XML-Schemaevolution

Änderung eines XML-Schemas mit automatisierter Adaption assoziierter XML-Dokumente

Dissertation

von

Dipl.-Inf. Thomas Nösinger

zur

Erlangung des akademischen Grades

Doktor-Ingenieur (Dr.-Ing.)

der Fakultät für Informatik und Elektrotechnik

der Universität Rostock

urn:nbn:de:gbv:28-diss2016-0094-3

Gutachter:
Prof. Dr. rer. nat. habil. Andreas Heuer (Universität Rostock)
Prof. Dr. Torsten Grust (Universität Tübingen)
Prof. Dr. rer. nat. habil. Gunter Saake (Universität Magdeburg)

Datum der Einreichung: 1. Februar 2016
Datum der Verteidigung: 7. Juli 2016

Kurzfassung

Die eXtensible Markup Language (**XML**) ist ein etabliertes und standardisiertes Hilfsmittel zum Austausch und Speichern strukturierter und semistrukturierter Informationen. Entspricht die Struktur eines XML-Dokuments dem Standard des W3C (World Wide Web Consortium), dann ist ein XML-Dokument wohlgeformt.

XML-Schema, abgekürzt XSD (XML Schema Definition), ist eine Schemasprache des W3C zur Spezifikation von Anforderungen bezüglich der Struktur und des Inhalts von XML-Dokumenten. Ein wohlgeformtes XML-Dokument ist gültig, wenn ein XML-Schema existiert und dessen Anforderungen realisiert sind.

Eine Änderung bzw. Evolution eines XML-Schemas (**XML-Schemaevolution**) kann unterschiedliche Ursachen haben, zum Beispiel die Korrektur von Fehlern, die Beseitigung von Unklarheiten, die Weiterentwicklung von Anwendungen oder im Allgemeinen die Anpassung der darzustellenden Informationen an aktuelle Anforderungen. Es ergibt sich als eine Konsequenz der XML-Schemaevolution die Fragestellung, ob vormals schemakonforme XML-Dokumente weiterhin gültig bezüglich des neuen, veränderten XML-Schemas sind. Dieses **Gültigkeitsproblem** und dessen Lösung sind Bestandteile der vorliegenden Dissertation.

Die grundlegende These der Dissertation ist, dass durch die Erfassung und Charakterisierung der Änderungen am XML-Schema die zur Wahrung oder Wiederherstellung der Gültigkeit notwendigen Adaptionen der assoziierten XML-Dokumente automatisiert hergeleitet werden können. Die nachfolgenden Ansätze und Mechanismen wurden zur Lösung des Gültigkeitsproblems entwickelt und angewendet.

Das konzeptuelle Modell **EMX** (Entity Model for XML-Schema) wird als Abstraktion von XML-Schema vorgestellt. Es entsteht eine Drei-Ebenen-Architektur mit Modell-, Schema- und Dokumentebene. Aufgrund der eindeutigen Korrespondenz werden Schemaänderungen stellvertretend auf dem EMX durchgeführt.

Änderungen werden durch die domainspezifische Transformationssprache **ELaX** (Evolution Language for XML-Schema) beschrieben und geloggt. Die Optimierung des Logs wird durch den regelbasierten Algorithmus **ROfEL** (Rule-based Optimizer for ELaX) vollzogen, indem unnötige, redundante und ungültige ELaX-Operationen erkannt und beseitigt werden. Das Ziel ist die Minimierung der Operationsanzahl für die anschließende, automatisierte Herleitung der Adaptionen.

Eine **Klassifikation** der Operationen ist die Grundlage für die Analyse der Auswirkungen von Schemaänderungen auf die Dokumentebene. Ist eine Dokumentanpassung notwendig bzw. aufgrund der mehrdeutigen Korrespondenz zwischen Schema- und Dokumentebene möglich, dann werden Transformationsschritte zur Adaption erzeugt. Die Transformation wird auf dem **DOM** (Document Object Model) der mit dem XML-Schema assoziierten XML-Dokumente durchgeführt.

Der Forschungsprototyp **CodeX** (Conceptual design and evolution of XML schemas) dient als Demonstrator, mit welchem die vorgestellten Ansätze und Mechanismen der vorliegenden Dissertation evaluiert werden können.

Abstract

The eXtensible Markup Language (**XML**) is a well-established and standardized format for exchanging and storing structured and semi-structured information. If the structure of an XML document complies with the standard of the W3C (World Wide Web Consortium), the XML document can be seen as well-defined.

The **XML Schema**, abbreviated XSD (XML Schema Definition), is one schema language of the W3C for specifying requirements for the structure and content of XML documents. An XML document is called valid, if it fulfills all restrictions and conditions of an associated XML Schema.

The modification or evolution of an XML Schema could have different reasons, for example the error correction, the clearance of obscurities, the further development of applications or in general the change of requirements for exchanged information. The resulting problem of modifying an XML Schema is that existing XML documents, which were valid against the former XML Schema, could consequently lose their validity and have to be adapted as well (**co-evolution**). The above mentioned **validity problem** and its solution are the main topics of this dissertation. The following approaches and mechanisms were developed and used for the solution of the validity problem.

The conceptual model **EMX** (Entity Model for XML-Schema) is a simplified representation of an XML Schema. The resulting three-layer architecture consists of a model, a schema and a document layer. Since a unique mapping between EMX and XSD exists, modifications are applied on the conceptual model.

Modifications are formally described by the domain-specific transformation language **ELaX** (Evolution Language for XML-Schema). The applied operations are logged. The ruled-based algorithm **ROfEL** (Rule-based Optimizer for ELaX) then reduces the number of logged ELaX operations by identifying and removing unnecessary, redundant and invalid operations. This reduction is an essential prerequisite for the following automatic derivation of transformation steps.

The **classification** of operations is the basis of an impact analysis of schema modifications to the document layer. If an adaption is necessary, or it's possible because of the ambiguous mapping of the schema and the document layer, XML document transformation steps are automatically derived. These steps are used for the adaption of the **DOM** (Document Object Model) of XML documents which are associated with the modified XML Schema.

The research prototype **CodeX** (Conceptual design and evolution of XML schemas) serves as demonstrator, which can be used for the evaluation of the above mentioned approaches and mechanisms presented in this dissertation.

Inhaltsverzeichnis

1. Einleitung

Die strukturierte Speicherung, Analyse und Darstellung von Informationen in Zeiten immer stärker anwachsender Datenmengen stellt eine nicht zu unterschätzende Thematik für zukünftige Entwicklungen dar. Die hohe Verfügbarkeit von teilweise un-/strukturiertem Wissen in heterogenen Quellen, sowie der effiziente Zugriff und Austausch von Informationen zum Zwecke der Verarbeitung, machen die Gesamtproblematik noch komplexer.

Beispiele in diesem Szenario sind das *Internet der Dinge* [MF10] mit dessen technologischen Herausforderungen an die Interoperabilität und das Datenvolumen, *Online Social Networks* [Hei10] mit unterschiedlichsten, nutzergenerierten Inhalten, sowie ganz allgemein *Big Data* [KTGH13] mit dem steigenden Volumen, der Vielzahl von Daten und deren Zuverlässigkeit.

Eine Möglichkeit zum allgemeinen Umgang mit heterogenen Daten sind "standardisierte und preiswerte Massentechnologien [..] der Web- und Internettechnologie" [MF10]. Die offenen Standards des World Wide Web Consortium (W3C) [W3C15a] sind diesbezüglich ein wichtiges Werkzeug und Hilfsmittel, allen voran die diversen XML-Technologien [W3C15b] zum effizienten Austausch, zur formalen Beschreibung, zur Transformation oder zur Anfrage von Informationen.

Die eXtensible Markup Language (**XML**) [BPSM+08] ist ein solches etabliertes und standardisiertes Hilfsmittel zum Austausch und Speichern strukturierter und semistrukturierter Daten bzw. Informationen. XML ist eine selbstbeschreibende, textbasierte Auszeichnungssprache (Markup-Sprache), "das heißt, Daten und Informationen über die Bedeutung der Daten (also Strukturinformationen) treten gemeinsam in einem Dokument auf" [KM03]. Ein XML-Dokument wird als wohlgeformt bezeichnet, wenn die im Standard [BPSM+08] definierten Regeln eingehalten werden. Dazu zählen unter anderem die Forderung korrekt geschachtelter, möglicherweise leerer Elementtags, das Vorhandensein eines Wurzelelements, sowie Einschränkungen bezüglich erlaubter Bezeichner von Elementen und Attributen.

Das XML-Beispiel 1.1 stellt ein wohlgeformtes, abstraktes XML-Dokument dar.[1] Es existiert ein Wurzelelement (*root*) mit weitergehenden Informationen über einen Namensraum (*xmlns*), einen Schemastandort, weitere Attribute (*a1*), Kinderelemente (*e1*) und zugeordnete Werte (*0*).

Die Struktur oder der Aufbau von XML-Dokumenten kann durch ein XML Schema beschrieben werden, wobei die grundlegenden die Document Type Definition

[1] Abstrakte Beispiele werden genutzt, um Diskussionen über die Relevanz beschriebener Änderungen in der Realität zu vermeiden und eine möglichst große Vielfalt von Konzepten darstellen zu können.

```
<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="example.xsd"
      a1="0">
    <e1>0</e1>
    <e1>0</e1>
</root>
```

XML-Beispiel 1.1: Wohlgeformtes XML-Dokument

(DTD) [BPSM+08], Relax-NG [CM01, JTC08], Schematron [JTC06] und XML Schema Definition (XSD) [FW04] sind. Die primären, vom W3C definierten XML Schema sind die DTD und XSD [W3C15c], wobei XSD (nachfolgend wie in [BL01] als **XML-Schema** bezeichnet) als Nachfolger[2] der DTD angesehen wird und wesentlich umfangreichere Darstellungsmöglichkeiten bietet. Die DTD ist "stark auf dokumentenorientierte Bedürfnisse zugeschnitten [..] Datentypaspekte spielen eine untergeordnete Rolle [..]" [LS04]. "XML-Schema behält die prinzipielle Mächtigkeit einer DTD zur Definition von Inhaltsmodellen bei und ergänzt diese um eine reichhaltige Möglichkeit, Datentypen zu definieren." [Lau05] "Ein grundlegendes Konzept von XML Schema (XML-Schema, Anm. d. Autors) ist, Typen zu definieren und dann über Deklarationen Namen zu Typen zuzuordnen, um auf diese Weise Beschränkungen für das Auftreten von Elementen und Attributen in entsprechenden XML-Dokumenten zu spezifizieren." [Sch03]

Ein wohlgeformtes XML-Dokument wird als gültig oder valide bezeichnet, wenn ein XML Schema existiert und dessen Einschränkungen realisiert sind. Das XML-Dokument des XML-Beispiels 1.1 ist valide zum XML-Schema 1.2.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="root" type="roottype"/>
    <xs:element name="e1" type="xs:decimal"/>
    <xs:element name="e2" type="xs:string"/>
    <xs:attribute name="a1" type="xs:decimal"/>
    <xs:attribute name="a2" type="xs:string"/>
    <xs:complexType name="roottype">
        <xs:sequence minOccurs="1" maxOccurs="2">
            <xs:element ref="e1" minOccurs="1" maxOccurs="2"/>
            <xs:element ref="e2" minOccurs="0" maxOccurs="2"/>
        </xs:sequence>
        <xs:attribute ref="a1" use="required"/>
        <xs:attribute ref="a2" use="optional"/>
    </xs:complexType>
</xs:schema>
```

XML-Beispiel 1.2: XML-Schema des XML-Dokuments 1.1

Es existieren ein Schemaelement (*xs:schema*), verschiedene Element- (*root*, *e1*, *e2*) und Attributdeklarationen (*a1*, *a2*), sowie eine Typdefinition eines komplexen Typs (*roottype*). Des Weiteren werden explizit Anforderungen an die Typen

[2] oder auch Erweiterung: "[..] XML Schemas can be seen as an extension of DTDs [..]" [BNdB04]

(*type='..'*), an die Reihenfolge (*xs:sequence*) und die Häufigkeit des Auftretens von Elementreferenzen (*minOccurs='..'*, *maxOccurs='..'*) und Attributreferenzen (*use='..'*) definiert.[3] Das XML-Dokument des XML-Beispiels 1.1 erfüllt alle gestellten Einschränkungen und beinhaltet alle notwendigen Strukturen, es ist somit gültig bezüglich XML-Schema 1.2.

Das eben vorgestellte XML-Schema wird nun verändert, es wird evolutioniert (**XML-Schemaevolution**). Änderungen können unterschiedliche Ursachen haben, zum Beispiel die Korrektur von Fehlern, die Beseitigung von Unklarheiten, die Weiterentwicklung von Anwendungen oder im Allgemeinen die Anpassung der darzustellenden Informationen an aktuelle Anforderungen. Als Konsequenz entsteht das nachfolgende XML-Schema im XML-Beispiel 1.3.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="root" type="roottype"/>
    <xs:element name="e1" type="xs:decimal"/>
    <xs:element name="e2" type="xs:string"/>
    <xs:attribute name="a1" type="xs:decimal"/>
    <xs:attribute name="a2" type="xs:string"/>
    <xs:complexType name="roottype">
        <xs:choice minOccurs="2" maxOccurs="2">
            <xs:element ref="e1" minOccurs="2" maxOccurs="2"/>
            <xs:element ref="e2" minOccurs="0" maxOccurs="2"/>
        </xs:choice>
        <xs:attribute ref="a1" use="optional"/>
        <xs:attribute ref="a2" use="prohibited"/>
    </xs:complexType>
</xs:schema>
```

XML-Beispiel 1.3: Verändertes XML-Schema 1.2

Das XML-Schema ist eine Kopie des XML-Beispiels 1.2, allerdings wurden einige, hier rötlich hervorgehobene Details angepasst. Die Anpassungen beziehen sich auf das Inhaltsmodell (*xs:choice*), die minimale Anzahl der Durchläufe der Auswahl und der Elementreferenz *e1* (jeweils *minOccurs='2'*), sowie Einschränkungen des Auftritts der Attributreferenzen (*use='optional'* und *use='prohibited'*).

Es ergeben sich als Folge der XML-Schemaevolution Probleme. Allen voran muss das **Gültigkeitsproblem** thematisiert werden, das heißt ob ein vormals schemakonformes XML-Dokument ebenso gültig bezüglich des neuen, veränderten XML-Schemas ist. Das XML-Dokument des XML-Beispiels 1.1 ist weiterhin gültig.

Des Weiteren ergibt sich die Fragestellung, ob aus den Änderungen am XML-Schema Rückschlüsse auf die Gültigkeit gezogen werden können. Zum Beispiel, ob die Änderungen allgemein *instanzverändernde*, *instanzerhaltende*, *instanzerweiternde* oder *instanzreduzierende* Operationen sind. Als letztes sollte die Gültigkeit, in dem Fall das diese nicht mehr garantiert werden kann, wieder hergestellt, zu mindestens aber analysiert und erkannt werden.

[3] Detaillierte Erläuterungen über XML-Schema im Allgemeinen folgen im Kapitel 2.1 (XML-Schema).

```
<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="example.xsd"
      a1="0"
      a2="string">
    <e1>0</e1>
    <e2>string</e2>
    <e1>0</e1>
</root>
```

XML-Beispiel 1.4: Nach Anpassung des XML-Schemas 1.2 ungültiges XML-Dokument

Die hohe Komplexität der Gesamtproblematik, welche durch die Einfachheit der Beantwortung bezüglich des XML-Dokuments des XML-Beispiels 1.1 geschlussfolgert werden könnte, kann mit einem minimal veränderten XML-Dokument illustriert werden. Das XML-Dokument des XML-Beispiels 1.4 wurde im Vergleich zu XML-Beispiel 1.1 an den rötlich hervorgehobenen Positionen verändert. Es wurde das Attribut *a2* hinzugefügt, das Element *e2* ergänzt sowie ein Element *e1* entfernt (visualisiert mittels durchgestrichener Komponente). Das XML-Dokument ist gültig bezüglich des XML-Schemas 1.2, allerdings ist es ungültig bezüglich des veränderten XML-Schemas 1.3.

Die angesprochene Komplexität ergibt sich daraus, dass einerseits XML-Schema eine hoch komplexe Spezifikation ist, die durch die erlaubte Optionalität (*use='optional'* und *minOccurs='0'*) und Flexibilität (z.B. *choice*) unterschiedlichste, gültige XML-Dokumente ermöglicht.

Während zum Beispiel die Anpassung der Attributreferenz *a1* keine gültigkeitsverletzende Operation ist, erfordert die Attributreferenz *a2* eine Löschung des entsprechenden Attributs im XML-Dokument (insofern vorhanden).

Andererseits sind die Änderungen kontextabhängig, das heißt eine Änderung kann nicht nur Auswirkungen auf die entsprechenden Komponenten im XML-Dokument haben (z.B. ein Attribut *a2* wird gelöscht), sondern ebenso abhängig vom Umfeld (z.B. *e1* und *e2* sind jeweils voneinander abhängig, s.u.) und weitergehenden Randbedingungen sein (z.B. das Wurzelelement ist *root*).

Das Ändern des Inhaltsmodells von einer Reihenfolge (*sequence*) auf eine Auswahl (*choice*) ist nur dann nicht gültigkeitsverletzend, wenn die Elementreferenzen *e1* und *e2* jeweils maximal zweimal auftreten (beide *maxOccurs='2'*). Pro Durchlauf des Inhaltsmodells würde genau eine der Referenzen ausgewählt werden. Tritt aber *e1* mindestens dreimal auf, und *e2* ist ebenso gegeben, dann ist das XML-Dokument nicht mehr gültig. Es ist des Weiteren denkbar, dass die Elementreferenzen abwechselnd im XML-Dokument auftreten (z.B. durch das Ignorieren der Streichung von *e1* im XML-Beispiel 1.4). Somit wäre eine Sortierung notwendig und zielführend, insofern die maximale Häufigkeit des Inhaltsmodells dies im Vergleich zu den Häufigkeiten der Elementreferenzen zulässt.

Darüber hinaus ergeben sich Folgeprobleme. Zum Beispiel muss das XML-Do-

kument des XML-Beispiels 1.4 erweitert werden, es fehlt ein Element *e1*. Es muss somit Wissen generiert werden, da der Datentyp *xs:decimal* keine leeren Elemente ermöglicht, zeitgleich aber die Deklaration von *e1* keine zusätzlichen Informationen beinhaltet (z.B. Defaultwerte). Nullwerte, die aus dem Bereich der Datenbanken bekannt sind, existieren nicht im XML-Schema. Alternativ könnte das vorhandene Element *e1* gelöscht und *e2* behalten werden, sodass zu mindestens dieser gültigkeitsverletzende Aspekt gelöst wäre. Löschungen bedeuten allerdings immer auch den Verlust von Informationen, was in diesem Beispiel zur Wiederherstellung der Gültigkeit kaum vermeidbar ist (Attribut *a2* und u.U. Element *e1*).

1.1. Problemstellung

Ein Überblick über die Problemstellung der **XML-Schemaevolution** ist in Abbildung 1.1 dargestellt. Unter der Annahme, dass ein XML-Schema (**XSD**) mit

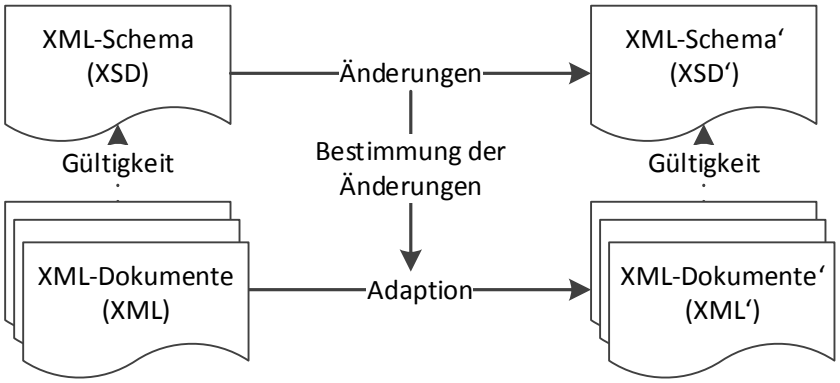


Abbildung 1.1.: Überblick der XML-Schemaevolution

wohlgeformten, gültigen XML-Dokumenten (**XML**) existiert, werden **Änderungen** an dem Ausgangsschema durchgeführt. Es entsteht als Resultat ein verändertes Zielschema (**XSD'**), sodass die Gültigkeit der ehemals schemakonformen XML-Dokumente nicht mehr zwingender Weise gewährleistet werden kann.

