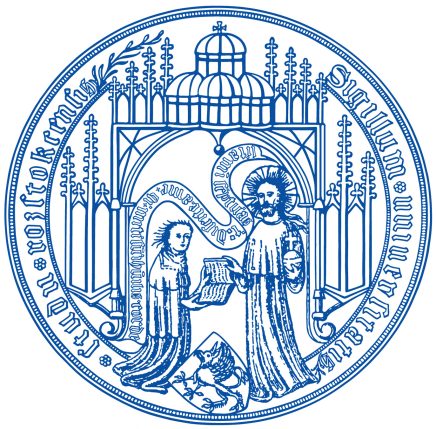
Integration von Integritätsbedingungen bei der XML-Schemaevolution

Masterarbeit

Universität Rostock
Fakultät für Informatik und Elektrotechnik
Institut für Informatik



vorgelegt von:	Hannes Grunert
Matrikelnummer:	7200076
geboren am:	30.08.1987 in Ribnitz-Damgarten
Gutachter:	PD Dr.-Ing. habil. Meike Klettke
	Prof. Dr.-Ing. habil. Peter Forbrig
Betreuer:	Dipl.-Inf. Thomas Nösinger
Abgabedatum:	29.04.2013

Selbständigkeitserklärung

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

Rostock, den 29. April 2013

Abstrakt

Die eXtensible Markup Language (XML) hat sich in den vergangenen Jahren als Austauschformat etabliert. Während XML-Schema vorwiegend genutzt werden, um die Struktur der XML-Dateien zu spezifizieren, wurden andere Schemasprachen wie Schematron entwickelt, um Integritätsbedingungen an die XML-Dateien zu stellen.

Mit v.1.1. wurde XML-Schema u.a. um Integritätsbedingungen erweitert, allerdings wird dieses Feature nur selten genutzt.

In dieser Arbeit wird untersucht, inwieweit sich die Nutzung von Integritätsbedingungen in XML-Schema durch deren Formulierung in einem konzeptionellen Modell verbessern lässt. Außerdem wird auch die Auswirkung durch die Einführung bzw. Veränderung von Integritätsbedingungen auf die XML-Dateien (Co-Evolution) betrachtet.

Abstract

The eXtensible Markup Language (XML) has become an established exchange-format over the last years. Being mainly used for defining the structur of XML-files, other schema-languages like Schematron focused on offering integrity constraints.

With v.1.1, this feature has been added to XML-Schema, but it is rarley used.

In this thesis it is examined how the use of integrity constraints in XML-Schema can be improved by formulating them on a conceptual model. The impact of adding or changing integrity constraints towards the XML-files (co-evolution) is also considered.

CR-Klassifikation

- D.2.8 Metrics
- D.3.3 Language Constructs and Features - Constraints
- F.3.1 Specifying and Verifying and Reasoning about Programs
- H.2.3 Languages - Query languages
- I.7.2 Document Preparation - Markup languages

Schlüsselwörter

XML Schema, Schema-Evolution, Integritätsbedingungen, konzeptionelles Modell

Keywords

XML Schema, schema-evolution, integrity constraints, conceptual model

Inhaltsverzeichnis

Kapitel 1

Einleitung

Der Austausch von Informationen zwischen Computersystemen und Programmteilen erfolgt häufig im XML-Format (eXtensible Markup Language). Für XML-Dateien existiert meist eine Beschreibung, wie XML-Schema, die die Struktur der Dokumente festlegt. Entspricht ein XML-Dokument allen Anforderungen des Schemas, so wird es als gültig bezeichnet.

Die Anforderungen an die XML-Dateien können sich im Laufe der Zeit ändern. Sollen neue Informationen gespeichert oder die Struktur verändert werden, so muss zunächst das Schema an die neuen Anforderungen angepasst werden. Anschließend müssen die zu dem Schema gehörigen Dokumente überprüft und gegebenenfalls verändert werden, damit ihre Gültigkeit weiterhin gewährleistet ist. Dieser Prozess wird als XML-Schemaevolution bezeichnet.

Neben der Struktur können in der Schemabeschreibung auch Integritätsforderungen an die XML-Dokumente formuliert werden. Für XML-Schema existieren bereits einige Möglichkeiten um Integritätsbedingungen zu formulieren. Dazu gehören z.B. Schlüssel-Fremdschlüssel-Beziehungen oder die Formulierung von logischen Konsistenzregeln in Assertion-Komponenten. Diese Regeln werden nur sehr selten verwendet, da ihre Formulierung relativ aufwendig ist. Durch eine geeignete Toolunterstützung bei der Formulierung solcher Regeln kann die Handhabung verbessert und Integritätsbedingungen beim Entwurf von XML-Schemata produktiv verwendet werden. Eine solche Unterstützung könnte z.B. durch die Integration der Integritätsbedingungen in ein konzeptionelles Modell, wie es CodeX anbietet, erfolgen. CodeX (Conceptual Design and Evolution for XML-Schema) ist ein grafischer Editor zur Modellierung und Evolution von XML-Schemata.

Im Rahmen dieser Masterarbeit wird untersucht, inwieweit CodeX um verschiedene Arten von Integritätsbedingungen erweitert werden kann. Dabei wird neben der Speicherung und Visualisierung auch die Überprüfung der XML-Dokumente gegen die Integritätsbedingungen untersucht. Zusätzliche Integritätsbedingungen haben verschiedene Auswirkungen auf die strukturellen Informationen eines XML-Schemas. Zum einen können sie redundant sein, d.h sie haben keinen Einfluss auf das Schema. Zum anderen ist es auch möglich, dass sie das Schema sinnvoll ergänzen, da sie Regeln für das Schema ausdrücken können, die allein durch die Struktur viel zu kompliziert darstellbar wären. Allerdings kann es auch sein, dass die in der Integritätsbedingung formulierte Forderung an das Schema auch zu einem Widerspruch führt. In diesem Fall muss bereits während der Formulierung der Integritätsbedingungen sichergestellt werden, dass keine widersprüchlichen Informationen entstehen und ggf. Alternativen vorgeschlagen werden.

Die Arbeit ist wie folgt gegliedert:

- In **Kapitel 2** werden die Grundlagen der Schemaevolution erläutert.
- **Kapitel 3** beschäftigt sich mit den verschiedenen Arten von Integritätsbedingungen und deren Umsetzung in anderen Systemen.
- Die Integration, Speicherung und Visualisierung von Integritätsbedingungen wird in **Kapitel 4** beschrieben.

- In **Kapitel 5** wird die Analyse, Kostenabschätzung, Bewertung und Korrektur von Integritätsbedingungen betrachtet.
- Anschließend wird in **Kapitel 6** der Algorithmus für die Anpassung der XML-Dateien via XSLT erklärt.
- Das Testszenario für die Überprüfung der Integritätsbedingungen ist Bestandteil des **7. Kapitels**.
- **Kapitel 8** fasst die Arbeit zusammen und gibt einen Ausblick auf zukünftige Entwicklungen.

Kapitel 2

Grundlagen

In diesem Kapitel werden die wichtigsten XML-Technologien vorgestellt. Dazu gehören die Grundlagen von XML, Schemasprachen zur Beschreibung des Aufbaus der Dokumente, sowie Transformations- und Anfragesprachen. Außerdem wird näher auf XML-Prozessoren und die XML-Schemaevolution eingegangen.

2.1 XML-Technologien

2.1.1 XML

Die eXtensible Markup Language (XML) ist ein häufig verwendetes Format zum Austausch von Informationen. Es besitzt einen einfachen, aus Elementen und Attributen bestehenden Aufbau, wodurch die Lesbarkeit für den Menschen vereinfacht wird. Ein einfaches Beispiel für eine XML-Datei ist in Abbildung 2.1 dargestellt.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<entry>
  <id>42</id>
  <title>Meine erste XML-Datei</title>
  <author>M. Mustermann</author>
  <published>2013-04-14T14:39:01</published>
</entry>
```

Abbildung 2.1: Beispiel für eine XML-Datei

Für XML existieren verschiedene Schemasprachen, welche die Struktur der Dokumentinstanzen festlegen. Neben dem häufig verwendeten XML-Schema, welches im nächsten Abschnitt beschrieben wird, existieren weitere Sprachen wie DTD (Document Type Definition), RelaxNG oder Schematron. Diese bieten eine vereinfachte Schemabeschreibung bzw. stellen zusätzliche Konstrukte bereit, die den Umfang von XML-Schema zu erweitern. Letztere Sprachen werden im nächsten Kapitel vorgestellt und miteinander verglichen.

2.1.2 XML-Schema

XML-Schema gilt als der Nachfolger von DTD und ermöglicht eine genauere Darstellung der Strukturinformationen einer Dokumentenkollektion. Es können beispielsweise durch die zusätzlichen Elementinfor-

mationen minOccurs und maxOccurs die genauen Kardinalitäten angegeben werden und Attribut- sowie Elementinhalte durch eine Reihe von vordefinierten Datentypen bzw. durch benutzerdefinierte Datentypen genauer definiert werden. Ein Beispiel für ein umfangreicheres XML-Schema, welches als laufendes Beispiel für diese Arbeit verwendet wird, ist in Anhang A angegeben.

Mit Version 1.1 wurde XML-Schema um einige Features erweitert:

- Assertions
- Bedingte Typisierung
- Schemaweite Attribute
- Flexibleres Schema durch <openContent> und <defaultOpenContent>
- Versionsverwaltung

Einen Überblick über XML-Schema und eine Zusammenfassung der Neuerungen in Version 1.1 ist in [W3C07a] und [Bor11b] gegeben. Da viele Werkzeuge noch nicht die neuen Features unterstützen bzw. sich in ihrer Entwicklung noch im Beta-Status befinden, erfolgt in dieser Arbeit keine nähere Untersuchung der Auswirkung der Schemaevolution auf diese Komponenten. Eine Ausnahme bilden hierbei die Assertions, da sie zusätzliche Integritätsbedingungen für das Schema bereitstellen und somit thematisch zu dieser Arbeit gehören.

Schlüssel-Fremdschlüssel-Beziehungen

In einem XML-Schema existieren drei Möglichkeiten, Schlüssel-Fremdschlüssel-Beziehungen auszudrücken. Dies sind die nachfolgend betrachteten Konstrukte *unique, key/keyref* und *id/idref*.

unique Soll für eine Menge von Elementen innerhalb eines Schemas sichergestellt werden, dass jedes ihrer Elemente eindeutig ist, so kann dies mittels einer *unique*-Angabe realisiert werden. Das Konstrukt wird als Kind innerhalb des Elementes eingefügt, für welche die Eindeutigkeit gelten soll. *Unique* verfügt über eine *selector*-Komponente, mit der das zu prüfende Element ausgewählt wird. Mittels einer oder mehrerer *field*-Angaben werden Attribute und Sub-Elemente selektiert. Auf den ausgewählten Feldern muss dann die Eindeutigkeit gelten. Die Adressierung der Elemente und Attribute erfolgt dabei mittels XPath und wird in den Attributwerten des *selector*- und der *field*-Elemente hinterlegt. Ein Beispiel für eine *unique*-Definition ist in Abbildung 2.2 dargestellt.

```
<xsd:element name="personen" type="personen">
  <xsd:unique name="PersonNameVorname">
    <xsd:selector xpath="person"/>
    <xsd:field    xpath="@name"/>
    <xsd:field    xpath="@vorname"/>
  </xsd:unique>
</xsd:element>
```

Abbildung 2.2: Beispiel für eine unique-Angabe

key/keyref Die Definition eines Primärschlüssels erfolgt in XML-Schema über das *key*-Element. Syntaktisch ist es unique sehr ähnlich (siehe Abbildung 2.3) und stellt ebenso sicher, dass das Element über die *selector*-field-Angaben eindeutig ist. Mit *keyref* ist es zudem möglich, dass die mit *key* und *unique* ausgezeichneten Elemente referenziert werden können. In dem *refer*-Attribut der *keyref*-Angabe wird dazu der Name des Primärschlüssels angegeben.

```
<xsd:element name="personen" type="personen">
  <xsd:unique name="PersonPnr">
    <xsd:selector xpath="person"/>
    <xsd:field    xpath="@pnr"/>
  </xsd:unique>
</xsd:element>
```

Abbildung 2.3: Beispiel für eine key-Angabe

id/idref In Elementen lassen sich *ID*-Attribute anlegen, damit sie über *IDREF*-Angaben referenziert werden können. Beide Attribute verfügen in XML-Schema über einen eigenen Datentyp. Zwar stellt die *ID* die Eindeutigkeit und Referenzierbarkeit der Elemente sicher, dennoch wird dieser Mechanismus nur noch selten verwendet, da mit *key/keyref* eine flexiblere Alternative geschaffen wurde, die auch auf Attributen anwendbar ist (siehe [W3C07b]).

Assertion

Um weiterführende Konsistenzbedingungen an XML-Dateien zu stellen, existiert in XML-Schema das Konstrukt Assertion. Dort ist es mittels XPath-Ausdrücken, z.b. Wertevergleiche und Aggregatfunktionen, möglich lokale als auch globale Bedingungen zu formulieren.

Zu diesen XPath-Ausdrücken gehören u.a folgende Operationen:

- Numerische Operationen
  - Addition, Subtraktion, Multiplikation, Division
  - Ganzzahlige Division, Modulo
  - Negation (*unary minus*)
  - Wertevergleiche (=, <, >)
  - Absolutbeträge
  - Rundungen (*ceiling, floor, round, round-half-to-even*)
- String-Funktionen
  - Lexikographische Ordnung (*compare*)
  - Verkettung
  - Sub-Strings
  - Länge der Zeichenkette
  - Groß- und Kleinschreibung (*upper-case, lower-case*)
  - Funktionen auf Sub-Strings (*contains, starts-with, ends with*, ...)
  - Pattern Matching (*matches, replace*)
- Bool'sche Vergleiche
  - true, false
  - Gleichheit (*equals*), Ungleichheit (*not, less-than, greater-than*)
- Funktionen auf Datums-Datentypen
  - Vergleiche (=, <, >)

  - Komponenten-Extraktion (*years-from-duration, seconds-from-time*)
  - Arithmetische Funktionen auf Längenangaben (+, -, *, /, *op:add-yearMonthDurations, ...*)

- Aggregatfunktionen
  - Anzahl (*count*)
  - Durchschnitt (*avg*), Summe (*sum*)
  - Minimum(*min*), Maximum(*max*)

Durch das Überladen vieler Operationen (Arithmetische Funktionen, Wertevergleiche) ist es möglich viele Operationen mit einem einzigen Operator zusammenzufassen, unabhängig vom konkreten Datentyp. Dadurch verringert sich der Implementationsaufwand. Außerdem reduziert sich der Aufwand für den Nutzer, da Assertions so einfacher zu formulieren sind.

Es ist unbedingt notwendig, dass die Assertions als Resultat einen bool'schen Wert zurückliefern, da sonst nicht ausgewertet werden kann, ob die Bedingung erfüllt ist oder nicht. Gängige Implementationen von XML-Validatoren, wie z.B. Apache Xerxec-J, liefern bei nicht-booleschen Werten immer als Ergebnis *true* zurück.

Ein Beispiel für eine Assertion ist in Abbildung 2.4 abgebildet. Es wird ein komplexer Typ *intRange* definiert, der über die Attribute *min* und *max* verfügt. In der Assertion wird im *test*-Attributes angegeben, welche zusätzliche Bedingung an den komplexen Typ gestellt wird. *@min* und *@max* referenzieren die oben definierten Attribute. Mittels *le* (less or equal; kleiner-gleich) wird festgelegt, dass der Wert des Attributes *min* kleiner oder gleich dem des Wertes *max* sein muss.

```
<xs:complexType name="intRange">
  <xs:attribute name="min" type="xs:int"/>
  <xs:attribute name="max" type="xs:int"/>
  <xs:assert test="@min le @max"/>
</xs:complexType>
```

Abbildung 2.4: Beispiel für eine Assertion in XML-Schema (aus [W3C07a])

2.1.3 XML-Schemaevolution

XML-Schemata für eine Anwendung können sich über die Zeit verändern. Die Gründe hierfür können vielfältig sein. Zum einen können sich die Anforderungen an die Anwendung ändern, aber auch Fehler beim Entwurf und deren Korrektur führen zu veränderten XML-Schemata. Neue Informationen werden abgespeichert, entfernt und andere nur leicht modifiziert. Durch diese Änderungsoperationen ist die Gültigkeit bestehender Dokumentinstanzen nicht mehr gesichert. Die Dokumente müssen auf ihre Gültigkeit hin geprüft und eventuell angepasst werden (Co-Evolution). Ein systematischer Überblick über die XML-Schemaevolution wird in Abbildung 2.5 dargestellt.

Für die Validierung der Dokumentinstanzen gegen ein XML-Schema existieren bereits viele Werkzeuge. Ein häufig verwendetes Tool ist z.B. Xerces-J, welches in der aktuellen Beta-Version auch XML-Schema v.1.1. unterstützt.

2.1.4 XSLT

Die Extensible Stylesheet Language for Transformations (XSLT) wurde entwickelt um XML-Dateien in verschiedenste Ausgabeformate zu transformieren, darunter auch XML. Durch den Sprachumfang von XSLT ist das Einfügen, Löschen und Verändern von Elementen und Attributen möglich, wodurch

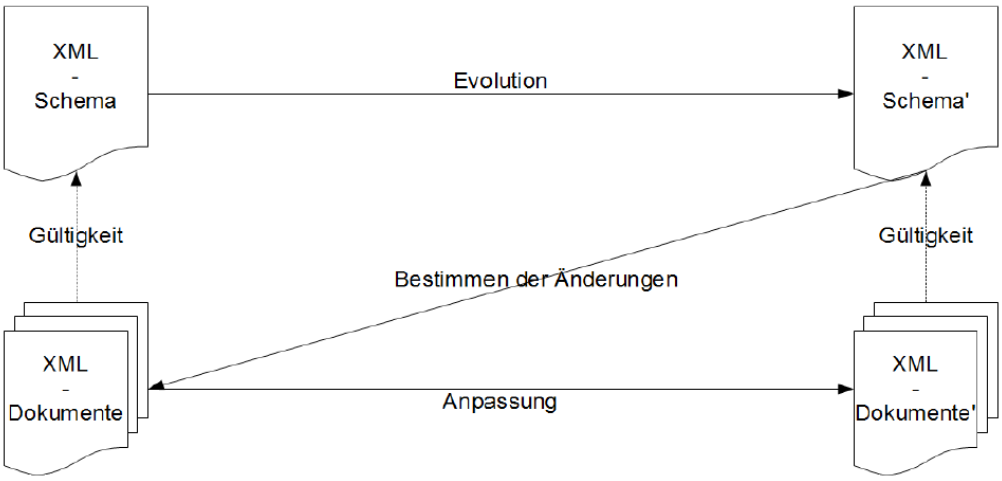


Abbildung 2.5: XML-Schemaevolution im Überblick

sich diese Sprache auch für die XML-Schemaevolution eignet. XSLT benutzt XPath zur Selektion der zu verändernden Knoten. Die einzelnen Transformationsregeln werden in sogenannten *Template*-Rules ausgedrückt.

Ein Beispiel für ein solches Template ist in Abbildung 2.6 angegeben. Der komplexe Typ *entryType* des Atom-Schemas (siehe Anhang A) wird um ein neues Element mit dem Namen *published* erweitert. Außerdem werden noch zusätzlich die Attribute *type, minOccurs und maxOccurs* für dieses Element erzeugt. Die Besonderheit an XSLT ist es, dass nur die Dokumentteile übernommen werden, die in den Template-Regeln durch das *match*-Attribut angegeben wurden. Um auch die Dokumentteile zu übernehmen, die durch die Schemaevolution nicht verändert wurden, muss eine spezielle Kopier-Regel angegeben werden. Diese kopiert durch das match-Attribut *match="@*/node()"* alle weiteren Attribute und Sub-Elemente.

Ein wichtiger XSLT-Prozessor im Java-Umfeld ist Apache Xalan, welcher unter [Fou06] bezogen werden kann.

2.1.5 XPath

Zur Adressierung von einzelnen Fragmenten von XML-Dokumenten wurde die Sprache XPath ([JC99] vom W3C entwickelt. XPath ist Bestandteil verschiedenster anderer XML-Standards wie der Transformationssprache XSLT oder XQuery. Durch die in XPath verwendeten Pfadausdrücke ist es möglich, eine bestimmte Knotenmenge oder atomare Werte aus einen größeren XML-Dokument zu ermitteln.

Es existieren für XPath verschiedene Navigationsachsen, durch die ausgehend vom aktuellen Kontextknoten und des aktuellen Teilpfades eine neue Knotenmenge selektiert wird. Nach Abarbeitung des gesamten Pfadausdruckes ist eine Knotenmenge bzw. ein atomarer Wert das Ergebnis der XPath-Anfrage. Folgende Navigationsachsen können zur Selektion der Knotenmenge genutzt werden:

- *self*: der Kontextknoten selbst
- *child*: die direkten Nachfolgeknoten unterhalb des Kontextknotens
- *descandant*: alle direkten und indirekten Nachfolgeknoten unterhalb des Kontextknotens
- *descandant-or-self*: alle Nachfolgeknoten inkl. dem aktuellen Kontextknoten

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns:
xs="http://www.w3.org/2001/XMLSchema" version="1.0">
<xsl:output method="xml" indent="yes" encoding="utf-8"/>
  <xsl:template match="//xs:complexType[@name='entryType']/xs:choice">
    <xsl:copy>
      <xsl:apply-templates select="attribute::*"/>
      <xsl:apply-templates select="child::node()"/>
      <xsl:element name="xs:element">
        <xsl:attribute name="name">published</xsl:attribute>
        <xsl:attribute name="type">atom:dateTimeType</xsl:attribute>
        <xsl:attribute name="minOccurs">1</xsl:attribute>
        <xsl:attribute name="maxOccurs">1</xsl:attribute>
      </xsl:element>
    </xsl:copy>
  </xsl:template>
  <xsl:template match="@*|node()">
    <xsl:copy>
      <xsl:apply-templates select="attribute::* | child::node()"/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

Abbildung 2.6: Beispiel für ein XSLT-Stylesheet

- *following*: alle Knoten nach dem Kontextknoten
- *following-sibling*: alle nachfolgenden Geschwisterknoten
- *preceding*: alle Knoten vor dem Kontextknoten
- *preceding-sibling*: alle vorhergehenden Geschwisterknoten
- *parent*: der direkt übergeordnete Knoten
- *ancestor*: der direkte und alle indirekten übergeordneten Knoten
- *ancestor-or-self*: alle ancestor-Knoten und der aktuelle Kontextknoten
- *attribute*: die Attribute des aktuellen Kontextknotens
- *namespace*: der Namensraum des aktuellen Kontextknotens

Die einzelnen Achsen können durch das Zeichen „/“ miteinander verbunden werden um komplexere Pfadausdrücke zu erzeugen. Zusätzlich zu den Navigationsachsen können Prädikat-Tests, logische Verknüpfungen und die oben erwähnten XPath-Functions (siehe Abschnitt Assertions) verwendet werden.

2.2 XML-Prozessoren

2.2.1 DOM

Das Document Object Model (DOM) dient zur Manipulation der Baumstruktur von Dokumenten. Es beschreibt Schnittstellen, mit denen gezielt bestimmte Teile (Elemente, Attribute und Text) eines Dokumentes ausgewählt werden können. Mittels DOM ist es ebenfalls möglich die Struktur eines Dokumentes

durch Einfüge-, Lösch- und Änderungs-Operationen grundlegend zu verändern. Dadurch eignet sich DOM zur Umsetzung von Schemaevolutionsschritten und der anschließenden Anpassung der XML-Instanzen. DOM wird in vielen XML-Prozessoren verwendet, u.a. in der Standard-Bibliothek des Java Development Kits (JDK).

2.2.2 SAX

Die Simple API for XML (SAX) ist ein Prozessor für die ereignisorientierte Abarbeitung von XML-Dokumenten. Durch die sequentielle Verarbeitung der Dokumente werden diese zeilenweise analysiert und der Beginn bzw. das Ende einzelner Elemente registriert. Durch diese Identifizierung der Dokumentbestandteile können benutzerdefinierte Methoden aufgerufen werden, die Teil eines Anwendungsprogramms sind.

SAX wird u.a. von XML-Validatoren verwendet, wie z.B. Apache Xerces-J.

2.3 CodeX

CodeX (Conceptual Design and Evolution for XML-Schema) ist ein graphischer Editor, der zur Modellierung und Evolution von XML-Schemata dient. Ursprünglich wurde der Editor als Plugin für Eclipse ([Kle07]) entwickelt, aktuell erfolgt die Umsetzung als Web-Applikation unter Nutzung des Google-Web-Toolkits ([Nös12]).

Der Editor in seiner aktuellsten Umsetzung ist in der Abbildung 2.7 dargestellt. Zentrales Element ist der Arbeitsbereich in der Mitte, der die Modellierung von XML-Schemata durch diverse Interaktionsmöglichkeiten (Drag and Drop, Kontextmenü, Mouseover...) ermöglicht. Auf der rechten Seite befindet sich eine Toolbar, durch die neue Komponenten in das Modell hinzugefügt werden können. Die linke Seite beinhaltet einen Explorer, über welchen Modelle, Schemata und XML-Dokumente verwaltet werden können.

Im unteren Bereich befindet sich ein Log, welches dem Anwender die wichtigsten Informationen angezeigt. Eine Menübar, in der die wichtigsten Funktionen aufgerufen werden können, befindet sich im oberen Bereich.

Diese Arbeit umfasst neben der theoretischen Ausarbeitung auch eine Implementation der entwickelten Konzepte. Diese Ergebnisse werden in CodeX integriert. Die Integration wird in Kapitel 4 näher erläutert.

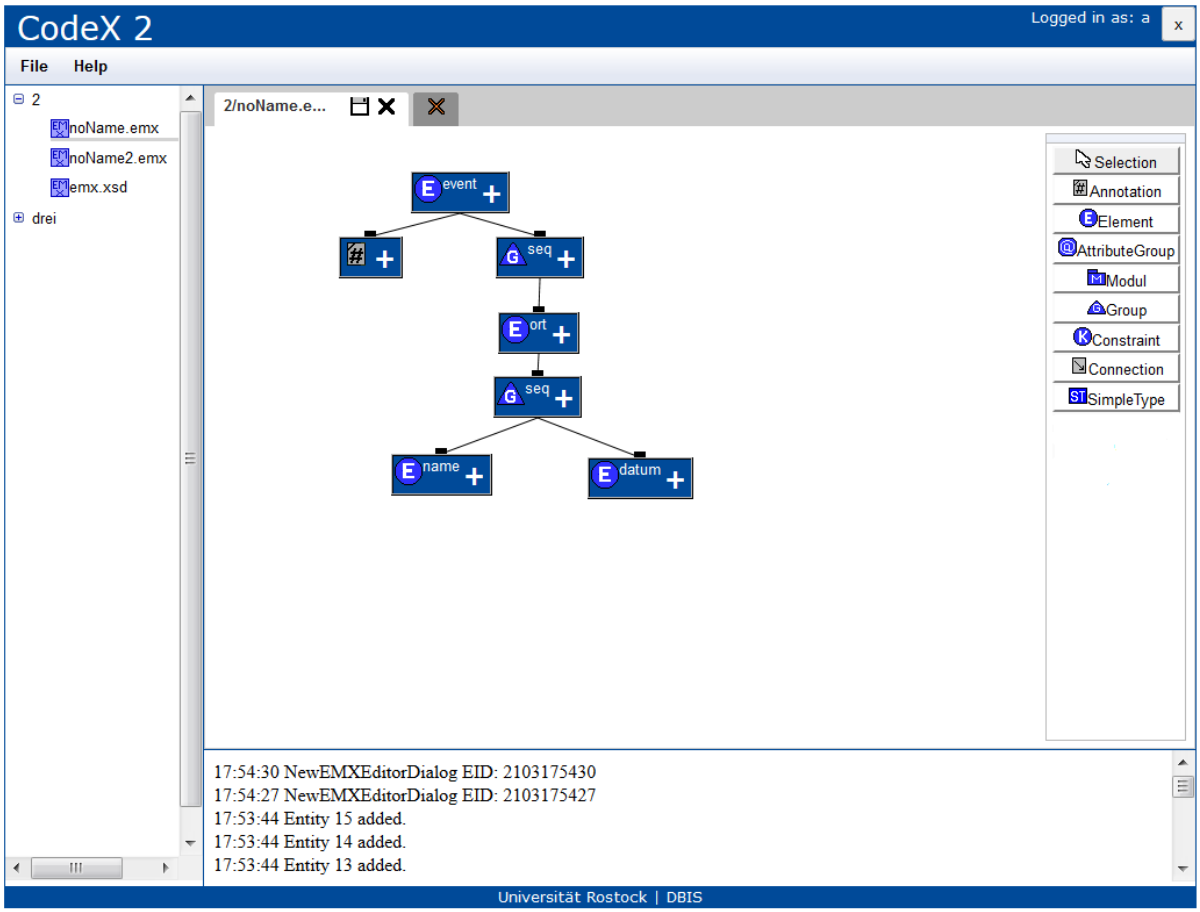


Abbildung 2.7: CodeX

Kapitel 3

Stand der Technik

In diesem Kapitel werden die verschiedenen Arten von Integritätsbedingungen erläutert und die Umsetzung von Integritätsbedingungen in Datenbanksystemen und Forschungsprototypen vorgestellt.

3.1 Arten von Integritäts- und Konsistenzbedingungen

Ein XML-Dokument wird als gültig bezeichnet, wenn es alle Bedingungen erfüllt, die in dem zugehörigen Schema (z.B. XML-Schema, DTD, RelaxNG) spezifiziert wurden. Wird mindestens eine Bedingung nicht erfüllt, so ist das XML-Dokument nicht mehr gültig.
In [Bor11a] werden Integritätsbedingungen kategorisiert. Dabei ergeben sich vier Hauptkategorien:

- Wertebereichsintegrität
- Strukturelle Integrität
- Schlüsselintegrität
- Logische Konsistenz

In XML-Schema 1.0 wurden bis auf die logische Konsistenz alle Integritätsbedingungen umgesetzt. Mit der Recommendation für XML-Schema 1.1 des W3C im April 2012 ist es möglich, weiterführende Integritätsbedingungen auf einfachen und komplexen Typen zu spezifizieren. Durch die Zuordnung jedes Attributes zu einem Datentyp können alle gängigen XML-Validatoren, die den Spezifikationstest des W3C bestanden haben (siehe http://www.w3.org/2001/03/webdata/xsv), eine Überprüfung des Wertebereiches durchführen.

3.1.1 Wertebereichsintegrität

In [AH00] wird der Wertebereich wie folgt definiert:
„Sei u eine nicht-leere, endliche Menge, das Universum der Attribute. Ein Element $A \in U$ heißt Attribut. Sei $D = D_1, ..., D_m$ eine Menge endlicher, nicht-leerer Mengen mit $m \in \mathbb{N}$. Jedes D_i wird Wertebereich oder Domäne genannt."
In XML-Schemata lassen sich verschiedene Attribute für Elemente spezifizieren. Dabei wird jedem Attribut ein Wertebereich zugeordnet, indem für das Attribut ein Datentyp angegeben wird. XML kennt verschiedene vordefinierte Datentypen, wie z. B. String oder Integer, aus denen zudem neue, benutzerdefinierte Datentypen erstellt werden können.
Werden für Attribute zudem final- oder default-Werte angegeben, so müssen diese ebenfalls überprüft werden. Viele der heutigen Systeme (beispielsweise XML-Parser wie Xerces) unterstützen diese Überprüfung nicht oder nur teilweise. Außerdem muss sichergestellt werden, dass für nicht-optionale Attribute

keine NULL-Werte auftreten. Werden die Bedingungen eines Datentyps für ein Attribut erfüllt, so ist die Bereichsintegrität sichergestellt.

3.1.2 Strukturelle Integrität

Neben dem Zugriff auf XML ist auch die Sicherstellung der strukturellen Integrität der XML-Daten von Bedeutung. In [W3C07a] werden die strukturellen Anforderungen eines Schemas an die Dokumente formuliert. Dazu wird ein abstraktes Modell eingeführt, welches für jede Komponente in XML die genaue Semantik und Eigenschaften spezifiziert. Der Standard des W3C definiert und dokumentiert dabei den Gebrauch, die Bedeutung und die Beziehungen der Schema-Komponenten. Dabei wird im Detail vorgegeben, welche Attribute und Sub-Elemente in den einzelnen Bestandteilen auftreten können und wie diese gebraucht werden. In Abbildung 3.1 wird am Beispiel eines komplexen Typs gezeigt, welche strukturellen Anforderungen an ihn gelten.

Schema Component: Complex Type Definition

`{annotations}`
A sequence of Annotation components.
`{name}`
An xs:NCName value. Optional.
`{target namespace}`
An xs:anyURI value. Optional.
`{base type definition}`
A type definition component. Required.
`{final}`
A subset of {*extension*, *restriction*}.
`{context}`
Required if {name} is ·absent·, otherwise MUST be ·absent·.
Either an Element Declaration or a Complex Type Definition.
`{derivation method}`
One of {***extension***, ***restriction***}. Required.
`{abstract}`
An xs:boolean value. Required.
`{attribute uses}`
A set of Attribute Use components.
`{attribute wildcard}`
A Wildcard component. Optional.
`{content type}`
A Content Type property record. Required.
`{prohibited substitutions}`
A subset of {***extension***, ***restriction***}.
`{assertions}`
A sequence of Assertion components.

Abbildung 3.1: Strukturelle Anforderungen an einen komplexen Typ (aus [W3C07a])

Die Prüfung der Gültigkeit eines Dokumentes zu seinem zugeordneten Schema wird in zwei Schritten durchgeführt:

1. Überprüfung der lokalen Schemagültigkeit

2. Ermittlung der globalen Gültigkeit

Lokale Gültigkeit Bei der lokalen Gültigkeit wird untersucht, ob für eine gegebene Element- oder oder Attributdefinition die notwendigen Informationen angegeben wurden. Beispielsweise muss für eine Attributdefinition das Attribut *name* mit seinem Wert angegeben werden, während die Angabe des Attributs *use* optional ist.

Globale Gültigkeit Ein Element oder Attribut ist global gültig, wenn es und seine Sub-Elemente lokal gültig sind. Außerdem müssen alle Sub-Elemente in der Deklaration des übergeordneten Elementes enthalten sein. Beispielsweise ist eine Attributdefinition nur dann gültig, wenn sie lokal gültig ist und außer einer Annotation keine weiteren Sub-Elemente besitzt.

Ein XML-Dokument ist hinsichtlich seiner Struktur gültig, wenn es die globale Gültigkeitsforderung erfüllt.

3.1.3 Schlüsselintegrität

Analog zu Datenbanksystemen lassen sich in XML-Schema Schlüssel-Fremdschlüssel-Beziehungen definieren. Dabei muss zum einen sichergestellt werden, dass der Primärschlüssel auf einem Objekt, in diesem Fall ein XML-Fragment, eindeutig ist (Entitätsintegrität) . Zum anderen muss für den Fremdschlüssel zugesichert werden, dass er nicht auf ein leeres Objekt, also einen nicht vorhandenen Knoten im XML-Baum, verweist (referentielle Integrität).

Entitätsintegrität

Primärschlüssel dienen zur eindeutigen Identifizierung von Objekten. In relationalen Datenbanksystemen werden dazu ein oder mehrere Attribute als Schlüssel (im SQL-Standard: *PRIMARY KEY*) ausgezeichnet. Häufig wird als Schlüssel auch ein zusätzliches Attribut in die Relationen eingefügt, ein sogenannter künstlicher Schlüssel. In [AH00] wird die identifizierende Attributmenge wie folgt definiert:

„Eine identifizierende Attributmenge für ein Relationenschema R ist eine Menge $K := B_1,...,B_k \subseteq R$*, so dass für jede Relation r(R) gilt:* $\forall t_1, t_2 \in r[t_1 \neq t_2 \Rightarrow \exists B \in K : t_1(B) \neq t_2(B)]$.“

In XML-Schema existieren zwei Varianten um die Entitätsintegrität sicherzustellen. Bei der ersten Variante wird ein künstlicher Schlüssel in Form einer ID in die betroffenen Elementdefinitionen eingefügt. In den jeweiligen Dokumentinstanzen muss diese ID eindeutig sein. Dies entspricht dem künstlichen Schlüssel aus dem Datenbankbereich. Alternativ kann eine Elementdefinition über die Schlüsselelemente *unique* und *key* ausgezeichnet werden. Beide Techniken ermöglichen die Angabe einer identifizierenden Subelement- und Attributmenge über einem Element. Die Elemente, die durch die Angabe dieses natürlichen Schlüssels ausgezeichnet wurden, müssen in den Dokumenten entsprechend verschieden sein um die Schlüsseleigenschaft zu erfüllen. Anders als in relationalen Datenbanken können einzelne Attribute des zusammengesetzten Schlüssels auch den Wert *NULL* annehmen.

Referentielle Integrität

In relationalen Datenbanken werden Beziehungen zwischen einzelnen Relationen häufig durch Fremdschlüssel (SQL: *FOREIGN KEY*) ausgedrückt. Dieser Fremdschlüssel referenziert ein Tupel aus einer anderen Relation. Der Fremdschlüssel nimmt dabei den Wert des Primärschlüssels des referenzierten Tupels an.

Die referentielle Integrität stellt sicher, dass der Wert des Fremdschlüssels entweder *NULL* ist bzw. dass ein Objekt mit einem solchen Schlüsselwert existiert. Außerdem wird die Beziehung zwischen den Relationen aufrechterhalten, auch wenn Daten der Relationen eingefügt, gelöscht oder verändert werden.

In XML-Schema wird die Beziehung zwischen Elementen durch die Mechanismen *IDREF* bzw. *keyref* hergestellt, allerdings erfolgt keine explizite Sicherstellung der referentiellen Integrität. Werden Elemente verändert, kann zwar im Nachhinein überprüft werden, ob die referentielle Integrität weiterhin besteht. Dessen ungeachtet erlauben die meisten (Web-)Anwendungen die Veränderung der Elemente, da sie über

