript, Universität Leipzig. Institut für Informatik – Betriebliche Informationssysteme. 2005.

[XLink01] **W3C**: *XML Linking Language (XLink) Version 1.0*, Web Reference,
Erscheinungsdatum: 27.06.2001, Abrufdatum: 01.06.2005, http://www.w3.org/TR/xlink/

[XMLSpy05] **Altova**: *Altova XML Spy 2005, Produktübersicht*, Erscheinungsdatum: 01.2005, Abrufdatum: 01.06.2005, http://www.altova.com/

[XPointer, 2001] **W3C**: *XML Pointer Language (XPointer) Version 1.0*, Working Draft,
Erscheinungsdatum: 08.01.2001, Abrufdatum: 01.06.2005,
http://www.w3.org/TR/WD-xptr

[XSD00] **W3C**: *XML Schema*, Web Reference,
Erscheinungsdatum: 01.04.2000, Abrufdatum: 01.11.2005,
http://www.w3.org/XML/Schema

[XSL05] **W3C**: *The Extensible Stylesheet Language Family (XSL)*,
Web Reference
Erscheinungsdatum: 13.05.2005, Abrufdatum: 01.06.2005,
http://www.w3.org/Style/XSL/

[XSLT99] **W3C**: *XSL Transformations (XSLT) – Version 1.0*, Web Reference, Erscheinungsdatum: 16.11.1999, Abrufdatum: 01.06.2005,
http://www.w3.org/TR/xslt