Es ergibt sich die Fragestellung, ob durch eine Charakterisierung und Erfassung (d.h. **Bestimmung**) der durchgeführten Änderungen am XML-Schema, die zur Wiederherstellung der Gültigkeit der XML-Dokumente notwendigen **Adaptionen** automatisch hergeleitet werden können.

Somit würde eine Möglichkeit existieren, sodass durch die Änderung eines XML-Schemas die damit assoziierten XML-Dokumente ebenfalls angepasst werden können. Die Gültigkeit der transformierten Instanzen (**XML'**) bezüglich des veränderten XML-Schemas (XSD') könnte gewährleistet werden, dies würde der Lösung des **Gültigkeitsproblems** im dargestellten Szenario entsprechen.

1.1.1. Zielsetzung der Arbeit

In der vorliegenden Arbeit wird die obige Problemstellung gelöst, indem *Änderungen* am XML-Schema bestimmt und zur *Adaption* der XML-Dokumente genutzt werden. Der Anspruch ist die XML-Schemaevolution als Ganzes zu behandeln, wodurch folgende Zielsetzungen definiert werden:

- Spezifikation und Erfassung von Änderungen, die ein Nutzer an einem XML-Schema vornimmt.
- Analyse, Optimierung und Bereinigung der erfassten Änderungen, sowie die weitestgehend automatische Erstellung von daraus resultierenden Transformationsschritten zur Adaption der XML-Dokumente.
- Unterstützung von Nicht-Experten[4] bei der hochkomplexen, fehleranfälligen Evolution durch ein geeignetes Tool und sinnvolle Abstraktionen.

1.1.2. Schwerpunkte der Arbeit

Es sind aus der Problemstellung drei Schwerpunkte hergeleitet worden, welche einen direkten Einfluss auf den Aufbau der Arbeit hatten. Dazu zählen:

- *Änderungen* (Kapitel 4 - Lösungsansatz)
  - Konzeptuelle Modellierung von XML-Schema
  - Verwaltung und Speicherung von Modellen
- *Bestimmung* (Kapitel 5 - Transformationssprache)
  - Spezifikation/Umsetzung von Änderungsoperationen
  - Definition einer Updatesprache und deren Optimierung
  - Logging der Nutzeraktion und deren Auswertung
- *Adaptionen* (Kapitel 6 - Adaption der Instanzen)
  - Automatisierte Erzeugung von Transformationsschritten zur Wahrung und/oder Wiederherstellung der Gültigkeit einer Datenbasis

Die Schwerpunkte werden hauptsächlich in den angegebenen Kapiteln thematisiert und behandeln die in der Problemstellung spezifizierten Fragestellungen. Es existieren allerdings auch implizite Abhängigkeiten zwischen den Schwerpunkten, sodass eine explizite Trennung, welche mit der obigen Auflistung suggeriert wird, nicht möglich ist. Die Erzeugung der Transformationsschritte ist zum Beispiel nur durch eine vorherige Bestimmung der angewendeten Änderungen möglich.

[4] Dies sind Anwender, die ein gewisses Grundwissen der notwendigen XML-Technologien besitzen, ohne dass jeder Aspekt in Tiefe bekannt sein muss. Eine Affinität zur Informatik wird vorausgesetzt.

Die ersten beiden Zielsetzungen werden durch die Schwerpunkte vollständig behandelt. In der dritten Zielsetzung wird ein geeignetes Tool gefordert, welches die unterschiedlichen Aspekte der obigen Schwerpunkte umsetzt. In Kapitel 7 wird ein entsprechender Prototyp beschrieben. Somit existieren analog zu den Schwerpunkten auch Abhängigkeiten zwischen den unterschiedlichen Zielsetzungen.

Sowohl die Fragestellungen der Problemstellung, als auch die Zielsetzungen und Schwerpunkte können folgerichtig nur als Ganzes behandelt werden. Dies erfolgt in der vorliegenden Arbeit, welche den nachfolgenden allgemeinen Aufbau hat.

1.2. Aufbau der Arbeit

Es werden Grundlagen bezüglich verwendeter Techniken in **Kapitel 2** überblicksartig erläutert, bevor klassische und aktuelle Ansätze der Evolution von Schemata allgemein, sowie speziell von XML-Schema in **Kapitel 3** vorgestellt werden.

Die darauf folgenden Kapitel orientieren sich an der obigen Problemstellung. Es sollen Änderungen an einem XML-Schema spezifiziert und erfasst werden, welche ein Anwender an diesem vornimmt. Dafür müssen Möglichkeiten der Abstraktion gefunden werden, um die Komplexität von XML-Schema handhaben zu können. Ein konzeptuelles Modell wird in **Kapitel 4** vorgestellt.

Des Weiteren ist eine Erfassung der Änderungen notwendig, um diese im Anschluss analysieren, kontextabhängig optimieren, sowie bei Bedarf bereinigen zu können. Ein notwendiger Zwischenschritt hierfür ist eine standardisierte Formulierung sowie Speicherung der Änderungen. Eine domainspezifische Transformationssprache und deren Optimierung wird in **Kapitel 5** erläutert. Die vollständige Übersicht der Transformationssprache ist in **Anhang B** enthalten.

Erfasste Änderungen werden im Anschluss zur automatisierten Erzeugung von Transformationsschritten zur Wahrung und/oder Wiederherstellung der Gültigkeit einer Datenbasis verwendet. Dafür ist eine detaillierte Betrachtung der Änderungen bezüglich deren Einfluss auf die Adaption von Instanzen notwendig. Die Klassifikation der Änderungen und deren Einfluss auf die Instanzen sind im **Kapitel 6** beschrieben. Des Weiteren wird die Adaption der Instanzen unter Anwendung der erzeugten Transformationsschritte in einem ausführlichen Beispiel erläutert.

Die obigen Erkenntnisse und Ansätze wurden in einem eigens entwickelten Forschungsprototypen weitestgehend umgesetzt und stehen somit Anwendern zur Verfügung. Der Prototyp und dessen Möglichkeiten werden in **Kapitel 7** präsentiert, zusätzliche Informationen sind in **Anhang C** beschrieben.

Die Arbeit schließt mit einer Schlussbetrachtung in **Kapitel 8** ab. Dabei wird die vorliegende Arbeit auf Basis der definierten Problemstellung analysiert und bewertet, bevor auf zukünftige, sinnvolle Erweiterungen eingegangen wird.

In **Anhang A** sind Übersichten und Abbildungen aus den obigen Kapiteln ergänzt worden, welche zusätzliche Informationen bzw. nur geringfügige Anpassungen im Vergleich zum entsprechenden Kapitel enthalten.

Hinweise zu Schriftschnitten, Fußnoten und Zitaten

In der vorliegenden Arbeit werden unterschiedliche **Schriftschnitte** verwendet. Ein Begriff wird **fett** hervorgehoben, wenn dieser erstmalig definiert und für die Gesamtarbeit notwendig ist. Dieser Begriff wird bei der nachfolgenden Verwendung gegebenenfalls *kursiv* markiert.

Wird ein Akronym eingeführt, dann wird dessen Langform in Klammern ergänzt, wobei die Buchstaben unterstrichen werden. Akronyme werden mehrfach in der Arbeit wiederholt, zu mindestens aber bei deren erstmaligen Einsatz in einem neuen Kapitel. In diesem Fall werden Akronyme ebenso kursiv hervorgehoben.

In der Arbeit werden unterschiedliche Abbildungen integriert und im Anschluss detailliert beschrieben. Sind Begriffe aus der Abbildung nachfolgend im Text übernommen worden, dann werden diese ebenso kursiv dargestellt. Als Ausnahme gelten die für die Gesamtarbeit notwendigen Begriffe, welche nach der obigen Regel fett statt kursiv hervorgehoben werden (z.B. Begriffe der Abbildung 1.1).

Es werden in der vorliegenden Arbeit lokale **Fußnoten** auf den Seiten ergänzt, insofern zusätzliche Informationen gegeben werden. Dies können sowohl der Verweis auf ein anderes Kapitel sein ("siehe auch: Kapitel [..]"), als auch die Markierung einer zentralen These ("siehe auch: These [..]") oder allgemeine Hinweise zur Bedienung ("Hinweis: [..]"). Die letzte Art der Fußnoten wird im Kapitel 7 angewendet und fungiert dort als best practice für den Umgang mit dem Prototypen.

In Kapitel 3 enthalten die Fußnoten zusätzlich die wörtlichen, zumeist englischen **Zitate** aus den angegebenen Quellen. Diese wurden ausgelagert, damit der Lesefluss nicht zu stark unterbrochen wird. Die inhaltlichen Übersetzungen sind im Text selbst enthalten und als Zitate in Hochkommas markiert. Auslassungen in wörtlichen Zitaten werden durch [..] gekennzeichnet.

Wird eine grobe Idee aus einer Quelle entnommen, ohne dass dies ein wörtliches Zitat rechtfertigt, dann wird die Quelle am Anfang oder Ende des Satzes ohne Hochkommas im Text integriert. Der Umgang mit Eigenzitaten aus erfolgreich veröffentlichten Arbeiten wird zu Beginn der entsprechenden Kapitel thematisiert, in welche Textbausteine übernommen worden sind.

Nach der Einleitung und Motivation, sowie der Problemstellung mit der Zielsetzung und den Schwerpunkten der Arbeit, werden im nächsten Kapitel grundlegende XML-Technologien zum verbesserten Verständnis überblicksartig präsentiert.

2. Grundlagen

In diesem Kapitel werden die für die vorliegende Arbeit notwendigen Grundlagen überblicksartig erläutert. Dies bezieht sich auf das XML-Schema als Ausgangspunkt der thematisierten XML-Schemaevolution in **Abschnitt 2.1**, auf XPath als Adressierungssprache zur Ermittlung von XML-Fragmenten in **Abschnitt 2.2**, sowie einen kurzen Vergleich der Evolution von XML-Schema gegenüber der Versionierung von diesen in **Abschnitt 2.3**.

2.1. XML-Schema

"XML Schema ist eine Schemasprache, mit der XML-Dokumente beschränkt und Typinformationen zu Teilen bereitgestellt werden können."[1] [Rys09] Die XML Schema Definition (XSD) [FW04] ist eine Möglichkeit des World Wide Web Consortium (W3C) [W3C15a] zur Spezifikation von Anforderungen bezüglich gültiger XML-Dokumente. Der innerhalb von zwei Jahren [vdV02] entwickelte Standard besteht aus zwei normativen Bestandteilen, der Struktur- ([TBMM04]) und der Datentypbeschreibung ([BM04]), sowie einem nicht normativen Teil als Einführung ([FW04]). Ziel war es eine formale Beschreibung der möglichen Strukturinformationen eines XML-Dokuments zu erhalten, welche nicht nur durch Menschen lesbar, sondern auch durch Maschinen ausführbar ist [W3C15c].

In [TBMM04] werden die unterschiedlichen Komponenten eines XML-Dokuments spezifiziert. Es kann diesbezüglich zwischen dem **Abstract Data Model (ADM)** und dem **Element Information Item (EII)** unterschieden werden. Das ADM ist konzeptuell und definiert eine generelle, von Implementierung oder Repräsentation unabhängige Beschreibung von Schemakomponenten. Das EII hingegen ist die Realisierung von diesen Schemakomponenten im XML-Schema, wobei Eigenschaften, XML Repräsentation, Einschränkungen und weitergehende Validierungsregeln und -hinweise spezifiziert werden.

2.1.1. Strukturbeschreibung des XML-Schemas

Das Abstract Data Model (ADM) unterscheidet zwischen primären und sekundären Komponenten, sowie kontextabhängigen Hilfskomponenten. Primär sind einfache und komplexe Typdefinitionen, sowie Attribut- und Elementdeklarationen.

[1] "XML Schema is a schema language that allows to constrain XML documents and provides type information about parts of the XML document." [Rys09]

Sekundär sind die Definitionen von Attributgruppen, Identity-Constraints und Modellgruppen, sowie die Deklarationen von Notationen.

Zu den Hilfskomponenten gehören die Annotationen, Modellgruppen[2], Partikel, Wildcards und Attributverwendungen ("Attribute Uses"). Der Zusammenhang zwischen Modellgruppen, Partikeln, Modellgruppendefinitionen und Attributverwendungen wird nachfolgend kurz erklärt, da dieser nicht sofort ersichtlich ist. Modellgruppen sind Listen von Partikeln mit drei unterschiedlichen Varianten: der Reihenfolgen mit exakter Abfolge, der Konjunktion mit beliebiger, aber ebenso vollständiger Abfolge und Disjunktion mit einer Auswahl. Partikel entsprechen dabei einem Elementinhalt, der eine Elementdeklaration, Wildcard oder Modellgruppe mit jeweils definierter Häufigkeit ist. Partikel innerhalb einer komplexen Typdefinition charakterisieren als Bestandteil dessen Inhaltsmodell.

Modellgruppendefinitionen sind benannte Zusammenfassungen von Elementen oder Attributen. Attributverwendungen spielen eine ähnliche Rolle wie die Partikel für den Elementinhalt, sie definieren innerhalb eines komplexen Typs unter anderem, welche Attributdeklarationen zwingend oder optional sind, bzw. nicht verwendet werden dürfen.

Attributdeklarationen

Das Element Information Item (EII) ist wie bereits angesprochen die Repräsentation der Schemakomponenten des ADM im XML-Schema, sodass für jede Komponente jeweils eine detaillierte Beschreibung vorliegt. Die Repräsentation von Attributdeklarationen[3] ist in EII-Beispiel 2.1 dargestellt.

```
<attribute
    default = string
    fixed = string
    form = (qualified | unqualified)
    id = ID
    name = NCName
    ref = QName
    type = QName
    use = (optional | prohibited | required) : optional
    {any attributes with non-schema namespace . . .}>
    Content: (annotation?, simpleType?)
</attribute>
```

EII-Beispiel 2.1: XML-Repräsentation eines Attributs nach [TBMM04]

Eine Attributdeklaration besitzt unterschiedliche, teilweise voneinander abhängende Eigenschaften (bzw. Attribute [vdV02]), welche eine Deklaration innerhalb eines XML-Schemas charakterisieren. Diese sind lexikografisch sortiert und enthalten im dargestellten Beispiel unter anderem *default-*, *fixed-*, *name-*, *ref-*, *type-* und *use-Attribute*. Den Eigenschaften können Datentypen gemäß [BM04] zugeordnet sein

[2] Hier sind nicht die Modellgruppendefinitionen der sekundären Komponenten gemeint.
[3] siehe auch: http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/#declare-attribute

(z.B. *string*, *ID*, *NCName*, *QName*), alternative, festgelegte Wertbelegungen (z.B. *use = optional*, *use = prohibited* oder *use = required*), sowie Defaultwerte, insofern eine Eigenschaft nicht im XML-Schema spezifiziert wurde (*use = [..] : optional*). Des Weiteren wird spezifiziert, welche weitergehenden Komponenten innerhalb einer Attributdeklaration erlaubt sind. Dieser Inhalt (*content*) kann eine Annotation und einfache Typdefinition enthalten (beide jeweils optional).

Gültigkeitsbereiche in XML-Schema

In Abhängigkeit des Gültigkeitsbereichs, wird zwischen globalen und lokalen Attributdeklarationen unterschieden. Das XML-Beispiel 2.2 illustriert die Gültigkeitsbereiche von Schemakomponenten.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:attribute name="a1" type="xs:string"/>
    <xs:complexType name="ct1">
        <xs:attribute ref="a1" use="required"/>
        <xs:attribute name="a2" type="xs:string"/>
    </xs:complexType>
    <xs:attributeGroup name="ag1">
        <xs:attribute name="a1" type="xs:string" use="required"/>
    </xs:attributeGroup>
</xs:schema>
```

XML-Beispiel 2.2: XML-Schema mit globalen und lokalen Attributen

Ein **globaler Gültigkeitsbereich** bedeutet, dass eine Schemakomponente *<schema>* als direkten Vorgänger hat. Dies sind im XML-Beispiel die erste Attributdeklaration *a1*, der komplexe Typ *ct1*, sowie die Attributgruppe *ag1*. Globale Deklarationen können innerhalb des gesamten XML-Schemas referenziert werden, genauso wie es die Attributreferenz *a1* in *ct1* macht. Dabei gilt allerdings, dass lokale Deklarationen die globalen überdecken.

Ein **lokaler Gültigkeitsbereich** bedeutet im Gegensatz zum globalen, dass eine Komponente *<schema>* nicht als direkten Vorgänger hat, sondern wie im XML-Beispiel 2.2 ersichtlich zum Beispiel den komplexen Typen oder die Attributgruppe. Die lokale Attributdeklaration *a1* der Attributgruppe überdeckt die globale Deklaration *a1*. Lokale Deklarationen können nicht im gesamten XML-Schema referenziert werden, sondern nur von Kinderkomponenten des eigenen Gültigkeitsbereichs.

Es existieren weiterhin unterschiedliche Einschränkungen bezüglich der definierten Eigenschaften von Attributdeklarationen. Im Bezug auf die obige Aufzählung (*default*, *fixed*, *name*, *ref*, *type* und *use*) ergeben sich unter anderem folgende Bedingungen: Globale Attributdeklarationen können kein *use* angeben, während lokale dies hingegen ermöglichen bzw. den Defaultwert *optional* verwenden. Die Attribute *ref* und *name*, *ref* und *type*, sowie *default* und *fixed* können nicht zeitgleich in einer Attributkomponente auftreten. Ist ein *ref* angegeben, dann ist der *type* implizit von der referenzierten Deklaration übernommen. Dies gilt auch für *default-*

oder *fixed-Angaben*. Die lokale Überdeckung gilt in diesem Fall allerdings nicht, das heißt wenn eine globale Attributdeklaration einen *fixed-Wert* definiert, dann ist dies lokal nicht mehr durch eine Referenz änderbar.

Elementdeklarationen

Elementdeklarationen sind eine weitere primäre Schemakomponente, deren Element Information Item in EII-Beispiel 2.3 dargestellt wird.

```
<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 = QName
    type = QName
    {any attributes with non-schema namespace . . .}>
    Content: (annotation?, ((simpleType | complexType)?,
           (unique | key | keyref)*))
</element>
```

EII-Beispiel 2.3: XML-Repräsentation eines Elements nach [TBMM04]

Elementdeklarationen[4] besitzen ähnlich zu den Attributdeklarationen Eigenschaften wie *default*, *fixed*, *name*, *ref* und *type*. Die Abhängigkeiten zwischen diesen Attributen gelten ebenso für Elemente, das heißt, dass zum Beispiel *fixed* und *default* nicht zeitgleich definiert sein können. Der Gültigkeitsbereich von Elementdeklarationen ist lokal oder global, wobei auch an dieser Stelle die gleichen Mechanismen bezüglich der Sichtbarkeit und Referenzierbarkeit gelten.

Die Attribute *minOccurs* und *maxOccurs* ermöglichen eine detaillierte Ausdrucksmöglichkeit der Häufigkeit eines Elements innerhalb eines XML-Dokuments. Es kann Optionalität (*minOccurs = 0*) und unbeschränkte Häufigkeit (*maxOccurs = unbounded*) spezifiziert werden. Dabei gilt, dass $minOccurs \leq maxOccurs$ sein muss, entsprechende Defaultwerte sind gegeben (jeweils *1*). Beide Attribute können nur bei lokalen Elementdeklarationen oder Elementreferenzen verwendet werden.

Die Nullwertfähigkeit (*nillable = true*) bedeutet, dass ein Elementinhalt dem Datentyp zuwider leer sein kann. Es wird "angezeigt, dass für ein Element im XML-Dokument kein Wert vorliegt"[5] [Cos02]. Das XML-Beispiel 2.4 zeigt dies.

[4] siehe auch: http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/#declare-element

[5] "nil: an instance document element may indicate no value is available" [Cos02]

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="root" type="roottype"/>
    <xs:element name="e1" type="xs:decimal" nillable="true"/>
    <xs:complexType name="roottype">
        <xs:sequence>
            <xs:element ref="e1" maxOccurs="2"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>

<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="exampleNil.xsd">
    <e1 xsi:nil="true"></e1>
    <e1>0</e1>
</root>
```

XML-Beispiel 2.4: XML-Schema eines nullwertfähigen Elements mit XML-Dokument

Die Elementreferenz *e1* ist nullwertfähig, da deren Deklaration das Attribut *nillable="true"* besitzt. Dadurch kann im XML-Dokument das erste Element von *e1* trotz des Datentyps *xs:decimal* einen leeren Elementinhalt besitzen. Dass dies ansonsten nicht möglich ist, wird durch das zweite Element *e1* ausgedrückt. Dieses ist zwar ebenso nullwertfähig, gibt dies aber nicht explizit an. Es muss das Attribut *xsi:nil="true"* angegeben werden, damit die in der Deklaration des XML-Schemas spezifizierte Nullwertfähigkeit greift. Somit wird ein Elementinhalt für *e1* gemäß Datentyp benötigt.