```
<simpleType name='triple'>
 <restriction base='integer'>
  <assertion test='$value mod 3 eq 0'/>
 </restriction>
</simpleType>
```

Abbildung 3.2: Definition eines Datentyps mittels Assertion (aus [W3C07a])

```
<simpleType name='triple'>
 <restriction base='integer'>
  <pattern value=
  '([0369]\|[147][0369]*[258]\|(([258]\|[147][0369]*[147])
  ([0369]\|[258][0369]*[147])*([147]\|[258][0369]*[258]))*'/>
 </restriction>
</simpleType>
```

Abbildung 3.3: Definition eines Datentyps mittels Pattern (aus [W3C07a])

keine eigene Validierungsfunktion verfügen. Eine Ausnahme bilden hierbei XML-Datenbanken wie DB2 ([Zha09]), welche die Mechanismen aus dem relationalen Umfeld übernommen haben.

3.1.4 Logische Konsistenz

Unter den Begriff Logische Konsistenz fallen diejenigen Integritätsbedingungen, die sich nicht oder nur sehr umständlich durch Integritätsbedingungen bezüglich des Wertebereichs, der Struktur oder durch Schlüssel-Fremdschlüssel-Beziehungen darstellen lassen.

In einigen Standards werden in den normativen Spezifikationen bezüglich der Struktur von Dokumenten verschiedene zusätzliche Integritätsbedingungen gestellt, diese werden aber in den dazugehörigen Schemata häufig wieder verworfen, da sie nicht oder nur sehr umständlich zu beschreiben sind. Dazu gehören auch Bedingungen, die erst auf Dokumentenebene überprüft werden können.

Logische Konsistenzbedingungen können z.B. wie folgt lauten:

- Eltern müssen vor ihren Kindern geboren sein.
- Die Ausgaben aller Abteilungen dürfen in der Summe nicht über den gesamten Einnahmen liegen.
- Falls Element X ein Subelement Y besitzt, dann muss Z das Attribut A besitzen.

Die logischen Konsistenzbedingungen vereinfachen auch die Einschränkung des Wertebereichs von Attributen. In ([W3C07a]) wird beispielsweise in XML-Schema ein Datentyp definiert, der alle durch 3 teilbaren Zahlen umfasst. Dabei erfolgt die Definition in einem Fall mittels Assertions (siehe Abbildung 3.2) und im anderen Fall mittels eines Pattern (siehe Abbildung 3.3). Wie leicht zu erkennen ist, ist die Formulierung des selben Sachverhalten als Pattern aufgrund der „umständlichen“ Schreibweise fehleranfälliger als die entsprechende Assertion.

Beispiel

Ein bekanntes, reales Beispiel kommt in der Hypertext Markup Language (HTML) vor. Für HTML existiert eine Recommendation des W3C ([W3C12]), als auch ein XML-Schema ([Mas02]), welches den Aufbau von HTML-Dateien beschreibt. Innerhalb der Recommendation beschreibt der Abschnitt *Element*

Prohibitions, welche Elemente nicht als Nachfolger eines anderen Elementes auftreten dürfen. Für das Element *a*, mit dem Links angegeben werden, gilt folgende Einschränkung: „**a** must not contain other a elements“ ([Mas02]).

Das Element darf, unabhängig von der Tiefe der Verschachtlung, kein weiteres a-Element enthalten. Im dazugehörigen XML-Schema wird diese Einschränkung nicht umgesetzt; bei der Spezifikation des a-Elementes wird lediglich in der Dokumentation darauf hingewiesen. Eine Lösung für dieses Problem ist es, dass das XML-Schema ohne das a-Element kopiert wird. Dieses neue Schema wäre dann für all jene HTML-Elemente zuständig, die unterhalb eines a-Tags stehen. Die Realisierung wäre durch die Einführung eines neuen Namespaces zwar einfach, jedoch nicht praktikabel. Einschränkungen existieren auch für weitere Elemente (*pre, button, label, form*), für die dann weitere Schemata erstellt werden müssten. Dies würde den Umfang des Schemas um ein Vielfaches erhöhen und eine Validierung des Schemas deutlich verlängern. Durch zusätzliche Integritätsbedingungen, in diesem Fall die Einschränkung der erlaubten Elemente, würden sowohl das Schema als auch die Zeit zur Validierung nur minimal größer werden, da die Zusatzbedingungen nur lokal an den betroffenen Stellen eingefügt bzw. überprüft werden.

Problem

Logische Konsistenzbedingungen wurden bereits im Datenbankumfeld näher untersucht. Can Türker untersucht in seiner Dissertation ([Tür99]) Integrationsbedingungen in föderierten Datenbanken. Dabei stellt er fest, dass Integritätsbedingungen aus mehreren Quellen bei der Föderation zu Widersprüchen führen können. Das Erkennen bzw. Auflösen dieser Inklusions- und funktionalen Abhängigkeiten ist laut Türker im Allgemeinen unentscheidbar bzw. in Spezialfällen nur in exponentieller Zeit lösbar.

> *„[...] the general problem of determining the relationship between general integrity constraints is undecidable. In some cases, the problem can only be solved in exponential time.“*

Türker stellt in seiner Arbeit einige Spezialfälle vor, deren Auflösung in Polynomialzeit möglich ist. Diese stellen aber nur einen kleinen Teil der möglichen Formulierungen dar. Wird das Problem auf andere Datenquellen übertragen, wie z.B. XML-Schemata, so soll die Auflösung der Abhängigkeiten noch schwerer sein.

> *„The problem becomes even worse if non-database sources should be federated“*

Dennoch werden logische Konsistenzbedingungen in vielen Systemen verwendet. Ein Überblick über diese Systeme gibt der nächste Abschnitt.

3.2 Bisherige Umsetzungen von Co-Constraints in XML

Die Einführung von XML-Schema brachte viele Verbesserungen zur Formulierung von Schemata, wie z.B. die Zuweisung von Datentypen. Es gab dennoch keine Möglichkeiten logische Konsistenzbedingungen (auch bekannt als Co-Constraints) zu formulieren. Verschiedene Systeme versuchten dies zu verbessern, wurden aber nur selten eingesetzt. Viele Ansätze (test-Attribut für Co-Constraints, alternative Datentypen, error-Datentyp) wurden in XML-Schema 1.1 übernommen. Im Folgenden wird eine kurze Übersicht über die bisherigen Erweiterungen von XML und Neuentwicklungen gegeben. Dabei zeigen Beispiele anhand des Atom-Schemas (siehe Anhang B) wie sich die Integritätsbedingungen in den einzelnen Sprachen formulieren lassen. Das Beispiel zeigt einen Nachrichten-Eintrag (*entry*), für den sichergestellt werden soll, dass entweder ein Autor (*author*) oder Herausgeber (*contributor*) angegeben ist.

3.2.1 Schematron

Schematron ([ISO11]) ist keine eigenständige Sprache, sondern dient vielmehr der Validierung von XML-Dateien. Über die Angabe eines XPath-Ausdruckes (*context*) können logische Konsistenzbedingungen (*rule*) formuliert werden. Diese *rules* prüfen ob für XML-Fragmente bestimmte Eigenschaften (*test*) gelten. Bei *test* handelt es sich in den meisten Fällen um XPath-Ausdrücke, die mit logischen Operatoren verknüpft werden. Außerdem ist es möglich, Fehlermeldungen anzugeben, damit der Anwender im Falle einer fehlgeschlagenen Validierung den Fehler schneller finden kann. Ein Beispiel für ein Schematron-Schema ist in Abbildung 3.4 angegeben.

```
<schema xmlns="http://www.ascc.net/xml/schematron">
  <title>A Schematron-Schema for Atom</title>
  <pattern name="Entry-Test">
    <rule context="entry">
      <assert test="author or contributor">
        An entry have at least one author or contributor.
      </assert>
    </rule>
  </pattern>
</schema>
```

Abbildung 3.4: Beispiel für ein Schematron-Schema

Die Arbeitsweise von Schematron wird von den Autoren in [ISO11] wie folgt zusammengefasst:

1. *„First, find a context nodes in the document (typically an element) based on XPath path criteria;“*
2. *„Then, check to see if some other XPath expressions are true, for each of those nodes.“*

Schematron wurde u.a. in Python und Perl implementiert, es existieren aber auch Validatoren, die auf XSLT aufbauen. In der letzteren Variante wird aus einer Schematron-Datei ein XSLT-Skript erzeugt. Dieses Skript kann anschließend auf ein XML-Dokument angewandt werden. Als Ausgabe wird ein Report im XML-Format erzeugt, welches die Informationen über die erfüllten und unerfüllten Integritätsbedingungen enthält.

3.2.2 RelaxNG

RelaxNG (Regular Language Description for XML New Generation, [Cla12]) ist eine kompakte Sprache zur Beschreibung der Struktur von XML-Dokumenten. RelaxNG besitzt wie Schematron und XML-Schema eine XML-Syntax. Neben der Struktur werden auch Datentypen, Namensräume und logische Konsistenzbedingungen unterstützt. Für Letztere ist der Umfang beschränkt; es besteht nur die Möglichkeit einfachere Pattern, wie in Abbildung 3.5 gezeigt, zu formulieren.

3.2.3 DSD

DSD (Document Structure Description, [NK00]) ist eine von AT&T [1] entwickelte Schemasprache zur Validierung von XML-Dokumenten. Wie auch die bereits vorgestellten Schemasprachen wird in DSD eine XML-Syntax benutzt. Neben Integritätsbedingungen bezüglich Wertebereich, Struktur und referentieller Integrität ist es in DSD möglich, weitere Konsistenzbedingungen durch logische Operatoren wie *and*, *or* und *not* zur formulieren. Diese können sowohl lokal für einzelne Elemente als auch global für das

[1] American Telephone and Telegraph Company

```
<element name="entry">
  <choice>
    <element name="author"/>
    <element name="contributor"/>
    </choice>
</element>
```

Abbildung 3.5: Beispiel für eine Konsistenzregeln in RelaxNG

gesamte Schema gelten. Ein Beispiel für das bereits mehrfach erwähnte Atom-Schema ist in Abbildung 3.6 angegeben.

```
<ElementDef ID="X" Name="entry">
  <And>
    <Not>
      <And>
        <Element Name="author"/>
        <Element Name="contributor"/>
      </And>
    </Not>
    <Or>
      <Element Name="author"/>
      <Element Name="contributor"/>
    </Or>
  </And>
</ElementDef>
```

Abbildung 3.6: Beispiel für eine Konsistenzregeln in DSD

3.2.4 xlinkit

xlinkit ([CN02]) ist eine Schemasprache, die sich auf die Validierung von verteilten Schemata (mehrere Schema-Dateien auf verschiedenen Computern) spezialisiert hat. Für die Validierung der XML-Dokumente werden Regeln (sogenannte *consistencyrules*) mittels XPath-Ausdrücken, Quantoren, Variablen und logischen Operatoren definiert. Die Ausgabe des Validierungsprozesses sind zwei Mengen von Elementen; eine Menge enthält alle Elemente, die allen Konsistenzregeln entsprechen, die zweite Menge enthält alle weiteren Elemente. Ein Beispiel für eine Konsistenzregel ist in Abbildung 3.7 abgebildet. Die Schemasprache xlinkit unterstützt alle vier Arten der Integrität.

3.2.5 SchemaPath

In [PM04] wird SchemaPath vorgestellt, eine Erweiterung zu XML-Schema. Ziel von SchemaPath ist die Ergänzung von XML-Schema um logische Konsistenzbedingungen. Die Autoren betrachten dazu die in den obigen Abschnitten vorgestellten Schemasprachen und vergleichen sie mit ihrem eigenen Sprachvorschlag. In SchemaPath lassen sich Regeln (*rule*) definieren, die eine Zusicherung (*assert* bzw. *report*) für ein über XPath ausgewähltes Kontext-Element (*context*) aufstellen. Der Test ist ein XPath-Ausdruck, der wahr muss um die Zusicherung zu erfüllen. Andernfalls wird eine benutzerdefinierte Fehlermeldung

```
<consistencyrule id="r1">
  <forall var="x" in="//entry">
    <and>
      <not>
        <and>
          <exists var="y" in="$x/author"/>
          <exists var="z" in="$x/contributor"/>
        </and>
      </not>
      <or>
        <exists var="y" in="$x/author"/>
        <exists var="z" in="$x/contributor"/>
      </or>
    </and>
  </forall>
</consistencyrule>
```

Abbildung 3.7: Beispiel für eine Konsistenzregeln in xlinkit

(Inhalt des *assert*-Elementes) ausgegeben. Ein Beispiel für eine solche Regel ist in Abbildung 3.8 angegeben.

```
<sch:rule context="entry">
  <sch:assert test="/author or /contributor">
    An entry must have an author or contributor.
  </sch:assert>
  <sch:report test="/published le /updated">
    An entry has to be published before updated.
  </sch:report>
</sch:rule>
```

Abbildung 3.8: Beispiel SchemaPath-Regel

Die Regeln werden dabei nicht wie bei RelaxNG durch Pattern-Matching aufgerufen, sondern werden wie bei Schematron direkt in das XML-Schema integriert. Die XML-Parser sollten nach Vorstellung der Autoren an die neuen Elemente angepasst werden. Unter [Vit04] wurde ein webbasiertes Tool zum Testen von SchemaPath bereitgestellt, allerdings ist dieses nicht mehr funktionsfähig. Der Validierungsprozess dieses Tools erfolge mittels XSLT. Dabei wurden die Integritätsbedingungen in XSLT-Stylesheets umgewandelt und anschließend auf die Dokumente angewandt. Dieses Vorgehen ist in Abbildung 3.9 skizziert. Es wird dargestellt wie das Schema S durch das XSLT-Skript T zu Schema S' verändert wird. Aus den Änderungen erzeugt das Skript MT die Adaption des Dokumentes X zu den angepassten Dokument X'.

Neben der Unterstützung von logischen Konsistenzbedingungen bietet SchemaPath auch die Möglichkeit zur Formulierung von alternativen Datentypen an. In XML-Schema 1.1 finden sich viele in SchemaPath entwickelte Ansätze wieder. Dazu zählen die Realisierung von Integritätsbedingungen durch XPath-Ausdrücke und deren direkte Einbettung in die einzelnen Elemente.

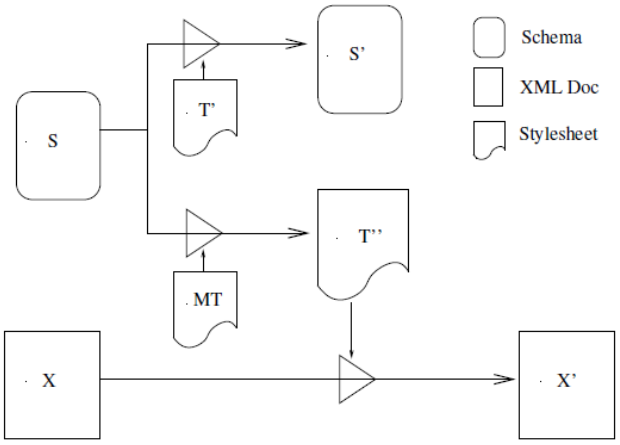


Abbildung 3.9: Validierungsprozess des SchemaPath-Tools

3.2.6 Object Constraint Language

Die Object Constraint Language (OCL) wird vorwiegend im UML-Umfeld eingesetzt. Da einige Forschungsprototypen der XML-Schemaevolution UML als konzeptionelles Modell verwenden, findet sich OCL auch zur Formulierung von Integritätsbedingungen für XML-Dateien wieder. OCL kann u.a. dazu verwendet werden, um:

- Invarianten auf Klassen und Datentypen zu formulieren (*entry inv: self.published > 1900-01-01*),
- Pre- und Post-Conditions für Methoden und Operationen anzugeben (*entry::updateEntry() post: self.updated>self.updated@pre*),
- eine Typhierarchie zu spezifizieren,
- Ableitungsregeln für Attributwerte anzulegen und
- Operationen auf Mengen, Mengenoperatoren $\forall, \exists$ ... darzustellen

Abbildung 3.10 illustriert ein Beispiel der vierten Verwendung. Es wird dargestellt wie abhängig vom Alter einer Person entweder das Einkommen der Eltern (parents.income) bzw. das eigene Einkommen durch die Ausübung eines Berufs als Gesamteinkommen ausgegeben werden.