Die Eigenschaften *abstract*, *block*, *final* und *substitutionGroup* ermöglichen die Definition von Stellvertretern von Elementen, sowie die Einschränkung von diesen. Zum Beispiel könnten eine Elementdeklaration *e1* und zwei weitere Elemente *e2* und *e3* spezifiziert werden, wobei die beiden letzten das Attribut *substitutionGroup = "e1"* und den gleichen Typen wie *e1* enthalten. Insofern nun eine Elementreferenz bezüglich *e1* existieren würde, könnte an dieser Stelle ebenso *e2* oder *e3* verwendet werden. Dabei sind unterschiedliche Randbedingungen bezüglich der Substitutionen zu beachten, die aufgrund der "besonders unscharfen Spezifikation von Erweiterungen und Einschränkungen"[6] [vdV02] von Substitutionsgruppen von Anwendungsempfehlungen abgelehnt werden ("DO NOT use substitution groups" [KAW01]). In [Oba03a] wird ebenso auf die Schwierigkeiten dieses Konzepts hingewiesen[7], besonders im Zusammenhang mit ungewollten, durch Einbindung weiterer XML-Schema verbundener Substitutionsmöglichkeiten.

[6] "Substitution groups can be seen as extensible element groups [..] the Recommendation is especially fuzzy on the extensibility of element groups and the restriction of substitution groups [..]" [vdV02]

[7] "Substitution groups [..] allow extensibility in directions the schema author may not have anticipated [..] it makes it harder to process documents based on such schemas." [Oba03a]

Einfache Datentypen

Die bereits angesprochenen Datentypen des zweiten, normativen Teils der Spezifikation sind in [BM04] dargestellt. Diese "beschreiben den Inhalt von Elementen (speziell von Textknoten im XML-Dokument) oder Attributen, und sind komplett unabhängig von anderen Knoten und somit vom Markup"[8] [vdV02]. Bei einfachen Typdefinitionen werden grundsätzlich drei Varianten unterschieden: atomare Typen ("atomic"), Listen- ("list") und Vereinigungstypen ("union").

Atomare Typen können entweder primitive oder abgeleitete Datentypen sein. Dazu gehören unter anderem die **built-in-Typen**, die in Abbildung 2.1 dargestellt sind. Primitive Datentypen sind unter anderem *string* und *decimal*, während

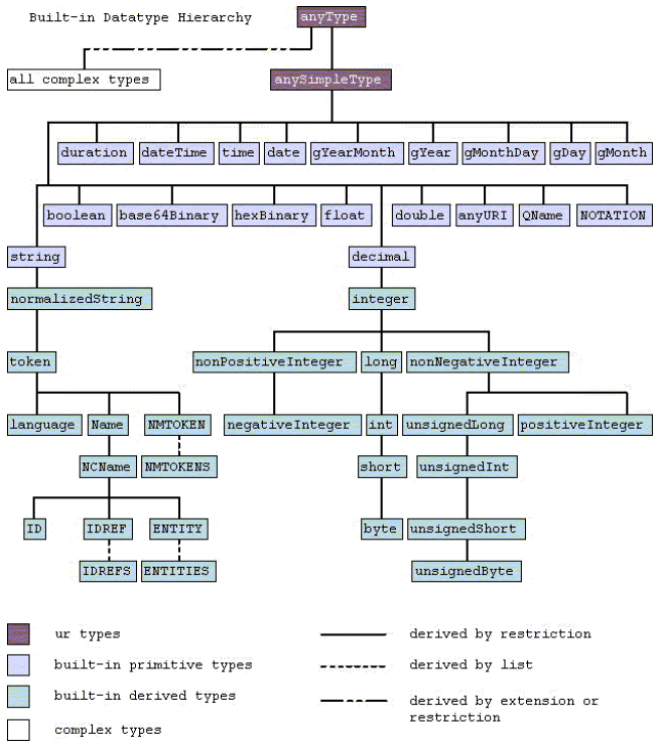


Abbildung 2.1.: XML-Schema built-in-Datentypen aus [BM04]

abgeleitete built-in-Typen *integer*, *long* etc. sind. Die Ableitung kann mittels Einschränkung und/oder Erweiterung oder Listenbildung geschehen, wobei Typhierarchien aufgebaut werden.

Ein **Listentyp** ist ein abgeleiteter Typ, wobei dieser als endlich langer, jeweils durch Leerräume separierter, atomarer Typ angesehen wird. Dieser atomare Typ eines Listentyps wird als *itemType* bezeichnet.

[8] "[..] describe the content of a text node or an attribute value. They are completely independent of the other nodes and, therefor, independent of the markup." [vdV02]

Die **Vereinigungstypen** sind ebenso abgeleitet, wobei der Wertebereich aus den unterschiedlichen, als Teilnehmer (*memberTypes*) spezifizierten Typen besteht. Aktuell existieren keine built-in-Vereinigungstypen.

Zusätzlich zu den bereits erwähnten, abgeleiteten Typen können einschränkende Typen (**Restriktionstypen**) definiert werden. Eine Restriktion besitzt einen Basistypen (*baseType*), der wiederum ein Listentyp, Vereinigungstyp, built-in-Typ oder sogar ein weiterer Restriktionstyp sein kann.

Die Einschränkung wird mittels **Facetten** getätigt, das heißt der Wertebereich des Basistyps wird zum Beispiel durch die explizite Angabe einer minimalen Länge (*minLength*) eingeschränkt. Die jeweils gültigen Facetten sind vom Datentypen des Basistypen abhängig, zum Beispiel kann die Länge (*length*), aber nicht der minimale, inklusive Wert (*minInclusive*) eines *string* Typs definiert werden. Dahingegen ist dies umgekehrt für den *decimal* Typ möglich (gültig: *minInclusive*, ungültig: *length*). Eine Zusammenfassung aller built-in-Datentypen mit deren gültigen Facetten ist im Anhang A Abbildung A.1 aufgelistet.

Das Element Information Item das Datentyps[9] wird in EII-Beispiel 2.5 gezeigt.

```
<simpleType
    final = (#all | List of (list | union | restriction))
    id = ID
    name = NCName
    {any attributes with non-schema namespace . . .}>
    Content: (annotation?, (restriction | list | union))
</simpleType>
```

EII-Beispiel 2.5: XML-Repräsentation eines einfachen Typs nach [BM04]

Der Inhalt (*content*) von einfachen, nutzerdefinierten Typen kann demzufolge eine Restriktion (*restriction*), ein Listentyp (*list*) oder ein Vereinigungstyp (*union*) sein. Die vordefinierten built-in-Typen sind nicht explizit aufgeführt, da diese im XML-Schema selber spezifiziert und somit direkt verwendbar sind.

Einfache Typdefinitionen können wie Deklarationen einen lokalen und globalen Gültigkeitsbereich besitzen. Built-in Datentypen sind global und im gesamten XML-Schema sichtbar, wobei eine Definition per qualifiziertem Namen (*QName*) referenziert wird. Ein solcher *QName* ist dabei ebenso ein primitiver built-in-Datentyp, der eine Zeichenkette darstellt. Diese besteht aus einem Präfix (üblicherweise *xs*), einer Schemaadresse (*http://www.w3.org/2001/XMLSchema*), einem Doppelpunkt und dem *NCNamen* des Datentyps (z.B. *string*). Der qualifizierte Name von *string* lautet demnach *xs:string*. Lokale Definitionen können innerhalb von Attribut- und Elementdeklarationen, sowie komplexen Typdefinitionen mit einfachem, einschränkendem Inhaltsmodell spezifiziert werden. Diese sind nur innerhalb deren Gültigkeitsbereiches sichtbar und somit referenzierbar.

[9] siehe auch: http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/#xr-defn

Komplexe Datentypen

Die noch verbleibende, primäre Schemakomponente der komplexen Typen wird nachfolgend erläutert. Im Gegensatz zu einfachen Datentypen, welche sowohl in Attribut- als auch bei Elementdeklarationen genutzt werden dürfen, können nur Elementdeklarationen einen komplexen Typ annehmen. Das Element Information Item ist in EII-Beispiel 2.6 dargestellt.

```
<complexType
    abstract = boolean : false
    block = (#all | List of (extension | restriction))
    final = (#all | List of (extension | restriction))
    id = ID
    mixed = boolean : false
    name = NCName
    {any attributes with non-schema namespace . . .}>
    Content: (annotation?, (simpleContent | complexContent |
           ((group | all | choice | sequence)?,
           ((attribute | attributeGroup)*, anyAttribute?))))
</complexType>
```

EII-Beispiel 2.6: XML-Repräsentation eines komplexen Typs nach [TBMM04]

Komplexe Typen[10] beschreiben das Markup, das heißt abhängig von deren *content* und der Eigenschaft *mixed* wird ein XML-Dokument strukturell spezifiziert. Das Attribut *mixed* = *true* bedeutet, dass zusätzlich zum Markup auch Textknoten zwischen dem Markup möglich sind, sodass "schwer zu analysierende XML-Dokumente entstehen können"[11] [Liq14]. Das XML-Beispiel 2.7 illustriert dies.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="root" type="roottype"/>
    <xs:element name="e1" type="xs:decimal"/>
    <xs:complexType name="roottype" mixed="true">
        <xs:sequence minOccurs="3" maxOccurs="3">
            <xs:element ref="e1" minOccurs="1" maxOccurs="1"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>

<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="exampleCT.xsd">
    Text dank mixed erlaubt
    <e1>0</e1> Text dank mixed erlaubt
    <e1>0</e1> Text dank mixed erlaubt
    <e1>0</e1> Text dank mixed erlaubt
</root>
```

XML-Beispiel 2.7: XML-Schema eines komplexen Typs mit XML-Dokument

[10] siehe auch: http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/#declare-type

[11] "Mixed content is something you should try to avoid when creating your XML schema. [..] However, it is difficult to parse and it can lead to unforeseen complexity in the XML document's data." [Liq14]

Es wird in dem Beispiel der *mixedContent* erlaubt, das heißt zwischen dem Markup der Elemente (*<e1>0</e1>*) stehen Textknoten, die passender Weise den Inhalt "*Text dank mixed erlaubt*" haben.

Es wird unterschieden zwischen einfachem (*simpleContent*) und komplexem Inhalt (*complexContent*). Ist weder *simpleContent* noch *complexContent* angegeben, so ist dies eine abkürzende Schreibweise eines komplexen Typs mit komplexen Inhalt als Einschränkung des *anyType* (siehe Abbildung 2.1). In diesem Fall wird eine Modellgruppe[12] des Abstract Data Model verwendet. Somit ist das **Inhaltsmodell** entweder eine Reihenfolge (*sequence*), eine Menge (*all*, als Konjunktion) oder eine Auswahl (*choice*, als Disjunktion). Dabei spielen die Partikel erneut eine wichtige Rolle, da diese die Anzahl der Durchläufe des Inhaltsmodells durch Angabe von *minOccurs* und *maxOccurs* festlegen. Eine Gruppe (*group*) als vierte Alternative ist an dieser Stelle eine Referenz auf eine globale Elementgruppe, die wiederum eine Zusammenfassung von Elementdeklarationen in einer Modellgruppe darstellt. Im XML-Beispiel 2.7 ist das Inhaltsmodell des komplexen Typs (*roottype*) eine Sequenz, die genau dreimal durchlaufen wird. Da die Elementreferenz innerhalb der Modellgruppe genau einmal auftreten kann, ergibt sich als Konsequenz, dass *e1* genau dreimal im XML-Dokument auftritt.

Neben dem Inhaltsmodell können in einem komplexen Typen wie bereits angesprochen Attributdeklarationen, sowie Referenzen auf globale Attributgruppen und eine Attributwildcard definiert werden.

Der *simpleContent* und *complexContent* ermöglicht die Einschränkung bzw. Erweiterung nutzerdefinierter komplexer Typen. Die Ableitung von komplexen Typen ist dabei ähnlich kompliziert wie die der Substitutionsgruppen von Elementdeklarationen und "führt zu mehr Problemen, als damit gelöst werden"[13] [Oba03c]. Dies gilt besonders für Einschränkungen, welche "nur mit Vorsicht zu benutzen sind"[14] [Oba03a].

Der *simpleContent* erweitert einfache Typen um Attribute, bzw. schränkt bereits vorhandene (einfache Typen mit Attributen) mittels Facetten und Einschränkungen von definierten Attributen ein. Das Ergebnis ist ein komplexer Typ mit einfachem Inhalt, allerdings erweitert um Attribute (insofern dies vollzogen wurde). Einschränkungen sind für lokale und globale Typen, Erweiterungen nur für globale Typen möglich.

Beim *complexContent* sind Einschränkungen bzw. Erweiterungen vom Inhaltsmodell und den Attributen möglich. Erweiterungen enthalten implizit das Inhaltsmodell und die Attribute des Basistypen, neue Komponenten werden jeweils am Ende der entsprechenden Modellgruppe mittels Sequenz angehängt. Einschränkungen sind beim komplexen Inhalt nur bei globalen Typen möglich. Dabei wird das Inhaltsmodell von diesem komplett wiederholt (explizit) und anschließend dessen

[12] siehe auch: http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/#declare-contentModel
[13] "[..] complex type derivation features [..] may add more problems than they solves [..]" [Oba03c]
[14] "DO use restriction of complex types carefully." [Oba03a]

Partikel eingeschränkt. Attribute werden allerdings anders behandelt, diese sind implizit enthalten und müssen bei einer Wiederholung eingeschränkt werden.

Weitere Schemakomponenten

Es wurden die primären Schemakomponenten des Abstract Data Model erläutert. Dies ist ein kleiner Ausschnitt der Möglichkeiten eines XML-Schemas, der allerdings einen großen Einfluss auf die Struktur der XML-Dokumente besitzt. Element Information Items existieren jedoch für alle erwähnten Schemakomponenten des ADM, dies ist neben den Einschränkungen und kontextabhängigen Eigenschaften der unterschiedlichen Attribute unter anderem ein Grund für die hohe Komplexität von XML-Schema.

Nicht ohne Grund wird in Anwendungsempfehlungen ([KAW01], [Oba03a]) darauf hingewiesen "DO NOT try to be a master of XML Schema. [..]" [KAW01]. Es werden in der vorliegenden Arbeit weitere EIIs anderer Schemakomponenten benötigt, an entsprechender Stelle wird auf die Spezifikation [TBMM04] verwiesen.

2.1.2. XML-Schema Version 1.1

Die XML-Schema 1.1 Spezifikation wurde am 5. April 2012 als Recommendation des W3C veröffentlicht [Cos09]. Damit wurden unterschiedliche, neue Konzepte aufgenommen bzw. alte überarbeitet, da XML-Schema 1.0 einige Lücken aufwies [Bor11] bzw. "Einschränkungen existierten"[15] [DGGN09], die zu "nicht intuitiven Schemadesigns"[16] [DGGN08] führten.

Der neue Standard besteht aus zwei Teilen, der Struktur- ([GSMT12]) und Datentypbeschreibung ([PGM+12]), wobei XML-Schema 1.1 als Obermenge von XML-Schema 1.0 zu verstehen ist [Cos09]. Abbildung 2.2 illustriert diesen Aspekt. Somit wurde sichergestellt, dass "ein XSD 1.0 konformes XML-Dokument mit ei-

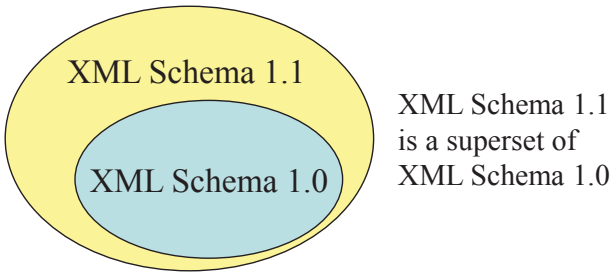


Abbildung 2.2.: Zusammenhang XSD 1.0 und XSD 1.1 aus [Cos09]

[15] "[..] XML Schema 1.0 has certain limitations." [DGGN09]
[16] "Schema authors often face certain challenges [..] resulting in counter-intuitive schema designs [..]" [DGGN08]

nem XSD 1.1 Validator geprüft werden kann, aber nicht anders herum"[17] [Cos09]. Für die Gegenrichtung (XSD 1.0 Validator, XSD 1.1 konformes XML-Dokument) werden Hinweise für die Anpassung des alten Validators in [Orc07] gegeben.[18]

Die sekundären Komponenten des Abstract Data Model wurde um Zusicherungen ("Assertions") und Typalternativen erweitert. Des Weiteren wurden Element Information Items für diese Komponenten mit in die Spezifikation [GSMT12] aufgenommen, sowie bereits vorhandene angepasst. Im Bezug auf die oben vorgestellten primären Schemakomponenten ergeben sich die in EII-Beispiel 2.8 dargestellten Änderungen.

```
<attribute>
+   targetNamespace = anyURI
</attribute>

<element>
+/- substitutionGroup = List of QName
+   targetNamespace = anyURI
    Content: (annotation?, ((simpleType | complexType)?,
+           alternative*,
            (unique | key | keyref)*))
</element>

<complexType>
+   defaultAttributesApply = boolean : true
    Content: (annotation?, (simpleContent | complexContent |
+           (openContent?,
            (group | all | choice | sequence)?,
            ((attribute | attributeGroup)*, anyAttribute?),
+           assert*)))
</complexType>
```

EII-Beispiel 2.8: Erweiterungen der primären Schemakomponenten nach [GSMT12]

In Beispiel 2.8 werden die Deltas im Bezug zu obigen Repräsentationen erfasst. Erweiterungen des EII werden mittels vorangestelltem *+*, Änderungen bereits vorhandener Eigenschaften mit *+/-* illustriert. Somit werden Attribut- und Elementdeklarationen jeweils um die direkte Möglichkeit zur Angabe des Zielnamensraums (*targetNamespace*) erweitert. Substitutionsgruppen von Elementen sind als Liste qualifizierter Namen spezifizierbar. Die Typalternative[19] des ADM wurde in den Inhalt (*content*) mit aufgenommen, das heißt eine bedingte Typisierung in Abhängigkeit von Attributwerten ist möglich [Bor11].

Komplexe Typen werden um Assertions und flexiblere Inhalte erweitert (eine gewisse Flexibilität ist bereits in XSD 1.0 möglich, siehe [Oba03b]). Durch die

[17] "An instance document conforming to a 1.0 schema can be validated using a 1.1 validator, but an instance document conforming to a 1.1 schema may not validate using a 1.0 validator." [Cos09]

[18] "The majority of this guide focuses on [XML Schema 1.1 Part 1] ([GSMT12], Anm. d. Autors) extensibility techniques that enable forwards-compatible versioning." [Orc07]

[19] siehe auch: http://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/#element-alternative

Angabe von *defaultAttributesApply* kann die im Schema[20] definierte Defaultattributgruppe (*defaultAttributes* = *QName*) referenziert werden, das heißt es werden in einen komplexen Typen automatisch alle in der Attributgruppe spezifizierten Attributdeklarationen übernommen. Dieses muss explizit abgewählt werden, da der Defaultwert *true* ist. Des Weiteren ist es möglich durch die Angabe von *openContent* Typen mit komplexem Inhalt (*complexContent*) zu erweitern. Es wird diesbezüglich definiert, an welcher Position im Inhaltsmodell welche zusätzlichen Elementdeklarationen (z.B. auch Wildcards) erlaubt sind. Ähnlich wie bei den Defaultattributgruppen existiert nun auch die Möglichkeit das "gesamte Schema als offen für beliebige andere Elemente" [Bor11] zu kennzeichnen, indem im Inhalt des Schemas die *defaultOpenContent* Komponente entsprechend definiert wird.

Das Element Information Item eines einfachen Typs wurde nicht erweitert und wurde daher im EII-Beispiel 2.8 nicht mit aufgenommen. Dennoch sind neue built-

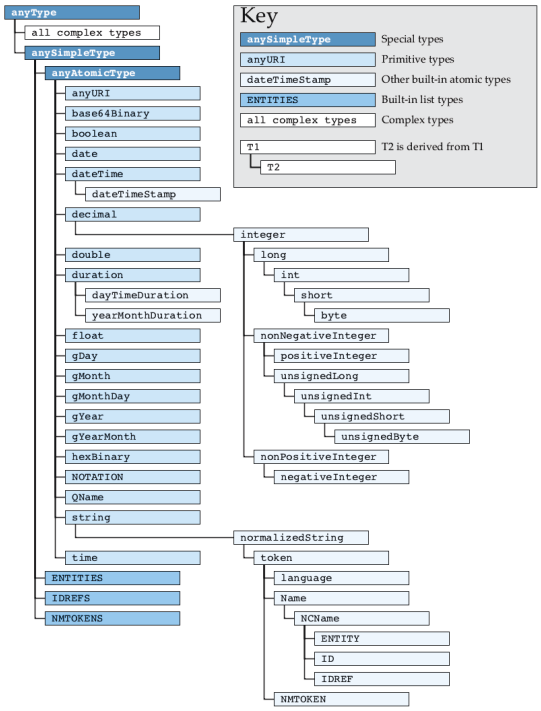


Abbildung 2.3.: XML-Schema built-in-Datentypen aus [PGM+12]

in-Typen mit in die Spezifikation aufgenommen worden, "um die Typsysteme mit anderen Spezifikationen des W3C abzustimmen. Dies sind *anyAtomicType*, *dayTimeDuration* und *yearMonthDuration*"[21] [DGGN08]. Die im Vergleich zu Abbil-

[20] siehe auch: http://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/#declare-schema

[21] "To align the type systems of XML Schema and these specifications (Alignment with XQuery 1.0 and XPath 2.0 data model types, Anm. d. Autors), the XML Schema 1.1 data types specification introduced [..] namely: anyAtomicType, dayTimeDuration, and yearMonthDuration." [DGGN08]

dung 2.1 angepasste Typhierarchie der built-in-Typen von XML-Schema 1.1 wird in Abbildung 2.3 dargestellt. Die oben erwähnte Zusammenfassung der built-in-Datentypen mit deren gültigen Facetten im Anhang A Abbildung A.1 bezieht sich auf selbige.

Die aufgezählten Änderungen betreffen nur die primären Schemakomponenten des Abstract Data Model. Es existieren allerdings weitere Anpassungen, die bei Bedarf an entsprechender Stelle in der vorliegenden Arbeit erwähnt werden.

2.1.3. Modellierungsstile von XML-Schema

Aus den Möglichkeiten Deklarationen und Definitionen global oder lokal zu definieren, ergeben sich unterschiedliche **Modellierungsstile**, die einen großen Einfluss auf die Struktur und die Eigenschaften eines XML-Schemas haben.

In [xml01][22] wurden drei Stile eingeführt: *Russian Doll*, *Salami Slice* und *Venetian Blind*. Der vierte Stil *Garden of Eden* wurde mit Verweis auf [xml01] in [Mal02] ergänzt. Abbildung 2.4 illustriert die Modellierungsstile von XML-Schema.

	Gültigkeits-bereich	Russian Doll	Salami Slice	Venetian Blind	Garden of Eden
Element- und Attributdeklaration	lokal	x		x	
	global		x		x
Typdefinition	lokal	x	x		
	global			x	x

Abbildung 2.4.: Modellierungsstile von XML-Schema nach [Mal02]

Der *Garden-of-Eden-Stil* zeichnet sich durch die Eigenschaft aus, dass sowohl die Attribut- und Elementdeklarationen als auch die einfachen und komplexen Typdefinitionen global spezifiziert werden (x). Dadurch ist "die Wiederverwendbarkeit aller Komponenten schemaintern und -übergreifend möglich"[23] [JW10], allerdings sind solche Schemata schwieriger von Menschen lesbar und im Vergleich zu anderen weniger kompakt. Des Weiteren müssen "globale Deklarationen einen eindeutigen Namen besitzen, ein praktischer Nebeneffekt"[24] [Mar03]. Diese Einschränkung bzw. Regel gilt zeitgleich für Typdefinitionen im selben Gültigkeitsbereich.

"Ist die Wiederverwendbarkeit nicht zwingend erforderlich, die Kompaktheit allerdings schon"[25] [JW10], wird zum Beispiel der *Russian-Doll-Stil* empfohlen. In diesem Stil sind alle Deklarationen und Definitionen lokal spezifiziert, sodass "entkoppelte, zusammenhängende Schemata"[26] [JW10] entstehen.

[22] siehe speziell: http://www.xfront.com/GlobalVersusLocal.html

[23] "By making every possible element, attribute, and type global, you create a scenario that maximizes reuse, both internally and between schemas [..]" [JW10]

[24] "[..] global declarations must be unique: No two global declarations can use the same [..] name." [Mar03]

[25] "If schema reuse is not imperative and minimizing size is, use the Russian doll style [..]" [JW10]

[26] "Russian doll style schemas [..] are also considered highly decoupled [..] and cohesive [..]" [JW10]

2.2. XPath

"XPath ist eine auf **Pfadausdrücken** basierende Sprache, die die Auswahl von Teilen eines gegebenen XML-Dokuments ermöglicht"[27] [HP09]. Seit 1999 existiert die Recommendation des W3C bezüglich der XML Path Language (XPath) [CD99]. XPath 1.0 ist eine Grundlage von XSD 1.0.[28] XPath 2.0 ([BBC+10]) wurde 2010 als Recommendation verabschiedet, eine Grundlage von XML-Schema 1.1.[29]

XPath modelliert XML-Dokumente als Baumstruktur mit Knotentypen. Die Typen sind gemäß des Datenmodells Elemente, Attribute, Texte, sowie Wurzel-, Namensraum-, Verarbeitungs- und Kommentarknoten. "Das grundlegende syntaktische Konstrukt von XPath ist ein Ausdruck"[30] [CD99]. Ausdrücke werden von "links nach rechts induktiv ausgewertet" [Lau05] und mit / miteinander verknüpft, wobei das Ergebnis eines Ausdrucks ein atomarer Wert oder eine gegebenenfalls leere Knotenmenge ist.

Die verwendete Recommendation spielt eine entscheidende Rolle, denn "XPath 1.0 verwendet duplikatfreie Knotenmengen, während XPath 2.0 diese zu Knotensequenzen mit Duplikaten generalisiert"[31] [BFM+10]. "Sofern nicht explizit anders definiert sind die Elemente einer Sequenz in **Dokumentordnung**." [Lau05] Des Weiteren nutzt XPath 2.0 das Typsystem von XML Schema, beinhaltet eine wesentlich umfangreichere Funktionsbibliothek, unterstützt Referenzen und kennt Dokumentkollektionen, sowie weitergehende Ausdrucksarten und Vergleiche, die nicht nur auf Werten basieren [KM03].

Lokationspfad

Pfadausdrücke (engl. Location Path) lassen sich absolut oder relativ angeben. "Ein absoluter Pfad wird von der Wurzel ausgehend ausgewertet. Er beginnt stets mit einem Schrägstrich / ." [KM03] Relative Pfade werden hingegen vom aktuellen **Kontextknoten** aus analysiert, wobei dieser durch die einzelnen Werte repräsentiert wird, die durch den vorangehenden XPath-Ausdruck gebildet werden [KM03].

"Ein Lokationsschritt (engl. Location Step, Anm. d. Autors) besteht aus einer Achse, einem Knotentest und, optional, aus einem oder mehreren Prädikaten." [Lau05] Ein Schritt hat den folgenden syntaktischen Aufbau:

Achse :: Knotentest [Prädikat]

[27] "XPath is a language based on path expressions that allows the selection of parts of a given XML document." [HP09]
[28] "XML Schema: Structures depends on the following specifications: [..] XPath [..]" [TBMM04]
[29] "XML Schema [..]: Structures depends on the following specifications: [..] XPath 2.0 [..]" [GSMT12]
[30] "The primary syntactic construct in XPath is the expression." [CD99]
[31] "Sequences replace node-sets from XPath 1.0. In XPath 1.0, node-sets do not contain duplicates. In generalizing node-sets to sequences in XPath 2.0, duplicate removal is provided by functions on node sequences." [BFM+10]

Die *Achse* "stellt die Beziehung zwischen den Kontextknoten und den zu selektierenden Knoten her" [KM03]. Eine Achse kann zum Beispiel den Vorgänger (*parent*), Nachfolger (*descendant*), den Knoten selbst (*self*) oder Kinderknoten (*child*) adressieren. Es existieren aktuell insgesamt 13 mögliche Varianten von Achsen.

Jede Achse hat einen speziellen Typen, der durch den *Knotentest* analysiert werden kann. Somit kann die ausgewählte Knotenmenge eingeschränkt bzw. das Vorhandensein eines bestimmten Typs verlangt werden. Zum Beispiel können unter anderem Textknoten (*text()*), Elementknoten (*node()*) oder beliebige Typen (*) ausgewählt werden. Die Angabe eines qualifizierten Namens ist ebenso möglich.

Mit Hilfe von *Prädikaten* können weitere Bedingungen an Knotenmengen formuliert werden. Die Prädikate müssen neben dem Knotentest erfüllt sein, damit ein entsprechender Knoten in die Knotenmenge übernommen wird. Prädikate können ebenso **Kontextpositionen** sein, die mit *1* beginnend die relative Position eines Knotens innerhalb einer Knotenmenge angeben. Die Mächtigkeit der vorliegenden Menge (Kontextgröße), sowie die Dokumentordnung sind dabei entscheidend.

"Da die vollständige Achsenangabe sehr gesprächig ist, gibt es eine verkürzende Schreibweise der Pfadangabe [..]" [KM03], sodass zum Beispiel statt *self::node()* '.', *parent::node()* '..' oder *attribute::* '@' verwendet werden kann.[32] Es existieren weitere Äquivalenzen, die unter anderem in [OMFB02] thematisiert werden.

2.3. Evolution und Versionierung

Die Schemaversionierung ist ein alternativer Ansatz zur Schemaevolution. "Die Versionierung beschäftigt sich mit der Anforderung aktuelle Daten zu bewahren, und der Fähigkeit diese abzufragen und zu aktualisieren."[33] [Rod09b]

Für die Versionierung ist charakteristisch, dass zu einem Zeitpunkt mehrere, verschiedene Ausprägungen einer Strukturbeschreibung existieren können. Zeitgleich sind vorhandene Instanzen (d.h. Dokumente) immer gültig bezüglich einer oder mehrerer Versionen. Ändert sich ein Schema aufgrund von korrigierender, adaptiver oder perfektionierender Wartung, wird eine neue Version erzeugt. Zukünftige Dokumente können dieses neue oder alternativ ein älteres Schema referenzieren.

Ein Vorteil aus Sicht der Daten ist, dass gültige Instanzen keine Anpassungen benötigen, ein Verlust von jenen ist ausgeschlossen. Allerdings können schnell eine Vielzahl von redundanten Schemata entstehen, deren Verwaltung komplex und aufwendig sein kann. Redundante Schemata sind unter anderem durch die Weiterentwicklung ausgehend von unterschiedlichen Versionen, sowie durch instanzerhaltende Schemaänderungen möglich. Ein weiterer, nicht zu unterschätzender Nachteil ist, dass korrigierende Anpassungen nicht zwingender Weise in allen Versionen

[32] siehe auch: http://www.w3.org/TR/1999/REC-xpath-19991116/#path-abbrev

[33] "Schema versioning deals with the need to retain current data, and the ability to query and update it, through alternate database structures." [Rod09b]

umgesetzt werden. Es können demnach fehlerhafte Schemaversionen vorliegen, die weiterhin referenziert werden können.

Die Evolution im Kontext der vorliegenden Arbeit[34] hat im Vergleich zur Versionierung das Ziel, "nur eine gültige Version eines Schemas zu besitzen"[35] [Rod09a]. Schemaänderungen können somit die Gültigkeit vorhandener Instanzen beeinträchtigen, eine Anpassung dieser kann bei *instanzverändernden*, *instanzerweiternden* oder *instanzreduzierenden* Operationen notwendig werden. Die Instanzanpassung kann zum Verlust von Daten führen, allerdings können somit alle Instanzen gültig bezüglich des neusten, korrigierten, adaptierten und perfektionierten Schemas sein. Des Weiteren ist eine Versionsverwaltung im Allgemeinen nicht notwendig.

Abschließende Betrachtung

In diesem Kapitel wurden *XML-Schema* und *XPath* vorgestellt. Dabei sind die primären Schemakomponenten und unterschiedlichen Modellierungsstile erläutert worden. Die Deklarationen und Definitionen sind in diesem Zusammenhang von Bedeutung. Anschließend wurde der allgemeine Aufbau von XPath vorgestellt. Mit dessen Lokalisierungspfaden werden Komponenten sowohl in XML-Schema als auch wohlgeformten XML-Dokumenten im Allgemeinen adressiert.

Zum Abschluss wurde eine Abgrenzung zur *Versionierung* als alternativer Ansatz der Evolution getätigt. Nachdem grundlegende Technologien überblicksartig präsentiert wurden, werden im folgenden Kapitel verwandte Arbeiten vorgestellt.

[34] siehe auch: Kapitel 1.1 (Problemstellung) und Abbildung 1.1 (Überblick der XML-Schemaevolution)
[35] "In all cases only one schema remained [..]" [Rod09a]

3. Stand der Technik

Die Schemaevolution ist nicht nur im Kontext von XML-Schema interessant. Es existieren ebenso Ansätze in relationalen bzw. objektorientierten Schemata, sowie in XML Schema allgemein. Daher werden in diesem Kapitel klassische, teils XML-Schema-fremde und aktuelle, XML-Schema-spezifische Ansätze in den **Abschnitten 3.1** und **3.2** thematisiert.

In diesem Zusammenhang wird ebenfalls in Hinblick auf die folgenden Kapitel dargelegt, welche Anforderungen, Konzepte und/oder Strategien gegebenenfalls übernommen und XML-Schema-spezifisch adaptiert werden. In **Abschnitt 3.3** werden die vorgestellten Ansätze abschließend kurz zusammengefasst.

3.1. Klassische Ansätze der Schemaevolution

Zu den klassischen Ansätzen wird die Möglichkeit der Schemaevolution im Umfeld des Relationenmodells und objektorientierter Schemata gezählt. Diese XML-Schema-fremden Modelle werden in den **Abschnitten 3.1.1** und **3.1.2** untersucht.

In **Abschnitt 3.1.3** wird die Evolution der Document Type Definition (DTD) [BPSM+08] thematisiert. Diese in Kapitel 1 bereits erwähnte Spezifikation des W3C [W3C15a] gilt als eingeschränkter Vorgänger von XML-Schema.

3.1.1. Relationenmodell

"Das von Codd 1970 eingeführte **Relationenmodell** [Cod70] ist das mittlerweile am weitesten verbreitete Datenbankmodell." [SSH13] Eine Veranschaulichung des Strukturteils ist in Abbildung 3.1 dargestellt. "Eine Relation kann anschaulich

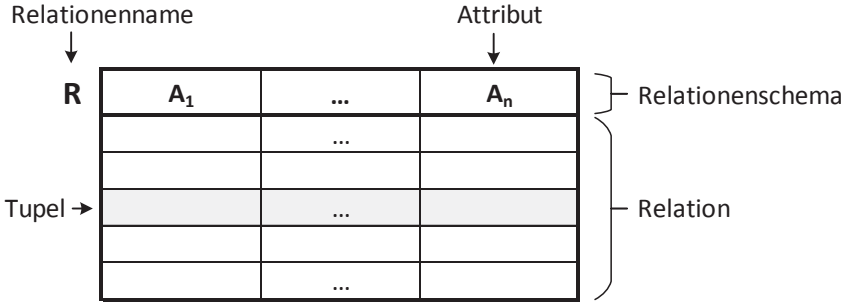


Abbildung 3.1.: Veranschaulichung eines Relationenschemas mit Relation nach [SSH13]

als Tabelle verstanden werden: Die Attribute des Relationenschemas bilden die Spaltenüberschriften der Tabelle, die Tupel sind die verschiedenen Zeilen, und die Einträge in den verschiedenen Tabellenpositionen gehören zu den jeweiligen Wertebereichen." [SSH13]

"**SQL** (Structured Query Language) ist die Datenbanksprache für relationale Systeme [..] und stellt als Datendefinitionssprache (engl. Data Definition Language, kurz **DDL**) eine ganze Reihe von Anweisungen zur Datendefinition zur Verfügung." [SSH13] "Die minimalen Anforderungen von SQL sind in ISO/IEC 9075-1 ([ISO11a]), ISO/IEC 9075-2 und ISO/IEC 9075-11 spezifiziert."[1] [ISO11d] Besonders die in [ISO11b][2] spezifizierten Möglichkeiten zur Schemadefinition und Manipulation sind im Kontext der Schemaevolution interessant, denn "SQL unterstützt mit der alter-Anweisung auch eine einfache Form der Schemaevolution" [SSH11]. Die **alter-table-Anweisung** ist in SQL-Beispiel 3.1 dargestellt.