```
context Person::income : Integer
  init: parents.income->sum() * 1% -- pocket allowance
  derive: if underAge
    then parents.income->sum() * 1% -- pocket allowance
    else job.salary %--income from regular job
    endif
```

Abbildung 3.10: OCL-Beispiel (aus http://www.omg.org/spec/OCL/2.3.1/PDF/

3.3 Forschungsprototypen

In den folgenden beiden Abschnitten werden bestehende Forschungsprototypen und Datenbanksysteme hinsichtlich ihrer Realisierung von Integritätsbedingungen im XML-Schema untersucht. Einen detaillierten Überblick über die einzelnen Systeme gibt die Arbeit von [Def12].

3.3.1 XSEM

Der Forschungsprototyp XSEM ([MN11]) stellt XML-Schema-Dateien auf einer konzeptionellen Ebene dar. Dabei werden die XML-Schemata in UML-Klassendiagramme überführt. Sowohl die Struktur als auch die Zuordnung der einzelnen Attribute zu einem Datentyp bleibt dabei erhalten. Auch die Schlüssel-Fremdschlüssel-Beziehungen werden in dem UML-Diagramm dargestellt. Zur Spezifizierung von logischen Konsistenzbedingungen wird eine modifizierte Object Constraint Language (OCL) genutzt. Diese Bedingungen werden als zusätzliche Elemente dargestellt und durch Verbindungslinien an die dazugehörigen Elemente gebunden.

3.3.2 XEM

In XEM ([Kra01], [DK01]) wird die Schemaevolution bezüglich einer veränderten DTD (Document Type Definition) betrachtet. Damit die Gültigkeit und Wohlgeformtheit der XML-Dokumente sichergestellt ist, werden für die einzelnen Operationen, die zu einer Evolution des Schemas führen, Pre- und Postconditions aufgestellt. Preconditions müssen vor der Ausführung eines Evolutionsschrittes erfüllt sein, ansonsten wird die Operation zurückgewiesen. Postconditions stellen sicher, dass nach der Ausführung der Evolution die zur DTD gehörigen XML-Dokumente angepasst werden. Der Prototyp bestimmt unter Berücksichtigung dieser Bedingungen die Änderungen an einer DTD und nimmt anschließend die Anpassung der Dokumente vor. Durch die semantische Analyse der Evolutionsschritte wird die strukturelle Konsistenz sicherstellt. Da die DTD keine Datentypen bereitstellt, erfolgt keine Überprüfung des Wertebereichs. Referentielle Integrität ist nur über den *ID/IDREF*-Mechanismus möglich; logische Konsistenzbedingungen fehlen ganz.

3.3.3 DTD-Diff

Ein weiteres Tool, welches die Evolution einer DTD betrachtet, ist DTD-Diff ([EL06]). Das Tool erhält als Eingabe zwei DTDs und ermittelt aus dem Unterschied zwischen den beiden Dateien eine textuelle Ausgabe der Evolutionsschritte. Da DTDs keine Datentypen oder logische Konsistenzbedingungen kennen, wird lediglich die strukturelle Integrität überprüft, die sich aus dem Test der Dokumente gegenüber der DTD ergibt.

3.3.4 DiffDog und XML-Spy

Altova Diffdog ([Alt11]) und XML-Spy ([Alt12]) bieten neben dem Vergleich zweier XML-Dokumente auch die Möglichkeit, die Dokumente auf ihre Gültigkeit bzgl. eines Schemas zu überprüfen. Um die Dokumente und das Schema darzustellen bietet DiffDog neben der textuellen Darstellung auch eine Visualisierung in Form einer Grid- oder Boxen-Ansicht. In beiden Ansichten ist es möglich, XML-Dokumente und Schemata zu validieren und Fehler bereits während des Bearbeitens zu bemerken. Fehler werden durch farbige Markierungen hervorgehoben, damit verletzte Integritätsbedingungen schneller aufgefunden werden. In Hinblick auf Integritätsbedingungen unterstützt der Schema-Editor von DiffDog und XML-Spy besonders die Formulierung von Schlüsselbedingungen, indem der Editor spezielle Dialogfelder für das Anlegen von *id-, key*, und *unique*-Constraints anbietet. Die Visualisierung einer Fremdschlüsselbedingung ist in Abbildung 3.11 dargestellt. Es wird ein Schlüssel definiert, der sicherstellt, dass für jedes *item* der wert von *note* für die gesamte *shiporder* eindeutig ist.

Logische Konsistenzbedingungen werden nicht umgesetzt, da der Editor XML-Schema v.1.1 noch nicht unterstützt und es somit nicht möglich ist, *assert*-Elemente in das Schema einzufügen.

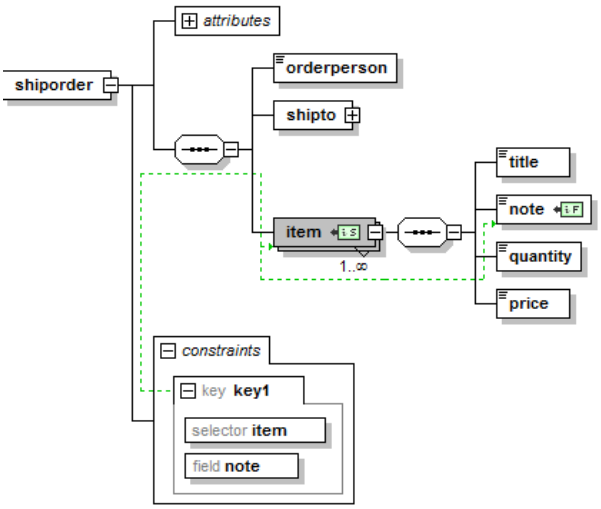


Abbildung 3.11: Fremdschlüsselbedingungen in XMLSpy

3.3.5 Clio

Ebenso wie DiffDog und DTD-Diff ermittelt Clio([Mot08]) aus einem gegebenen Quell- und Zielschema Evolutionsschritte und versucht ein Mapping zwischen den Dokumentinstanzen des neuen und alten Schemas zu erzeugen. Dabei erfolgt der Vergleich nicht auf den Dateien, sondern auf einen aus den Dateien generierten Metamodell. Clio fokussiert sich auf das Mapping von Attributen, insbesondere für den Fall, wenn sich Datentypen im Quell- und Zielschema unterscheiden. Tritt dies ein, so wird geprüft, ob die Datentypen zueinander vollständig kompatibel, teilweise kompatibel oder nicht kompatibel sind. Existiert keine Konvertierungsregel zwischen den Datentypen, so wird die vorläufig Evolution zurückgewiesen. Der Nutzer muss anschließend entscheiden, ob er für die betroffenen Attribute Skolem-Regeln (Ermittlung eines Wertes aus den anderen Attributwerten) aufstellt, Default-Werte angibt, oder den Attributwert freilässt, sofern dieser optional ist. Die gleichen Optionen gelten auch für Mapping-Problemen von Schlüsseln und Fremdschlüsseln bei der Evolution, wenn z.B. Fremdschlüsselbeziehungen im Quellschema noch nicht existieren. Für die Transformation der Schlüssel aus dem Metamodell zurück in ein XML-Schema werden vorrangig key/keyref-Elemente verwendet, alternativ wird auch der id/idref-Mechanismus angeboten. Die Visualisierung der Schlüsselbedingungen ist in Abbildung 3.12 dargestellt. Es wird z.B. eine Fremdschlüsselbedingung zwischen dem Attribut *gehaltenVon* des Elementes *Lehrveranstaltung* zu dem Attribut *name* von *ProfessorIn* definiert. Treten während der Evolution des Schemas Fehler auf, meldet Clio diese über PopUp-Fenster.

3.3.6 UML-to-XML-Framework

Das UML-to-XML-Framework ([ED11]) arbeitet ebenso wie Clio nicht direkt auf den XML-Dateien, sondern benutzt UML als Metamodell. Änderungen auf dem Modell werden anschließend wieder auf die XML-Schemata angewandt. Zur Transformation des Schemas und der dazugehörigen Dokumente wird XSLT verwendet. Das Framework bietet die Möglichkeit, komplexere Konsistenzbedingungen zu

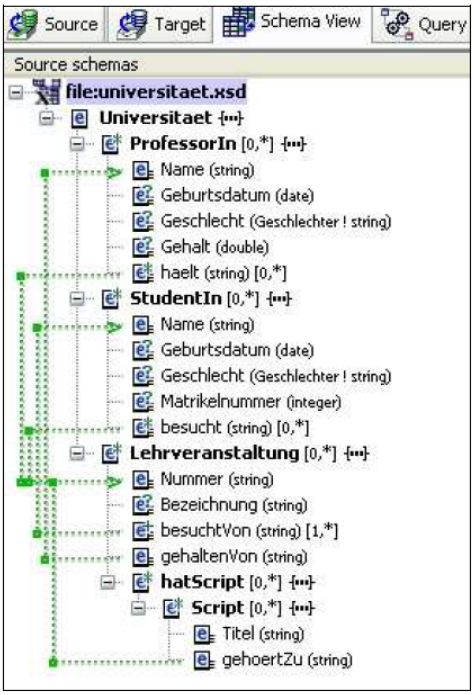


Abbildung 3.12: Fremdschlüsselbedingungen in Clio

formulieren. Dabei wird wie in XSEM die Object Constraint Language benutzt. Die Visualisierung des UML-Metamodells einschließlich der OCL-Bedingungen ist in Abbildung 3.13 dargestellt. Zum Beispiel sollen alle Departments einen eindeutigen Namen haben (*Department::allInstances-> isUnique(name)*). Eine konkrete Implementierung dieses Frameworks wurde u.a. in Oracle-Datenbanken umgesetzt.

3.3.7 X-Evolution

X-Evolution ([GG08], [GG06]) ist ein webbasiertes Tool, welches die Evolution von XML-Schemata durch eine grafische Benutzeroberfläche ermöglicht. In dem Modell ist es möglich, die Beziehung zwischen den einzelnen Elementen des Schemas (strukturelle und referentielle Integrität) als auch die Elemente (Struktur und Wertebereich) zu verändern. Da das Tool auf XML-Schema 1.0 basiert und keine weiteren Schemasprachen unterstützt, ist die Formulierung von logischen Konsistenzbedingungen in X-Evolution nicht möglich.

3.3.8 XCase

In [Klí10] wird das Tool XCase vorgestellt, welches die Evolution von XML-Schemata in einer Fünf-Ebenen-Architektur ermöglicht. Die oberste, konzeptionelle Ebene (Platform-Independent Level, PIM) gestattet es dem Anwender das Schema durch ein UML-Modell zu modellieren. Durch die Nutzung von UML zusammen mit der Object Constraint Language (OCL) lassen sich in XCase alle Arten von Integritätsbedingungen umsetzen. Anpassung der Dokumente an das neue Schema wird mittels XSLT-Skripts realisiert.

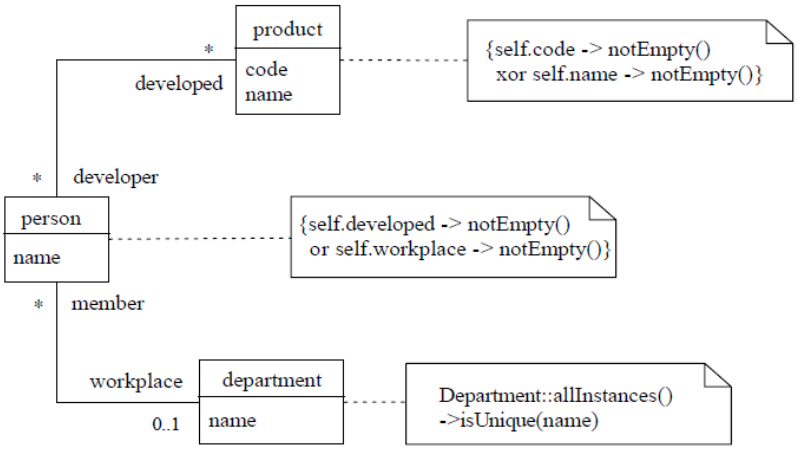


Abbildung 3.13: Integritätsbedingungen in UML-to-XML-Framework

3.4 Datenbanksysteme

Viele Datenbanksysteme verfügen über eine Erweiterung um XML-Dokumente zu verwalten. Darüber hinaus werden Metoden bereitgestellt um Schemata zu registrieren und die Dokumente gegen dieses Schema zu validieren. Da einige dieser Systeme zudem die Evolution von XML-Schemata teilweise unterstützen, wird im folgenden ein kleiner Überblick darüber gegeben, inwieweit diese Systeme die Integritätsbedingungen umsetzen.

3.4.1 IBM DB2

In DB2 ([IBM11]) ist die XML-Schemaevolution nur beschränkt möglich. Die erlaubten Evolutionsschritte beschränken sich nur auf solche Operationen, die sicherstellen, dass bisher gültige Dokumente weiterhin gültig bleiben. Zu diesen Operationen zählen solche Schritte, die die Informationskapazität erhöhen. Beispiele für solche Operationen sind z.B. das Setzen des *use*-Wertes eines Attributes von *required* auf *optional* oder das Erhöhen des *maxOccurs*-Wertes einer Elementdefinition. Integritätsbedingungen bezüglich Wertebereich, Struktur, Schlüsselintegrität und logischer Konsistenz sind durch den Validierungsprozess zwar umsetzbar, durch die Einschränkungen jedoch nur in geringen Umfang verfügbar.

3.4.2 Oracle

Oracle ([Ada12]) bietet in ihrem Datenbanksystem mehrere Schnittstellen für die Schemaevolution an, darunter auch eine API für Java und einige Kommandozeilentools. Die Transformation des Schemas (DTD oder XML-Schema) und der dazugehörigen Dokumente erfolgt mittels XSLT. Der XSD-Validator von Oracle arbeitet auf einem DOM-Baum, der die Pfade der ungültigen Dokumente auf einem Stack abgelegt. Zu den ungültigen Elementen wird zusätzlich die genaue Position (Zeilen- und Spaltennummer) im XML-Dokument gespeichert. Die Validierung selbst erfolgt mittels SAX. Dabei werden Struktur, Wertbereich und Schlüsselbeziehungen überprüft. Optional ist es auch möglich die Validierung abzuschalten, dabei werden die XML-Dokumente lediglich auf ihre Wohlgeformtheit geprüft. Eine Unterstützung von XML-Schema 1.1 und den damit verbundenen logischen Konsistenzbedingungen ist noch nicht realisiert, da der Schema-Validator von Oracle auf die JDK-Bibliothek aufsetzt, welche momentan noch mit der alten

Version von XML-Schema arbeitet. Eine alternative Implementation mittels Apache-Xerces bzw. Saxon wird von den Entwicklern in Betracht bezogen, wurde aber noch nicht umgesetzt (siehe [Men11]).

3.4.3 Microsoft SQL Server 2008

Der SQL Server 2008 ([Mic08]) von Microsoft ermöglicht die Speicherung von XML-Dateien und die Registrierung von XML-Schemata. Die Schemaevolution ist nur begrenzt mit SQL Server umsetzbar. Zwar existiert eine von Microsoft entworfene Erweiterung zu XQuery (XML DML), welche die Manipulation von XML-Dokumenten ermöglicht, allerdings ist diese nicht explizit auf die Evolution eines Schemas inkl. der Adaption der Dokumente ausgelegt. Die Validierung eines Dokumentes erfolgt gegen alle registrierten XML-Schemata. Entspricht das Dokument einer dieser Schemabeschreibungen, so ist es gültig. Da Microsofts Datenbanksystem auch in der aktuellen Version (SQL Server 2012) bisher nur XML-Schema 1.1 unterstützt, erfolgt keine Prüfung von logischen Konsistenzbedingungen.

3.4.4 MySQL 6.0

MySQL ([MyS12]) bietet zwar die Möglichkeit XML-Dokumente im Datenbanksystem zu verwalten, allerdings ist die Registrierung eines XML-Schema nicht möglich. Dadurch ist weder eine Schemaevolution noch eine Validierung der Dokumente möglich.

3.4.5 Software AG Tamino

Tamino ([Sch05]) bietet zur Modellierung von XML-Schemata einen grafischen Editor an, der XML- bzw. Schema-Dateien in einem konzeptionellen Modell darstellt. Zur Sicherstellung der Integritätsbedingungen bzgl. Wertebereich und Schlüssel-Fremdschlüssel-Beziehungen werden verschiedene graphische Komponenten wie Dropdown-Listen, Checkboxen, Property-Value-Tabellen (siehe Abbildung3.14), Dialoge und Toolbars eingesetzt. Dadurch wird der Anwender bei der Eingabe von Werten und Strukturierung des Schemas unterstützt. Die strukturelle Integrität wird im Modell durch Einschränkung der erlaubten Verbindung zwischen den Schema-Elementen erzwungen. Zum Beispiel können Attribute nur Annotationen als Nachfolge-Element besitzen. Logische Konsistenzbedingungen werden zur Zeit nicht unterstützt. Für einzelne Schema-Komponenten und deren Integritätsbedingungen beruft sich Tamino auf die Einhaltung XML Schema Part 0 und Part 1. Zu deren Umsetzung erfolgen keine näheren Angaben, da es sich bei Tamino um ein kommerzielles System handelt.

Die XML-Schemaevolution ist in Tamino nur begrenzt möglich, da nur solche Schemaanpassungen erlaubt sind, die die Gültigkeit der Dokumente erhalten. Tamino ermöglicht die Validierung der Dokumente gegen das neu entwickelte Schema. Die Ausgabe der Validierung erfolgt in einen eigens dafür vorgesehenen Panel (*Output Panel*), in dem detaillierte Informationen zu den ungültigen XML-Dokumenten aufgelistet werden.

3.4.6 eXist

eXist ([Mei09]) ist eine native Open Source XML-Datenbank, die über zwei Möglichkeiten verfügt XML-Dokumente gegen eine DT oder ein XML-Schema zu validieren.

Bei der impliziten Validierung werden die XML-Dokumente bereits beim Laden in Datenbank überprüft, während die explizite Validierung über XQuery-Anweisungen erfolgt. Das exist-Datenbanksystem unterstützt die Schemaevolution, indem sie es dem Anwender ermöglicht XQuery-Update-Anweisungen zu formulieren.

Der Validierungs- und Evolutionsprozess basiert auf XML-Parser Apache Xerces und einigen Java Bibliotheken (java.xml.validation, javax.xml.parsers). Durch die Nutzung von Xerces gestattet eXist die Formulierung von logischen Konsistenzbedingungen.

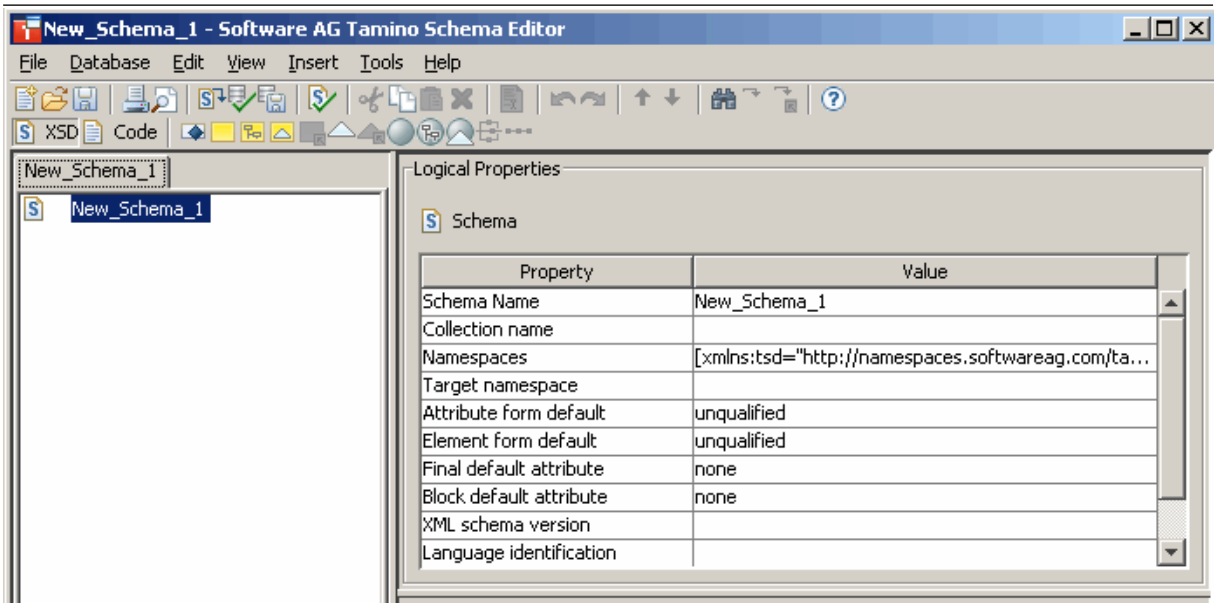


Abbildung 3.14: Eine Property-Value-Tabelle von Tamino

Das Ergebnis der Validierung wird in XML-Dokumenten, den sogenannten Validation reports, gespeichert. Ist das Dokument bezüglich eines XML-Schemas gültig, wird ein Report mit dem Status *valid* erzeugt (siehe Abbildung 3.15). Im Falle eines ungültigen Dokumentes wird eine Report mit dem Status invalid und mehreren Fehlermeldungen erzeugt (siehe Abbildung 3.16). Diese Fehlermeldungen bestehen aus der Zeile und Spalte in der der Fehler auftrat, sowie eine kurze Erklärung für den Grund des Fehlers.

```
<?xml version='1.0'?>
<report>
  <status>valid</status>
  <namespace>MyNameSpace</namespace>
  <duration unit="msec">106</duration>
</report>
```

Abbildung 3.15: Ausgabe für eine erfolgreiche Validierung durch eXist

3.5 Vergleich der Systeme

Tabelle 3.1 gibt einen Überblick über die betrachteten Forschungsprototypen und Datenbanksysteme. Dabei wird dargestellt, welches Schema von dem jeweiligen System unterstützt wird und welche Arten von Integritätsbedingungen umgesetzt werden.

Viele der vorgestellten Systeme unterstützen die logischen Konsistenzbedingungen noch nicht. Grund hierfür ist, dass die neue Version von XML-Schema noch nicht lange (April 2012) existiert bzw. die Systeme mit ihren eigenen Schemasprachen oder mit OCL arbeiten. In dieser Arbeit wird gezeigt, wie sich zusätzliche Integritätsbedingungen in einem konzeptionellen Modell formulieren lassen um sie anschließend in XML-Schema-Dateien zu integrieren und die Validierung der Dokumente zu ermöglichen. Dabei

```
<?xml version='1.0'?>
<report>
  <status>invalid</status>
  <namespace>MyNameSpace</namespace>
  <duration unit="msec">39</duration>
  <message level="Error" line="3" column="20">
    cvc-datatype-valid.1.2.1: 'aaaaaaaa' is not a valid value for '
    decimal'.
    </message>
  <message level="Error" line="3" column="20">
    cvc-type.3.1.3: The value 'aaaaaaaa' of element 'c' is not valid.
  </message>
</report>
```

Abbildung 3.16: Ausgabe für eine nicht erfolgreiche Validierung durch eXist

werden außerdem die Wertebereichsintegrität, die strukturelle Integrität und die referentielle Integrität in das Modell implementiert.

System	Schema	Wertebereich	Struktur	Schlüsselintegrität	Logische Konsistenz
XSEM	XML-Schema	x	x	x	x
XEM	DTD		x	(x), nur ID/IDREF	
DTD-Diff	DTD		x	(x)	
XML-Spy/DiffDog	XML-Schema	x	x	x	
Clio	XML-Schema	x	x	x	
UML-to-XML-Framework	XML-Schema	x	x	x	x
X-Evolution	XML-Schema	x	x	x	
XCase	XML-Schema	x	x	x	x
IBM DB2	XML-Schema	x	x	x	x
Oracle	DTD, XML-Schema	x	x	x	
Microsoft SQL Server	XML-Schema	x	x	x	
MySQL 6.0	(kein Schema)				
Software AG Tamino	XML-Schema	x	x	x	
eXist	XML-Schema, DTD	x	x	x	x

Tabelle 3.1: Vergleichstabelle

Kapitel 4

Umsetzung in CodeX

In diesem Kapitel wird die Integration, Speicherung und Visualisierung von Integritätsbedingungen in das konzeptionelle Modell von CodeX vorgestellt.

4.1 CodeX

Der an der Universität Rostock entwickelte Forschungsprototyp CodeX (Conceptual Design and Evolution for XML-Schema) wurde auf Basis des Google Web Toolkits (GWT) entwickelt. GWT ermöglicht es, Client-Server-Anwendungen zu entwickeln, bei denen der Client eine vollständig im Internet-Browser lauffähige Website ist, während serverseitig Java-Programme für die weiterführende Verarbeitung zuständig sind.

Die Programmierung des Clients, des Servers und der Kommunikation via Remote Procedure Calls (RPC) erfolgt dabei komplett in Java. Das Toolkit erzeugt während des Übersetzungsprozesses aus dem Java-Quellcode des Clients die für den Browser lesbaren Dateien (HTML, CSS und JavaScript). Der Übersetzungsprozess ist in Abbildung 4.1 schematisch dargestellt.

Bei der Integration der Integritätsbedingungen muss auf die Client-Server-Architektur geachtet werden. Nicht alle der in Java verfügbaren Bibliotheken und Funktionen lassen sich für die clientseitige Nutzung übersetzen, wodurch nicht alle Integritätsbedingungen clientseitig überprüft werden können.

Andererseits würde eine vollständige Überprüfung der Bedingungen auf Serverseite zu einer hohen Kommunikationslast zwischen Client und Server führen und den Server durch die Berechnungen auslasten.

Es muss entsprechend geprüft werden, welche Integritätsbedingungen sich bereits clientseitig lösen lassen und welche sich nur auf dem Server überprüfen lassen. Im folgenden Abschnitt werden die verschieden Arten von Integritätsbedingungen auf ihre clientseitige Umsetzbarkeit überprüft.

4.2 Integration und Speicherung

4.2.1 Architekturen zur Integritätssicherung

In [AH00] werden folgende Varianten der Integritätssicherung in Datenbankmanagementsystemen unterschieden:

1. Integritätssicherung durch Anwendung
2. Integritätsmonitor als Komponente des DBMS
3. Integritätssicherung durch Einkapselung

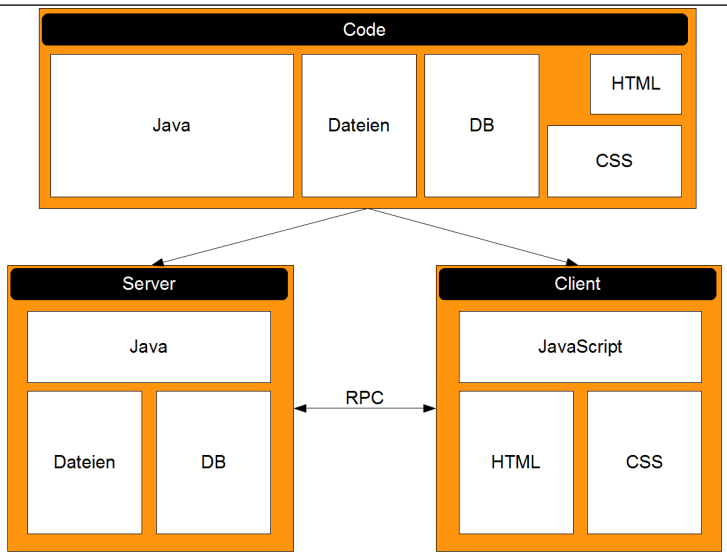


Abbildung 4.1: Übersetzung des GWT-Quellcodes

Die Architekturen werden in den folgenden Abschnitten kurz vorgestellt. Ziel ist es, eine geeignete Variante für die Integration von Integritätsbedingungen zu finden.

Integritätssicherung durch Anwendung

Stellt eine Kernkomponente die Integritätssicherung nicht im vollen Umfang zur Verfügung, kann diese auch durch Teilsysteme oder externe Anwendungen sichergestellt werden. Der Nachteil an dieser Architektur ist zum einen die entstehende Redundanz, da jede neue Teilkomponente oder externe Anwendung die Integrität der Daten sicherstellen muss. Durch die dezentrale Überprüfung kann es des Weiteren zu Inkonsistenzen in den XML-Dokumenten kommen, da die verschiedenen Systeme unter Umständen die Integrationsbedingungen unterschiedlich korrigieren.

Integritätssicherung durch Kapselung

Bei der Integritätssicherung durch Kapselung wird zwischen den Anwendungen und dem Kernsystem eine weitere Architekturebene eingefügt. Diese ermöglicht eine zentrale Überprüfung der Integritätsbedingungen, ist aber vom konkreten System unabhängig. Die Anwendungen greifen bei dieser Architektur nicht mehr auf das Kernsystem zu, sondern arbeiten auf der Zwischenschicht. Durch diese Umsetzung entstehen weder Inkonsistenzen noch Redundanzen, dafür ist die Rechenzeit höher, da für Operationen zwei Bearbeitungsschritte (Anwendung $\rightarrow$ Kapselung, Kapselung $\rightarrow$ Kernsystem) notwendig sind. Außerdem ist mit der Kapselung in Hinsicht auf die Client-Server-Architektur von CodeX eine höherer Kommunikationslast verbunden, da nicht alle Integritätsbedingungen clientseitig umgesetzt werden können.

Integritätssicherung durch Integritätsmonitor

Für Integritätssicherung in CodeX wird ein Integritätsmonitor verwendet. Dadurch ist sichergestellt, dass sowohl externe Anwendungen als auch Teil-Komponenten von CodeX (bereits integriert oder für

zukünftige Entwicklungen), die auf die XSD/XML-Dateien oder die relationale Speicherung des EMX-Modells zugreifen, stets gültige Ergebnisse erhalten.

Durch die Verwendung eines Integritätsmonitors ergeben sich viele Vorteile. Zum einen gestattet der Monitor die optimierte Überprüfung der Integritätsbedingungen durch das System. Wie auch bei der Kapselung entstehen keine Redundanzen und Inkonsistenzen durch mehrfache Implementierungen. Da eine Verteilung der Algorithmen auf Client und Server möglich ist, können Integritätsbedingungen bzgl. Wertebereich und Struktur bereits zur Laufzeit überprüft werden. Dadurch bemerkt der Anwender Fehler bereits während des Modellierungsprozesses.

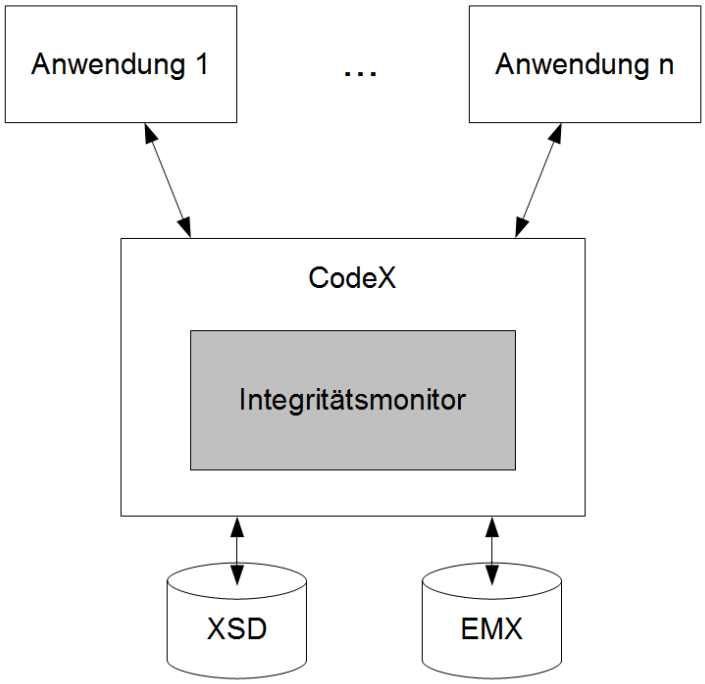


Abbildung 4.2: Integritätsmonitor in CodeX

4.2.2 Bereichsintegrität

Auch: Typintegrität In XML-Schema stehen bereits mehrere vordefinierte Datentypen zur Verfügung. Außerdem können aus bestehenden Datentypen neue, benutzerdefinierte Datentypen erzeugt werden.

Vordefinierte Datentypen

Die vordefinierten Datentypen wurden in einer Klassenhierarchie angeordnet, die der Spezifikation des W3C entspricht (siehe [W3C04]). Jede Klasse verfügt über eine Methode *checkDomainIntegrity*, welche für einen gegebenen String überprüft, ob dieser mit dem Wertebereich des jeweiligen Datentyps übereinstimmt.

Die Überprüfung erfolgt in drei Schritten:

1. Überprüfung, ob die Eingabe zu dem Wertebereich des übergeordneten Datentyps gehört.
2. Vergleich der Eingabe mit einem regulären Ausdruck.

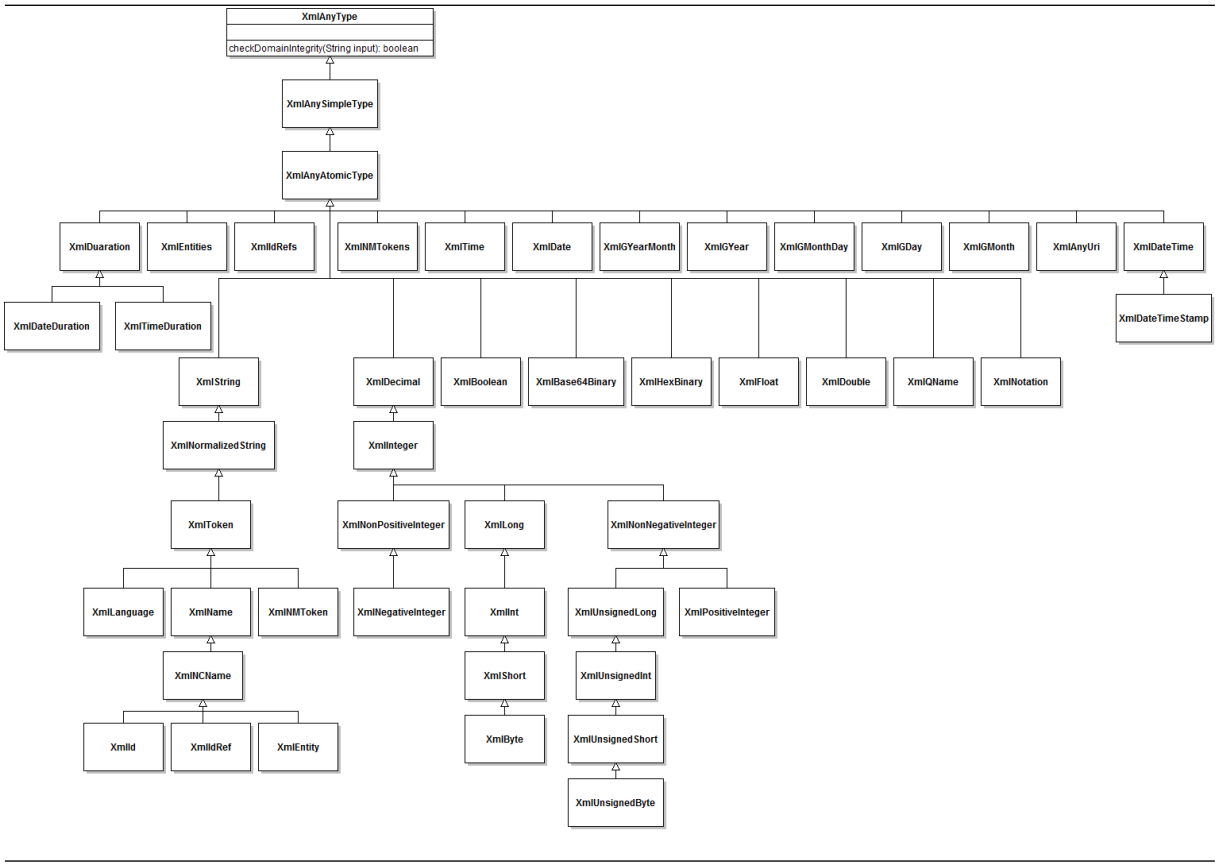


Abbildung 4.3: Klassenhierarchie XML-Datentypen

3. Auswertung der Eingabe mit String- oder Integer-Operationen.

Gehört eine Eingabe bereits nicht in den Wertebereich des übergeordneten Datentyps, so kann sie auch nicht zum spezielleren Datentyp gehören. Falls doch, so wird mittels regulären Ausdrücken überprüft, ob der Wert ein bestimmtes Pattern erfüllt. Erfüllt die Eingabe auch das Pattern, erfolgt eine Überprüfung mit in Java vordefinierten Operationen auf Basisdatentypen. Dies ist notwendig, da bei einigen XML-Datentypen der dazugehörige reguläre Ausdruck zu komplex wäre, beispielsweise bei der Schaltjahr-Regelung auf Datums-Typen. Bei einigen Datentypen kann die letzte Überprüfung auch entfallen.
Beispiel für GDay (Kalender-Tag):