```
<alter table statement> ::=
  ALTER TABLE <table name> <alter table action>
<alter table action> ::=
  <add column definition>
    | <alter column definition>
    | <drop column definition>
    | <add table constraint definition>
    | <alter table constraint definition>
    | <drop table constraint definition>
    | <add system versioning clause>
    | <alter system versioning clause>
    | <drop system versioning clause>
```

SQL-Beispiel 3.1: alter-table-Anweisung nach [ISO11b]

Mit der alter-Anweisung können Spalten, Constraints und Versionierungsklauseln hinzugefügt, verändert oder gelöscht werden. Im Allgemeinen gilt, dass von den Änderungen betroffene Tabellen **nicht referenzierbar** sein dürfen und dies zum aktuellen Zeitpunkt auch nicht sind.[3] Diese starke Einschränkung ist notwendig, damit die Konsistenz einer Datenbank erhalten bleibt. Allerdings könnten kaskadierend referenzierende Strukturen angepasst werden (was zum Teil auch möglich ist), dies sollte bei der XML-Schemaevolution übernommen werden.

Es existieren weitergehende Einschränkungen, die exemplarisch für jede Variante (d.h. *drop*, *alter* und *add*) der Spaltenänderung dargestellt werden soll. Eine Spalte darf nur gelöscht werden, wenn diese nicht die einzige der Tabelle ist.[4] Datentypen

[1] "ISO/IEC 9075-1, ISO/IEC 9075-2 and ISO/IEC 9075-11 encompass the minimum requirements of the language. Other parts define extensions." [ISO11d]

[2] Mit Ausnahme von [ISO11a] werden die Final Drafts des Standards referenziert, da die Beschaffung des Originalstandards mit nicht unerheblichen, finanziellen Aufwand verbunden ist.

[3] "T shall not be a referenceable table or a system-versioned table." [ISO11b]

[4] "C shall be a column of T and C shall not be the only column of T." [ISO11b]

von Spalten dürfen nur dahingehend geändert werden, dass der neue Typ dem aktuellen in Hinblick auf Länge, Genauigkeit, Maßstab etc. entspricht oder diesen erweitert.[5] Werden Spalten hinzugefügt, muss die Namenseindeutigkeit beachtet werden.[6] Weitere syntaktische, allgemeine, sowie Zugriffs- und Konformitätsregeln existieren für jede Klausel der alter-Anweisung.

Ein weiterer Aspekt der Schemaevolution sind **Defaultwerte**, die vorhandene Tupel beim Hinzufügen von Spalten anpassen. Es existiert generell die Möglichkeit, Nullwerte zu verwenden, allerdings können auch beliebige Zeichenketten, Katalogdaten, Nutzer-, Sitzungs- oder Systeminformationen, sowie Datentypspezifische Werte verwendet werden. Dieser Aspekt sollte bei der XML-Schemaevolution übernommen und XML-Schema-spezifisch angepasst werden.

In [HH06] wird das Problem der Datenbankschemaevolution und dessen Folgen im Kontext aktuell verwendeter Informationssysteme (i.A. relationale Datenbanken) aus Sicht von Entwicklern thematisiert. Die bereits vorgestellte *alter-table-Anweisung* aus SQL-Beispiel 3.1 wird unter anderem angewendet. Es werden Transformationen in drei Ebenen (d.h. konzeptuell, logisch und physisch) in einem "nicht temporalen Ansatz betrachtet, in welchem alle Komponenten (d.h. Schemata, Daten und Programme) durch eine neue Version ersetzt werden"[7] [HH06]. Besonders die Forderung der vollständigen **History** von Transformationen, sowie die konsekutive Analyse, Bereinigung und Normalisierung von dieser, sind Ansätze, die übernommen werden sollen. In diesem Zusammenhang wird die **eindeutige Identifikation** von sich ändernden Objekten gefordert. Ein abstrakter, eindeutiger Zeitstempel wird diesbezüglich eingeführt. Eine Identifikation von transformierten Schemakomponenten spielt ebenso in der XML-Schemaevolution eine wichtige Rolle und könnte mit Hilfe von Surrogatschlüsseln realisiert werden.

3.1.2. Objektorientierte Schemata

"Bei Datenbanken bewirkt die Einführung der Objekt-Orientierung eine Abkehr von zahlreichen Beschränkungen, die dem Anwender durch die vorher verfügbare (insbesondere die relationale) Technologie auferlegt wurden [..]." [LV96]

Neben einfachen Tabellen sind somit komplexe Strukturen und Attributwerte, sowie eine variable Semantik durch die Spezifikation von Methoden möglich. "Die Interpretation und Anzahl der objektorientierten Konzepte ist in den OODMs (objektorientierte Datenbankmodelle, Anm. d. Autors) sehr unterschiedlich." [Heu97] Grundanforderungen eines objektorientierten Datenbankmodells werden allerdings im "Manifesto" in [ABD+89] festgelegt, dazu zählen zwingende (z.B. komplexe Objekte, Objektidentität, Kapselung, Typen- und Klassenhierarchien, u.a.) und optionale Features (z.B. Mehrfachvererbung, Typprüfungen und -ableitungen, u.a.). Die

[5] New type "shall be greater than or equal to declared type of C" [ISO11b]

[6] "The <column name> in the <column definition> shall not be equivalent to the <column name> of any other column of T." [ISO11b]

[7] "non-temporal approach [..] all the application components [..] are replaced by new versions" [HH06]

Schemaevolution ist interessanterweise eines der Features, welches nicht eindeutig als zwingend oder optional eingestuft wurde. Die Anforderung der **Objektidentität** ist wiederum zur Identifikation von sich ändernden Objekten sehr hilfreich.

"Die Beschreibung der Struktur und das Verhalten einer Objektdatenbank wird als **Objektdatenbankschema** bezeichnet. Zu einem Objektdatenbankschema gehören demnach: Klassen- und Typdefinitionen, Spezialisierungshierarchien, Methoden(-implementierungen), Funktionen und Trigger (ECA-Regeln), Integritätsbedingungen, sowie Sichtdefinitionen." [SST97]

Die "langfristige Verwaltung von Daten impliziert, dass das DBMS (Datenbankmanagementsystem, Anm. d. Autors) mit sich ändernden Gegebenheiten umgehen kann. Es muss deshalb möglich sein, Änderungen der betrachteten Miniwelt oder der Anforderungen der Benutzer durch Schemaevolution nachzuvollziehen und existierende Daten an neue Schemata anzupassen." [Gep02]

Ein Überblick der Schemaevolution in objektorientierten Datenbanken mit deren zentralen Fragestellungen und Anforderungen wird in [Li99] gegeben. Eine Fragestellung bezieht sich auf die Möglichkeit der Anpassung mit Hilfe von Operationen. Diese **Schemaoperationen** können gemäß [Heu97] die Klasse bzw. Klassenintension (d.h. die Attributmenge oder Methoden), die Klassenhierarchie oder die Menge der Klassen ändern. In [SST97] wird eine ähnliche Auflistung gemacht, die entsprechende Liste der Operationen ist in Abbildung 3.2 dargestellt. Die Klas-

Evolutionsebene	Änderungsoperationen
Datenbankschema	neue Klasse einfügen
	bestehende Klasse löschen
	bestehende Klasse umbenennen
	neue Superklasse einfügen
	bestehende Superklasse entfernen
	Superklassenreihenfolge ändern
Klassenintension	neues Attribut hinzufügen
	bestehendes Attribut löschen
	bestehendes Attribut umbenennen
	Sichtbarkeit eines Attributs ändern
	Typ/Wertebereich eines Attributs ändern
	Defaultwert eines Attributs ändern
	neue Methode hinzufügen
	bestehende Methode löschen
	Sichtbarkeit einer Methode ändern
	Signatur einer Methode ändern
	Implementierung einer Methode ändern

Abbildung 3.2.: Auflistung relevanter Operationen der Schemaevolution nach [SST97]

sifikation von Operationen ist in der XML-Schemaevolution ebenso notwendig, besonders gilt dies für die Betrachtung und Adaption von Hierarchien und Attributmengen.

Es existieren unterschiedliche Schema-Evolutionsmechanismen, die Grundlage

von den später "verbreitetsten kommerziell vertriebenen Objekt-Datenbanksystemen" [Tre95] sind. Dazu zählen zum Beispiel ORION [Kim88] oder O_2 [FMZ+95]. In ORION "wurde der erste systematische Ansatz, Schemaevolution durch Konvertierung zu realisieren, vorgestellt" [Kol99]. Die Schemaänderungen von O_2 entsprechen denen von ORION, wobei der Schwerpunkt auf die strukturelle und verhaltensmäßige Konsistenz gelegt wird [Kol99]. "Bei der Änderung, Einfügung oder Löschung von Methoden (d.h. verhaltensmäßige Konsistenz, Anm. d. Autors) läuft diese Forderung auf eine inkrementelle Typüberprüfung hinaus." [Heu97] Eine **inkrementelle Typüberprüfung**, sowie ein **interaktives Dialogsystem** (Interactive Consistency Checker - ICC), das bei der Anwendung von Schemaänderungen in O_2 ausgeführt wird, sind Konzepte, die bei der XML-Schemaevolution verwendet werden können.

Ein für die automatisierte Änderung von Instanzen interessanter Ansatz ist der Mechanismus von COCOON [Tre95], bei dem Änderungen am Schema auf die Datenebene propagiert werden. "Die Änderungstaxonomie ähnelt wiederum der von ORION." [Kol99] Ein weiterer Aspekt ist die Einteilung der Operationen gemäß deren **Informationskapazität**. Eine Schemaevolution ist entweder kapazitätserhaltend, -erweiternd, -reduzierend oder -ändernd. Durch diese ebenso für die XML-Schemaevolution interessante Einteilung kann bereits frühzeitig entschieden werden, ob vorhandene Instanzen (d.h. die Datenbasis) angepasst werden müssen oder nicht. Des Weiteren wird mit COOL-SML (Schema Manipulation Language) eine **Schemaevolutionssprache** eingeführt, die die deklarative Beschreibung der Evolution, die Überprüfung von Strukturregeln, die Übersetzung in Elementaroperationen, sowie die Reoptimierung von impliziten Hierarchien ermöglicht.

"Eine beliebige Restrukturierung von Typen ist in den meisten Schema-Evolutionsmechanismen nicht vorgesehen." [Heu97] Darüber hinaus werden die sowieso nur rudimentären Schemaänderungsoperationen teilweise abgelehnt, wenn Instanzen vorliegen. Eine Typänderung in O_2 kann zum Beispiel "nur dann sinnvoll unterstützt werden, wenn alle Objekte der betroffenen Klasse vorher gelöscht werden" [SST97]. Der damit verbundene, manuelle Aufwand zur Zwischenspeicherung und erneuten Zuordnung der Datenbasis kann nicht verlangt werden. Des Weiteren werden im Allgemeinen nicht alle Konzepte eines Objektdatenbankschemas berücksichtigt, dazu gehören unter anderem die Integritätsbedingungen oder Sichtdefinitionen. Zum Beispiel werden im weitreichenden Ansatz bzw. Mechanismus in [Tre95] schon einfache Constraints wie Kardinalitäten weggelassen, damit keine Widersprüche oder Zyklen in der Anwendung entstehen.

3.1.3. Document Type Description - DTD

Die Evolution der Document Type Definition (DTD) [BPSM+08] wird in [SKC+01] thematisiert. Ausgehend von den Schemaoperationen, die in ORION [Kim88] eingeführt wurden, werden **DTD Change Primitives** und **XML Data Operati-**

ons vorgestellt. Diese sind in dem **XML Evolution Manager (XEM)** umgesetzt und sollen die Konsistenz von XML-Dokumenten bei strukturellen oder Constraint betreffenden Änderungen erhalten, sowie gültige DTDs erzeugen.

Die DTD Change Primitives werden unterteilt in Änderungen bezüglich der Definition des Dokuments bzw. der Element- und Attributtypen. Eine solche **Operationsunterteilung von Schemakomponenten** sollte ebenso in der XML-Schemaevolution übernommen werden. Die Operationen sind hier allerdings sehr eingeschränkt, was durch die Definition von Vorbedingungen bestimmt wird. So dürfen keine beliebigen Elemente gelöscht werden[8], neue bzw. zwingende Elemente oder Attribute benötigen generell Defaultwerte[9] und komplexe Elemente (d.h. Gruppen) können nicht auf zwingend gesetzt werden[10].

Die XML Data Operations dienen der Anpassung von XML-Dokumenten und resultieren aus den obigen Primitiven. Diese werden in [SKC+01] nur kurz angeführt, sind allerdings in [Kra01] bzw. [SKR02] ausführlicher charakterisiert.[11] Es gibt laut [SKC+01] Einfüge-, Lösch- und Änderungsoperationen für Elemente und Attribute eines XML-Dokuments. Die Identifizierung von den entsprechenden Positionen innerhalb einer Instanz geschieht unter Anwendung von **XPath-Ausdrücken**. **Lokationspfade** werden im Abschnitt 2.2 vorgestellt und gleichfalls in der XML-Schemaevolution angewendet.

Die XML Data Operations werden in **Exemplar**, der Prototyp-Implementation von XEM, abschließend abgeändert. Aufgrund der Implementierung von Attributen als Membervariablen in Java wird das Löschen dem Nullsetzen und das Einfügen dem modifiziertem Ändern gleichgesetzt. Somit verbleibt neben den Elementoperationen explizit nur die Änderung von Attributen als XML Data Operation.

In [LHBM06] wird ein alternativer Ansatz zur Ermittlung von DTD Änderungen vorgestellt, dies ist der Algorithmus **DTD-Diff**. Ausgehend von zwei DTDs werden durch ein Matching übereinstimmende Paare von Elementtyp- (ETD), Attribut- (AD) und Entitätsdeklarationen (ED) im **DTD Data Model** ermittelt. Die dabei anwendbaren Operationen werden in [LHBM05] erläutert und sind in Abbildung 3.3 dargestellt. Es ist möglich, Kardinalitäten (*cardinality*), komplexe Inhalte als Teilbäume[12](*leaf node* und *subtree*), die Sortierung (*order*), sowie spezielle externe Entitäten (*external ED*) zu ändern. Des Weiteren wird die Verschiebung von ganzen Teilkomponenten durch die *Move*-Operation ermöglicht. Die eingeführten Operationen sind ebenso in der XML-Schemaevolution notwendig.

DTD-Diff erzeugt aus dem Matching ein **Änderungsskript** zur Überführung der einen DTD in die andere. Die dazu notwendigen Operationen werden in folgender Reihenfolge analysiert: Verschieben (*move*), Löschen (*delete*), Einfügen (*in-*

[8] "Element E must be a non-nested element with empty or PCDATA content model." [SKC+01]
[9] "The default value must not be null." [SKC+01]
[10] "We not allow the new quantifier to represent a required constraint if the old did not." [SKC+01]
[11] Die Operationen unterscheiden sich im Vergleich, die ausführlichste Liste befindet sich in [SKC+01].
[12] Die Autoren in [LHBM05] repräsentieren den Inhalt von Elementtypen als Bäume.

Element Type Decl. (ETD)	Attribute Declaration (AD)
Insertion of a new ETD	Insertion of a new AD
Deletion of an ETD	Deletion of an AD
Insertion of a leaf node	Insertion of a new attribute
Deletion of a leaf node	Deletion of an attribute
Insertion of a subtree	Update of attribute type
Deletion of a subtree	Update of default value
Move a leaf node	**Entity Declaration (ED)**
Move a subtree	Insertion of a new ED
Update of order	Deletion of an ED
Insertion of cardinality	Update of replacement text of internal ED
Deletion of cardinality	Update of location of external ED
Update of cardinality	Update of content notation of external ED

Abbildung 3.3.: Typen von Änderungen des DTD Data Models nach [LHBM05]

sert), Kardinalitätsänderung (*cardinality update*), Umsortieren (*local order move*), Attributlistenänderung (*attribute list change*), sowie Entitätsänderung (*entity declaration change*). Diese Reihenfolge wird nicht weitergehend erläutert, allerdings ist diese notwendig, da ansonsten zum Beispiel eine Verschiebung durch ein Löschen und Einfügen realisiert werden könnte und somit überflüssig wäre.

In [LHBM06] wird das möglichst effiziente Erkennen von Änderungen thematisiert. Allerdings wird explizit darauf hingewiesen, dass die erkannten Änderungen als Grundlage für die Anpassung von XML-Dokumenten, zur inkrementellen Anpassung von relationalen Schemata und/oder zur XML Schema Integration genutzt werden können. Die Herleitung von Anpassungen von XML-Dokumenten aus Schemaänderungen ist ein wesentlicher Bestandteil der XML-Schemaevolution.

3.2. Aktuelle Ansätze der XML-Schemaevolution

Zu den aktuellen Ansätzen gehören die Möglichkeiten der XML-Schema-spezifischen Systeme. Es werden in **Abschnitt 3.2.1** die XML Funktionalitäten der großen Datenbankhersteller untersucht, bevor in **3.2.2** das XML-Tool *DiffDog von Altova* bezüglich der XML-Schemaevolution analysiert wird. Die spezialisierten XML-Datenbanken werden in **3.2.3** erläutert.

Anschließend werden die Ansätze und Prototypen anderer Forschungsgruppen beschrieben. Dies sind *X-Evolution* in **Abschnitt 3.2.4**, das *GEA-Framework* in **3.2.5** und *XCase* in **3.2.6**. Es werden die entsprechenden Publikationen des Umfelds der zugehörigen Gruppen unter den Aspekten der in Kapitel 1.1 thematisierten Problemstellung betrachtet. Dies beinhaltet gemäß Abbildung 1.1 hauptsächlich die Art und Weise der Erfassung von Schemaänderungen (*Änderungen*), deren Analyse und Charakterisierung (*Bestimmung der Änderungen*), sowie die Anpassung der XML-Dokumente (*Adaption*).

In **Abschnitt 3.2.7** werden weitere Arbeiten vorgestellt, die so nicht in die Struktur der Arbeit passen, aber interessante Ansätze und Strategien beinhalten.

3.2.1. XML-Schema in Datenbanksystemen

Die Datenbanken von **Oracle** (12c Release 1 [Ada14]), **IBM** (DB2 LUW v10.5 [IBM13]) und **Microsoft** (SQL Server 2014 [Mic14]) bieten jeweils eine sehr große Vielfalt an Funktionalitäten. Diese beinhalten nicht nur das klassische Relationenmodell oder objektrelationale Aspekte, es werden auch XML spezifische Inhalte angeboten. Dies ist unter anderem dem Standard ISO/IEC 9075-14 [ISO11c] geschuldet, in welchem "Wege definiert werden wie die Datenbanksprache SQL in Verbindung mit XML genutzt werden kann"[13] [ISO11c].

In [Sch03] wurden die Versionen *Oracle 9i Release 2*[14] , *IBM UDB mit XML Extender 7*[15] und *Microsoft SQL Server 2000*[16] bezüglich deren Möglichkeiten zur Schemaevolution untersucht. IBM und Microsoft ermöglichten diese Eigenschaft. In Übereinstimmung mit dem vorgestellten Umfang von Oracle in [CSK01] wurde vermerkt, dass diese Funktionalität nicht vorgesehen sei.

In aktuelleren Versionen dieser Datenbanken ist dies allerdings verändert, so wird zum Beispiel in [Ora09] im direkten Vergleich von *Oracle 11g* und *IBM LUW v9.5* der Oracle-Datenbank die Evolutionsmöglichkeit zugesprochen.[17]

Oracle Database

Oracle XML DB unterstützt zwei Möglichkeiten zur Durchführung einer XML-Schemaevolution. Dazu wurden jeweils PL/SQL (Procedural Language/Structured Query Language) Prozeduren umgesetzt, welche in [Ada14] nebst entsprechender Einschränkungen und Leitfäden erläutert werden.[18]

Die **Copy-Based Schema Evolution** wird durch die Prozedur *DBMS_XMLSCHEMA.copyEvolve* ermöglicht, durch die "alle Instanzdokumente, die zum alten XML-Schema gültig sind, in einen temporären Bereich kopiert werden. Das alte Schema wird gelöscht und das modifizierte XML-Schema wird registriert, bevor die Instanzdokumente aus dem temporären Bereich an die entsprechende Stelle zurück geschrieben werden."[19] [Ada14]

Der Nachteil dieser Methode ist, dass alle XML-Schemas sowie zugeordnete XML-Dokumente während der Evolution kopiert und gelöscht werden. Dies ist deaktivierbar, dennoch wird explizit auf die Erzeugung eines Backups vor Anwendung der Prozedur hingewiesen.[20] Trotz Backups gehen allerdings weitergehende

[13] "defines ways in which Database Language SQL can be used in conjunction with XML" [ISO11c]
[14] "Oracle 9i Release 2: Eigenschaft Schema-Evolution - Grad der Erfüllung nicht vorgesehen" [Sch03]
[15] "IBM UDB mit XML Extender 7.2: Schema-Evolution - Grad der Erfüllung nicht möglich" [Sch03]
[16] "Microsoft SQL Server 2000: Eigenschaft Schema-Evolution - Grad der Erfüllung möglich" [Sch03]
[17] "Support for Schema Extension, Versioning and Evolution - Oracle Yes / IBM Limited" [Ora09]
[18] In [Def12] werden die Oracle PL/SQL Prozeduren zusätzlich mit einem komplexeren Beispiel getestet.
[19] "Copy-based schema evolution, in which all instance documents that conform to the schema are copied to a temporary location in the database, the old schema is deleted, the modified schema is registered, and the instance documents are inserted into their new locations from the temporary area." [Ada14]
[20] "Before executing procedure DBMS_XMLSCHEMA.copyEvolve, always back up all registered XML

Datenbankstrukturen mit Bezug zum Schema wie Indizes, Triggers, Constraints, Metadaten vom XML Typtabellen, usw. generell verloren.

Neben diesen Problemen existieren Einschränkungen bezüglich der Schemaänderungen, so sind Umbenennungen oder Löschungen von globalen Elementen nicht vorgesehen und verlangen einen manuellen Eingriff.[21] Dies ist besonders bei global dominierten Modellierungsstilen[22] problematisch. Wird die Gültigkeit von XML-Dokumenten nach dem Einfügen aus dem temporären Bereich verletzt, wird ein XSL-Dokument (Extensible Stylesheet Language) benötigt, welches die Dokumente entsprechend transformiert.[23] Diese Transformationsskripte müssen wiederum erzeugt und vom Anwender bereitgestellt werden. Die in der vorliegenden Arbeit angestrebte XML-Schemaevolution wird nicht durch die *Copy-Based Schema Evolution* der Prozedur *DBMS_XMLSCHEMA.copyEvolve* realisiert.

Als zweite Möglichkeit wird die **In-Place XML Schema Evolution** mit der Prozedur *DBMS_XMLSCHEMA.inPlaceEvolve* bereitgestellt, bei der "kein Kopieren, Löschen und Einfügen existierender Daten notwendig ist. Dadurch ist diese Evolution viel schneller im Vergleich zur vorherigen Möglichkeit. Allerdings gilt unter anderem die Einschränkung, dass vorhandene Dokumente durch Schemaänderungen nicht ungültig werden dürfen."[24] [Ada14]

Die *In-Place Evolution* erzeugt eine neue Version des XML-Schemas durch die Anwendung der Änderungen eines *diffXML-Dokuments*, welches vom Anwender entsprechend bereitgestellt werden muss. Dies stellt neben der zwingenden *Rückwärtskompatibilitätsanforderung*[25] erneut einen Nachteil dar, da ein solches Dokument erzeugt werden muss. Läuft die Evolution erfolgreich, wird das alte Schema gelöscht, sodass auch in diesem Fall explizit auf "die Erschaffung eines Backups vor Ausführung hingewiesen wird. Dieses wird auch empfohlen, da eventuell nicht gewollte, aber korrekte Änderungen nicht zurück genommen werden können bzw. umkehrbar sind."[26] [Ada14] Die Prozedur ermöglicht Testläufe von *diffXML-Dokumenten* im *Trace Mode*, bei denen die Evolution ohne abschließende, dauerhafte Ersetzung des alten Schemas sowie der Anpassung interner Strukturen durch DDL

schemas and all XML documents that conform to them." [Ada14]

[21] "Procedure DBMS_XMLSCHEMA.copyEvolve assumes that top-level elements have not been dropped and that their names have not been changed in the new XML schemas." [Ada14]

[22] siehe auch: Kapitel 2.1.3 (Modellierungsstile von XML-Schema)

[23] "After you modify a registered XML schema, you must update any existing XML instance documents that use the XML schema. You do this by applying an XSLT stylesheet to each of the instance documents. The stylesheet represents the difference between the old and new XML schemas." [Ada14]

[24] "In-place schema evolution, which does not require copying, deleting, and inserting existing data and thus is much faster than copy-based evolution, but which has restrictions that do not apply to copy-based evolution. In general, in-place evolution is permitted if you are not changing the storage model and if the changes do not invalidate existing documents." [Ada14]

[25] "a given XML schema can be evolved in place in only a backward-compatible way" [Ada14]

[26] "Make sure that you back up your data before performing in-place XML schema evolution, in case the result is not what you intended. There is no rollback possible after an in-place evolution. If any errors occur during evolution, or if you make a major mistake and need to redo the entire operation, you must be able to go back to the backup copy of your original data." [Ada14]

(Data Definition Language) Ausdrücke erfolgt. Ein Anwender könnte sich daher an das erwünschte Ergebnis schrittweise annähern.

Die Einschränkung der erlaubten Schemaänderungen durch die Rückwärtskompatibilität ist wie bereits erwähnt ein weiterer Nachteil dieser Prozedur. Die unterstützten Schemaänderungen sind in Abbildung 3.4 aufgelistet. Es ist im Allgemei-

Schemaoperation	Bedingung
Optionales Element in komplexen Typen oder Gruppe einfügen	
Optionales Attribut in komplexen Typen oder Attributgruppe einfügen	
Einfachen Typen in komplexen Typen mit einfachem Inhalt ändern	Speichermodell ist binäres XML
Existierenden maxLength Attributwert ändern	Erhöhung des Werts
Zusätzlichen Aufzählungswert einfügen	Am Ende der Aufzählungsliste
Globales Element einfügen	
Globales Attribut einfügen	
Globalen, komplexen Typen einfügen oder löschen	
Globalen, einfachen Typen einfügen oder löschen	
minOccurs Attributwert ändern	Verminderung des Werts
maxOccurs Attributwert ändern	Erhöhung des Werts UND Speichermodell ist binäres XML
Globale Gruppe oder Attributgruppe einfügen oder löschen	
xdb:defaultTable Attributwert ändern	Nicht auf Wert eines anderen xdb Namensraumattributs
Kommentar oder Verarbeitungsanweisung einfügen, löschen oder ändern	

Abbildung 3.4.: Unterstützte Operationen der In-Place Evolution nach [Ada14]

nen möglich optionale Komponenten in vorhandene Strukturen einzufügen, globale Komponenten zu deklarieren und zu definieren oder kapazitätserweiternde Änderungen (u.a. Verminderung von *minOccurs* und/oder Erhöhung von *maxOccurs*) zu vollziehen. Globale Typen und Gruppen können ebenso gelöscht werden, was im ersten Fall aufgrund der vorhandenen Typhierarchien[27] und einer Kompensation durch den *anyType* möglich ist. Im zweiten Fall dürfte dies allerdings aufgrund der notwendigen Rückwärtskompatibilität nur für leere oder nicht referenzierte Komponenten zulässig sein.

Die angestrebte XML-Schemaevolution wird aufgrund der geringen Anzahl unterstützter Operationen nicht durch die *In-Place XML Schema Evolution* der Prozedur *DBMS_XMLSCHEMA.inPlaceEvolve* realisiert. Somit bietet die Datenbank von **Oracle** (12c Release 1 [Ada14]) aufgrund der Einschränkungen nicht die gewünschte Funktionalität, dennoch ist im gewissen Umfang und mit zusätzlichen, manuellen Aufwand eine XML-Schemaevolution möglich.

IBM DB2

In *IBM pureXML* werden XML-Schemata nach deren Registrierung als *XSR-Objekte* (XML-Schema-Repository) behandelt. "Ein im XML-Schema-Repository registriertes Schema kann zu einem neuen, kompatiblen XML-Schema weiterentwickelt werden, ohne dass bereits gespeicherte XML-Instanzdokumente erneut auf Gültigkeit geprüft werden müssen." [IBM13] "Um ein XML-Schema im XML-Schema-Repository weiterentwickeln zu können, müssen das ursprüngliche XML-

[27] siehe auch: Kapitel 2.1.1 (Strukturbeschreibung des XML-Schemas) Abbildung 2.1 bzw. Kapitel 2.1.2 (XML-Schema Version 1.1) Abbildung 2.3

Schema und das neue, für die Aktualisierung verwendete XML-Schema hinreichend ähnlich sein. Wenn die beiden XML-Schemata nicht kompatibel sind, schlägt die Aktualisierung fehl und es wird eine Fehlernachricht generiert." [IBM13]

Zur Sicherstellung der Kompatibilität werden in [IBM13] zehn Anforderungen erläutert, welche in Abbildung 3.5 aufgelistet werden. In Abbildung 3.5 wird je-

Kriterium	Altes Schema	Neues Schema
Attributinhalt	Attribut vorhanden	Muss vorhanden sein
	Attribut nicht vorhanden	Neues Attribut darf nur optional sein
Elementinhalt	Element vorhanden	Muss vorhanden sein
	Element nicht vorhanden	Neues Element darf nur optional sein
Fassettenkonflikt	Einfacher Typ vorhanden	Wertebereich muss trotz Änderungen von Fassetten kompatibel sein
Inkompatibler Typ	XML-Instanzen gültig	Erneute Gültigkeitsprüfung darf nicht scheitern
	Annotation vorhanden	Annotation darf sich nicht unterscheiden
Mixed-Content (Inhaltsmodell)	Attribut mixed='true'	Attribut mixed='true' muss gegeben sein
Nullwertfähigkeit	Attribut nillable='true'	Attribut nillable='true' muss gegeben sein
Entferntes Element	Global deklariertes Element	Muss vorhanden und nicht abstrakt sein
Entfernter Typ	Global definierter Typ	Muss vorhanden sein
Einfacher und komplexer Inhalt	Einfacher Inhalt gegeben	Neuer Inhalt darf nicht komplex sein
Einfacher Typ	Basisdatentyp gegeben	Basisdatentyp muss übereinstimmen

Abbildung 3.5.: Kompatibilitätsanforderungen nach [IBM13]

weils ein Kriterium mit deren Anforderungen an das alte und neue, gegebenenfalls modifizierte XML-Schema gegeben. Zum Beispiel müssen Element- und Attributdeklarationen, die im alten Schema vorhanden waren, ebenso im neuen Schema vorhanden sein. Werden neue Deklarationen eingefügt, so dürfen diese nur optional sein. Im Allgemeinen sind somit keine Schemaoperationen erlaubt, die die Gültigkeit vorhandener XML-Dokumente verletzen könnten. Dieses entspricht der Rückwärtskompatibilität der *Oracle In-Place Evolution*.

Die in der vorliegenden Arbeit angestrebte XML-Schemaevolution wird wegen der geringen Anzahl unterstützter Operationen nicht realisiert. Die Datenbank von **IBM** (DB2 LUW v10.5 [IBM13]) bietet somit aufgrund der restriktiven Kompatibilitätsanforderungen nicht die gewünschte Funktionalität.

Microsoft SQL Server

"*SQL Server* bietet eine leistungsstarke Plattform zum Entwickeln umfassender Anwendungen zur Verwaltung halbstrukturierter Daten. Alle Komponenten in SQL Server bieten XML-Unterstützung." [Mic14]

"XML-Werte können systemeigen in einer xml-Datentypspalte [sic] gespeichert werden, die gemäß einer Auflistung von XML-Schemas typisiert oder nicht typisiert werden kann." [Mic14] "SQL Server verwendet die zugeordnete XML-Schemaauflistung außerdem im Fall von typisiertem xml [sic], um die XML-Instanz zu überprüfen. Wenn die XML-Instanz dem Schema entspricht, lässt die Datenbank das

Speichern der Instanz und ihrer Typinformation im System zu. Anderenfalls wird die Instanz abgelehnt." [Mic14]

Eine Schemaauflistung wird mit Hilfe der *CREATE-XML-SCHEMA-COLLECTION-Anweisung* erzeugt, wobei ein oder mehrere XML-Schemata importiert werden können. Es werden bei Ausführung der Anweisung "verschiedene Schemakomponenten in die Datenbank importiert. Zu den Schemakomponenten gehören Schemaelemente, -attribute und -typdefinitionen."[28] [Mic14] Ein XML-Schema wird demnach nicht als vollständige Datei gespeichert, sondern ist "wie eine Tabelle in der Datenbank eine Metadatenentität" [Mic14]. Mit der Anweisung *ALTER XML SCHEMA COLLECTION* können zu vorhandenen Schemata weitere Komponenten oder zu Auflistungen neue Schemas hinzugefügt werden.

"Die Anweisung *DROP XML SCHEMA COLLECTION* löscht alle in der Auflistung enthaltenen Schemas und entfernt das Auflistungsobjekt." [Mic14] Diese Löschung ist allerdings nur dann möglich, wenn die Auflistung weder in einer typisierten Spalte zugeordnet, noch in einer Tabelleneinschränkung (*table constraint*) angegeben, noch in einer schemagebundenen Funktion oder gespeicherten Prozedur referenziert ist.

Ein Nachteil der Methode ist, dass gemäß [Mic14] vorhandene Komponenten nicht nachträglich verändert, sondern nur als neue Komponente mit angepassten Namensraum registriert werden. Des Weiteren existieren allgemein Einschränkungen bezüglich der Schemaauflistungen auf dem Server. Es werden zum Beispiel keine sekundären Schemakomponenten[29] wie Identity-Constraints (*<key>*, *<keyref>* und *<unique>*)[30], die unbeschränkte Häufigkeit (*maxOccurs = unbounded*)[31] oder Einschränkungen vom einfachen Vereinigungstypen[32] unterstützt.

Ein weiterer Nachteil ist, dass nur komplette Auflistungen entfernt werden können. Dies ist auch nur dann möglich, wenn unter anderem keine zugeordneten XML-Instanzen in Form typisierter Spalten existieren. Die Löschung einzelner Komponenten ist nicht vorgesehen.

Somit sind nur kapazitätserweiternde Schemaoperationen auf einem reduzierten Umfang möglicher Schemakomponenten von XML-Schema realisiert, die die Gültigkeit vorhandener XML-Instanzen nicht verletzen. Die in der Arbeit angestrebte XML-Schemaevolution ist demnach nicht durch die Datenbank von **Microsoft** (SQL Server 2014 [Mic14]) umgesetzt.[33]

[28] "Die in der Datenbank gespeicherten Schemakomponenten fallen in folgende Kategorien: Element, Attribut, TYPE (für einfache oder komplexe Typen), ATTRIBUTEGROUP, MODELGROUP" [Mic14]

[29] siehe auch: Kapitel 2.1.1 (Strukturbeschreibung des XML-Schemas)

[30] "Zurzeit unterstützt SQL Server diese XSD-basierten Einschränkungen zum Erzwingen der Eindeutigkeit oder zum Einrichten von Schlüsseln oder Schlüsselverweisen nicht. XML-Schemas, die diese Elemente enthalten, können nicht registriert werden." [Mic14]

[31] "Die Werte für minOccurs- und maxOccurs-Attribute müssen in ganze 4-Byte-Zahlen passen. Schemas, die diese Bedingung nicht erfüllen, werden vom Server zurückgewiesen." [Mic14]

[32] "SQL Server unterstützt keine Einschränkungen aus union-Datentypen." [Mic14]

[33] "Die direkte Anpassung von XML-Schema auf dem Server wird nicht unterstützt, es wird davon ausgegangen, dass eine Anwendung existiert, die das übernimmt (vgl. [..] und [Cas09])." [Def12]

3.2.2. Altova DiffDog

"Mit XMLSpy ([Alt15d], Anm. d. Autors) und den anderen preisgekrönten XML-Tools von Altova steht Entwicklern eine robuste und umfassende Umgebung zum Erstellen von Applikationen, in denen XML und zusätzliche Technologien [..] zum Einsatz kommen, zur Verfügung." [Alt15c]

Besondere Bedeutung im Zusammenhang mit der XML-Schemaevolution besitzt das XML-Tool **DiffDog** [Alt15a], mit welchem unter anderem strukturelle Unterschiede zwischen zwei XML-Schemata ermittelt werden können. "Ein Unterschied zwischen zwei verglichenen XML-Schemas tritt auf, wenn in einem XML-Schema Elemente existieren, die im anderen XML-Schema fehlen, oder wenn der Name der Elemente unterschiedlich sind." [Alt15a] "Zusammen mit den Funktionen zum Vergleich von XML-Dateien bietet DiffDog XML-Schemavergleichsfunktionen zum Anpassen von XML-Dateien an geänderte XML-Schemas." [Alt15b]

Es wird in DiffDog ein *Mapping* erstellt, wobei dieser Prozess jedem Element in der linken Vergleichskomponente (*Ausgangsschema*) ein Element in der rechten Vergleichskomponente (*Zielschema*) zuordnet. Dies kann entweder automatisch durch DiffDog oder manuell vom Anwender erfolgen. Abbildung 3.6 veranschaulicht ein solches Mapping, wobei dieses durch blaue bzw. graue Verbindungen zwischen den jeweils zugeordneten Komponenten dargestellt ist. Das Mapping er-

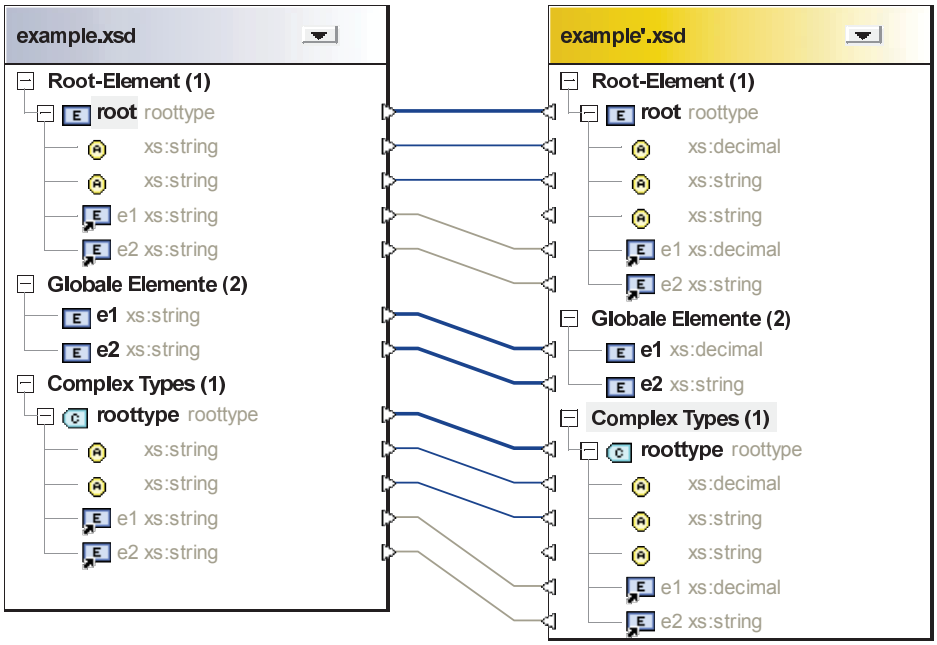


Abbildung 3.6.: DiffDog-Mapping eines Ausgangs- und Zielschemas

folgte in diesem Fall automatisch, nachdem ein globales Element als Root-Element spezifiziert wurde. Diese Notwendigkeit ist bei global dominierten Modellierungs-

stilen[34] mit einer großen Anzahl von globalen Elementen problematisch, da die Auswahl einen großen Einfluss auf nachfolgende Prozesse hat. Die verwendeten XML-Schemata sind in den XML-Beispielen A.1 und A.2 in bekannter Form abgebildet. Die Schemata unterscheiden sich im einfachen Datentyp des Elements *e1* und Attributs *a1* (*xs:string* bzw. *xs:decimal*), im Inhaltsmodell des komplexen Typen *roottype* (*xs:sequence* bzw. *xs:choice*) und im *use-Attribut* des Attributs *a3* im *roottype* (*prohibited* bzw. *required*).

Aus dem Mapping kann ein XSLT-Dokument (Extensible Stylesheet Language Transformation) [Kay01] erzeugt werden, "damit die Änderungen im XML-Schema auch in jenen XML-Dateien zur Anwendung kommen, die früher mit Hilfe dieses Schemas erzeugt wurden" [Alt15a]. Das XSLT-Dokument ist als XML-Beispiel A.3 im Anhang dargestellt und wurde *von links nach rechts generiert*.[35]

Das XML-Beispiel 3.2 ist ein gültiges XML-Dokument zum Ausgangsschema.