```
public boolean checkDomainIntegrity(String input){
  if(input.matches("[0-9]{2}")){
    int day = Integer.parseInt(input);
    if(day>31||day==0){
      return false;
    }
    return true;
  }
  return false;
}
```

Abbildung 4.4: Code für Überprüfung des Wertebereiches GDay

Benutzerdefinierte Datentypen

Benutzerdefinierte Datentypen werden durch einen Basisdatentypen und ein oder mehrere Facetten beschrieben, wobei nicht jede Facette für jeden Datentyp erlaubt ist. Eine genaue Auflistung welche Facetten für welche Datentypen erlaubt sind gibt die Spezifikation des W3C (siehe [W3C04]). Die Überprüfung eines benutzerdefinierten Datentyps geschieht in drei Schritten:

1. Überprüfung, ob Wert Element des Wertebereichs des Basisdatentyps ist.
2. Kontrolle, ob Facetten für den Basisdatentyp erlaubt sind.
3. Überprüfung, ob Wert durch die Facetten weiterhin erlaubt ist.

Einbindung in CodeX

Die Überprüfung eines einzelnen Dialogfeldes wird mittels eines ValueChange-Handlers realisiert. Dieser prüft, ob eine Eingabe in einem Textfeld verändert wurde und ruft daraufhin die Methode onValueChange auf. Innerhalb dieser Methode werden folgende Schritte ausgeführt:

1. Ermittlung des Eingabe-Strings (getValue())
2. Erzeugung eines Datentyp-Objektes (z.B. new XmlNonNegativeInteger())
3. Überprüfen des Wertebereichs der Eingabe (checkDomainIntegrity())
4. Abhängig vom Ergebnis der Überprüfung entweder
   - gültige Eingabe: Fehlermeldung ausblenden und Modell blau (=fehlerfrei) einfärben
   - ungültige Eingabe: Fehlermeldung einblenden und Modell rot (=Fehler) einfärben

```
Label lblMaxoccurs = new Label("maxOccurs");
flexTable.setWidget(2, 0, lblMaxoccurs);

maxError = new ErrorLabel("");
flexTable.setWidget(2, 2, maxError);

textBoxMaxOccurs = new TextBox();
textBoxMaxOccurs.addValueChangeHandler(new ValueChangeHandler<String>() {
  public void onValueChange(ValueChangeEvent<String> event) {
    String value = event.getValue();
    XmlNonNegativeInteger tester = new XmlNonNegativeInteger();
    maxValid = tester.checkDomainIntegrity(value);
    if(maxValid || value.equals("unbounded")){
      maxError.setEnabled(false);
      entity.setHeaderCorrect(true);
    }
    else{
      maxError.setTooltip("maxOccurs does not match the datatype
      NonNegativeInteger (Integer greater or equal to zero) or \"unbounded
      \".");
      maxError.setEnabled(true);
      entity.setHeaderCorrect(false);
    }
  }
});
textBoxMaxOccurs.setText(this.maxOccurs);
flexTable.setWidget(2, 1, textBoxMaxOccurs);
```

Abbildung 4.5: Überprüfung des Wertebereiches auf Modellebene am Beispiel für maxOccurs

4.2.3 Strukturelle Integrität

CodeX stellt dem Benutzer verschiedene grafische Repräsentationen für die einzelnen XML-Schema-Komponenten bereit. Diese können in der Toolbox auf der rechten Seite des EMX-Editors ausgewählt werden und per Klick in den Editierbereich eingefügt werden. Eine Übersicht über die Komponenten des EMX-Modells ist in Abbildung 4.6 skizziert.

Über das Tool *Connection* ist es dem Nutzer möglich die strukturellen Beziehungen zwischen den einzelnen Komponenten zu spezifizieren. Da die Verbindung zwischen den Komponenten aufgrund der Struktur von XML-Schema nicht beliebig sein kann, ist es notwendig, dass nur gültige Beziehungen zwischen den Komponenten erstellt werden dürfen.

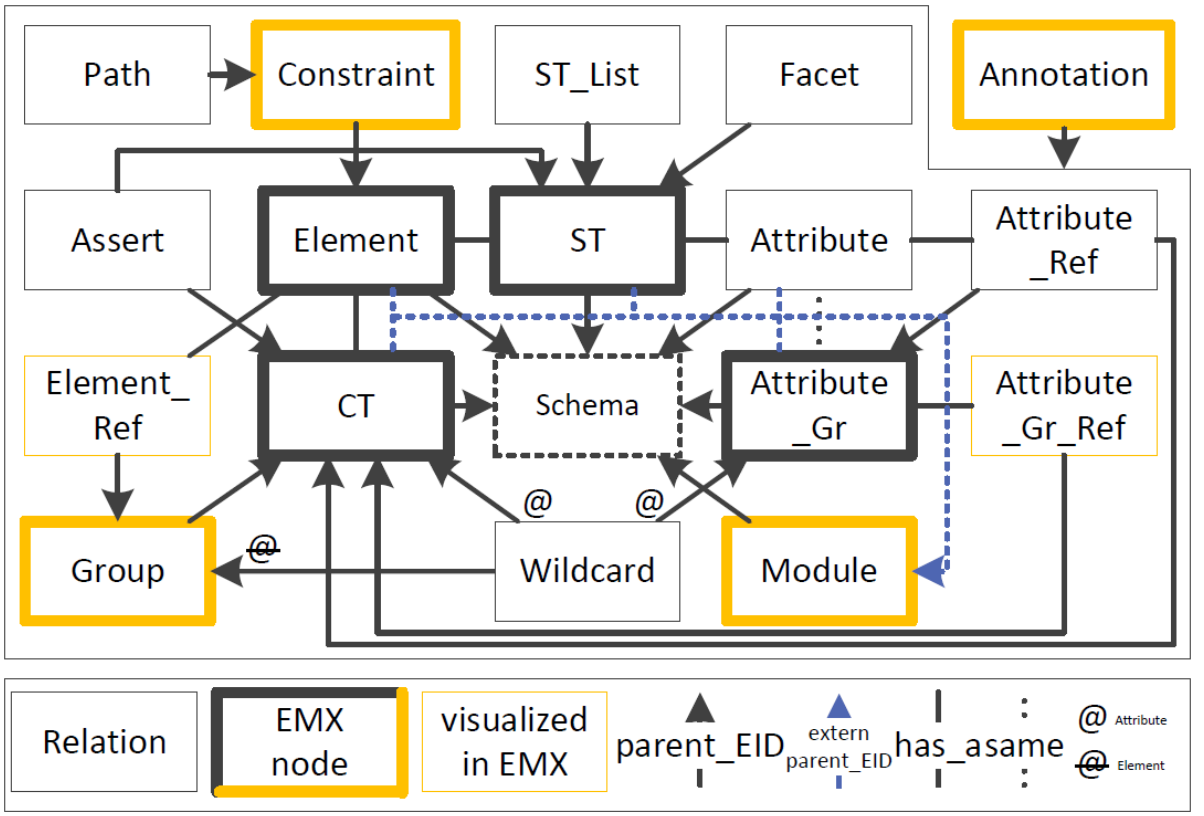


Abbildung 4.6: Logische Struktur des EMX-Modells (aus [TN13])

In Tabelle 4.1 ist eine Übersicht über die erlaubten Verbindungen im EMX-Modell gegeben. Die Richtung des Pfeiles gibt dabei an, von welcher EMX-Komponente eine Verbindung zu einer anderen Komponente erlaubt ist.

Im Editor wird nach der Auswahl des Tools *Connection* intern gespeichert, welche Elemente ausgewählt wurden. Das zuerst ausgewählte Element wird dabei in der Variable *firstClicked*, das zweite in der Variable *secondClicked* gespeichert. Beide Variablen sind vom Typ AbstractEmxEntityModel, welches die in der Vererbungshierarchie übergeordnete Klasse aller EMX-Modelle ist.

In der Methode *createConnection* (siehe Abbildung 4.7) wird anschließend mittels des Java- Schlüsselwortes *instanceof* für firstClicked und secondClicked geprüft, welche konkrete Klasse vorliegt. Für jede Kombination wird anschließend geprüft ob sie zulässig ist. Falls eine zulässige Verbindung erstellt wird,

	element	attribute-group	group	annotation	constraint	module
element		←	←↑	↑	↑	←
attribute-group	↑			↑		←
group	←↑			↑		
annotation	←	←	←		←	←
constraint	←			↑		
module	↑	↑		↑		

Tabelle 4.1: Erlaubte Verbindungen

erfolgt der Methodenaufruf von *addAndDrawValidConnection(firstClicked, secondClicked)*, wodurch die Verbindung der beiden Komponenten gespeichert und gleichzeitig auch visualisiert wird. Ist die Verbindung der Komponenten prinzipiell erlaubt, aber in der falschen Reihenfolge angegeben, so wird die obige Methode mit vertauschten Parametern aufgerufen. Im Falle einer ungültigen Verbindung werden die Speicher-Variablen zurückgesetzt und keine Verbindung erstellt.

```
private void createConnection(){
  ...
  if(firstClicked instanceof EmxElementModel){
    if(secondClicked instanceof EmxAnnotationModel){
      addAndDrawValidConnection(firstClicked, secondClicked);
      return;
    }
    if(secondClicked instanceof EmxElementModel){
      resetClickCounter();
      return;
    }
    if(secondClicked instanceof EmxGroupModel){
      addAndDrawValidConnection(secondClicked, firstClicked);
      return;
    }
    ...
  }
  ...
}
```

Abbildung 4.7: Code-Ausschnitt für ClickHandler

4.2.4 Schlüsselintegrität

Zur Formulierung von Primär- und Fremdschlüsseln wurde ein Dialogfenster programmiert, welches im Abschnitt zur Visualisierung im Detail vorgestellt wird. Das Anlegen eines neuen Schlüssels erfolgt über das Tool „Constraints“ welches sich in der Toolbar des EMX-Editors befindet. Die Speicherung der

Schlüssel erfolgt in der gleichen Datenbank, welche auch die Informationen über die anderen Schema-Komponenten beinhaltet. Dabei werden folgende Daten abgespeichert:

- EID (eindeutige Identifizierung im Modell)
- Parent-EID (Zuordnung zu einem Element)
- Position (x- und y-Koordinaten für Darstellung im Modell)
- Typ (*id, idref, key, keyref* oder *unique*)
- Wert des Schlüssels

In den folgenden Abschnitten werden die Algorithmen zur Sicherstellung der Schlüsselintegrität vorgestellt.

Enititätsintegrität

Beim Anlegen eines Primärschlüssels wird über das *selector*-Attribut angegeben, für welche Elemente die identifizierbare Attributmenge ausgewählt werden soll. Die einzelnen Attribute der Menge werden anschließend in die *field*-Attribute eingetragen.

Damit ein Element eindeutig identifizierbar ist, darf jede Kombination der Attributwerte des Primärschlüssels maximal einmal vorkommen. Dabei müssen die XML-Elemente, die durch die *field*-Komponente ausgewählt wurden paarweise miteinander verglichen werden. Der Vergleich selbst prüft jede selector-Komponente für sich. Ist eine selector-Kopmonente verschieden, so sind die beiden Elemente unterschiedlich und das nächste Paar von Elementen wird geprüft. Sind sie gleich, wird die nächste Komponente geprüft. Sind alle Komponenten gleich, so sind die Elemente gleich und die identifizierende Attributmenge erfüllt die Schlüsseleigenschaft nicht. Der Vergleich aller Elemente wird abgebrochen, sobald ein Duplikat gefunden wurde (negatives Ergebnis, *false*) oder alle Elemente miteinander verglichen wurden (positives Ergebnis, *true*). Der dazugehörige Algorithmus ist in Abbildung 4.8 dargestellt. Für den Fall, dass keine Duplikate vorliegen eine Laufzeit von $O(n^2 * log(m))$, wobei n die Anzahl der zu vergleichenden Elemente und m die Anzahl der *field*-Komponenten ist. Falls Duplikate vorkommen liegt die Laufzeit bei $O(n * log(n) * log(m))$, da nicht alle Elemente miteinander verglichen werden müssen.

Alternativ kann Primärschlüssel auch durch die Eingabe einer *ID* erzeugt werden. Die Überprüfung auf doppelte Elemente erfolgt dabei durch die Java-Klasse *java.util.Set* aus der JDK-Bibliothek. Dabei wird eine Menge aus *ID*s erzeugt und automatisch geprüft, ob eine neue *ID* bereits in der Menge enthalten ist.

Referentielle Integrität

Die referentielle Integrität wird durch die Angabe eines Fremdschlüssels mittels *keyref* bzw. *idref* erzeugt. Die Formulierung dieser Fremdschlüssel erfolgt in der gleichen Art von Dialogfenster wie bei *unique, key* und *id*. Zusätzlich zu der Schlüsseleigenschaft muss sichergestellt werden, dass beim Ändern oder Löschen eines Schlüssels die entsprechenden Fremdschlüssel angepasst werden. Die Realisierung der referentiellen Integrität erfolgt durch ein Trigger-ähnliches System, welches bei jeder Veränderung eines Schlüssels die Fremdschlüssel überprüft.

4.2.5 Logische Konsistenz

Integration in den EMX-Editor

Logische Konsistenzbedingungen können sowohl für komplexe, als auch für benutzerdefinierte Datentypen formuliert werden. Komplexe Typen werden im graphischen Modell durch eine Gruppe (group) dargestellt. In Abbildung 4.9 wird gezeigt, wie über das Kontextmenü der Gruppe das Assertion-Menü

```
//elements: Menge von Elementen, die durch selector ausgewählt wurden
//field: Ausgewählte Sub-Elemente durch field
Menge von field-Elementen
for(int i=0; i<elements.length-1; i++){
  for(int j=i+1; j<elements.length; j++){
    boolean equal = true;
    for(int k=0; k<field.length; k++){
      if(i.getField(k) != j.getField(k)){
        equal = false;
        break;
      }
    }
    if(equal){
      return false;
    }
  }
  return true;
}
```

Abbildung 4.8: Code zur Duplikaterkennung von XML-Elementen

aufgerufen wird. Über dieses Menü lassen sich die Assertions angelegt, verändert oder gelöscht werden. Eine detaillierte Beschreibung folgt im Abschnitt Visualisierung. Bei benutzerdefinierten Datentypen stellen Assertions eine besondere Art der Facetten dar. Facetten werden beim Anlegen eines neuen Datentyps in CodeX formuliert. Dazu wird in der Auswahl des Typs der Facette der Eintrag „Assertions“ gewählt und anschließend die Integritätsbedingung angegeben.

Speicherung der Assertion-Einträge in die Datenbank

Die Speicherung der assert- bzw assertion-Einträge erfolgt wie für die anderen EMX-Elemente in einer relationalen Datenbank. Die Tabelle für die logischen Konsistenzbedingungen hat den in Tabelle 4.2 gezeigten Aufbau. Neben der für die meisten EMX-Elemente gültigen Attribute wie den künstlichen Schlüssel (*EID*) und den Verweise auf das Modell (*file_ID*), werden für die logischen Konsistenzbedingungen zusätzlich der Schlüssel für den betroffenen komplexen oder einfachen Typen (*parent_EID*) gespeichert. Außerdem wird die ausformulierte Integritätsbedingung im Attribut *test_value* hinterlegt.

file_ID	INT NOT NULL	Schlüssel des Modells
EID	INT NOT NULL PRIMARY KEY	Primärschlüssel
parent_EID	INT	Schlüssel des zugeordneten Elementes
test_value	VARCHAR(200)	Konsistenzbedingung
xpathDefaultNS	VARCHAR(200)	Namespace-Angabe

Tabelle 4.2: Relationale Speicherung von Assertions

Um die Client-Server-Kommunikation zu minimieren, werden alle Integritätsbedingungen beim Laden des Modells in Java-Objekte überführt. Diese verfügen über die gleichen Attribute wie in der relationalen Speicherung. Zudem wurden Getter- und Setter-Methoden angelegt um die Werte auszulesen und zu verändern.

Das Mapping zwischen den Daten aus der Datenbank und den im EMX-Editor verwendeten Java-Objekten erfolgt mittels JDBC.

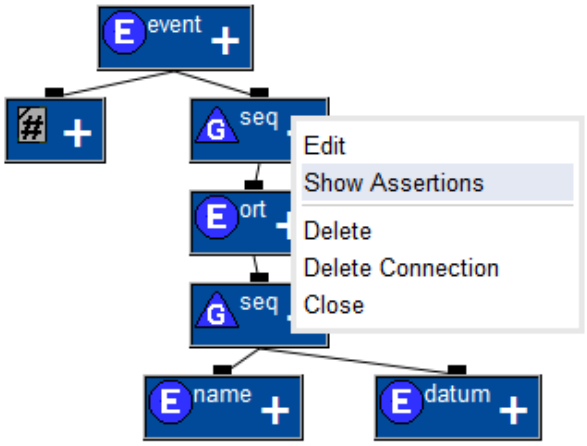


Abbildung 4.9: Einbindung von Assertions auf Modellebene

Speicherung im XML-Schema

Die Integritätsbedingungen werden in XML-Schema als assert- bzw. assertion-Elemente hinterlegt. Die Übersetzung erfolgt durch die Ausführung folgender Schritte:

1. Erzeugung eines neuen *assert*- bzw. *assertion* Elementes
2. Namespace *xs* hinzufügen
3. Erzeugung eines neuen Attributes *test* für das Element e mit dem Wert aus *test_value*
4. Hinzufügen des assert- bzw. assertion-Elementes unter das Parent-Element, welches durch die parent_EID bestimmt ist

In Abbildung 4.10 wird gezeigt, wie die Transformation mittels JDOM in Java umgesetzt ist.

```
public void transformSingleAssertion(String assertionString, Element
element){
 Element assertElement = new Element("assert");
 assertElement.setNamespace(xs);
 assertElement.setAttribute("test", assertionString);
 element.addContent(assertElement);
}
```

Abbildung 4.10: Übersetzung einer assert-Klausel

4.3 Visualisierung

In diesem Abschnitt wird die Visualisierung der Integritätsbedingungen bzgl. Wertebereich, Schlüssel-Beziehungen und logischer Konsistenz vorgestellt. Die strukturelle Integrität wird nicht zusätzlich visualisiert, da die dazugehörigen Algorithmen lediglich korrigierende Maßnahmen ergreifen.

4.3.1 Wertebereich

Verletzte Integritätsbedingungen im EMX-Editor werden durch die Warnfarbe Rot hervorgehoben. Ist eine Komponente fehlerhaft, so wird ihr normalerweise dunkelblauer Hintergrund rot gefärbt. Dadurch ist es dem Nutzer beim Überblicken des Modells möglich die Fehler schnell zu finden. Wird das Detailfenster eines Elementes aufgerufen, so werden die fehlerhaften Attribute mit roten Ausrufezeichen markiert. Bewegt der Anwender den Mauszeiger über dieses Ausrufezeichen erscheint ein Tooltip, der die Ursache für den Fehler erklärt. Dadurch wird dem Nutzer eine Hilfestellung zur Behebung des Problems gegeben (siehe Abbildung 4.11).

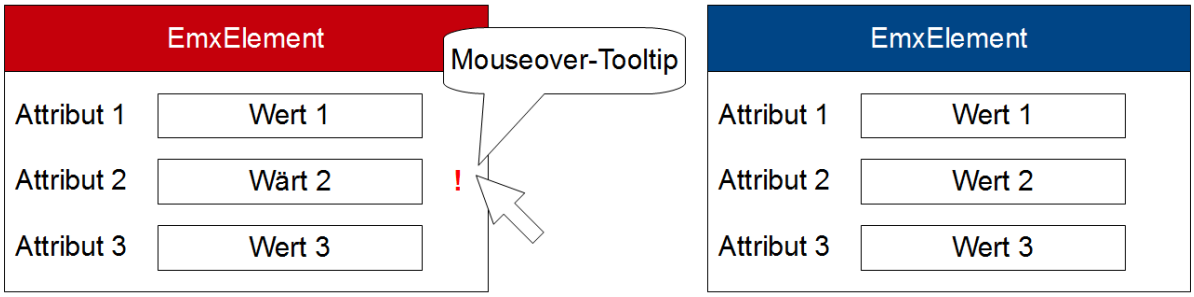


Abbildung 4.11: Visualisierung von fehlerhaften Entitäten

Die Kontrolle erfolgt mittels eines ValueChangedHandler, welcher für jedes Textfeld, in dem ein Wert eingegeben wird, angelegt wird. Ein Beispiel für die Implementation eines solchen Handlers ist in Abbildung 4.12 angegeben.

```
TextBox textBoxName = new TextBox();
textBoxName.addValueChangeHandler(new ValueChangeHandler<String>() {
 @Override
 public void onValueChange(ValueChangeEvent<String> event) {
  String value = event.getValue();
  XmlNCName tester = new XmlNCName();
  textBoxNameValid = tester.checkDomainIntegrity(value);
  if(textBoxNameValid){
   textBoxNameError.setEnabled(false);
   entity.setHeaderCorrect(true);
  }
  else{
   textBoxNameError.setTooltip("Name does not match the datatype NCName ([a
   -zA-Z_][a-zA-Z0-9._-]*).");
   textBoxNameError.setEnabled(true);
   entity.setHeaderCorrect(false);
  }
 }
});
```

Abbildung 4.12: Beispiel für einen ValueChangedHandler

Die Überprüfung erfolgt sofort bei jeder einzelnen Änderung eines Attributwertes. Dadurch ist es bereits

bei der Eingabe möglich Fehler aufzudecken und korrigierende Maßnahmen zu ergreifen.

4.3.2 Schlüssel

Zur Definition von (Fremd-)Schlüsseln wurde ein neues Dialogfenster erschaffen. In diesem wird der Typ des Schlüssels in einer Drop-Down-Box ausgewählt. Abhängig von der Auswahl passen sich die anderen UI-Elemente an. Für den *id/idref*-Mechanismus erscheint eine Textbox, welche die Angabe des Modell-Elementes ermöglicht, für welches *ID*s vergeben bzw. auf welche referenziert werden soll. Für *key, keyref* und *unique* wird eine Textbox zur Angabe des *selector*-Elementes und eine Textbox für die Angabe eines *field*-Elementes dargestellt. Weitere *field*-Elemente können per Button eingefügt bzw. bestehende *field*s wieder gelöscht werden. Beim Löschen wird sichergestellt, dass mindestens eine *field*-Angabe erhalten bleibt. Um die Angabe der Elemente zu vereinfachen wurde zudem ein weiteres Dialogfenster angelegt, mit dessen Hilfe ein Überblick über die im Modell definierten Elemente gegeben wird. Aus dieser Übersicht lassen sich die gesuchten Elemente auswählen. Nachdem ein Schlüssel angelegt wurde, wird zwischen dem Schlüssel-Modell und dem ausgewählten Schlüssel-Element eine grüne Verbindungslinie gezogen um die Referenz zu visualisieren.

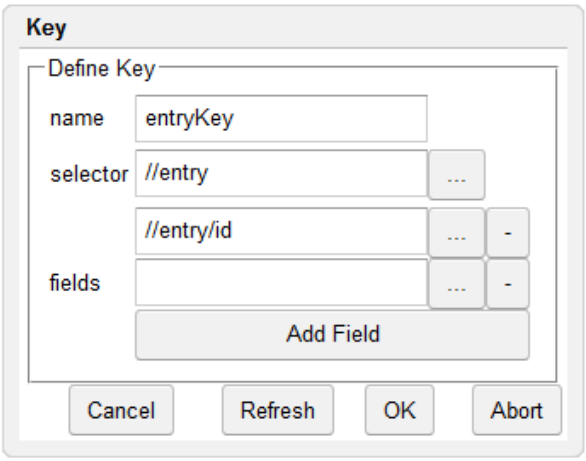


Abbildung 4.13: Visualisierung von Schlüsseln

4.3.3 Logische Konsistenz

Abbildung 4.14 zeigt das Dialogfenster, welches für einen komplexen Typ alle bisher definierten Assertions darstellt. In dieser Übersicht lassen sich neue Assertions anlegen („Add Assertion“), bestehende Assertions bearbeiten („Edit“). Außerdem ist es auch möglich Assertions wieder zu entfernen („-“).

Zum Definieren einer einzelnen Assertion wurde eine Toolbox (siehe Abbildung 4.15) implementiert, über welche der Benutzer unterstützen soll. In der oberen Hälfte des Dialogfensters befindet sich eine Sammlung der gebräuchlichsten Funktionen, welche nach ihrer logischen Zugehörigkeit gruppiert wurden. Da insbesondere Wertevergleiche auf (aggregierten) Werten verschiedener Elemente in den Konsistenzbedingungen umgesetzt werden sollen, wurden diese in die Toolbox mit aufgenommen. Zudem ist es möglich, über eine Dropdown-Liste die im Schema definierten Elemente und Attribute auszuwählen und in die Assertion einzufügen. Der untere Teil des Dialoges enthält eine TextBox, in der die Assertion formuliert wird. Diese Box kann entweder direkt editiert oder durch die Funktions-Buttons erweitert werden. Das Einfügen der Funktionen erfolgt an der Stelle, an der sich die Einfügemarke der Textbox

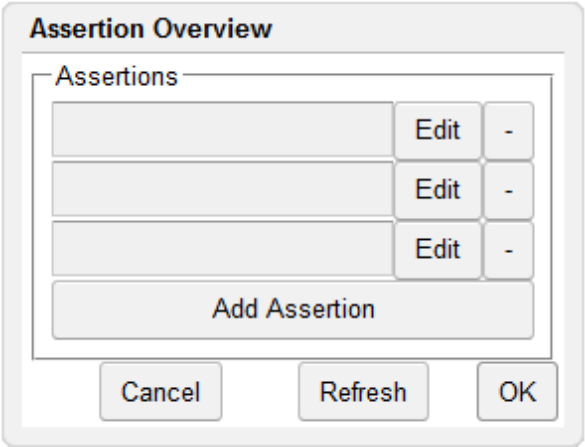


Abbildung 4.14: Übersicht über Assertions eines Typs

befindet. Infix-Operationen, wie mathematische Operationen oder Vergleichsoperatoren werden direkt eingefügt, während alle weiteren Funktionen die aktuelle Auswahl als ersten Parameter übernehmen und den Funktionsnamen sowie eine öffnende Klammer vor der Auswahl einfügen. Weitere Parameter werden zusammen mit der schließenden Klammer nach der Auswahl eingefügt.

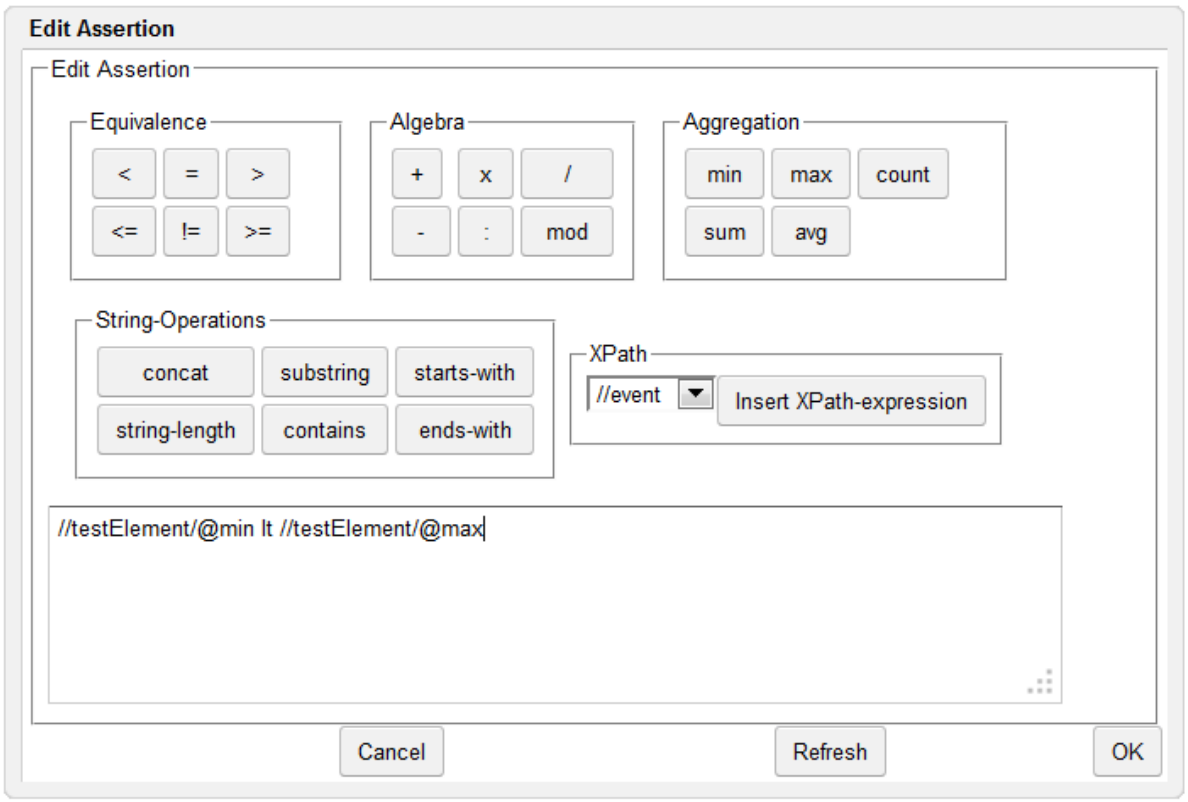


Abbildung 4.15: Tool zur Erstellung einer Assertion

Kapitel 5

Analyse von Integritätsbedingungen

In diesem Kapitel werden die Auswirkungen der Integritätsbedingungen hinsichtlich Kostenabschätzung, Bewertung und Korrektur der Dokumente untersucht.

Die Kostenabschätzung baut auf der Arbeit von [Gru11] auf. Der dort vorgestellte Algorithmus berechnet die Kosten für alle Arten von Schemaevolutionsschritten. Dabei wird ermittelt, wie viele Elemente innerhalb eines Dokumentes durch einen Evolutionsschritt durchschnittlich betroffen sind. Zusätzlich werden die Kosten für das Verändern (bzw. Löschen oder Hinzufügen) eines einzelnen Elementes berechnet. Die Gesamtkosten ergeben sich aus dem Produkt von Anzahl und Einzelkosten.

5.1 Wertebereichsintegrität

Die Überprüfung, ob die Werte von Attributen und Elementinhalten aus dem Wertebereich des zugeordneten Datentyps stammen, erfolgt bereits durch XML-Prozessoren. Fehlerhafte Attributwerte und Elementinhalte können durch eine Anpassung ihres Wertes korrigiert werden. Dabei muss zum einen geprüft werden, ob der eingegebene Attributwert aus dem Wertebereich des Basisdatentyps stammt und ob evtl. Facetten den Datentyp weiter einschränken. Die Kosten für das Ändern eines Attributwertes bzw. Elementinhaltes ergibt sich aus der Anzahl der zu korrigierenden Werte.

5.2 Strukturelle Integrität

Die Analyse der strukturellen Integrität im Rahmen der XML-Schemaevolution und die daraus resultierenden Evolutionsschritte wurde bereits in vielen Arbeiten behandelt ([Wil06, Gru11]). Dabei wurden Algorithmen entwickelt, die die Adaption der Dokumente hinsichtlich der Struktur des neuen Schemas ermöglichen. Detaillierte Informationen zu den Auswirkungen hinsichtlich Informationskapazität, Korrekturschritten und der Kostenanalyse werden insbesondere in [Gru11] vorgestellt.

5.3 Schlüsseleigenschaften

Die Schlüsseleigenschaft eines XML-Elementes ist gegeben, wenn es durch seine Attributwerte bzw. Subelemente eindeutig identifizierbar ist. Damit jedes Element aus einer Elementmenge eindeutig ist, lassen sich Duplikate durch die folgenden Vorgehensweisen eliminieren:

1. Löschen der Duplikate
2. Minimale Änderung der Attributwerte bzw. Subelemente

Löschen der Duplikate Durch das Entfernen eines bzw. mehrerer Duplikate wird die Schlüsseleigenschaft wieder sichergestellt.Durch das Löschen der Duplikate ergibt sich aber ein Informationsverlust, da die Duplikate in den Nicht-Schlüssel-Attributen evtl. weitere Informationen enthalten. In diesem Fall sollte der Anwender überprüfen, ob die identifizierende Attributmenge des Schlüssels korrekt angegeben wurde. Außerdem muss darauf geachtet werden, dass die minimale Anzahl an Elementen (*minOccurs*) nicht unterschritten wird. Die Kosten für das Anpassen der Dokumente ergibt sich aus der Anzahl der gelöschten Duplikate und den Kosten für das Löschen eines einzelnen Duplikates.

Änderung der Attribute bzw. Subelemente Alternativ lassen sich die Duplikate auch so abändern, dass ihre Attribute bzw. Sub-Elemente eindeutige Werte annehmen. Im Idealfall muss nur ein Attributwert angepasst werden (nur möglich wenn Wertebereich unendlich groß bzw. größer als Elementmenge ist). Bei der Anpassung muss darauf geachtet werden, dass die Änderung keine neuen Duplikate erzeugt. Wird der *id/idref*-Mechanismus zur Angabe von Schlüsseln benutzt, so muss lediglich der *ID*-Wert des Duplikates geändert werden. Die Anpassungskosten der Dokumente ergeben sich aus der Gesamtzahl veränderter (Attribut-)Werte.

5.4 Logische Konsistenz

Ausgehend von logischen Konsistenzbedingungen werden insert-, update-, und delete-Operationen erzeugt. Wird das *test*-Attribut einer Integritätsbedingung zu *true* ausgewertet, so muss keine Anpassung der Dokumente erfolgen. Wird das Attribut hingegen zu *false* (z.B. durch eine nicht erfüllte Gleichung) ausgewertet, so muss eine Menge von Operationen gefunden werden, die die Dokumente mit minimalen Aufwand verändert. Eine Übersicht über die Ermittlung der notwendigen Schemaevolutionsschritte erfolgt in den nächsten Abschnitten. Zunächst erfolgt die Angabe einiger Beispiele um das Vorgehen zu verdeutlichen:

- test = “count(xPath) gt x“
  - falls false, so müssen (count(xPath) - x +1)-Elemente eingefügt werden
- test = “count(xPath) le x“
  - falls false, so müssen (x - count(xPath))-Elemente hinzugefügt werden
- test = “@min le @max“
  - falls false, so muss einer der Attributwert für jedes Vorkommen des Attributes angepasst werden
- test = “fn:exists(xPath)“
  - falls false, so muss ein Element eingefügt werden, dass den XPath-Ausdruck erfüllt.

5.4.1 Größenvergleiche von Attribute

Werden mittels <, <=, ==, !=, >=, > zwei Attributwerte verglichen, so muss für jedes Element, welches die Assertion enthält, überprüft werden, ob der als Ergebnis true oder *false* zurückgeliefert wird. Ist das Ergebnis für ein Vorkommen true, so muss keines der Attribute angepasst werden und es entstehen an dieser Stelle keine Kosten. Wird jedoch *false* zurückgeliefert, so muss eines der Attribute angepasst werden. Fehlt eines der Attribute (z.B. durch Optionalität), so wird der Vergleich ignoriert, wodurch das jeweils andere Attribut (sofern vorhanden) nicht angepasst werden muss.

Arithmetik

Durch die Anwendung von arithmetischen Operationen wie Addition, Multiplikation, Absolutbeträge und Rundungen werden lediglich die Werte der Attribute bzw. Elementinhalte verändert, die Auswertung durch die Vergleichsoperatoren bleibt jedoch unverändert. Es besteht ebenso keine Auswirkung der Auswertung von Aggregatfunktionen, die im folgenden beschrieben wird.

5.4.2 Anzahl von Knoten

Mittels *count(xPath)* lässt sich die Anzahl der Knoten ermitteln, die den XPath-Ausdruck erfüllen. Durch numerische Vergleichsoperatoren lässt sich die Knotenanzahl gegen eine Konstante, eine Formel, einer weiteren *count*-Funktion oder einer zusammengesetzten Funktion testen. Wird die Bedingung erfüllt, ist keine weitere Anpassung der Dokumente erforderlich. Ist die Bedingung jedoch nicht erfüllt, so müssen zusätzliche Knoten in die Dokumente eingefügt oder bestehende Knoten aus dem Dokument entfernt werden. In den folgenden Abschnitten erfolgt eine Betrachtung von Integritätsbedingungen mit einem oder mehreren *count*-Aufrufen. Sofern nicht anders angegeben, treffen diese Betrachtungen auch für alle nachfolgenden Operationen zu und werden nicht wiederholt erwähnt. Insbesondere bei Funktionen die numerische Werte zurückliefern brauchen insbesondere Größenvergleiche nicht explizit wiederholt werden.

Bedingungen mit einem count-Aufruf

Enthält die *test*-Bedingung der Integritätsbedingung lediglich eine *count*-Funktion, so lässt sich die Gleichung/Ungleichung leicht lösen. Angenommen, die *count*-Funktion befindet sich auf der linken Seite der Formel, dann ergeben sich folgende Fälle:

Fall 1: < oder <= Da die *count*-Funktion einen größeren Wert zurückliefert als gefordert, muss die Anzahl der Elemente soweit reduziert werden, bis die Anzahl der durch den XPath-Ausdruck spezifizierten Knoten klein genug ist. Zudem muss überprüft werden, ob die minOccurs-Angabe des betroffenen Elementes nicht unterschritten wird. Um die Kosten zu ermitteln, werden die Anzahl der zu entfernenden Knoten benötigt. Diese Zahl wird mit den Kosten für das Entfernen eines einzelnen Elementes multipliziert.

Fall 2: > oder >= Analog zu der kleiner(-gleich)-Ungleichung müssen zur Erfüllung der Integritätsbedingung solange Elemente hinzugefügt werden, bis die Anzahl größer (oder gleich) dem Ausdruck auf der rechten Seite ist. Zudem muss überprüft werden, ob die *maxOccurs*-Angabe des betroffenen Elementes nicht überschritten wird. Die Kosten ergeben sich aus der Multiplikation der Anzahl der Elemente mit den Kosten für das Hinzufügen eines einzelnen Elementes.

Fall 3: != Damit diese Ungleichung erfüllt ist, muss im Dokument entweder ein einzelnes Element hinzugefügt oder gelöscht werden. Damit die Kosten möglichst gering ausfallen, sollte die Option gewählt werden, die das Dokument am wenigsten verändert. Im Blick auf den Informationsgehalt sollte vom Nutzer abgewägt werden, ob er die bestehenden Informationen behalten (Hinzufügen statt Löschen) oder auf den Aufwand zusätzliche Informationen hinzufügen zu müssen verzichten möchte (Löschen statt Hinzufügen).

Fall 4: == Um Gleichheit zu erhalten, muss zunächst geprüft werden, ob zu viele oder zu wenige Elemente vorhanden sind. Falls zu viele Elemente im Dokument gegenwärtig sind, muss die Anzahl entsprechend reduziert werden, ansonsten müssen weitere Elemente eingefügt werden. Es besteht auch die Möglichkeit, dass die Gleichung nicht erfüllbar ist, wie z.B 5*count(xPath) == 4. Diese Formel wäre äquivalent zu count(xPath) == 0.8. Da die *count*-Funktion nur ganze Zahlen zurückliefern kann, ist die Ungleichung nicht auflösbar. Eine automatische Konfliktauflösung durch die XSD-Prozessoren, die bereits XML-Schema 1.1 unterstützen, ist noch nicht umgesetzt worden. Um diesen Konflikt zu umgehen wird

die Anzahl der Elemente nur solange erhöht/reduziert bis die Gleichung erfüllt ist oder die neue Anzahl der Elemente größer/kleiner dem Ausdruck der rechten Seite ist

Bedingungen mit mehreren count-Aufrufen

Enthält eine logische Konsistenzbedingung mehr als einen *count*-Aufruf, kann im Allgemeinen analog vorgegangen werden. Dazu werden bis auf einen *count*-Aufruf alle Elemente fixiert, d.h. die Anzahl der Elemente bleibt konstant. Für die verbliebene *count*-Funktion wird der Algorithmus aus den obigen Abschnitt angewendet.

Dieses Vorgehen eignet sich nicht für solche Integritätsbedingungen, bei denen *count* sowohl auf Elemente und deren Subelemente ausgeführt wird, da sich Veränderungen auf dem übergeordneten Element auf die Sub-Elemente auswirken. Alternativ könnten auch alle Varianten von Wertepaaren für *count* überprüft werden, allerdings skaliert dieses Vorgehen bei großen Wertebereichen und einer großen Anzahl an Funktionsaufrufen sehr schlecht.

5.4.3 Summe

Mittels *sum(x)* lässt sich die Summe aller Attribute bzw. Elementinhalte für eine Knotenmenge bestimmen. Diese kann anschließend mit einen anderen Wert oder Ausdruck durch einen Vergleichsoperator verglichen werden.

Schlägt dieser Vergleich fehl, so muss die Summe derart abgeändert werden, dass sie den Vergleich erfüllt. Da nicht auf einzelnen Elementen sondern auf einer Knotenmenge gearbeitet wird, kann dieses Problem nicht automatisch gelöst werden. Ähnliche Problemfälle existieren bereits länger, im Bereich der Datenbanken ist dies beim Update von Sichten der Fall. Da dies häufig zu Anomalien führt, erlauben viele Datenbanksysteme meist nur lesende Zugriffe auf Sichten und verhindern den Schreibzugriff.

Prinzipiell können folgende drei Instanzanpassungen vorgenommen werden um die verletzte Integritätsbedingung zu erfüllen:

1. Löschen bestehender bzw. Einfügen neuer Elemente
2. Ändern des Wertes eines Attributes oder Elementes
3. Ändern aller betroffenen Attribute bzw. Elemente

Die Auswahl der entsprechenden Operation muss dabei der Anwender treffen. Dies kann entweder manuell nach der Überprüfung der Integritätsbedingungen geschehen oder in den benutzerdefinierten Einstellungen voreingestellt werden.

Fall 1

Fall 1.1: Summe zu groß Ist die Summe zu groß, d.h die Vergleichsoperatoren $<$, $<=$, $==$ liefern *false* zurück, kann die Summe durch das Löschen von ein oder mehreren Elementen mit positiven Werten verringert werden. Dabei muss eine evtl. *minOccurs*-Angabe für die Anzahl der Elemente eingehalten werden. Die Kosten für die Anpassung ergeben sich aus den Kosten durch das Löschen eines betroffenen Elementes multipliziert mit der Anzahl der gelöschten Elemente.

Alternativ können auch zusätzliche Elemente mit negativen Werten eingefügt werden, wodurch die Summe verringert wird. Dabei muss eine evtl. *maxOccurs*-Angabe für die Anzahl der Elemente eingehalten werden. Außerdem muss der Wert des eingefügten Elementes bzw. Attributes dem zugeordneten Datentyp entsprechen. Die Kosten für die Anpassung ergeben sich aus den Kosten durch das Hinzufügen eines betroffenen Elementes multipliziert mit der Anzahl der hinzugefügten Elemente.

Fall 1.2: Summe zu klein Ist die Summe zu klein, d.h die Vergleichsoperatoren $>$, $>=$, $==$ liefern *false* zurück, kann die Summe durch das Hinzufügen von ein oder mehreren Elementen mit positiven Werten vergrößert werden. Dabei muss eine evtl. *maxOccurs*-Angabe für die Anzahl der Elemente eingehalten werden. Die Einhaltung des Wertebereichs des zugeordneten Datentyps ist ebenfalls notwendig. Die Kosten für die Anpassung ergeben sich aus den Kosten durch das Hinzufügen eines betroffenen Elementes multipliziert mit der Anzahl der hinzugefügten Elemente.

Wenn das nicht möglich ist, können auch Elemente mit negativen Werten gelöscht werden, wodurch die Summe erhöht wird. Dabei muss eine evtl. *minOccurs*-Angabe für die Anzahl der Elemente eingehalten werden. Die Kosten für die Anpassung ergeben sich aus den Kosten durch das Löschen eines betroffenen Elementes multipliziert mit der Anzahl der gelöschten Elemente.

Fall 1.3 Summe erfüllt Ungleichheit nicht In diesem Fall können sowohl Elemente gelöscht oder hinzugefügt werden oder der Wert eines Elementes verringert oder vergrößert werden. Die Wahl der Anpassung erfolgt durch den Anwender muss muss abhängig vom Kontext getroffen werden. Integritätsbedingungen die diesen Operator verwenden, werden in den nachfolgenden Betrachtungen nicht weiter erläutert, da sie immer durch beliebige Anpassungen der betroffenen Elementmenge erfüllt werden können.

Fall 2 Sollen keine Elemente gelöscht oder hinzugefügt werden, so kann ein bestehender Attributwert bzw. Elementinhalt auch modifiziert werden. Dadurch wird das Dokument der kleinstmöglichen Änderung unterzogen. Dies ist somit die kostengünstigere Vorgehensweise, da lediglich eine Veränderung eines Wertes erfolgt anstatt komplette Elementstrukturen einzuführen bzw. zu löschen. Allerdings erfordert dies kontextabhängige Werte, welche allerdings nicht einfach bestimmbar sind.

Fall 3 Ähnlich zu Fall 2 werden durch das gleichmäßige Anpassen aller Werte keine komplexeren Elemente gelöscht oder eingefügt. Allerdings wird das Dokument etwas stärker verändert, da jeder betroffene Wert verändert wird, wodurch die Kosten sich mit jedem veränderten Wert weiter erhöhen. Das Finden kontextabhängiger Werte ist auch hier ein Problem.

5.4.4 Durchschnitt

Der Durchschnitt ist das Ergebnis aus der Summe geteilt durch die Anzahl der Elemente für einen bestimmten XPath-Ausdruck (*avg(x) = sum(x) div count(x)*). Um Integritätsbedingungen durch Anpassung der Elemente zu erfüllen, stehen drei Möglichkeiten zur Verfügung.

1. Verändern eines oder mehrerer Attribute bzw. Elementinhalte
2. Löschen bzw. Hinzufügen eines oder mehrerer Elemente
3. Kombinationen aus Punkt 1 und 2

Fall 1 Dieses Vorgehen entspricht Fall 2 bzw. 3 der Betrachtung zur Summe. Es sind dabei die gleichen Probleme und Anpassungskosten verbunden.

Fall 2 Werden Elemente hinzugefügt oder gelöscht verändert sich sowohl die Summe der Werte als auch die Anzahl der Elemente. Wird dieses Vorgehen in Betracht gezogen, müssen die *minOccurs*- bzw. *maxOccurs*-Angaben für das betroffene Element beachtet werden. Außerdem muss beachtet werden, dass der *count*-Wert erhöht wird, was sich wiederum auf den Durchschnitt auswirkt.

Sei y := count(x) die Anzahl und z = sum(x) die Summe der Elemente vor dem Einfügen bzw. Löschen der Elemente. Sei y' := count(x) die Anzahl nach dem Einfügen bzw. Löschen der Elemente. Dann ergibt sich für die Summe z' = sum(x) nach dem Löschen bzw. Einfügen der Elemente: $z' := z * \frac{y'}{y}$. Das neue

eingefügte Element bzw. das gelöschte Element muss entsprechend einen Wert von mindestens oder genau (bei Gleichheit) z' haben. Die Kosten ergeben sich aus der gelöschten bzw. hinzugefügten Elemente.

Fall 3 Der Durchschnitt lässt sich auch verändern, indem sowohl die konkreten Werte angepasst werden, als auch die Anzahl der Elemente verändert wird. Die Kosten für diese Anpassungsmöglichkeit der Dokumente ergibt sich aus der Anzahl der veränderten Werte und den Kosten für das Löschen bzw. Hinzufügen der Elemente.

5.4.5 Minimum und Maximum

Die Funktionen *min(x)* bzw. *max(x)* ermitteln aus einer Menge von Attributwerten bzw. Elementinhalten den minimalen bzw. maximalen Wert. Ist eine mit *min(x)* oder *max(x)* formulierte Integritätsbedingung (z.B. Test mit einem Vergleichsoperator) nicht erfüllt, so bestehen folgende Möglichkeiten zur Dokumentadaption:

1. Überschreitung des Maximalwertes bzw. Unterschreitung des Minimalwertes
   (a) Löschen aller Elemente, deren Werte das angegebene Maximum überschreiten bzw. das Minimum unterschreiten
   (b) Ändern der entsprechenden Werte
   (c) Einfügen eines neuen Elementes, welches den Vorgaben entspricht (nur für Minimum)
2. Unterschreitung des Maximalwertes bzw. Überschreitung des Minimalwertes
   (a) Löschen aller Elemente, deren Werte das angegebene Maximum unterschreiten bzw. das Minimum überschreiten
   (b) Ändern der entsprechenden Werte
   (c) Einfügen eines neuen Elementes, welches den Vorgaben entspricht (nur für Maximum)

Fall 1 Dieser Fall tritt ein, wenn das Ergebnis von max(x) mit den Vergleichsoperatoren $<$, $<=$ oder $==$, bzw. wenn das Ergebnis von *min(x)* mit den Vergleichsoperatoren $>$, $>=$ oder $==$ gegen einen anderen Wert verglichen wird.

Fall 1.a In diesem Fall müssen alle Elemente entfernt werden, die das gegebene Maximum überschreiten bzw. das Minimum unterschreiten. Die Kosten entsprechen der Anzahl der zu entfernenden Elemente multipliziert mit den Kosten für das Entfernen eines Elements.

Fall 1.b Werden alle Werte so abgeändert, dass sie den Vorgaben für das Minimum bzw. Maximum entsprechen, erfüllen sie die Integritätsbedingung. Die Kosten ergeben sich aus der Anzahl der veränderten Attributwerte bzw. Elementinhalte.

Fall 1.c Eine weitere Möglichkeit, die Integrationsbedingung *min(x)* zu erfüllen und evtl. auch geringe Adaptionskosten als Fall 1.a und 1.b zu erzeugen, ist das Einfügen eines neuen Elementes. Dabei muss sichergestellt werden, dass der neue zu überprüfende Minimum-Wert den Vergleich mittels $>$, $>=$ bzw. $==$ erfüllt.

Fall 2 Dieser Fall tritt ein, wenn das Ergebnis von *min(x)* mit den Vergleichsoperatoren $<$, $<=$ oder $==$, bzw. wenn das Ergebnis von *max(x)* mit den Vergleichsoperatoren $>$, $>=$ oder $==$ gegen einen anderen Wert verglichen wird.

Fall 2.a Dies entspricht Fall 1.a, allerdings sind Minimum und Maximum bezüglich Unterschreitung und Überschreitung vertauscht.

Fall 2.b In diesem Fall kann wie in Fall 1.b vorgegangen werden, es sind jedoch andere Elemente von der Integritätsbedingung betroffen.

Fall 2.c Dieses Vorgehen ähnelt Fall 1.c mit dem Unterschied, dass der Vergleich des neuen Wertes gegen die Operatoren <, <= bzw. == erfolgt.

5.4.6 Lexikographische Ordnung

Die Funktion *compare*(string1, string2 [,collation]) ermöglicht einen genaueren Vergleich von zwei String-Werten als die Vergleichsoperatoren <, ==, usw.. Durch die Angabe einer Sortierreihenfolge (*collation*) ist es möglich, dass Strings gleich sein können, obwohl sie aus einer unterschiedlichen Zeichenfolge bestehen.

In [AM10] wird folgendes Beispiel für die Verwendung der Funktion compare angegeben:

fn:compare('Strasse', 'Straße', 'deutsch') returns 0 if the collation identified by the relative URI constructed from the string value 'deutsch' includes provisions that equate 'ss' and the (German) character 'ß' ('sharp-s'). (Otherwise, the returned value depends on the semantics of that collation.).

Die Funktion liefert als Ergebnis -1, 0 oder 1 zurück, je nachdem, ob das erste Argument kleiner, gleich oder größer als das zweite Argument war. Dieser Wert lässt sich anschließend über die gängigen Vergleichsoperatoren abfragen.

Schlägt die Validierung dieser Integritätsbedingung fehl, so lässt sich durch die Anpassung eines der beiden Attributwerte bzw. Elementinhalte die Gültigkeit wiederherstellen. Die Kosten ergeben sich aus der Anzahl der veränderten Werte.

5.4.7 Verkettung von Strings, Funktionen zur Bildung von Substrings

Die XPath-Functions stellen eine Reihe von Funktionen zur Verfügung um Strings zu Verketten (*concat*) und um Teile von Strings zu extrahieren. Eine Übersicht über diese Funktionen ist in Tabelle 5.1 aufgelistet.

Diese Art von Funktionen haben als Funktionswert Attributwerte bzw. Elementinhalte und vergleichen diese mit anderen Parametern. Als Ergebnis wird entweder *true/false* zurückgegeben oder das Ergebnis ist ein neuer String, der anschließend weiter verwendet wird. Ist eine mit diesen Funktionen definierte Integritätsbedingung nicht erfüllt, so kann sie korrigiert werden, indem der Attributwert bzw. der Elementinhalt angepasst wird. Die Kosten ergeben sich aus der Anzahl der veränderten Werte.

5.4.8 Länge eines Strings

Mittels der Funktion *fn:length* ist es möglich die Länge eines String zu bestimmen. In Verbindung mit einem Vergleichsoperator kann die Länge des Elementinhaltes bzw. des Attributwertes gegen eine Konstante verglichen werden. Auch wenn diese Art der Integritätsbedingung in den meisten Fällen bereits durch einen benutzerdefinierten Datentypen mit eingeschränkten Wertebereich (*length, minLength...*) überprüft wird, existieren einige Anwendungsfälle bei denen eine Assertion notwendig ist. Beispielsweise kann mit *fn:length* eine Assertion formuliert werden, die sicherstellt, dass alle Elementinhalte die gleiche Länge haben, auch wenn die Länge des Strings an sich variabel sein kann. Ist eine auf diese Weise formulierte Integritätsbedingung nicht erfüllt, besteht für Anpassung der Dokumentwerte folgende Möglichkeit zur Korrektur:

- Bei zu langen Eingaben:
  - String nach Überschreitung der geforderten Länge abschneiden

Funktionsname	Beschreibung	Beispiel
fn:concat	Verbindet zwei Strings zu einen	fn:concat('Ha', 'llo') == 'Hallo'
fn:substring	Ermittelt den Substring von einer Position y für n Zeichen	fn:substring('Hallo', 2, 3) == 'all'
fn:upper-case	Wandelt alle Buchstaben in Großbuchstaben um	fn:upper-case('HallO') == 'HALLO'
fn:lower-case	Wandelt alle Buchstaben in Kleinbuchstaben um	fn:lower-case('HallO') == 'hallo'
fn:contains	Ermittelt, ob eine Zeichenkette in einem String vorkommt	fn:contains('Hallo', 'all') == true
fn:starts-with	Prüft, ob ein String mit einer bestimmten Zeichenkette beginnt	fn:starts-with('Hallo', 'Ha') == true
fn:ends-with	Prüft, ob ein String mit einer bestimmten Zeichenkette endet	fn:ends-with('Hallo', 'llo') == true
fn:substring-before	Ermittelt den Substring vor dem Auftreten einer bestimmten Zeichenkette	fn:substring-before('Hallo', 'llo') == 'Ha'
fn:substring-after	Ermittelt den Substring nach dem Auftreten einer bestimmten Zeichenkette	fn:substring-after('Hallo', 'Ha') == 'llo'
fn:matches	Prüft, ob ein String einen regulären Ausdruck erfüllt	fn:matches('XPath-Functions', '*at.*Fun*') == true
fn:replace	Ersetzt das Vorkommen einer Zeichenkette durch eine andere	fn:replace('Hallo', 'Hal', 'Maha') == 'Mahalo'

Tabelle 5.1: XPath-Functions zur Bildung und Bearbeitung von Strings

  - String komplett löschen (sofern Datentyp dies erlaubt)
  - String durch Konstante ersetzen (z.B. '####')

- Bei zu kurzen Eingaben:
  - String mit Füllzeichen auffüllen (z.B. '#')
  - String durch Konstante ersetzen

Die Kosten ergeben sich aus der Anzahl der Werte, die durch eine dieser Anpassungen verändert wurden.

5.4.9 Funktionen auf Datums-Datentypen

Für die Datentypen Duration, Date, Time und allen ähnlichen Datentypen existieren im XQuery- and XPath-Functions-Standard verschiedene Funktionen um Wertevergleiche und Berechnungen auf Teilkomponenten durchzuführen. Diese Komponenten (Tag, Monat,... Zeitzone) können anschließend mit Konstanten verglichen oder in andere Integritätsbedingungen eingesetzt werden. Tabelle 5.2 gibt einen Überblick über die erlaubten Vergleichsoperatoren auf den Datums-Datentypen. Zum Beispiel sind auf dem Datentyp yearMonthDuration die Vergleiche kleiner-gleich (*less-than*) und größer-gleich (*greater-than*) erlaubt, Gleichheit (*equals)* hingegen nicht.

In der Tabelle 5.3 wird gezeigt, aus welchen Datentypen Teilkomponenten extrahiert werden können.

Für den Fall, dass eine Integritätsbedingung nicht erfüllt ist, können die entsprechenden Dokumente durch eine Veränderung der jeweiligen Datums-Komponente angepasst werden. Dies entspricht den Kosten für das Ändern eines einzelnen Wertes.

Datentyp	**less-than**	**greater-than**	**equals**
yearMonthDuration	x	x	
dayTimeDuration	x	x	
duration			x
dateTime	x	x	x
date	x	x	x
time	x	x	x
gYearMonth			x
gYear			x
gMonthDay			x
gMonth			x
gDay			x

Tabelle 5.2: Erlaubte Wertevergleiche auf Datums-Datentypen

Ursprungs-Datentyp \ **extrahierter Datentyp**	**gYear**	**gMonth**	**gDay**	**hour**	**minute**	**second**	**timezone**
duration	x	x	x	x	x	x	
dayTime	x	x	x	x	x	x	x
date	x	x	x				x
time				x	x	x	x

Tabelle 5.3: Komponentenextraktion von Datums-Datentypen

Kapitel 6

Anpassung der XML-Dokumente

In diesem Kapitel wird gezeigt wie sich XML-Dokumente bezüglich der Integritätsbedingungen anpassen lassen. In Kapitel 5 wurde gezeigt, dass Konflikte zwischen logischen Konsistenzbedingungen nicht einfach aufgelöst werden können. Aus diesem Grund erfolgt die Transformation der Dokumente in zwei Schritten. Im ersten Schritt werden die Dokumente unabhängig von den logischen Konsistenzbedingungen an das neue Schema angepasst. Anschließend erfolgt die Umsetzung der logischen Konsistenzbedingungen. Dabei werden die von beiden Transformationen betroffenen Schema-Fragmente gekennzeichnet und dem Nutzer zur manuellen Überprüfung vorgelegt, damit dieser die Dokumente ggf. anpassen kann.

6.1 Transformation ohne logische Konsistenzbedingungen

Zunächst werden alle Transformationsschritte ausgeführt, welche die Struktur, die Schlüssel-Beziehungen und den Wertebereich betreffen. In der bisherigen Implementation von CodeX als Eclipse-Plugin werden die Schemaevolutionsschritte geloggt und in einer eigenen Update-Sprache (XSEL[1], siehe [Tie05]) formuliert. Diese Evolutionsschritte werden anschließend mit dem in [Gru12] vorgestellten Algorithmus in XSLT-Skripts übersetzt werden. Durch die XSLT-Skripts ist die Adaption der zum Schema gehörigen XML-Dokumente möglich. Viele der in Kapitel 2 vorgestellten Prototypen verwenden ebenfalls XSLT zur Dokumentanpassung, wodurch sich diese Technik in der Schemaevolution etabliert hat.

In [TN13] wird ein aktualisierten Sprachvorschlag für eine Update-Sprache vorgestellt. Die Vorteile von ELaX[2] sind eine Vereinfachte Syntaxdarstellung und die Unterstützung der Assertions aus XML-Schema 1.1. Die aktuelle Implementation von CodeX wird diese Sprache zur Umsetzung der Schemaevolution nutzen, allerdings befindet sich die Integration in CodeX noch in der Entwicklung. Aus diesem Grund kann in der momentanen Umsetzung von CodeX keine Adaption der Dokumente erfolgen.

6.2 Überprüfung der logischen Konsistenzbedingungen

Nachdem der erste Schritt zur Adaption der Dokumente erfolgte, werden nun die XML-Dateien hinsichtlich ihrer logischen Konsistenz überprüft und angepasst. Die Einhaltung der logischen Integritätsbedingungen wird durch den Xerces-Parser von Apache getestet. Dabei tritt aber ein kleineres Problem auf, da XML-Parser ihre Validierung beim ersten Vorkommen eines Fehlers stoppen. Grund hierfür ist die XML-Spezifikation des W3C:

„The cases described above are the only types of error which this specification defines. With respect to the processes of the checking of schema structure and the construction of schemas corresponding to schema documents, this specification imposes no restrictions on processors in the presence of errors, beyond the

[1] XML-Schema Evolution Language
[2] Evolution Language for XML-Schema

requirement that if there are errors in a schema, or in one or more schema documents used in constructing a schema, then a conforming processor must report the fact. However, any further operations performed in the presence of errors are outside the scope of this specification and are not ·schema-validity assessment· as that term is defined here“ [W3C04].

Zusätzliche Fehler werden somit meist erst nach dem Beheben des ersten Fehlers erkannt. Der in der Implementation verwendete Parser Xalan-J benutzt einen Default-ErrorHandler für die Validierung der Dokumente gegen ein Schema, dieser wirft bei dem zuerst auftreten schwerwiegenden Fehler eine Exception und bricht den Validierungsprozess ab. Die ist ein standardmäßiges Vorgehen, welches auch in der Javadoc beschrieben ist:

„For XML processing errors, a SAX driver must use this interface in preference to throwing an exception: it is up to the application to decide whether to throw an exception for different types of errors and warnings. Note, however, that there is no requirement that the parser continue to report additional errors after a call to fatalError. In other words, a SAX driver class may throw an exception after reporting any fatalError. Also parsers may throw appropriate exceptions for non-XML errors. For example, XMLReader.parse() would throw an IOException for errors accessing entities or the document.“

Dieses Verhalten kann umgangen werden, indem ein eigener ErrorHandler programmiert wird (siehe Abbildung 6.1, der die Exceptions auffängt und nach der gesamten Validierung zurückgibt.

6.3 Auflösung von Konflikten

Die Validierung eines XML-Dokumentes kann durch einen Rechtsklick auf die dazugehörige Datei im Projekt-Explorer gestartet werden. Es öffnet sich ein Dialogfenster, in welchem ein XML-Schema aus dem gleichen Projekt dem Dokument zugeordnet werden kann. Das Ergebnis der Validierung wird anschließend in einem weitere Dialogfenster (siehe Abbildung 6.2) angezeigt.

6.3.1 Assertions in einfachen Typen

Werden Assertsions als Facetten für einfache Typen angegeben, können diese zu den bereits definierten Facetten im Widerspruch stehen. Beispielsweise kann ein die Länge eines Attributwertes sowohl durch die Facette *length* als auch durch die XPath-Funktion *fn:length* festgelegt werden. Sind die Angaben dazu unvereinbar, kann ohne eine Entscheidung des Nutzers nicht entschieden werden, welche Angabe stimmt. Die Entdeckung solcher Widersprüche ist im Allgemeinen nur schwer erkennbar, wodurch erst bei der Adaption festgestellt werden kann, dass ein Fehler begangen wurde. Die Auflösung solcher Konflikte wird am Ende dieses Kapitels vorgestellt.

6.3.2 Mehrere Assertions in komplexen Typen

Werden mehrere logische Konsistenzbedingungen für einen einfachen oder komplexen Typen verfasst, so können sich diese unter Umständen widersprechen. O.B.d.A. ist es ausreichend das Konfliktpotenzial zwischen zwei Assertions zu betrachten. Dabei können folgende Fälle unterschieden werden:

1. Beide Assertions verändern die Dokumente nicht.
2. Eine Integritätsbedingung verändert die Dokumente, die andere hingegen nicht.
   (a) Die zuerst angewandte Regel verändert die Dokumente.
   (b) Die zweite Regel verändert die Dokumente.
3. Die Integritätsbedingungen verändern unterschiedliche Teile der Dokumente.
4. Beide Konsistenzbedingungen verändern das gleiche Element.