```
<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="example.xsd"
    a1="a1" a2="a2">
    <e1>e1</e1>
    <e1>e1</e1>
    <e2>e2</e2>
    <e2>e2</e2>
</root>
```

XML-Beispiel 3.2: Gültiges XML-Dokument für XML-Schema des XML-Beispiels A.1

Es sind die Attribute *a1* und *a2*, sowie die Elemente *e1* und *e2* enthalten, wobei entsprechende Werte passend zum einfachen Datentyp *xs:string* vergeben wurden. Des Weiteren enthält das Element *root* die zur Gültigkeitsprüfung notwendigen Informationen zum XML-Schema (Ausgangsschema: *example.xsd*).

Das XML-Dokument 3.2 muss angepasst werden, damit es zum Zielschema gültig ist. Eine Möglichkeit ist in XML-Beispiel 3.3 dargestellt.

```
<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="example'.xsd"
    a1="1" a2="a2" a3="">
    <e1>1</e1>
    <e1>1</e1>
</root>
```

XML-Beispiel 3.3: Gültiges XML-Dokument für XML-Schema des XML-Beispiels A.2 (manuell konvertiert aus XML-Dokument des XML-Beispiels 3.2)

Wichtige Anpassungen sind: Attribut *a1* benötigt einen Dezimalwert (*xs:decimal*), Attribut *a3* muss gegeben sein, das Inhaltsmodell ist nun eine Auswahl (*xs:choice*), das heißt entweder wird das Element *e1* (mit zugehöriger Anpassung auf den neuen Datentypen *xs:decimal*) oder *e2* behalten, und zuletzt muss eine Änderung der Information über das zugeordnete Schema erfolgen (Zielschema: *example'.xsd*).

[34] siehe auch: Kapitel 2.1.3 (Modellierungsstile von XML-Schema)

[35] Rechts nach links liefert das gleiche Ergebnis; DiffDog-Dateivergleich: "Keine Unterschiede gefunden!"