```
public String validateXML(File xmlFile, File schemaFile){
  try{
      final ArrayList<String> errors = new ArrayList<String>();

    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    factory.setNamespaceAware(true);
    DocumentBuilder builder = factory.newDocumentBuilder();
    Document doc = builder.parse(xmlFile);

    SchemaFactory schemaFactory = SchemaFactory.newInstance(
      "http://www.w3.org/XML/XMLSchema/v1.1");
    Source schemaSource = new StreamSource(schemaFile);
    Schema schema = schemaFactory.newSchema(schemaSource);
    Validator validator = schema.newValidator();
    ErrorHandler errorHandler = new ErrorHandler() {

        @Override
        public void warning(SAXParseException arg0) throws SAXException {
          errors.add(arg0.getMessage());
        }

        @Override
        public void fatalError(SAXParseException arg0) throws SAXException
        {
          errors.add(arg0.getMessage());

        }

        @Override
        public void error(SAXParseException arg0) throws SAXException {
          errors.add(arg0.getMessage());
        }
      };
    validator.setErrorHandler(errorHandler);
    validator.validate(new DOMSource(doc));

    for(int i=0; i<errors.size(); i++){
      System.out.println(errors.get(i));
    }

    return "true";
  }
  catch(Exception e){
    return "XSD is not valid, because of the following reason: "+e.
    getMessage();
  }
}
```

Abbildung 6.1: Code für Anlegen eines neuen ErrorHandlers

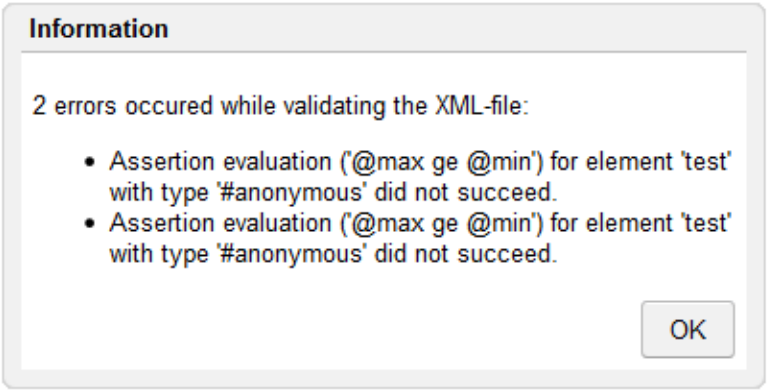


Abbildung 6.2: Fehlgeschlagene Validierung eines XML-Dokumentes

Sei d ein XML-Dokument vor Anwendung der ersten Regel, d' das Dokument, welches sich aus der Evolution von d' nach Anwendung der ersten Regel ergibt und d" das Dokument, welches sich aus d' nach Anwendung der zweiten Regel ergibt.

- $d \overset{Regel1}{\longrightarrow} d' \overset{Regel2}{\longrightarrow} d''$

Fall 1: Haben beide Integritätsregeln keinen Einfluss auf eine Dokumentinstanz, d.h. sie verändern das Dokument nicht, so entstehen auch keine Konflikte die aufgelöst werden müssen. Die Schemaevolution wird ohne Warnungen oder Fehlermeldungen fortgesetzt. Die Reihenfolge in der die Regeln angewendet werden, hat dabei keinen Einfluss auf das Ergebnis. Es gilt:

- $d = d' = d''$
- $d \overset{Regel1,Regel2}{\longrightarrow} d'' \hat{=} d \overset{Regel2,Regel1}{\longrightarrow} d''$
- $d \longrightarrow d$

Fall 2:

Fall 2.1: Verändert die zweite Regel die Dokumente nicht, so bleiben die Elemente nach Anwendung der ersten Regel gleich. Es gilt:

- $d' = d''$
- $d \overset{Regel1,Regel2}{\longrightarrow} d'' \hat{=} d \overset{Regel1}{\longrightarrow} d''$
- $d \longrightarrow d''$

Fall 2.2: In diesem Fall ist die Reihenfolge der Regeln vertauscht. Um die Dokumentinstanzen von Fall 2.1 zu unterscheiden, werden die veränderten Dokumente mit d* bzw. d** bezeichnet. Die zuerst angewandte Regel verändert die Dokumente nicht. Die Adaption der Dokumente erfolgt erst mit Anwendung von Regel 2:

- $d \overset{Regel1}{\longrightarrow} d* \overset{Regel2}{\longrightarrow} d**$

- $d = d*$
- $d \overset{Regel1,Regel2}{\longrightarrow} d**$
- $d \longrightarrow d**$

Die Reihenfolge der Anwendung von Integritätsbedingungen hat Einfluss auf die Transformation der Dokumente und liefert unterschiedliche Ergebnisse. Sowohl bei der Formulierung als auch bei der Umsetzung von Integritätsbedingungen ist auf Auswirkung einer veränderten Reihenfolge zu achten. Dass Auflösen solcher Konflikte ist aufgrund der Probleme, die im nächsten Abschnitt behandelt werden, nicht automatisiert möglich.

Fall 3: Auch wenn verschiedene Teile des Dokumentes verändert werden, muss überprüft werden, ob sich die Regeln gegenseitig beeinflussen. Da eine Konfliktauflösung vor der Adaption der Dokumente im Allgemeinen nicht berechenbar ist, muss eine Validierung während des Adaptionsprozesses stattfinden.

Fall 4: Es ist möglich, mehrere Integritätsbedingungen an das gleiche Element zu stellen. Dabei kann das Element selbst, seine Attributwerte oder seine Sub-Elemente durch die Integritätsbedingung verändert werden. Werden verschiedene Teile des Elementes verändert, so muss geprüft werden ob andere, zuvor erfüllte Integritätsbedingungen dadurch (erneut) verletzt werden. Dies entspricht dann Fall 3. Wird das Element selbst verändert, muss geprüft werden, ob die Reihenfolge der Regeln Einfluss auf die Werte und Struktur des Elementes hat (siehe Fall 2).

6.3.3 Konflikte zwischen Assertions und anderen Integritätsbedingungen

Logische Konsistenzbedingungen können im Widerspruch zu Integritätsbedingungen hinsichtlich Wertebereich, Struktur und referentieller Integrität stehen. In [Tür99] wurde gezeigt, dass das Erkennen und Auflösen solcher Beziehungen im Allgemeinen nicht berechenbar ist. Aus diesem Grund wird lediglich geprüft, ob ein Bestimmtes Element bei der Adaption der XML-Dokumente durch verschiedene Arten von Integritätsbedingungen verändert wird. Dazu werden, nachdem das Dokument durch die erste Transformation verändert wurde, die veränderten Elemente markiert. Anschließend erfolgt die Anwendung der Integritätsbedingungen auf die XML-Dokumente. Dabei wird verglichen, ob bereits veränderte Elemente erneut verändert werden. Ist dies der Fall, so wird der Anwender darüber informiert, dass evtl. Unstimmigkeiten bei der Transformation aufgetreten sind.

6.3.4 Allgemeine Probleme bei der Auflösung

Werden komplexere Integritätsbedingungen formuliert, kann nicht sichergestellt werden, dass für die Erfüllung der Gleichungen immer Lösungen existieren. Ursache hierfür sind mathematische Problem, die im folgenden kurz vorgestellt werden.

10. Hilbert'sches Probleme

„Eine Diophantische Gleichung mit irgend welchen Unbekannten und mit ganzen rationalen Zahlencoefficienten sei vorgelegt: man soll ein Verfahren angeben, nach welchem sich mittelst einer endlichen Anzahl von Operationen entscheiden. läßt, ob die Gleichung in ganzen rationalen Zahlen lösbar ist.“ ([Hil00]).

Das ein Algorithmus zum Lösen solcher Gleichungen nicht existiert wurde 1970 von Yuri Matiyasevich bewiesen ([MD74]). Ein Spezialfall der Diophantischen Gleichungen ist der Große fermatsche Satz, für dessen Gleichungen keine Lösungen existieren.

Großer fermatscher Satz

Pierre de Fermat stellte im 17. Jahrhundert folgende Vermutung auf:
$a^n + b^n = c^n$ mit a,b,c $\in \mathbb{N}$ besitzt für keine natürliche Zahl n>2 eine Lösung.
In [Wil95] erfolgte 1995 der Beweis der Vermutung Fermats.
Wird folgende Integritätsbedingung an ein XML-Dokument formuliert
$count(//a) * count(//a) * count(//a) + count(//b) * count(//b) * count(//b) =$
$count(//c) * count(//c) * count(//c)$
entspricht dies in Potenzschreibweise
$count(//a)^3 + count(//b)^3 = count(//c)^3$.
Da count() stets ganzzahlige Lösungen zurückliefert und die Potenz 3>2 ist, kann die Gleichung nicht erfüllt werden.

Query Containment Problem

Das Query Containment Problem (QCP) ist ein NP-vollständiges Problem, welches in vielen Bereichen der Informatik vorkommt. Dieses Problem tritt z.B. bei der Optimierung von Datenbank-Anfragen auf, wenn beispielsweise die Äquivalenz von Anfragen gezeigt werden soll.

QCP ist wie für Datenbanken folgt definiert ([JP98]): Sei R ein Datenbankschema und I eine beliebige Instanz von R. Weiterhin seien q_1, q_2 Anfragen gegen I und $q_i(I)$, i = 1,2 die von q_i auf I berechneten Tupel. Dann ist
$q_1 \subseteq q_2 \Leftrightarrow q_1(I) \subseteq q_2(I)$.

Für die Anwendung in XML-Schemata, speziell für die Formulierung von Integritätsbedingungen, ergibt sich somit das Problem, dass die von einem Anwender formulierten Assertions sich widersprechen können oder redundant sind.

In [Tür99] werden Algorithmen vorgestellt, die für einige Spezialfälle in Polynomialzeit das QCP entscheiden können. Die durch den XPath-Functions bereitgestellten Funktionalitäten besitzen aber eine weit höhere Mächtigkeit, wodurch diese Algorithmen nicht anwendbar sind. Für die praktische Anwendung bedeutet dies, dass der Anwender für die redundanzfreie Formulierung von Integritätsbedingungen verantwortlich ist. Insbesondere muss darauf geachtet werden, dass keine Formeln formuliert werden, die unter keiner möglichen Belegung erfüllt werden können (Kontradiktion). Ein Beispiel für eine solche Kontradiktion ist in Abbildung 6.3 angegeben. Es wird der paarweise Größentest zwischen den Attributen a, b und c formuliert. Jede einzelne Bedingung ist an sich erfüllbar, aber aus der Transitivität des Operators „lt“ (<) ergibt sich:
$a < b \wedge b < c \rightarrow a < c$.

Da nach Integritätsbedingung 3 gilt: c < a stellt dies einen Widerspruch zu a < c dar. Dadurch ist die gesamte Assertion nicht erfüllbar.

```
<assert test='@a lt @b'>
<assert test='@b lt @c'>
<assert test='@c lt @a'>
```

Abbildung 6.3: Eine nicht erfüllbare Integritätsbedingung

Kapitel 7

Testszenarien

In diesem Kapitel wird beschrieben, wie entwickelten Algorithmen zur Modellierung und Überprüfung der Integritätsbedingungen getestet wurden. Viele dieser Tests verwenden dabei JUnit (junit.org), ein Framework zum automatisierten Testen von Java-Methoden und -Klassen. Für GWT existiert dazu ein spezielles Plugin, welches das Testen von graphischen Oberflächen ermöglicht.

Als Beispiel-Schema wurde dazu *Atom*, eine Beschreibungssprache für News-Einträge, verwendet. Das vollständige Atom-Schema ist in Anhang B abgebildet. In den folgenden Abschnitten wird zur Veranschaulichung nur ein Teil des Schema dargestellt.

Die einzelnen Abschnitte stellen die Testverfahren für die jeweilige Art der Integritätsbedingung vor. Im Anschluss wird das Ergebnis der Tests zusammengefasst und eine Übersicht über die dabei auftretenden Probleme gegeben.

7.1 Test der Integritätsbedingungen

7.1.1 Wertebereich

Die Überprüfung, ob die einzelnen Algorithmen zur Überprüfung des Wertebereiches der einzelnen Datentypen erfolgte mittels JUnit-Tests. Ein solcher Test erzeugt jeweils ein Objekt des Typs DomainValidator, welches mittels der Methode *checkPredefinedType* für eine gegebene Eingabe und einen gegebenen Datentyp überprüft, ob die Eingabe zum Wertebereich gehört. Die Methode assertTrue, welche von JUnit bereitgestellt wird, überprüft anschließend ob das Ergebnis wahr oder falsch ist. Beispiele für solche Tests sind in Abbildung 7.1 abgebildet.

```
@Test
public void testDecimal() {
  DomainValidator v = new DomainValidator();
  assertTrue(v.checkPredefinedType("123.456", new XmlDecimal()));
}