```
<root a1="a1" a2="a2">
    <e1>e1</e1>
    <e1>e1</e1>
    <e2>e2</e2>
    <e2>e2</e2>
</root>
```

XML-Beispiel 3.4: XML-Dokument nach Anwendung des DiffDog XSLT-Dokuments (ausgehend von XML-Dokument des XML-Beispiels 3.2)

Das XML-Beispiel 3.4 zeigt das Ergebnis der Anwendung des von DiffDog erzeugten XSLT-Dokuments. Die Anwendung erfolgte mit Hilfe von XMLSpy. Das Ergebnis ist ein ungültiges XML-Dokument, bei welchem die Änderungen des Ausgangsschemas nicht entsprechend auf vorhandene Instanzen angewendet wurden. Es fehlen notwendige Schemainformationen für eine Gültigkeitsprüfung, die Datentypen wurden nicht beachtet (*a1* und *e1* mit *xs:decimal*), das neue Inhaltsmodell (*xs:choice*) des komplexen Typen *roottype* wurde nicht angepasst und das zwingende Attribut *a3* wurde nicht ergänzt.

Dies ist allerdings durch die Analyse des XSLT-Dokuments nachvollziehbar. Unter Verwendung von *Transformationsregeln* (*Templates*: <xsl:template>) wird in Beachtung der vorliegenden, erlaubten Elemente und Attribute der komplette Inhalt eines XML-Dokuments kopiert. Dabei erfolgt eine Identifizierung über die Namen der Komponenten. Eine Anpassung von Werten findet allerdings nicht statt, sondern es wird die vorliegende Zeichenkette (oder der Textknoten) unverändert übernommen. Sind Komponenten im Zielschema nicht länger vorhanden oder werden ergänzt, fehlt ein entsprechendes Template. Wird eine Änderung in der Semantik (in dem Fall Auswahl *xs:choice* statt Sequenz *xs:sequence*) vollzogen, wird dies nicht erkannt[36], sondern in der Reihenfolge der Komponenten des Ausgangsschemas alles kopiert. Somit ist auch zu erklären, dass Element *e1* vor *e2* im transformierten XML-Dokument auftaucht. Warum allerdings solch elementare Attribute wie die Schemainformationen nicht mit übernommen und angepasst werden, ist nicht verständlich. Diese sind innerhalb des XML-Schemas zwar nicht vorhanden, sodass kein Template erstellt wird, aber zur Anpassung an ein geändertes XML-Schema wäre dies für eine Gültigkeitsprüfung zwingend notwendig.

Das XML-Tool **DiffDog** [Alt15a] von Altova realisiert die in der Arbeit angestrebte XML-Schemaevolution nicht. Es ist möglich durch ein Mapping unterschiedlicher Versionen eines XML-Schemas ein XSLT-Dokument zu erzeugen, "wodurch die Konvertierung von XML-Daten erleichtert wird" [Alt15b]. Die Gültigkeit bezüglich des Zielschemas nach dessen Änderung ausgehend vom Ausgangsschema wird allerdings nicht erreicht, wie mit Hilfe eines kleinen Beispiels gezeigt wurde.

[36] Dies ist scheinbar auch nicht vorgesehen, da es um den strukturellen XML-Schemavergleich geht.

3.2.3. XML-Datenbanken

"Eine XML-Datenbank ist eine Datenbank, deren kleinste logische Einheit ein XML-Dokument ist, d.h. XML ist das grundlegende logische Konstrukt, auf dem die Datenbank aufgebaut ist." [Dat12]

In [ST01] werden Anforderungen an XML-Datenbanken gestellt, dazu zählen ein gut definiertes Datenmodell[37], sowie die Möglichkeiten der Datendefinition und Datenmanipulation. Im Zusammenhang mit der Datenmanipulation werden Transformationen thematisiert, wobei die Schemaevolution explizit als eines der Anwendungsszenarien erwähnt wird.[38]

Eine umfangreiche Übersicht über *native XML-Datenbanken* und Produkte wird in [Ron10] gegeben.[39] Eine reduzierte, allerdings aktuellere Teilmenge von Datenbanken ist in [Wik15] aufgelistet. Aus dieser Übersicht wurden die nachfolgenden nativen XML-Datenbanken ausgewählt: **Tamino** (v4.2.1 [Sch05]), **Sedna** (v3.5 [Sed11]) und **eXist-db** (v2.2 [eXi14a]).[40]

Tamino

Tamino wurde bereits in [ST01] analysiert und ist im Vergleich zu ähnlich betagten Produkten dahingehend interessant, dass explizit auf die Schemaevolution eingegangen wird. In [Sch05] heißt es: "Ist Schema Evolution unterstützt? Ja. Das neue Schema sollte so sein, dass bestehende Dokumente nicht ungültig werden. Der Benutzer ist für die Validierung der bestehenden Dokumente gegen das neue Schema verantwortlich."

Somit ist ein Anwender entweder für die manuelle Anpassung nicht schemakonformer Dokumente verantwortlich, oder er verzichtet auf Schemaoperationen, die die Gültigkeit vorhandener XML-Dokumente verletzen könnten. **Tamino** (v4.2.1 [Sch05]) realisiert die in der Arbeit angestrebte XML-Schemaevolution nicht.

[37] "A well-defined database system is based on a well-defined data model." [ST01]

[38] "Schema evolution. [..] Because changes in document type definitions are quite common, there is often a need to transform existing data to correspond to a new definition." [ST01]

[39] "A number of native XML DBMSs have been suggested [Ron10]." [And08]

[40] Es wurde unter Beachtung von [Wik15] auf Entwicklungen von **Oracle** und **IBM** in Hinblick auf Kapitel 3.2.1 (XML-Schema in Datenbanksystemen) verzichtet, ebenso auf das "eingefrorene **MonetDB/XQuery** Projekt" [mon11] und das "sich im Ruhestand befindende **Xindice**" [Apa11]. **BaseX** ist laut [Grü10] eine "ausgereifte XML-Speicher- und Query-Architektur", bei der wohlgeformte Dokumente zwingend notwendig sind ("The command (Database Operation: CREATE DB, Anm. d. Autors) fails [..] if one of the documents to be added is not well-formed [..]" [Bas15]), Schemakonformität allerdings eher als optional angesehen wird ("It is one key feature of XML that no schema needs to be defined at all [..]" [Grü10]).

Sedna

Sedna als weitere native XML-Datenbank bietet eine Update-Sprache, welche auf der Diplomarbeit von Patrick Lehti [Leh01] basiert.[41] Es werden die folgenden Update-Ausdrücke angeboten: *insert*, *delete*, *delete_undeep*, *replace* und *rename*. Mit diesen Ausdrücken ist es möglich XML-Dokumente anzupassen, wobei "das Ergebnis solcher Updates weder die Wohlgeformtheit noch die Gültigkeit vorhandener XML-Entitäten verletzen darf. Andererseits wird ein Fehler generiert."[42] [Sed11]

XML-Dokumente können *standalone* gespeichert werden. Alternativ können diese in einer benannten *Collection* gesammelt werden, wobei ein *einheitliches, beschreibendes Schema* ("common descriptive schema") "dynamisch aufgebaut wird. Dieses beschreibt die Struktur der gespeicherten XML-Dokumente und wird inkrementell bei Änderung der Collection angepasst."[43] [Wik14]

Das einheitliche, beschreibende Schema entspricht keinem XML-Schema und kann manuell nicht geändert werden. Eine XML-Schemaevolution ist demnach in **Sedna** (v3.5 [Sed11]) nicht vorgesehen bzw. möglich.

eXist-db

In **eXist-db** sind eine *implizite* und *explizite Validierung* möglich. Die erste Variante wird automatisch bei entsprechender Konfiguration beim Einfügen von XML-Dokumenten ausgeführt. Dabei werden ungültige Dokumente, unter der Voraussetzung ein Schema ist spezifiziert und im System registriert, abgelehnt. Die entsprechenden Schemata müssen als *OASIS Katalogdateien* [OAS05] in der Datenbank registriert werden.[44] Die explizite Validierung wird unter der Verwendung von *XQuery-Erweiterungsfunktionen* ("XQuery extension functions") umgesetzt, wodurch eine nachträgliche Validierung von bereits eingefügten XML-Dokumenten möglich ist.[45]

"Aus Gründen der Effizienz werden Schemata vor dem ersten Gebrauch kompiliert und in einem Cache zwischengespeichert."[46] [eXi14b] Wird durch eine Weiterentwicklung ein Schema geändert, das heißt es kommt zu einer Schemaevolution, "wird die manuelle Löschung dieses Caches empfohlen."[47] [eXi14b]

[41] "The update language is based on the XQuery update proposal by Patrick Lehti [..]" [Sed11]

[42] "The result of each update statement, shouldn't break the well-formedness and validness of XML entities, stored in the database. Otherwise, an error is raised." [Sed11]

[43] "[..] the descriptive schema is generated from data dynamically (and is maintained incrementally) and represents a concise and an accurate structure summary for data." [Wik14]

[44] "All grammars (XML schema, DTD) that are used for implicit validation must be registered with eXist using OASIS catalog files. These catalog files can be stored on disk and/or in the database itself." [eXi14b]

[45] Eine Übersicht der Funktionen nebst Beschreibung notwendiger Parameter ist in [eXi14c] aufgelistet.

[46] "The XML parser (Xerces) compiles all grammar files (dtd, xsd) upon first use. For efficiency reasons these compiled grammars are cached and made available for reuse [..]" [eXi14b]

[47] "[..] it may be desirable to manually clear this cache [..]" [eXi14b]

Die XML-Schemaevolution wird durch **eXist-db** (v2.2 [eXi14a]) nicht umgesetzt. Wird ein Schema geändert, muss dieses aus dem Cache gelöscht und neu registriert werden. Des Weiteren muss eine explizite Validierung durchgeführt werden. Anschließend kann eine manuelle Anpassung der eventuell ungültigen XML-Dokumente mit Hilfe der Operationen *insert*, *replace*, *update value*, *delete* und *rename* erfolgen.[48]

3.2.4. X-Evolution

X-Evolution [GM08] ist ein webbasierter Prototyp, der in "einer ersten Version in [MCSG06] als Demo präsentiert wurde"[49] [GM08]. Dieser benötigt ein *kommerzielles, XML-fähiges Datenbankmanagementsystem*[50], in welchem XML-Schemata und XML-Dokumente gespeichert sind.

X-Evolution ermöglicht die Anwendung der in [GMR05, GMR06] vorgestellten *Primitiven* mittels einer *GUI* (Graphical User Interface) oder alternativ unter Verwendung einer speziellen, in [CGM08a] eingeführten Schemamodifikationssprache *XSchemaUpdate* [Cav09][51].

Die Primitive sind in Abbildung 3.7 dargestellt. Es wird unterschieden zwischen

	Insertion	Modification	Deletion
Simple Type	*insert_glob_simple_type** *insert_new_member_type**	*change_restriction* *change_base_type* *rename_type** *change_member_type* *global_to_local** *local_to_global**	*remove_type** *remove_member_type**
Complex Type	*insert_glob_complex_type** *insert_local_elem* *insert_ref_elem* *insert_operator*	*rename_local_elem* *rename_global_type** *change_type_local_elem* *change_cardinality* *change_operator* *global_to_local** *local_to_global**	*remove_element* *remove_operator* *remove_substructure* *remove_type**
Element	*insert_glob_elem*	*rename_glob_elem** *change_type_glob_elem* *ref_to_local** *local_to_ref**	*remove_glob_elem**

Abbildung 3.7.: Primitive zur Modifikation von XML-Schema aus [MCSG06]

drei atomaren Primitiven. Diese sind das Einfügen (*Insertion*), Update (*Modification*) und Löschen (*Deletion*) der Schemakomponenten einfacher Typ (*Simple*

[48] "eXist-db provides an extension to XQuery for updating nodes in the database. The extension makes the following operations possible [..]: insert, delete, replace, update value, and rename." [eXi14d]

[49] "A demo of a first version of X-Evolution has been presented in [MCSG06] [..]" [GM08]

[50] "commercial XML-enabled DBMS" [GM08]

[51] Die zitierte Publikation "F. Cavalieri, G. Guerrini, M. Mesiti. XSchemaUpdate: Schema Evolution and Document Adaptation. TR, DISI, Universit'a di Genova, 2008" wurde nicht gefunden und ist in [Gue15] auch nicht aufgelistet. Daher wird die Masterarbeit [Cav09] von F. Cavelieri verwendet, die im Vergleich ein Jahr später erschien und die Spezifikation von XSchemaUpdate vollständig enthält.

Type), komplexer Typ (*Complex Type*) und Element. "Die mit * gekennzeichneten Modifikationen beeinträchtigen nicht die Gültigkeit von XML-Dokumenten."[52] [MCSG06] Die Anwendung dieser Primitive unterliegt *Anwendbarkeitsbedingungen* ("applicability conditions"), damit die Gültigkeit eines XML-Schemas nicht verletzt wird. Dazu zählt zum Beispiel, dass globale Typen nur dann gelöscht werden können (u.a. *remove_type*[53]), wenn diese nicht referenziert werden.

Modifikationssprache XSchemaUpdate

Neben der Anwendung der Primitive auf eine Baumdarstellung eines XML-Schemas[54], können Änderungen mittels *XSchemaUpdate* spezifiziert werden. Ein Ausdruck dieser Sprache hat den allgemeinen Aufbau, der in Abbildung 3.8 dargestellt ist. Die komplette Spezifikation von XSchemaUpdate gemäß [Cav09] ist im Anhang in Abbildung A.2 dargestellt.