@Test
public void testDecimal2() {
  DomainValidator v = new DomainValidator();
  assertTrue(!v.checkPredefinedType("123.456a", new XmlDecimal()));
}
```

Abbildung 7.1: JUnit-Tests zur Überprüfung des Wertebereichs

Nachdem alle Tests durchgelaufen sind, wird das Ergebnis in der Entwicklungsumgebung Eclipse visualisiert (siehe Abbildung 7.2). Dabei wird für die einzelnen Testfälle angegeben, welche erfolgreich abgeschlossen wurden. Für die fehlerbehafteten Testfälle wird eine kurze Übersicht (Failure Trace) angezeigt, damit der Entwickler feststellen kann, an welche Methoden zum Fehlschlagen des Tests geführt haben.

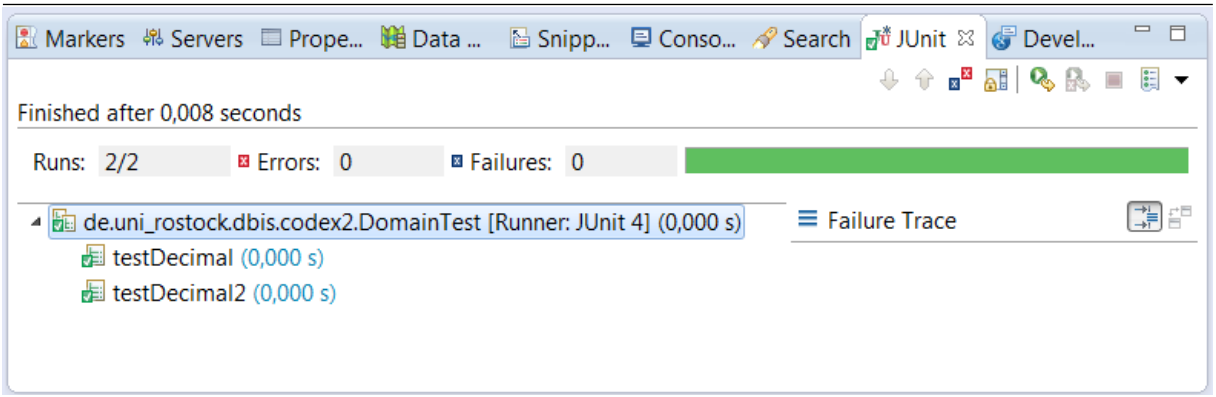


Abbildung 7.2: Ergebnis der JUnit-Tests in Eclipse

Da der Wertebereich für einige Datentypen unendlich groß ist, konnten natürlich nicht alle möglichen Eingabewerte überprüft werden. Für jeden einzelnen Datentyp wurden einige Testfälle entworfen, darunter die Beispielwerte des W3C ([W3C04]) und die Grenzwerte der einzelnen Datentypen (z.B. die Werte -128 und 127 für den Datentyp Byte).

Die Verbindung zwischen den Methoden zur Datentypüberprüfung und den Dialogfenstern in CodeX wurde ebenfalls getestet. Dazu wurden die Eingaben der Testfälle aus den JUnit-Tests in die einzelnen Textboxen der Dialoge übernommen und geprüft, ob das dazugehörige Infofeld im Falle eines ungültigen Werte erscheint. Dabei ergaben sich keine Unterschiede zu den JUnit-Tests.

7.1.2 Strukturelle Integrität

Die Überprüfung der strukturellen Integrität auf Modellebene erfolgte durch einen einfachen Test, bei dem versucht wurde, alle Kombinationen von EMX-Modell-Komponenten paarweise gerichtet zu verbinden. Dabei wurden folgende Eigenschaften geprüft:

- Versucht der Anwender eine unerlaubte Verbindung zwischen Komponenten zu erstellen, so wird keine Verbindungslinie erzeugt. Im Log wird eine Fehlermeldung ausgegeben, welche dem Anwender auf den Fehler hinweist.
- Versucht der Anwender eine erlaubte Verbindung zwischen Komponenten zu erstellen, deren Richtung aber falsch angegeben wurde, so wird die Verbindung mit korrigierter Richtung erzeugt. Im Log wird eine Warnung ausgegeben.
- Wird eine erlaubte, korrekt angegebene Verbindung erstellt, so wird diese Verbindung dauerhaft gespeichert. Im Log wird eine Erfolgsmeldung angezeigt.

Zusätzlich wurde überprüft, ob die internen Variablen *clickCounter, firstClicked* und *secondClicked* nach dem dem Versuch eine Verbindung zu erzeugen zurückgesetzt wurden Gleiches gilt für den vorzeitigen Abbruch, z.B. wenn bereits *firstClicked* gesetzt wurde, dann aber ein anderes Tool ausgewählt wurde. Die strukturelle Integrität zwischen XML-Dokumenten und Schema wird durch Apache-Xerces geprüft. Zum

Testen wurden Beispieldokumente angelegt, die entweder bzgl. des Schemas gültig oder ungültig waren. Die Ausgabe der Überprüfung wurde anschließend in einem Dialog ausgegeben (siehe Kapitel 4.3). Bei den abschließenden Test konnten keine Fehler bezüglich der strukturellen Integrität festgestellt werden.

7.1.3 Referentielle Integrität

Der Test von Primär- und Fremdschlüsseln erfolgte zum einen auf Modell- zum anderen auf Schema-Ebene. Die Überprüfung, ob die Schlüsselbeziehungen auf Dokumentebene den Vorgaben des Schemas entsprechen erfolgte durch den XML-Parser Apache Xerces. Dabei wurden Beispieldokumente gegen ein Schema geparst. Dabei auftretende Verletzungen der Integrität wurden in einem Dialogfenster ausgegeben.

Auf Modell-Ebene wurde überprüft, ob die Formulierung der (Fremd-)Schlüssel ordnungsgemäß funktioniert. Dabei wurde insbesondere darauf geachtet, dass Referenzen nur auf referenzierbare Elemente erstellt werden konnten. Bei der Angabe einer *key/keyref*-Angabe wurde zudem überprüft, dass *field*-Angaben nur für die Subelemente des im *selector* angegebenen Elementes möglich sind.

7.1.4 Logische Konsistenz

In Abbildung 7.3 ist der komplexe Typ *entryType* vor der Schemaevolution dargestellt. Dieser Typ verfügt u.a. über die Elemente *published* und *updated* mit dem Datentyp *dateTime*. Ein dazugehöriges XML-Fragment ist in 7.4 abgebildet. Für die Veranschaulichung des Beispiels wird angenommen, dass ein entry-Eintrag immer erst veröffentlicht (*published*) und später editiert (*updated*) werden kann. Ohne die Einführung zusätzlicher Integritätsbedingungen ist es möglich, dass das updated-Datum zeitlich vor dem des published-Datum liegt, was z.B durch einen Schreibfehler passieren kann. Das Dokument bleibt durch diesen Fehler zwar gegenüber dem Schema gültig, ist im Bezug auf die Bedeutung von *updated* und *published* semantisch falsch.

Durch die Einführung einer logischen Konsistenzbedingung, welche überprüft, ob das *published*-Datum vor der dem *updated*-Datum liegt, kann ein solcher Fehler erkannt und behoben werden. Die Formulierung einer solchen Bedingung könnte wie folgt lauten: *test="published le updated"*. Das durch die Integritätsbedingung angepasste Schema ist in Abbildung 7.5 dargestellt, das angeglichene XML-Fragment ist in 7.6 abgebildet.

Die Überprüfung von gültigen bzw. ungültigen Dokumenten erfolgte erneut durch Apache-Xerces. Die erwarteten unerfüllten logischen Konsistenzbedingungen wurden korrekt in dem Ausgabedialog angezeigt. Auf Modell-Ebene wurde u.a. die Funktionsweise des XPath-Functions-Editor geprüft. Dabei wurde geprüft, ob die Operatoren und Funktionen ordnungsgemäß in die Integritätsbedingung eingefügt wurde. Außerdem wurde die Datenbank-Speicherung und das Mapping in das entsprechende XSD-Fragment überprüft. Für den Übersetzungsprozess von Modell in Schema-Fragment wurden JUnit-Tests erstellt, die überprüfen, ob das erstellte Fragment mit dem erwarteten Ergebnis übereinstimmen.

7.2 Ergebnis

In der fertiggestellten Erweiterung von CodeX befinden sich nach Abschluss der Tests keine erkennbaren Fehler. Während der Entwicklung wurden einige Fehler entdeckt. Eigene Programmierfehler konnten dadurch behoben werden. Fehler, die durch die Verwendung von externen Java-Bibliotheken entstanden wurden entweder durch eigene Implementierungen behoben. Leider sind in der aktuellen Beta-Version von Apache-Xerces-J einige Funktionen noch nicht vollständig umgesetzt worden. Beispielsweise kann nach dem Validierungsprozess zu einer Fehlermeldung nicht die dazugehörige Spalten- bzw. Zeilennummer ausgelesen werden, wodurch die Lokalisierung des Fehlers im XML-Dokument nicht möglich ist. Dies führte dazu, dass einige Funktionen, wie die Markierung der fehlerhaften Elemente im Dokument nicht umgesetzt werden konnten. Wird eine neue Version von Xerces-J herausgebracht, sollte überprüft werden ob diese Funktionalitäten umgesetzt wurden.

```
<xs:complexType name="entryType">
  <xs:annotation>
    <xs:documentation>
      The Atom entry construct is defined in section 4.1.2 of the
      format spec.
    </xs:documentation>
  </xs:annotation>
  <xs:choice maxOccurs="unbounded">
    <xs:element name="author" type="atom:personType" minOccurs="0"
    maxOccurs="unbounded"/>
    <xs:element name="category" type="atom:categoryType" minOccurs="0"
     maxOccurs="unbounded"/>
    <xs:element name="content" type="atom:contentType" minOccurs="0"
    maxOccurs="1"/>
    <xs:element name="contributor" type="atom:personType" minOccurs="0
    " maxOccurs="unbounded"/>
    <xs:element name="id" type="atom:idType" minOccurs="1" maxOccurs="
    1"/>
    <xs:element name="link" type="atom:linkType" minOccurs="0"
    maxOccurs="unbounded"/>
    <xs:element name="published" type="atom:dateTimeType" minOccurs="0
    " maxOccurs="1"/>
    <xs:element name="rights" type="atom:textType" minOccurs="0"
    maxOccurs="1"/>
    <xs:element name="source" type="atom:textType" minOccurs="0"
    maxOccurs="1"/>
    <xs:element name="summary" type="atom:textType" minOccurs="0"
    maxOccurs="1"/>
    <xs:element name="title" type="atom:textType" minOccurs="1"
    maxOccurs="1"/>
    <xs:element name="updated" type="atom:dateTimeType" minOccurs="1"
    maxOccurs="1"/>
    <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded"/>
  </xs:choice>
  <xs:attributeGroup ref="atom:commonAttributes"/>
</xs:complexType>
```

Abbildung 7.3: entryType vor Schemaevolution

```
<entry>
  <id>4711</id>
  <title>Atom-Example-Pre-Evolution</title>
  <published>2013-03-17T20:39:45</published>
  <updated>1999-12-31T23:59:59</updated>
</entry>
```

Abbildung 7.4: entry-Eintrag vor Schemaevolution

```
  <xs:complexType name="entryType">
  <xs:annotation>
    <xs:documentation>
      The Atom entry construct is defined in section 4.1.2 of the
      format spec.
    </xs:documentation>
  </xs:annotation>
  <xs:choice maxOccurs="unbounded">
    <xs:element name="author" type="atom:personType" minOccurs="0"
    maxOccurs="unbounded"/>
    <xs:element name="category" type="atom:categoryType" minOccurs="0"
     maxOccurs="unbounded"/>
    <xs:element name="content" type="atom:contentType" minOccurs="0"
    maxOccurs="1"/>
    <xs:element name="contributor" type="atom:personType" minOccurs="0
    " maxOccurs="unbounded"/>
    <xs:element name="id" type="atom:idType" minOccurs="1" maxOccurs="
    1"/>
    <xs:element name="link" type="atom:linkType" minOccurs="0"
    maxOccurs="unbounded"/>
    <xs:element name="published" type="atom:dateTimeType" minOccurs="0
    " maxOccurs="1"/>
    <xs:element name="rights" type="atom:textType" minOccurs="0"
    maxOccurs="1"/>
    <xs:element name="source" type="atom:textType" minOccurs="0"
    maxOccurs="1"/>
    <xs:element name="summary" type="atom:textType" minOccurs="0"
    maxOccurs="1"/>
    <xs:element name="title" type="atom:textType" minOccurs="1"
    maxOccurs="1"/>
    <xs:element name="updated" type="atom:dateTimeType" minOccurs="1"
    maxOccurs="1"/>
    <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded"/>
  </xs:choice>
  <xs:attributeGroup ref="atom:commonAttributes"/>
  <xs:assert test="published le updated"/>
</xs:complexType>
```

Abbildung 7.5: entryType nach Schemaevolution

```
<entry>
  <id>4711</id>
  <title>Atom-Example-After-Evolution</title>
  <published>2013-03-17T20:39:45</published>
  <updated>2013-03-17T20:39:45</updated>
</entry>
```

Abbildung 7.6: entry-Eintrag nach Schemaevolution

Kapitel 8

Fazit und Ausblick

8.1 Zusammenfassung

Im Rahmen dieser Masterarbeit wurde untersucht, inwieweit XML-Schemata durch die Integration von Integritätsbedingungen auf einem konzeptionellen Modell ergänzt werden können. Zunächst erfolgte eine Betrachtung der Umsetzung der verschiedenen Arten von Integritätsbedingungen in Datenbanken und Forschungsprototypen. Davon ausgehend wurden Konzepte entwickelt, wie diese in einem konzeptionellen Modell umgesetzt werden können. Dabei wurde der Einfluss der Schemaevolution auf das Modell und die dazugehörigen Dokumente untersucht.

Dabei wurden Methoden entworfen, diese Integritätsbedingungen in den Forschungsprototypen CodeX zu integrieren und zu visualisieren. Ausgehend von dem Modell wurden die Integrationsbedingungen in XML-Schema-Komponenten umgewandelt. Bezüglich der Schemaevolution wurden zudem die Auswirkungen auf XML-Dokumente untersucht und dabei ermittelt, welche Kosten bei der Anpassung der Dokumente entstehen. Im Rahmen eines größeren Testszenarios wurde sichergestellt, dass die entwickelte Erweiterung für den Prototyp CodeX auch korrekt arbeitet.

8.2 Ausblick

Kernstück dieser Arbeit war die Integration von Integritätsbedingungen auf dem konzeptionellen Modell eines XML-Schemas. Die Anpassung der XML-Dokumente an die logischen Konsistenzbedingungen wurde in Kapitel 6 behandelt. Dabei ergaben sich einige Probleme, die bereits aus der Datenbanktheorie (z.B. das Update auf Sichten) bekannt sind.

- In der momentanen Umsetzung erfolgt keine Auflösung der Konflikte zwischen verschiedenen Integritätsbedingungen. Die Möglichkeiten der logischen Konsistenzbedingungen sind unter diesem Aspekt noch zu erweitern, sofern sie berechenbar und effizient umsetzbar sind.

- Die Anpassung der Dokument an die Integritätsbedingungen erfolgt momentan noch durch den Nutzer. Der für den Validierungsprozess zuständige Parser Apache Xerces erkennt zwar welche Integritätsbedingung verletzt ist, kann aber keine konkreten Aussagen zu der eigentlichen Ursache treffen. Sobald Xerces über den Beta-Status hinaus ist, sollte dies realisierbar sein.

- Die Integritätsbedingungen können ebenfalls auch für die Formulierung der Bedingungen für alternative Datentypen genutzt werden. Dieses Feature ist ebenfalls neu in XML-Schema 1.1. Durch die Angabe einer bzw. mehrerer Bedingungen können unterschiedliche Datentypen für Attribute bzw. Elementinhalte vergeben werden. Dies gestattet eine flexiblere Gestaltung des XML-Schemas.

- Die Umsetzung der weiteren Features von XML-Schema 1.1 sollte angestrebt werden, damit CodeX den Standard vollständig unterstützt.

Anhang A

Laufendes Beispiel

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://www.w3.org/2005/Atom"
elementFormDefault="qualified" attributeFormDefault="unqualified" xmlns:
atom="http://www.w3.org/2005/Atom" xmlns:xs="http://www.w3.org/2001/
XMLSchema">
  <xs:annotation>
    <xs:documentation>
      This version of the Atom schema is based on version 1.0 of the
      format specifications,
      found here http://www.atomenabled.org/developers/syndication/atom-
      format-spec.php.
    </xs:documentation>
  </xs:annotation>
  <xs:import namespace="http://www.w3.org/XML/1998/namespace"
  schemaLocation="http://www.w3.org/2001/03/xml.xsd"/>
  <xs:annotation>
    <xs:documentation>
      An Atom document may have two root elements, feed and entry, as
      defined in section 2.
    </xs:documentation>
  </xs:annotation>
  <xs:element name="feed" type="atom:feedType"/>
  <xs:element name="entry" type="atom:entryType"/>
  <xs:complexType name="textType" mixed="true">
    <xs:annotation>
      <xs:documentation>
        The Atom text construct is defined in section 3.1 of the format
        spec.
      </xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:any namespace="http://www.w3.org/1999/xhtml" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="type">
      <xs:simpleType>
        <xs:restriction base="xs:token">
```

```
        <xs:enumeration value="text"/>
        <xs:enumeration value="html"/>
        <xs:enumeration value="xhtml"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <xs:attributeGroup ref="atom:commonAttributes"/>
</xs:complexType>
<xs:complexType name="personType">
  <xs:annotation>
    <xs:documentation>
      The Atom person construct is defined in section 3.2 of the
      format spec.
    </xs:documentation>
  </xs:annotation>
  <xs:choice minOccurs="1" maxOccurs="unbounded">
    <xs:element name="name" type="xs:string" minOccurs="1" maxOccurs="
    1"/>
    <xs:element name="uri" type="atom:uriType" minOccurs="0" maxOccurs
    ="1"/>
    <xs:element name="email" type="atom:emailType" minOccurs="0"
    maxOccurs="1"/>
    <xs:any namespace="##other"/>
  </xs:choice>
  <xs:attributeGroup ref="atom:commonAttributes"/>
</xs:complexType>
<xs:simpleType name="emailType">
  <xs:annotation>
    <xs:documentation>
      Schema definition for an email address.
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:normalizedString">
    <xs:pattern value="\w+@(\w+\.)+\w+"/>
  </xs:restriction>
</xs:simpleType>
<xs:complexType name="feedType">
  <xs:annotation>
    <xs:documentation>
      The Atom feed construct is defined in section 4.1.1 of the
      format spec.
    </xs:documentation>
  </xs:annotation>
  <xs:choice minOccurs="3" maxOccurs="unbounded">
    <xs:element name="author" type="atom:personType" minOccurs="0"
    maxOccurs="unbounded"/>
    <xs:element name="category" type="atom:categoryType" minOccurs="0"
     maxOccurs="unbounded"/>
    <xs:element name="contributor" type="atom:personType" minOccurs="0
    " maxOccurs="unbounded"/>
```

```
    <xs:element name="generator" type="atom:generatorType" minOccurs="
    0" maxOccurs="1"/>
    <xs:element name="icon" type="atom:iconType" minOccurs="0"
    maxOccurs="1"/>
    <xs:element name="id" type="atom:idType" minOccurs="1" maxOccurs="
    1"/>
    <xs:element name="link" type="atom:linkType" minOccurs="0"
    maxOccurs="unbounded"/>
    <xs:element name="logo" type="atom:logoType" minOccurs="0"
    maxOccurs="1"/>
    <xs:element name="rights" type="atom:textType" minOccurs="0"
    maxOccurs="1"/>
    <xs:element name="subtitle" type="atom:textType" minOccurs="0"
    maxOccurs="1"/>
    <xs:element name="title" type="atom:textType" minOccurs="1"
    maxOccurs="1"/>
    <xs:element name="updated" type="atom:dateTimeType" minOccurs="1"
    maxOccurs="1"/>
    <xs:element name="entry" type="atom:entryType" minOccurs="0"
    maxOccurs="unbounded"/>
    <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded"/>
  </xs:choice>
  <xs:attributeGroup ref="atom:commonAttributes"/>
</xs:complexType>
<xs:complexType name="entryType">
  <xs:annotation>
    <xs:documentation>
      The Atom entry construct is defined in section 4.1.2 of the
      format spec.
    </xs:documentation>
  </xs:annotation>
  <xs:choice maxOccurs="unbounded">
    <xs:element name="author" type="atom:personType" minOccurs="0"
    maxOccurs="unbounded"/>
    <xs:element name="category" type="atom:categoryType" minOccurs="0"
     maxOccurs="unbounded"/>
    <xs:element name="content" type="atom:contentType" minOccurs="0"
    maxOccurs="1"/>
    <xs:element name="contributor" type="atom:personType" minOccurs="0
    " maxOccurs="unbounded"/>
    <xs:element name="id" type="atom:idType" minOccurs="1" maxOccurs="
    1"/>
    <xs:element name="link" type="atom:linkType" minOccurs="0"
    maxOccurs="unbounded"/>
    <xs:element name="published" type="atom:dateTimeType" minOccurs="0
    " maxOccurs="1"/>
    <xs:element name="rights" type="atom:textType" minOccurs="0"
    maxOccurs="1"/>
    <xs:element name="source" type="atom:textType" minOccurs="0"
    maxOccurs="1"/>
```

```
    <xs:element name="summary" type="atom:textType" minOccurs="0"
    maxOccurs="1"/>
    <xs:element name="title" type="atom:textType" minOccurs="1"
    maxOccurs="1"/>
    <xs:element name="updated" type="atom:dateTimeType" minOccurs="1"
    maxOccurs="1"/>
    <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded"/>
  </xs:choice>
  <xs:attributeGroup ref="atom:commonAttributes"/>
</xs:complexType>
<xs:complexType name="contentType" mixed="true">
  <xs:annotation>
    <xs:documentation>
      The Atom content construct is defined in section 4.1.3 of the
      format spec.
    </xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="type" type="xs:string"/>
  <xs:attribute name="src" type="xs:anyURI"/>
  <xs:attributeGroup ref="atom:commonAttributes"/>
</xs:complexType>
<xs:complexType name="categoryType">
  <xs:annotation>
    <xs:documentation>
      The Atom cagegory construct is defined in section 4.2.2 of the
      format spec.
    </xs:documentation>
  </xs:annotation>
  <xs:attribute name="term" type="xs:string" use="required"/>
  <xs:attribute name="scheme" type="xs:anyURI" use="optional"/>
  <xs:attribute name="label" type="xs:string" use="optional"/>
  <xs:attributeGroup ref="atom:commonAttributes"/>
</xs:complexType>
<xs:complexType name="generatorType">
  <xs:annotation>
    <xs:documentation>
      The Atom generator element is defined in section 4.2.4 of the
      format spec.
    </xs:documentation>
  </xs:annotation>
  <xs:simpleContent>
    <xs:extension base="xs:string">
      <xs:attribute name="uri" use="optional" type="xs:anyURI"/>
      <xs:attribute name="version" use="optional" type="xs:string"/>
      <xs:attributeGroup ref="atom:commonAttributes"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

```
<xs:complexType name="iconType">
  <xs:annotation>
    <xs:documentation>
      The Atom icon construct is defined in section 4.2.5 of the
      format spec.
    </xs:documentation>
  </xs:annotation>
  <xs:simpleContent>
    <xs:extension base="xs:anyURI">
      <xs:attributeGroup ref="atom:commonAttributes"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
<xs:complexType name="idType">
  <xs:annotation>
    <xs:documentation>
      The Atom id construct is defined in section 4.2.6 of the format
      spec.
    </xs:documentation>
  </xs:annotation>
  <xs:simpleContent>
    <xs:extension base="xs:anyURI">
      <xs:attributeGroup ref="atom:commonAttributes"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
<xs:complexType name="linkType" mixed="true">
  <xs:annotation>
    <xs:documentation>
      The Atom link construct is defined in section 3.4 of the format
      spec.
    </xs:documentation>
  </xs:annotation>
  <xs:attribute name="href" use="required" type="xs:anyURI"/>
  <xs:attribute name="rel" type="xs:string" use="optional"/>
  <xs:attribute name="type" use="optional" type="xs:string"/>
  <xs:attribute name="hreflang" use="optional" type="xs:NMTOKEN"/>
  <xs:attribute name="title" use="optional" type="xs:string"/>
  <xs:attribute name="length" use="optional" type="xs:positiveInteger"
  />
  <xs:attributeGroup ref="atom:commonAttributes"/>
</xs:complexType>
<xs:complexType name="logoType">
  <xs:annotation>
    <xs:documentation>
      The Atom logo construct is defined in section 4.2.8 of the
      format spec.
    </xs:documentation>
  </xs:annotation>
  <xs:simpleContent>
    <xs:extension base="xs:anyURI">
```

```
      <xs:attributeGroup ref="atom:commonAttributes"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
<xs:complexType name="sourceType">
  <xs:annotation>
    <xs:documentation>
      The Atom source construct is defined in section 4.2.11 of the
      format spec.
    </xs:documentation>
  </xs:annotation>
  <xs:choice maxOccurs="unbounded">
    <xs:element name="author" type="atom:personType" minOccurs="0"
    maxOccurs="unbounded"/>
    <xs:element name="category" type="atom:categoryType" minOccurs="0"
     maxOccurs="unbounded"/>
    <xs:element name="contributor" type="atom:personType" minOccurs="0
    " maxOccurs="unbounded"/>
    <xs:element name="generator" type="atom:generatorType" minOccurs="
    0" maxOccurs="1"/>
    <xs:element name="icon" type="atom:iconType" minOccurs="0"
    maxOccurs="1"/>
    <xs:element name="id" type="atom:idType" minOccurs="0" maxOccurs="
    1"/>
    <xs:element name="link" type="atom:linkType" minOccurs="0"
    maxOccurs="unbounded"/>
    <xs:element name="logo" type="atom:logoType" minOccurs="0"
    maxOccurs="1"/>
    <xs:element name="rights" type="atom:textType" minOccurs="0"
    maxOccurs="1"/>
    <xs:element name="subtitle" type="atom:textType" minOccurs="0"
    maxOccurs="1"/>
    <xs:element name="title" type="atom:textType" minOccurs="0"
    maxOccurs="1"/>
    <xs:element name="updated" type="atom:dateTimeType" minOccurs="0"
    maxOccurs="1"/>
    <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded"/>
  </xs:choice>
  <xs:attributeGroup ref="atom:commonAttributes"/>
</xs:complexType>
<xs:complexType name="uriType">
  <xs:simpleContent>
    <xs:extension base="xs:anyURI">
      <xs:attributeGroup ref="atom:commonAttributes"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
<xs:complexType name="dateTimeType">
  <xs:simpleContent>
    <xs:extension base="xs:dateTime">
      <xs:attributeGroup ref="atom:commonAttributes"/>
```

```
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
  <xs:attributeGroup name="commonAttributes">
    <xs:attribute ref="xml:base"/>
    <xs:attribute ref="xml:lang"/>
    <xs:anyAttribute namespace="##other"/>
  </xs:attributeGroup>
</xs:schema>
```

Anhang B

Übersicht über die Datentypen in XML-Schema

Die folgenden Informationen existieren auch in [W3C07a], hier werden sie jedoch in kompakterer Form dargestellt. Zunächst werden die einzelnen Datentypen vorgestellt, anschließend wird ihre Verwendung in XML-Schema und ihre Einbindung in CodeX dargelegt.

B.1 Wertebereich der einzelnen Datentypen

AnyType und AnySimpleType Für die beiden XML-Datentypen AnyType und AnySimpleType sind keine Einschränkungen an den Wertebereich angegeben. Aus diesem Grund wird jede Eingabe akzeptiert.

String Nach der Definition von [W3C07a] besteht ein String aus einer endlichen Folge von beliebigen Unicodezeichen, mit Ausnahme der sogenannten Surrogat-Blöcke. Ein Zeichen (engl. char) kann folgende hexadezimalen Werte annehmen:
$Char ::= \#x9|\#xA|\#xD|[\#x20 - \#xD7FF]|[\#xE000 - \#xFFFD]|[\#x10000 - \#x10FFFF]$.
Entsprechend wird der Wert eines String zeichenweise auf seine Gültigkeit geprüft.

NormalizedString Der Wertebereich eines NormalizedStrings ist „the set of strings that do not contain the carriage return (#xD), line feed (#xA) nor tab (#x9) characters.“ (siehe [W3C07a]). Entsprechend der Typhierarchie wird zunächst überprüft, ob es sich beider Eingabe um einen String handelt. Anschließend wird getestet, ob der String eines der drei Steuerzeichen enthält. Ist dies der Fall, so ist die Eingabe fehlerhaft, andernfalls nicht.

Token Ein Token ist nach der Definition von [W3C07a] ein String, der die Anforderungen an einen NormalizedString erfüllt und zusätzlich weder mit einen Leerzeichen beginnt oder endet oder mehr als ein Leerzeichen hintereinander enthält. Die Überprüfung hinsichtlich des Wertebereichs an diese Spezifikation wird mittels der Java-Funktionen startsWith(“ “), endsWith(“ “) und contains(“ “) realisiert.

Language Der Datentyp Language basiert auf den Typ Token. Außerdem muss er noch den regulären Ausdruck „[a-zA-Z]1,8(-[a-zA-Z0-9]1,8)*“ erfüllen, um der Definition des W3C zu entsprechen. Der Algorithmus für die Überprüfung der Eingabe ist in Abbildung B.1 dargestellt.

```
public boolean checkDomainIntegrity(String input)
{
  XmlToken t = new XmlToken();
  if(t.checkDomainIntegrity(input)){
    if(input.matches("[a-zA-Z]{1,8}(-[a-zA-Z0-9]{1,8})*")){
      return true;
    }
  }
  return false;
}
```

Abbildung B.1: Code für Überprüfung des Wertebereiches Language

Name Um den Wertebereich des Datentyps Name zu überprüfen, muss zunächst getestet werden, ob der dem übergeordneten Typ Token entspricht. Außerdem muss die Eingabe folgender Produktionsregel des W3C entsprechen:
Name ::= (Letter | '_' | ':') (NameChar)*
NameChar ::= Letter | Digit | '.' | '-' | '_' | ':' | CombiningChar | Extender
Dies entspricht dem regulären Ausdruck „[a-zA-Z_:][a-zA-Z0-9.:_-]*“. Der Algorithmus zur Überprüfung entspricht, mit Ausnahme des regulären Ausdruckes, dem Algorithmus in Abbildung B.1.

NCName Der Datentyp NCName basiert auf dem Datentyp Name, darf allerdings keinen Doppelpunkt enthalten (non-colonized). Es gelten folgende Produktionsregeln:
NCName ::= (Letter | '_') (NCNameChar)*
NCNameChar ::= Letter | Digit | '.' | '-' | '_' | CombiningChar | Extender.
Daraus ergibt sich folgender regulärer Ausdruck: „[a-zA-Z_][a-zA-Z0-9._-]*“.

ID, IDRef und IDRefs Diese Datentypen werden speziell für die id- und idref-Attribute in XML-Dokumenten verwendet. Die Datentypen ID und IDRef entsprechen in ihrem Wertebereich den Typ NCName. Für ID und IDRef muss lediglich überprüft werden, ob die Eingabe als Wert für NCName gültig ist. Bei IDRefs handelt es sich um ein oder mehrere IDRef's die durch ein Leerzeichen voneinander getrennt sind. Der Algorithmus für IDRefs ist in Abbildung B.2 dargestellt.

Entity und Entities Der Datentyp Entity wird vorrangig in DTDs eingesetzt und spielt in XML Schema nur eine untergeordnete Rolle, da dadurch das Lesen als auch die Auswertung der XML-Dokumente erschwert wird. Die Ursache dafür ist das Fehlen eines Entity-Elementes in XML-Schema. Als Ersatz dafür können Elemente verwendet werden. Ein Beispiel für dieses Vorgehen ist in Abbildung B.3 dargestellt.

Wird der Datentyp trotzdem verwendet, so entspricht sein Wertebereich dem des NCName. Entities sind eine Folge von Entity, die durch Leerzeichen voneinander getrennt sind. Die Überprüfung erfolgt analog zu IDRefs.

NMToken und NMTokens NMToken basiert auf den Datentyp Token. Seine Produktionsregel ist nach [W3C07a] wie folgt definiert:
Nmtoken ::= (NameChar)+
NameChar ::= Letter | Digit | '.' | '-' | '_' | ':' | CombiningChar | Extender
Das entspricht dem regulären Ausdruck [a-zA-Z0-9.:_-]+.

```
public boolean checkDomainIntegrity(String input)
{
  XmlToken t = new XmlToken();
  if(t.checkDomainIntegrity(input)){
    String[] idrefs = input.split(" ");
    if(idrefs.length==0){
      return false;
    }
    for(int i=0; i<idrefs.length; i++){
      XmlIdRef r = new XmlIdRef();
      if(!r.checkDomainIntegrity(idrefs[i])){
        return false;
      }
    }
    return true;
  }
  return false;
}
```

Abbildung B.2: Code für Überprüfung des Wertebereiches IDRefs

```
<xsd:element name="eacute" type="xsd:token" fixed="é"/>

<town>Montr<c:eacute/>al</town>
```

Abbildung B.3: Beispiel für die Verwendung einer Entity in XML-Schema (aus [W3C07a])

Decimal Decimal umfasst alle gebrochenen Zahlen. Es dürfen beliebig viele Vor- und Nachkommastellen angegeben werden. Der Wertebereich ist durch folgenden regulären Ausdruck definiert: (+|-)?([0-9]+(.[0-9]+)?)

Integer Der Datentyp Integer umfasst alle ganzzahligen Dezimal-Werte. Zur Überprüfung des Wertebereiches wird zunächst getestet, ob die Eingabe bezüglich der Oberklasse (Decimal) gültig ist. Ist dies der Fall, so wird weiterhin getestet, ob die Eingabe einen Punkt enthält. Enthält der Wert keinen Punkt und ist die Eingabe eine Dezimalzahl, dann sind die Anforderungen an den Wertebereich des Datentyps Integer erfüllt.

Long, Int, Short und Byte Die Datentypen Long, Int, Short und Byte schränken den Typ Integer hinsichtlich des maximales bzw. minimalen Wertes ein. Die genauen Werte lassen sich der Tabelle B.1 entnehmen. Der Wertebereich wird überprüft, indem zunächst getestet wird, ob es sich um einen Integer-Wert handelt. Anschließend wird überprüft, ob der Wert in das jeweilige Intervall fällt.

Datentyp	Minimaler Wert	Maximaler Wert
Long	-9223372036854775808	9223372036854775807
Int	-2147483648	2147483647
Short	-32768	32767
Byte	-128	127

Tabelle B.1: Einschränkungen des Datentyps Integer

NonPositiveInteger und NegativeInteger In den Wertebereich dieser Datentypen gehören alle ganzzahligen, negativen Integerwerte mit (NonPositiveInterger) und ohne (NegativeInteger) Null. Entsprechend wird geprüft, ob es sich bei der Eingabe um einen Integerwert handelt und ob dieser kleiner bzw. kleiner gleich Null ist.

NonNegativeInteger und PositiveInteger Der Wertebereich dieser Datentypen beinhaltet alle ganzzahligen, positiven Integerwerte mit (NonNegativeInterger) und ohne (PositiveInteger) Null. Zunächst wird getestet, ob der eingegebene Wert ein Integer ist. Anschließend wird geprüft, ob dieser größer bzw. größer gleich Null ist.

UnsignedLong, UnsignedInt, UnsignedShort und UnsignedByte Diese Datentypen stellen eine Einschränkung des Datentyps NonNegativeInteger hinsichtlich des minimalen und maximalen Wertes dar. Die Anzahl der möglichen Werte gleicht denen der vorzeichenbehafteten Datentypen Long, Int, Short und Byte, durch das fehlende Vorzeichen ist der Wertebereich unterschiedlich. Die genauen Werte lassen sich der Tabelle B.2 entnehmen.

Datentyp	Minimaler Wert	Maximaler Wert
UnsignedLong	0	18446744073709551615
UnsignedInt	0	4294967295
UnsignedShort	0	65535
UnsignedByte	0	255

Tabelle B.2: Einschränkungen des Datentyps NonNegativeInteger

```
public boolean checkDomainIntegrity(String input)
{
  if(input.matches("[-]?P([0-9]+Y)?([0-9]+M)?([0-9]+D)?
  (T([0-9]+H)?([0-9]+M)?([0-9]+(\\.[0-9]+)S)?)?")){
    if((input.endsWith("T"))||(input.endsWith("P"))||(input.contains("PT"))
    ){
      return false;
    }
    return true;
  }
  return false;
}
```

Abbildung B.4: Code für Überprüfung des Wertebereiches Duration

```
-?([1-9][0-9]{3,}|0[0-9]{3})
-(0[1-9]|1[0-2])
-(0[1-9]|[12][0-9]|3[01])
T(([01][0-9]|2[0-3]):[0-5][0-9]:[0-5][0-9](\.[0-9]+)?|(24:00:00(\.0+)?))
(Z|(\+|-)((0[0-9]|1[0-3]):[0-5][0-9]|14:00))?
```

Abbildung B.5: Regulärer Ausdruck für dateTime

Duration, YearMonthDuration, DayTimeDuration Das W3C (siehe [W3C07a]) definiert den Datentyp Duration wie folgt: „The lexical representation for duration is the [ISO 8601] extended format PnYn MnDTnH nMnS, where nY represents the number of years, nM the number of months, nD the number of days, 'T' is the date/time separator, nH the number of hours, nM the number of minutes and nS the number of seconds. The number of seconds can include decimal digits to arbitrary precision. [...] An optional preceding minus sign ('-') is allowed, to indicate a negative duration.“ Jede der Zeitkomponenten ist optional, jedoch muss sichergestellt werden, dass

1. mindestens eine Zeitkomponente angegeben wird
2. wenn der Separator 'T' auftaucht, jeweils eine Komponente links und rechts des Separators steht.

Zur Realisierung dieser Anforderung werden Java-String-Operationen verwendet. Der erste Punkt wird dadurch gelöst, dass geprüft wird ob der String mit 'P' endet. Ist dies nicht der Fall, so muss mindestens noch eine Komponente folgen. Um sicherzustellen, dass links und rechts vom Separator 'T' Zeitangaben stehen, wird geprüft, ob der String mit 'T' endet (rechte Seite vorhanden) und kein 'PT' (linke Seite nicht leer) enthält. Durch die Verwendung der Java-Funktionalitäten verringert sich die Komplexität des regulären Ausdruckes, welcher zusammen mit dem kompletten Algorithmus in Abbildung B.4 angegeben ist.

Der Datentyp DayTimeDuration leitet sich aus Duration ab, besitzt aber keine Angaben zum Jahr und Monat. YearMonthDuration hingegen besteht nur aus der Angabe zu Jahr und Monat.

DateTime, DateTimeStamp, Time und Date DateTime besteht aus den Komponenten Jahr, Monat, Tag, Stunde, Minute, Sekunde und einer optionalen Zeitzonenangabe. Der reguläre Ausdruck zur Überprüfung des Wertebereiches ist in Abbildung B.5 angegeben. Bei DateTimeStamp ist die Angabe zur Zeitzone nicht optional.

Time leitet sich aus aus DateTime ab, besitzt jedoch keine Angaben zu Jahr, Monat und Tag. Analog dazu besitzt Date keine Angaben zu Stunde, Minute und Sekunde.

GYearMonth Soll eine Datumsangabe ohne Angabe des genauen Tages erfolgen, so wird der Datentyp GYearMonth benutzt. Dieser verfügt über eine optionale Zeitzonenangabe. Der Wertebereich wird mittels des Patterns -?([1-9][0-9]3,|0[0-9]3)-(0[1-9]|1[0-2])(Z|(+|-)((0[0-9]|1[0-3]):[0-5][0-9]|14:00))? überprüft.

GYear Der Datentyp GYear stellt den Wertebereich für die Angabe eines Jahres (mit optionaler Angabe einer Zeitzone) bereit. Die Überprüfung, ob eine Eingabe in diesen Wertebereich fällt, wird mittels des Patterns -?([1-9][0-9]3,|0[0-9]3)(Z|(+|-)((0[0-9]|1[0-3]):[0-5][0-9]|14:00))? überprüft.

GMonthDay Eine Datumsangabe bestehend aus Monat und Tag wird durch den Datentyp GMonthDay angegeben. Der Wertebereich wird durch das Pattern –(0[1-9]|1[0-2])-(0[1-9]|[12][0-9]|3[01])(Z|(+|-)((0[0-9]|1[0-3]):[0-5][0-9]|14:00))? überprüft, es findet dabei jedoch keine Überprüfung statt, ob für einen bestimmten Monat der 31. Tag erlaubt ist. Gleiches gilt für den Monat Februar. Ein optionaler Zeitzonenangabe ist ebenfalls erlaubt.

GDay Der Tag im Gregorianischen Kalender wird durch folgendes Pattern im Wertebereich festgelegt: —(0[1-9]|[12][0-9]|3[01])(Z|(+|-)((0[0-9]|1[0-3]):[0-5][0-9]|14:00))?. Die Angabe einer Zeitzone ist wie in allen anderen Zeitformaten optional.

GMonth GMonth stellt die Monatsangabe im Gregorianischen Kalender dar. Eine optionale Angabe der Zeitzone ist erlaubt. Der Wertebereich wird durch folgendes Pattern überprüft: –(0[1-9]|1[0-2])(Z|(+|-)((0[0-9]|1[0-3]):[0-5][0-9]|14:00))?.

Boolean Boolean umfasst die Werte true und false sowie deren analog verwendeten Darstellungen 1 und 0. Entspricht die Eingabe keiner dieser Werte, so gehört sie auch nicht in diesen Wertebereich.

Base64Binary und HexBinary Binär codierte Daten können mit den Datentypen Base64Binary und HexBinary dargestellt werden. Die Überprüfung des Wertebereiches erfolgt mittels Pattern. Für HexBinary erfolgt die Darstellung in Tupeln der Form [0-9a-fA-F]2, während für Base64-codierte Daten je nach Länge drei verschiedene Pattern existieren:

- ([A-Za-z0-9+/]4)*[A-Za-z0-9+/]4
- ([A-Za-z0-9+/]4)*[A-Za-z0-9+/]2[AEIMQUYcgkosw048][=]
- ([A-Za-z0-9+/]4)*[A-Za-z0-9+/][AQgw][=][=]

Float und Double Gleitkommazahlen werden mittels der Datentypen Float und Double repräsentiert. Sie bestehen aus einer (evtl. vorzeichenbehafteten) Mantisse und einem (evtl. vorzeichenbehaftetem) Exponenten, die durch ein e (oder auch E) voneinander getrennt sind. Es ist auch möglich, dass der Wert unendlich groß oder (INF, -INF) oder keine Zahl (NaN) ist. Für den Exponenten des Typs Float muss gelten, dass er größer als -149 und kleiner als 104 sein muss. Für Double erhöhen sich diese Werte auf -1075 bzw. 970. Die Überprüfung des Wertebereichs erfolgt mittels einer Kombination aus Pattern und String-Operationen . Der Algorithmus zur Überprüfung des Datentyps Float ist in Abbildung B.6 dargestellt.

```
public boolean checkDomainIntegrity(String input)
{
  if((input.equals("-INF"))||((input.equals("INF"))||(input.equals("NaN")))
  )
    return true;
  else{
    if(input.matches("(([\\+|-]?[0-9]+)|([0-9]*))(\\.[0-9]+)?([eE
    ][\\+|-]?[0-9]+)?")){
      String exponent = null;
      int e = 0;
      if(input.contains("e"))
        exponent= input.split("e")[1];
      if(input.contains("E"))
        exponent = input.split("E")[1];
      if((exponent!=null)&&((exponent.contains("+"))||(exponent.contains("-
      "))))
        e = Integer.parseInt(exponent.substring(1));
      if(exponent!=null)
        if((e>104)||(e<-149))
          return false;
      return true;
    }
  }
  return false;
}
```

Abbildung B.6: Code für Überprüfung des Wertebereiches Float

```
public boolean checkDomainIntegrity(String input){
  // RFC 2396
  try{
    new URI(input);
    return true;
  }
  catch (Exception e){
    return false;
  }
}
```

Abbildung B.7: Code für Überprüfung des Wertebereiches anyURI

anyURI Für die Angabe von Uniform Resource Identifier (URI) wird der Datentyp anyURI verwendet. In der RFC 2396 wird beschrieben, wie eine URI aufgebaut ist. Zur Überprüfung in Java wird ein URI-Objekt mit dem gegebenen Eingabestring erzeugt. Entspricht der String keiner URI, so wird eine Exception ausgelöst (siehe Abildung B.7).

QName und NOTATION Der Datentyp QName stellt XML-qualifizierte Namen dar. Jedes Tupel, welches die Form NCName:NCName hat, gehört in den Wertebereich dieses Datentyps. sollen mehrere QNames in einer Menge angegeben werden, so wird dafür der Datentyp NOTATION verwendet.

B.2 Verwendung der Datentypen in XML-Schema

Dieser Abschnitt gibt einen kurzen Überblick über die Zuordnung von Datentypen zu Attributen die in XML-Schema existieren. Dabei werden für die einzelnen Schemakomponenten die möglichen Attribute aufgelistet. Zugeordnete Datentypen werden durch die Angabe ihres Namens angegeben. Falls für ein Attribut statt eines Datentyps eine Auswahl von Attributwerten angegeben ist, so ist dies in Anführungszeichen („value“) dargestellt. Falls ein default-Wert für ein Attribut existiert, so wird dieser ebenfalls angegeben.

attribute

- default = string
- fixed = string
- form = („qualified“ | „unqualified“)
- id = ID
- name = NCName
- ref = QName
- targetNamespace = anyURI
- type = QName
- use = („optional“ | „prohibited“ | „required“) : „optional“
- inheritable = boolean

element

- abstract = boolean : false
- block = („#all“ | List of (extension | restriction | substitution))
- default = string
- final = („#all“ | List of (extension | restriction))
- fixed = string
- form = (qualified | unqualified)
- id = ID
- maxOccurs = (nonNegativeInteger | „unbounded“) : „1“
- minOccurs = nonNegativeInteger : „1“
- name = NCName
- nillable = boolean : „false“
- ref = QName
- substitutionGroup = List of QName
- targetNamespace = anyURI
- type = QName

complexType

- abstract = boolean : „false“
- block = („#all“ | List of (extension | restriction))
- final = („#all“ | List of (extension | restriction))
- id = ID
- mixed = boolean
- name = NCName
- defaultAttributesApply = boolean : „true“

attributeGroup

- id = ID
- name = NCName
- ref = QName

group

- id = ID
- maxOccurs = (nonNegativeInteger | „unbounded“) : „1“
- minOccurs = nonNegativeInteger : „1“
- name = NCName
- ref = QName

all

- id = ID
- maxOccurs = („0“ | „1“) : „1“
- minOccurs = („0“ | „1“) : „1“

choice

- id = ID
- maxOccurs = (nonNegativeInteger | „unbounded“) : „1“
- minOccurs = nonNegativeInteger : „1“

sequence

- id = ID
- maxOccurs = (nonNegativeInteger | „unbounded“) : „1“
- minOccurs = nonNegativeInteger : „1“

any

- id = ID
- maxOccurs = (nonNegativeInteger | „unbounded“) : „1“
- minOccurs = nonNegativeInteger : „1“
- namespace = ((„##any“ | „##other“) | List of (anyURI | („##targetNamespace“ | „##local“)))
- notNamespace = List of (anyURI | („##targetNamespace“ | „##local“))
- notQName = List of (QName | („##defined“ | „##definedSibling“))
- processContents = („lax“ | „skip“ | „strict“) : „strict“

anyAttribute

- id = ID
- namespace = ((„##any“ | „##other“) | List of (anyURI | („##targetNamespace“ | „##local“)))
- notNamespace = List of (anyURI | („##targetNamespace“ | „##local“))
- notQName = List of (QName | „##defined“)
- processContents = („lax“ | „skip“ | „strict“) : „strict“

unique

- id = ID
- name = NCName
- ref = QName

key

- id = ID
- name = NCName
- ref = QName

keyref

- id = ID
- name = NCName
- ref = QName
- refer = QName

selector

- id = ID
- xpath = a subset of XPath expression
- xpathDefaultNamespace = (anyURI | („##defaultNamespace“ | „##targetNamespace“ | „##local“))

field

- id = ID
- xpath = a subset of XPath expression
- xpathDefaultNamespace = (anyURI | („##defaultNamespace“ | „##targetNamespace“ | „##local“))

XPath-Selector

- Selector = Path ('|' Path)*
- Path = ('.' '//')? Step ('/' Step)*
- Step = '.' | NameTest
- NameTest = QName | '*' | NCName ':*'

XPath-Field

- Path = ('.' '//')? (Step '/')* (Step | '@' NameTest)

alternative

- id = ID
- test = an XPath expression
- type = QName
- xpathDefaultNamespace = (anyURI | („##defaultNamespace“ | „##targetNamespace“ | „##local“))

assertion

- id = ID
- test = an XPath expression
- xpathDefaultNamespace = (anyURI | („##defaultNamespace“ | „##targetNamespace“ | „##local“))

notation

- id = ID
- name = NCName
- public = token
- system = anyURI

annotation

- id = ID

appinfo

- source = anyURI

documentation

- source = anyURI
- xml:lang = language

simpleType

- final = („#all“ | List of (list | union | restriction | extension))
- id = ID
- name = NCName

restriction

- base = QName
- id = ID

list

- id = ID
- itemType = QName

union

- id = ID
- memberTypes = List of QName

schema

- attributeFormDefault = („qualified“ | „unqualified“) : „unqualified“
- blockDefault = („#all“ | List of (extension | restriction | substitution)) : „“
- defaultAttributes = QName
- xpathDefaultNamespace = (anyURI | („##defaultNamespace“ | „##targetNamespace“ | „##local“)) : „##local“
- elementFormDefault = („qualified“ | „unqualified“) : „unqualified“
- finalDefault = („#all“ | List of (extension | restriction | list | union)) : „“
- id = ID
- targetNamespace = anyURI
- version = token
- xml:lang = language

defaultOpenContent

- appliesToEmpty = boolean : „false“
- id = ID
- mode = („interleave“ | „suffix“) : „interleave“

include

- id = ID
- schemaLocation = anyURI

redefine

- id = ID
- schemaLocation = anyURI

override

- id = ID
- schemaLocation = anyURI

import

- id = ID
- namespace = anyURI
- schemaLocation = anyURI

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Ada12] Adams, Drew: *Oracle XML DB Developer's Guide; 11g Release 2 (11.2).* `http://docs.oracle.com/cd/E11882_01/appdev.112/e23094/xdb07evo.htm`, 2012. zuletzt aufgerufen am 21.09.2012.

[AH00] Andreas Heuer, Gunther Saake: *Datenbanken: Konzepte und Sprachen.* mitp-Verlag, Bonn, 2000.

[Alt11] Altova GmbH: *Altova DiffDog.* `http://www.altova.com/de/diffdog/xml-schema-difftool.html`, 2011. zuletzt aufgerufen am 19.09.2012.

[Alt12] Altova GmbH: *Altova XML Spy.* `http://www.altova.com/xmlspy.html`, 2012. zuletzt aufgerufen am 19.09.2012.

[AM10] Ashok Malhotra, Jim Melton, Norman Walsh Michael Kay: *XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition).* `http://www.w3.org/TR/xpath-functions`, 2010. zuletzt aufgerufen am 16.04.2013.

[Bor11a] Boritz, J. Efrim.: *Views on Core Concepts of Information Integrity.* International Journal of Accounting Information Systems, 2011.

[Bor11b] Born, Matthias: *Überblick: W3C XML Schema 1.1.* 2011.

[Bri] Briganti, Domenico.

[Cla12] Clark, James: *The Design of RELAX NG.* `http://www.thaiopensource.com/relaxng/design.html`, 2012. zuletzt aufgerufen am 20.04.2012.

[CN02] Christian Nentwich, Licia Capra, Wolfgang Emmerich Anthony Finkelstein: *xlinkit: A Consistency Checking and Smart Link Generation Service.* ACM Transaction on Internet Technology, 2002.

[Def12] Deffke, Jan: *XML-Schema Evolution: Evolution in der Praxis.* Bachelorarbeit, Universität Rostock, 2012.

[DK01] Diane Kramer, Li Chen, Kajal Claypool Elke A. Rundensteiner Hong Su: *XEM: Managing the Evolution of XML Documents.* Technischer Bericht, Department of Computer Science; Worcester Polytechnic Institute, 2001.

[ED11] Eladio Domínguez, Jorge Lloret, Beatriz Pérez Áurea Rodríguez Ángel L. Rubio María A. Zapata: *Evolution of XML schemas and documents from stereotyped UML class models: A traceable approach.* Technischer Bericht, Universidad de Zaragoza, 2011.

[EL06] Erwin Leonardi, Tran T. Hoai, Sourav S. Bhowmick Sanjay Madria: *DTD-Diff: A Change Detection Algorithm for DTDs.* Technischer Bericht, Nanyang Technological University, Singapore; University of Missouri-Rolla, 2006.

[Fou06] Foundation, The Apache Software: *Apache Xalan-Java Version 2.7.1.* http://xml.apache.org/xalan-j/, 2006. zuletzt aufgerufen am 14.04.2012.

[GG06] Giovanna Guerrini, Marco Mesiti, Daniele Rossi: *XML Schema Evolution.* Technischer Bericht, Università di Genova, Università di Milano, Italy, 2006.

[GG08] Giovanna Guerrini, Marco Mesiti: *X-Evolution: A Comprehensive Approach for XML Schema Evolution.* Technischer Bericht, Università di Genova, Università di Milano, Italy, 2008.

[Goo12] Google: *Google Web Toolkit.* https://developers.google.com/web-toolkit/, 2012. zuletzt aufgerufen am 19.09.2012.

[Gru11] Grunert, Hannes: *XML-Schemaevolution: kategorisierung und Bewertung.* Masterarbeit, Universität Rostock, 2011.

[Gru12] Grunert, Hannes: *Übersetzung von XSEL-Ausdrücken in andere XML-Updatesprachen.* Projektarbeit, Universität Rostock, 2012.

[Hil00] Hilbert, David: *Mathematische Probleme.* Vortrag, gehalten auf dem internationalen Mathematiker-Kongress zu Paris, 1900.

[IBM11] IBM: *IBM DB2 Database for Linux, UNIX and Windows - Informationszentrale.* http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/index.jsp, 2011. zuletzt aufgerufen am 21.09.2012.

[ISO11] ISO/IEC, Rick Jelliffe: *Schematron.* http://www.schematron.com/, 2011. zuletzt aufgerufen am 19.09.2012.

[JC99] James Clark, Steve DeRose: *XML Path Language (XPath) Version 1.0.* http://www.w3.org/TR/xpath/, 1999. zuletzt aufgerufen am 14.04.2012.

[JP98] J.R. Parama, N.R. Brisaboa, H.J. Hernandez: *Containment of Conjunctive Queries with Built-in Predicates with Variables and Constants over Any Ordered Domain.* 1998.

[Klí10] Klímek, Jakub: *Integration and Evolution of XML Data via Common Data Model.* Doktorarbeit, Charles University in Prague, 2010.

[Kle07] Klettke, Meike: *Conceptual XML Schema Evolution - the CoDEX approach for Design and Redesign.* 2007.

[Kra01] Kramer, Diane: *XEM: XML Evolution Management.* Masterarbeit, Worcester Polytechnic Institute, 2001.

[Mas02] Masayasu, Ishikawa. http://www.w3.org/TR/2002/NOTE-xhtml1-schema-20020902/, 2002. zuletzt aufgerufen am 28.01.2013.

[MD74] M. Davis, Y. Matiyasevich, J. Robinson: *Hilbert's tenth problem. Diophantine equations: Positive aspects of a negative solution.* 1974.

[Mei09] Meier, Wolfang M.: *Open Source Native XML Database Documentation.* http://exist.sourceforge.net/documentation.html, 2009. zuletzt aufgerufen am 21.09.2012.

[Men11] Menon, Ramkumar: *XML Schema 1.1 - What you need to know.* https://blogs.oracle.com/rammenon/entry/xml_schema_11_what_you_need_to, 2011. zuletzt aufgerufen am 22.04.2013.

[Mic08] Microsoft: *Verwenden von XML in SQL Server.* http://technet.microsoft.com/dede/library/ms190936(SQL.90).aspx, 2008. zuletzt aufgerufen am 21.09.2012.

[MN11] Malý, Jakub und Martin Necaský: *XML Document Versioning, Revalidation and Constraints.* Technischer Bericht, Università di Genova, Università di Milano, Italy, 2011.

[Mot08] Motschiunigg, Oliver: *Evaluierung von Clio zur Transformation von Metamodellen.* Diplomarbeit, Universität Wien, 2008.

[MyS12] MySQL: *MySQL 5.6 Reference Manual, Functions and Operators, XML Functions.* http://dev.mysql.com/doc/refman/5.6/en/xml-functions.html, 2012. zuletzt aufgerufen am 21.09.2012.

[NK00] Nils Klarlund, Anders Møller, Michael I. Schwartzbach: *DSD: A Schema Language for XML.* Proceedings of the third workshop on Formal methods in software practice, Portland, 2000.

[Nös12] Nösinger, Thomas: *Evolution von XML-Schemata auf konzeptioneller Ebene - Uebersicht: Der CodeX-Ansatz zur Loesung des Gültigkeitsproblems.* 2012.

[PM04] Paolo Marinelli, Claudio Sacerdoti, Coen Fabio Vitali: *SchemaPath: Extending XML Schema for Co-Constraints.* Technischer Bericht, Department of Computer Science, University of Bologna, 2004.

[Sch05] Schmidhauser, Dr. Arno: *Tamino XML-Datenbank.* Technologie memo, 2005.

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

[TN12] Thomas Nösinger, Meike Klettke, Andreas Heuer: *Evolution von XML-Schemata auf konzeptioneller Ebene - Übersicht: Der CodeX-Ansatz zur Lösung des Gültigkeitsproblems.* Ingo Schmitt, Sascha Saretz and Marcel Zierenberg (Hrsg.), Grundlagen von Datenbanken, volume 850 of CEUR Workshop Proceedings, pages 29-34, CEUR-WS.org, 2012.

[TN13] Thomas Nösinger, Meike Klettke Andreas Heuer: *Automatisierte Modelladaptionen durch Evolution - (R)ELaX in the Garden of Eden.* Technischer Bericht, Institut für Informatik, Universität Rostock, 2013.

[Tür99] Türker, Can: *Semantic Integrity Constraints in Federated Database Schemata*, 1999.

[UR12] Universität Rostock, Lehrstuhl für Lehrstuhl für Datenbank-und Informationssysteme: *CodeX -Conceptual design and evolution of XML-Schema.* https://emma.informatik.uni-rostock.de/Trac/CoDex/, 2012. zuletzt aufgerufen am 19.09.2012.

[Vit04] Vitali, Fabio: *The testing page of SchemaPath.* http://tesi.fabio.web.cs.unibo.it/cgi-bin/twiki/bin/view/Tesi/TestingSchemaPath, 2004. zuletzt aufgerufen am 21.04.2013.

[W3C04] W3C: *XML Schema Part 2: Datatypes Second Edition.* http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/, 2004. zuletzt aufgerufen am 02.11.2012.

[W3C07a] W3C: *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures.* http://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/, 2007. zuletzt aufgerufen am 19.09.2012.

[W3C07b] W3C: *XML Schema Constraints vs. XML 1.0 ID Attributes.* http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/#schemaConstraintsVsXML1, 2007. zuletzt aufgerufen am 23.04.2013.

[W3C07c] W3C: *XSL Transformations (XSLT) Version 2.0.* http://www.w3.org/TR/xslt20/, 2007. zuletzt aufgerufen am 19.09.2012.

[W3C12] W3C. http://www.w3.org/TR/2002/REC-xhtml1-20020801/#prohibitions, 2012. zuletzt aufgerufen am 28.01.2013.

[Wil95] Wiles, Andrew: *Modular Elliptic Curves and Fermat's last theorem.* 1995.

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

[Zha09] Zhang, Guogen. https://www.ibm.com/developerworks/community/blogs/purexml/entry/xml_schema_support_in_db2?lang=en, 2009. zuletzt aufgerufen am 20.04.2013.

Thesen

1. Mechanismen zur Integritätssicherung wurden in vielen Datenbanksystemen implementiert. Im XML-Bereich haben sich solche Systeme bisher nicht durchsetzen können.
2. Integritätsbedingungen für XML-Dokumente lassen sich in XML-Schema v.1.1. definieren, werden gegenwärtig aber nur sehr selten verwendet. Durch eine geeignete Toolunterstützung kann ein Nutzer bei der Formulierung der Bedingungen unterstützt werden.
3. Eine Visualisierung verletzter Integritätsbedingungen hilft dem Nutzer dabei, Fehler in seinem Schema schneller aufzufinden und zu beheben. Werden neben der Fehlerstelle zudem Informationen über die Ursache des Fehlers eingeblendet, steigert sich die Produktivität des Nutzers weiter.
4. Durch die Einführung bzw. das Verändern von Integritätsbedingungen unterliegt das XML-Schema einer Evolution. Bisher gültige XML-Dokumente müssen erneut auf ihre Validität hin geprüft und ggf. angepasst werden.
5. Durch die Anpassung der XML-Dokumente können Informationen verloren gehen. Eine Bewertung über eine Metrik, welche die Anpassungskosten der Dokumente abschätzt, zeigt dem Nutzer auf, wie stark der Einfluss der Integritätsbedingungen auf die Daten ist.