```
UPDATE SCHEMA ObjectSpec
UpdateSpec
AdaptSpec?
```

Abbildung 3.8.: Allgemeiner Aufbau eines XSchemaUpdate-Ausdrucks aus [GM08]

Die Identifizierung eines Objektes (*ObjectSpec*) erfolgt durch die Anwendung eines *XSPath*-Ausdrucks [CGM08a, CGM08b], eine aus "*XPath*[55] hergeleitete"[56] Pfadausdruckssprache für XML-Schema. "XSPath wurde speziell für die Pfadnavigation in XML-Schemas entwickelt, da die Anwendung von XPath in der Spezifikation komplexer Ausdrücke resultieren würde, die nicht den Erwartungen eines Anwenders an eine Anfrageformulierung entsprechen würden."[57] [CGM08a] Ein anzuwendendes Primitiv für das identifizierte Objekt wird in *UpdateSpec* angegeben, wobei eine "nutzerfreundliche Syntax"[58] für diese in [Cav09] entwickelt wurde. Die optionale *AdaptSpec*-Klausel steht im direkten Zusammenhang mit der Adaption der Dokumente.

[52] "Primitives marked with '*' do not alter the validity of the document instance." [MCSG06]
[53] Eine detaillierte Beschreibung aller Primitive aus Abbildung 3.7 kann [GMR06] entnommen werden.
[54] Ein XML-Schema wird als hierarchische Baumstruktur visualisiert, bei der zulässige Primitive über ein Kontextmenü auf eine Komponente (d.h. auf einen Knoten des Baums) angewendet werden können.
[55] siehe auch: Kapitel 2.2 (XPath)
[56] "XSPath, a language that derives from XPath [..]" [CGM08b]
[57] "XSPath [CGM08a] has been tailored for specifying path expressions on XSD schemas because the use of XPath over a schema would result in the specification of complex expressions that do not reflect the user expectation in query formulation." [CGM08a]
[58] "A userfriendly syntax has been developed for the specification of the modification primitives." [GM08]

Adaption von XML-Dokumenten

In [GMS07] werden unterschiedliche Ansätze zur *inkrementellen Validierung* und *automatischen Anpassung* vorgestellt. "Die inkrementelle Validierung, welche abhängig von der Einhaltung der Abhängigkeitsbedingungen ist, prüft nacheinander für jedes einzelne Primitiv in Kombination mit dem XML-Schema, ob ein gegebenes Dokument noch gültig ist oder nicht."[59] [GMS07] Ist dies nicht der Fall, wird je nach verwendetem Primitiv die Gültigkeit durch "Dokumentmodifikationen wieder hergestellt. Es wird dabei unterschieden zwischen dem Umbenennen, vollständigen Löschen und Einfügen von Elementen."[60] [GMS07] "Die Bestimmung von Werten beim Einfügen von Elementen wird als eines der Kernthemen der automatischen Adaption gesehen."[61] [GM08] "*Defaultwerte* kommen dabei abhängig vom Datentyp zum Einsatz, während bei komplexen Typen die einfachsten Strukturen generiert werden."[62] [GM08]

Ist dieser Ansatz nicht geeignet, das heißt die automatische Anpassung ist nicht möglich oder sinnvoll, wird die *anfragebasierte Adaption* ("Query-based Adaption") mittels *XSchemaUpdate* und *AdaptSpec*-Klausel vorgeschlagen.[63] Dies ist vor allem "bei *semantischen Änderungen* [TG04] notwendig, bei denen komplexere, den Sinn und Zweck eines XML-Schemas ändernde Anpassungen vorgenommen werden"[64] [GM08]. Bei der anfragebasierten Adaption kann der Anwender den neuen, notwendigen Inhalt angeben. Die AdaptSpec-Klausel besteht in diesem Fall aus einem Ausdruck zur Dokumentmodifikation. Diese wird durch die *XQuery Update Facility* [RCD+11] spezifiziert, "mit deren Hilfe persistente Änderungen an Instanzen des XPath-Datenmodells ermöglicht werden"[65] [RCD+11]. Das heißt die Baumstruktur eines wohlgeformten XML-Dokuments wird durch das Hinzufügen, Löschen, Modifizieren oder Kopieren von Knoten dahingehend verändert, dass diese Struktur wieder gültig bezüglich des durch die Primitive veränderten XML-Schemas ist.

[59] "The algorithm, relying on the applicability conditions of the evolution primitives being satisfied, tries to determine document validity from the applied evolution primitive and the schema [..]" [GMS07]

[60] "Document modifications can be of different types: element renaming, removal of an element with all its content, insertion of an element." [GMS07]

[61] "A key issue in the adaptation process is the determination of the values to assign when new elements should be inserted in the documents to make them valid." [GM08]

[62] "[..] for simple types, default primitive values are assigned, whereas, for complex types, the simplest structure of that type is extracted and associated with the element [..]" [GM08]

[63] "When this is not appropriate, the query-based adaptation facility should be used." [GM08]

[64] "By contrast (bzgl. Automatischer Adaption, Anm. d. Autors), query-based adaptation is needed to appropriately handle more complex schema changes, and also to account for semantic changes [TG04], when the change is in what the schema element represents." [GM08]

[65] "The XQuery Update Facility provides expressions that can be used to make persistent changes to instances of the XQuery 1.0 and XPath 2.0 Data Model (siehe [BFM+10], Anm. d. Autors)." [RCD+11]

Optimierung von Schemaoperationen

In [CGMO11] wird ein Algorithmus zur *Reduktion von Sequenzen von Operationen* eingeführt. Im Vergleich zu obigen Primitiven sind diese atomaren Operationen: *Insert*, *Delete*, *Update* und *Move*. Der Algorithmus ist ein baumbasierter Ansatz, bei welchem die Knoten des Ausgangs- und Zielschemas mittels des *dynamischen Bezeichnungsschemas* ("dynamic labeling scheme") *DDE* (Dynamic DEwey [XLWB09]) eindeutig identifiziert werden. "DDE garantiert dabei, dass trotz Anpassung des Ausgangsschemas alle Knoten das ursprüngliche Label und somit deren Identität behalten."[66] [XLWB09]

Die angewendeten Operationen, welche unter anderem eine Zuordnung zum angepassten Knoten durch Angabe des betreffenden Labels besitzen, werden dahingehend untersucht, ob deren Anwendung durch andere Operationen desselben Labels bzw. Knotens überflüssig sind. Ist dies der Fall, können unter bestimmten Nebenbedingungen Operationen aus der Sequenz entfernt werden, da diese keinen Einfluss auf die Adaption der Instanzen haben.[67]

Weiterentwicklung EXup

"**EXup** [Cav10] ist eine Weiterentwicklung von X-Evolution, wobei die Unterstützung von *XSUpdate*-Ausdrücken eingeführt wurde, nebst verschiedener Ansätze zur effektiven Anwendung innerhalb nativer Datenbankmanagementsystemen."[68] [CGM11c] Da XSUpdate eine verkürzte Schreibweise von XSchemaUpdate ist, wird diese hier nicht erneut betrachtet.[69]

Die Konzepte bezüglich der Adaption von XML-Dokumenten in EXup ähneln denen von X-Evolution, wobei die Möglichkeit der "*No Adaptation*" eingeführt wurde. Damit wird angegeben, dass keine Anpassung von Dokumenten gewünscht ist. "Dies hat allerdings zur Folge, dass Schemaoperationen beim Auftreten von ungültigen Instanzen zurück genommen werden. Alternativ kann auch spezifiziert werden, dass ungültige Instanzen gelöscht und die Schemaoperation durchgeführt werden soll."[70] [Cav10]

Die automatische Adaption wurde übernommen, während die anfragebasierte Adaption von X-Evolution durch eine *nutzerdefinierte Adaption* ("user-defined Adaptation") abgelöst wurde. In diesem Zusammenhang wurden *Umgebungen* ("environments") eingeführt, mit denen spezifischere Anpassungen von XML-Dokumen-

[66] "For static documents, the labels of DDE are the same as dewey which yield compact size and high query performance. When updates take place, DDE can completely avoid re-labeling [..]" [XLWB09]

[67] Ein ähnliche Ansatz wird in Kapitel 5.4 (Optimierung der Transformationssprache) vorgestellt.

[68] "The EXup system extends the X-Evolution system [GM08] by introducing support for XSUpdate statements and providing different approaches for making effective the updates in off-the-shelf native DBMSs." [CGM11c]

[69] XSPath wird in [Cav09] interessanter Weise analog zu XSchemaUpdate noch XSchemaPath genannt.

[70] "If the documents are no longer valid the operation is rollback unless the REMOVE INVALID option is specified. In this case documents are removed from the instances of the schema." [Cav10]

ten in Abhängigkeit des Umfelds möglich sind. Zum Beispiel kann somit definiert werden, dass bei einem bestimmten, von der Schemaoperation betroffenem Knoten ein gewisser Elementwert eingefügt wird, während in einem anderem Umfeld dieser Wert abweicht.[71]

Weitere Arbeiten

In [CGM11b] wird die Forschungsarbeit bezüglich des Dissertationsprojekts von Herrn Federico Cavalieri zusammenfassend vorgestellt, dazu gehören neben der *Schemaevolution* mit den entsprechenden obigen Ansätzen auch das "Dynamic Reasoning" von XML-Updates in [CGM11a], sowie der Rolle der "Provenance" von XML-Updates.

Ein weiterer Ansatz zur Analyse von Dokumentanpassungen wird in [SDG12] vorgestellt. Dort wird auf einer reduzierten Auswahl von Primitiven[72] mittels Automaten versucht zu entscheiden, ob ausgehend vom Ausgangs- und Zielschema, sowie den Dokumentanpassungsoperationen die Gültigkeit von XML-Dokumenten gewährleistet werden kann. "Dieses kann eine kostspielige Revalidierung von XML-Dokumenten verhindern."[73] [SDG12]

Zusammenfassung

X-Evolution (und ebenso **EXup**) bietet die Möglichkeit eine XML-Schemaevolution durchzuführen. Änderungen am Ausgangsschema können dabei entweder über eine *GUI* oder unter Anwendung von *XSchemaUpdate*-Ausdrücken (bzw. *XSUpdate*) erfolgen. Ein wichtiger Bestandteil ist die für die Pfadnavigation in Schemas entwickelt Sprache *XSPath*.

Änderungen werden mit Hilfe von Primitiven beschrieben, wobei zwischen *Insertion*, *Modification* und *Deletion* unterschieden wird. Jedes dieser atomaren Primitive ist weiter unterteilt und kann, insofern bestimmte *Anwendbarkeitsbedingungen* erfüllt sind, auf einfache und komplexe Typen, sowie Elemente angewendet werden. Eine Charakterisierung im Bezug auf die Notwendigkeit einer Adaption von XML-Dokumenten bei entsprechender Anwendung erfolgt ebenso.

Die Adaption von XML-Dokumenten kann bei weniger komplexen Änderungen automatisch geschehen, bzw. zusätzlich bei EXup komplett verhindert werden. Bei komplexeren Szenarien steht eine *Query-based* (bzw. *User-defined*) *Adaptation* zur Verfügung, bei der ein Anwender den Evolutionsprozess der Dokumente durch die Spezifikation verschiedener Parameter und Angabe von weitergehenden Informationen aktiv beeinflussen kann. Die daraus resultierenden Dokumentmodifikatio-

[71] Ein Umfeld kann z.B. durch das Auftreten bestimmter Nachbarknoten bzw. -elemente angegeben sein.
[72] Dies sind die in [CGM11a] vorgestellten Primitive einer PUL (Pending Update List) der XQuery Update Facility [RCD+11].
[73] "[..] avoiding the very expensive run-time revalidation of the set of involved documents [..]" [SDG12]

nen werden in *XQuery Update* Ausdrücke übersetzt, mit denen die Gültigkeit von wohlgeformten XML-Dokumenten gegebenenfalls wieder hergestellt wird.

Bewertung

Der Ansatz der Forschungsgruppe zeichnet sich dadurch aus, dass durchgehend von der Erfassung der Änderungen, über deren Charakterisierung, bis hin zur Adaption von XML-Dokumenten ein System zur Verfügung steht. Dennoch existieren Einschränkungen, die nachfolgend thematisiert und in den späteren Kapiteln beachtet und vermieden werden.

Ein Nachteil ist, dass wesentliche Bestandteile des *Standards von XML-Schema* nicht umgesetzt sind. Dazu zählen allen voran die *abgeleiteten Typen*, welche einen Großteil der primären Komponente des *Abstract Data Models* einfacher Typen darstellen.[74] Somit ist es nicht möglich Datentypen wie zum Beispiel *ID*, *NCName*, *token*, *short* etc. abzubilden bzw. zu behandeln.[75] Des Weiteren werden keine *externen Definitionen und Deklarationen* berücksichtigt, welche unter anderem aus anderen XML-Schemata importiert werden könnten und für die Modularität von Spezifikationen notwendig sind. Abschließend fehlt jegliche Möglichkeit zur Behandlung von *Constraints*, sowie *Wildcards*. Besonders letztere dienen der Flexibilisierung des Inhaltsmodells von komplexen Typen.

Die Sprache *XSchemaUpdate* mit *XSPath* wurde speziell für die Schemaevolution entwickelt. Mit Hilfe der *UpdateSpec*-Klausel sollen unter anderem komplexe Typen eingefügt werden können. Der Spezifikation aus Abbildung A.2 gemäß muss ein entsprechender Ausdruck lauten: "UPDATE SCHEMA" *ObjectSpec* "INSERT TYPE {" XMLTypeDef "}" *AdaptSpec?*. Was allerdings *XMLTypeDef* darstellen soll, wird in [Cav09] und nachfolgenden Publikationen nicht erläutert. Es wird an dieser Stelle angenommen, dass dort komplexere Strukturen eingefügt werden sollen. Dies müssten der XSchemaUpdate-Spezifikation folgend korrekt geschachtelte Sequenzen von XSUpdExpr sein, eine Ersetzung von XMLTypeDef hin zu *XSUpdExpr (", " XSUpdExpr)?* wäre notwendig. Ob dieses Vorgehen allerdings auch im Hinblick auf den gesamten Evolutionsprozess möglich ist und ein Ergebnis anschließend mit den vorliegenden Algorithmen auswertbar wäre, kann hier nicht geklärt werden. Die Behandlung von Elementen, Attributen und einfachen Typen konnten nach einer Gewöhnung an die Syntax nachvollzogen werden.

Im Allgemeinen müsste geklärt werden, ob *XSchemaUpdate* überhaupt notwendig ist, da dessen Ausdrücke in *XQuery Update* übersetzt werden.[76] Es könnte somit beim direkten, notwendigerweise Tool-gestützten Formulieren von XQuery

[74] "We have focused on the key features of XML Schema (global and local element declarations, simple and complex type definitions, references, arbitrary nesting of sequence, all, choice grouping operators). However, specific support should be included for other XML Schema peculiarities (like derived types, group elements, substitution groups, abstract definitions, uniqueness and keys)." [Cav10]

[75] siehe auch: Kapitel 2.1.1 (Strukturbeschreibung des XML-Schemas) Abbildung 2.1

[76] "[..] XSchemaUpdate statements are translated in XQuery update statements." [GM08]

Update auf die Übersetzung verzichtet werden.

Als Nebeneffekt wäre *XSPath* eventuell überflüssig. Es würde allerdings das fehlerhafte Vermischen mit XPath verhindert werden, welches mit der syntaktischen Ähnlichkeit der Konstrukte beider Sprachen begründbar ist.[77] XSPath wird abschließend in eine Menge von XPath-Ausdrücken übersetzt.[78] Diese Übersetzung würde ebenso eingespart werden. Alternativ zum Verzicht könnte eine Erweiterung von XQuery Update um XSPath angestrebt werden. Inwieweit dies allerdings realisierbar ist, müsste gegebenenfalls geprüft werden.

Ein weiterer Aspekt der nicht eindeutig dargestellt wird, ist der *Zeitpunkt der Anpassung* von XML-Dokumenten. Angenommen es werden gültigkeitsverletzende Schemaoperationen durchgeführt und die "No Adaptation" Option ist nicht spezifiziert, wann erfolgt daraufhin die Adaption von Dokumenten? In [RAJB+00] werden die folgenden Zeitpunkte für einen solchen Prozess aufgelistet: sofort, verzögert, später und niemals.[79] Der wahrscheinlichste Fall ist hier die sofortige Anpassung der Dokumente.[80] Dies resultiert allerdings darin, dass komplexe Operationen durch die Anwendung mehrerer, logisch zusammenhängender Primitive nicht möglich wären. Des Weiteren könnte es somit zu überflüssigen Adaptionen kommen, die zum Beispiel durch das "Herumprobieren" unwissentlich erfolgen.

Ein letzter Aspekt sind die *Defaultwerte* bei der *automatischen Adaption*. In [Cav10] wird speziell für den *string* eine leere Zeichenkette vorgeschlagen.[81] Ob dieses Vorgehen aber bei einer existierenden Mindestlängenbeschränkung (*minLength*)[82] umgesetzt wird oder aufgrund von Anwendbarkeitsbedingungen abgelehnt wird, müsste nochmals thematisiert werden.

Was bei Defaultwerten und einfachen Zeichenketten angemessen erscheint, wird bei numerischen Werten zum Problem. In diesem Fall müssten Werte definiert werden, die laut Datentyp keine leere Zeichenkette sind. Allerdings ist zum Beispiel die Semantik einer 0 im Kontext eines Schuldenelements wesentlich erfreulicher als bei einem Einkommenselement. In beiden Fällen müsste auf die automatische zugunsten einer anfrage- bzw. nutzerbasierten Adaption verzichtet werden.

[77] siehe auch: [CGM08a] Kapitel 3 (XSPath Specification)

[78] "The XSPath expressions are translated into an union of XPath expressions [..]" [GM08]

[79] "Finally, for the latter question (When?, Anm. d. Autors), the answers are immediately, delayed, later and never. For example, schema changes can result in the immediate conversion of data to the new format, a scheduled conversion at some later time, lazy conversion in which data are changed only when accessed or the use of filters to simulate change." [RAJB+00]

[80] Der Rückschluss erfolgte aufgrund der Beschreibung der "No Adaptation" Klausel in [Cav10]. Hier werden Schemaoperationen zurückgenommen, wenn die Gültigkeit von vorhandenen XML-Dokumenten verletzt wird. Eine verzögerte oder spätere Adaption ergibt in dem Zusammenhang keinen Sinn.

[81] "[..] empty string (the default value for the string type) [..]" [Cav10]

[82] *minLength* kann dem Datentyp *string* standardkonform als Facette mittels XSchemaUpdate zugeordnet werden, z.B.: "UPDATE SCHEMA" *ObjectSpec* "ADD RESTRICTIONS minLength = 1"

3.2.5. GEA-Framework

Das **GEA-Framework** (Generic Evolution Architecture) [DLP+11] ist ein Framework für den modell-getriebenen ("model-driven") Entwicklungskontext. "GEA ist eine Generalisierung der in [DLRZ08] vorgestellten, speziell für die Evolution von Datenbankkomponenten entwickelten Architektur *MeDEA* (Metamodel-based Database Evolution Architecture)."[83] [DLP+11]

Mit GEA wird ein "Evolutionsframework vorgestellt, in welchem XML-Schema und XML-Dokumente inkrementell upgedatet werden. Die dafür notwendigen Änderungen werden auf einem *konzeptuellen Modell* erfasst, wobei Klassendiagramme der *Unified Modeling Language* (UML) als Grundlagen dienen."[84] [DLP+11] Für das spezielle UML-zu-XML-Anwendungsszenario wird eine *Oracle-Datenbank* benötigt, da die *Copy-Based Schema Evolution* mit der Prozedur *DBMS_XMLSCHEMA.copyEvolve* angewendet wird.[85]

Generische Evolutionsarchitektur

In Abbildung 3.9 ist die allgemeine Architektur von GEA dargestellt. Es existieren

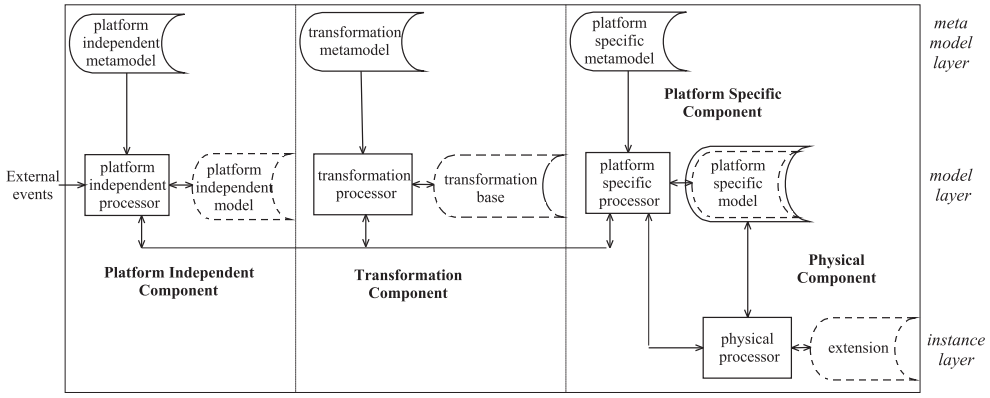


Abbildung 3.9.: GEA - Generic Evolution Architecture aus [DLP+11]

eine *Plattform-unabhängige*, eine *Transformations-* und eine *Plattform-spezifische Komponente*. Im vorgestellten Szenario entspricht erstere dem *UML-Klassenmodell mit Stereotypen und Profilen*[86] [OMG11], während die letzte Komponente allgemein die aus dem UML-Klassenmodell *transformierten XML-Schemata* repräsen-

[83] "GEA, standing for Generic Evolution Architecture, which is a generalization of a previous traceable architecture called MeDEA, devoted to the specific database context [DLRZ08]." [DLP+11]

[84] "[..] evolution framework by which the XML schema and documents are incrementally updated according to the changes in the conceptual model (expressed as a UML class model)." [DLP+11]

[85] siehe auch: Kapitel 3.2.1 (XML-Schema in Datenbanksystemen)

[86] "[..] this component deals with stereotyped UML class models and profiles componente." [DLP+11]

tiert[87]. "Im XML-Kontext wird das Plattform-spezifische Modell mit Hilfe von XML-Schema in eine *textuelle Struktur* transformiert, sodass die *Extension* der *physikalischen Komponente* mit zum spezifischen XML-Schema konformen XML-Dokumenten bestückt werden kann."[88] [DLP+11]

"Die Transformationskomponente ist ein wesentlicher Beitrag von GEA, da hier sowohl Verweise zwischen Elementen der Plattform-unabhängigen und anderen Plattform-spezifischen Elemente gespeichert werden, als auch explizite Informationen zur Transformation von diesen."[89] [DLP+11] "Durch diese Komponente wird die Nachvollziehbarkeit (*traceability*) des Transformationsprozesses sicher gestellt."[90] [DLP+11]

"Es wird davon ausgegangen, dass der Transformations- und Evolutionsprozess immer in der Plattform-unabhängigen Komponente beginnt."[91] [DLP+11]

UML-XML-Transformationsregeln

Es existieren unterschiedliche Ansätze zur Transformation von UML in XML-Schema, welche zusammenfassend in [DLP+07] aufgelistet werden. In [KK03] wurden die in Abbildung A.3 dargestellten Regeln eingeführt. Diese wurden für GEA adaptiert und in einem *Transformationsalgorithmus* vereinigt. Die entsprechenden Regeln sind in Abbildung 3.10 dargestellt. Es werden demnach Klassen (*class*)

UML block	XML item(s)
class	element, complex type, with ID attribute, and key
attribute	subelement of the corresponding class complex type
association	reference element, with IDREF attribute referencing the associated class and keyref for type safety (key/keyref references)
generalization	complex type of the subclass is defined as an extension of the complex type of the superclass

Abbildung 3.10.: Regeln zur Generierung von XML-Schema aus UML nach [DLRZ05]

eines UML-Diagramms zu *Elementdeklarationen* im XML-Schema transformiert, welche einen *komplexen Typen*, ein *ID-Attribut*, sowie einen Schlüssel (*key*) enthalten. Assoziationen werden als *Elementreferenzen* interpretiert, wobei die ID zur

[87] "[..] this component univocally represents the XML schemas obtained as a transformation of the UML class models and profiles defined at the platform independent level." [DLP+11]

[88] "Within the XML context, the platform specific model is transformed into a textual structure using XML schema, so that the extension can be populated with textual XML documents conforming to the specific XML schema." [DLP+11]

[89] "The transformation component, which is one of the main contributions of our architecture, stores the links that trace the transformation from elements of the platform independent component to others of the platform specific component. This component also stores explicit information about the way in which specific platform independent elements are transformed into platform specific ones." [DLP+11]

[90] "It ensures the traceability of the transformation process [..]" [DLP+11]

[91] "GEA has been created following a forward engineering pattern, which implies that the transformation and evolution processes always start at the platform independent component." [DLP+11]

Darstellung Verknüpfung mit der zugeordneten Klasse (d.h. der Elementdeklaration) genutzt wird, während Schlüssel-/Fremdschlüsselbeziehungen die Einhaltung von Datentypen (*type safety*) sicher stellen.

GEA "setzt den Modellierungsstil *Venetian Blind*[92] um"[93] [DLP+11], wobei die "Wurzelklasse auf ein Element mit komplexen Typen abgebildet wird. Unterklassen werden als komplexe Typen dargestellt, welche per Erweiterung des komplexen Typen des Wurzelelements erzeugt werden."[94] [DLP+07]

Durch die Anwendung der Transformationsregeln wird nicht nur ein XML-Schema erzeugt, sondern die Transformationskomponente mit *elementaren Transformationen* befüllt. Das heißt wird ein Element aus einer Klasse erzeugt, werden innerhalb der Transformationskomponente folgende Informationen gespeichert: der Typ der elementaren Transformation (*Class2Element*), die durch Namen identifizierten UML- und XML-Elemente, sowie der Name der für die Erzeugung verantwortlichen Prozedur (*trClass*)[95]. Durch die so gesammelten Informationen kann zu einem späteren Zeitpunkt nachvollzogen werden, welche Elemente aus welchen Klassen erzeugt wurden. Dadurch können Komponenten identifiziert werden, die zum Beispiel bei einer Änderung einer Klasse betroffen sind oder nicht.

Propagierungsalgorithmus

Zur "inkrementellen Aufrechterhaltung der Konsistenz von UML-Klassenmodellen und XML-Schemata bei evolutionären Änderungen der Plattform-unabhängigen Komponente"[96] [DLP+11], wurde ein *Algorithmus zur Propagierung* entwickelt. Dieser wurde, ebenso wie der Transformationsalgorithmus[97], im Kontext von Datenbankschemata erstmals in [DLRZ04] eingeführt.

"Die Evolution des UML-Klassenmodells der Plattform-unabhängigen Komponente wird zu den nachfolgenden Komponenten durch den Propagierungsalgorithmus propagiert, welcher in zwei Subalgorithmen unterteilt wird: der *Plattformspezifische Subalgorithmus* und der *physikalische Subalgorithmus*."[98] [DLP+11]

[92] siehe auch: Kapitel 2.1.3 (Modellierungsstile von XML-Schema)

[93] "[..] the resulting XML schemas following a 'Venetian Blind' pattern [Mal02]." [DLP+11]

[94] "The root class is mapped to an element and a complex type. The subclasses are mapped to complex types derived by extension from the complex type of the superclass." [DLP+07]

[95] Die Prozedur trClass garantiert u.a. bei Ausführung, dass z.B. beim Abbilden einer Klasse ein Element und gemäß Abbildung 3.10 zusätzlich ein komplexer Typ, eine ID und ein Schlüssel erzeugt werden.

[96] "[..] to incrementally maintain the consistency between UML class models and XML schemas and documents when evolution changes happen in the platform independent component." [DLP+11]

[97] Dieser wurde als "translation algorithm" eingeführt, da [DLRZ04] folgend nicht nur eine traditionelle Übersetzung eines konzeptuellen in ein logisches Modell erfolgte, sondern zeitgleich Informationen in einer "translation base" (in GEA die Transformationskomponente) gespeichert werden.

[98] "The UML class model evolution is propagated to the rest of the components by the propagation algorithm, which is split into two subalgorithms: the first for the transformation and platform specific components, which we call platform specific subalgorithm and the second for the physical component, which we call physical subalgorithm." [DLP+11]

Eine Propagierungsregel des Plattform-spezifischen Subalgorithmus ist in Abbildung 3.11 dargestellt. Die *newElementForComplexType* Regel wird angewendet,

Name	newElementForComplexType
Event	createClass(name)
Condition	the XML complex type ct has been previously created for the new class cl created by the event
Action	(1) `createElement(name) AS vElement` (2) `createElementaryTransformation(cl, vElement, trClass, Class2Element)` (3) `assignComplexTypeToElement(ct, vElement)` (4) `rootct = getRootType` (5) `addElementToComplexType(vElement, rootct)`

Abbildung 3.11.: Beispielregel des Plattform-spezifischen Subalgorithmus aus [DLP+11]

wenn im Klassenmodell eine neue Klasse erschaffen wird (*createClass*). Unter der Voraussetzung (*condition*), dass ein komplexer Typ vorliegt, werden unterschiedliche Aktionen ausgelöst. Diese Aktionen passen unter anderem die Transformationskomponente an, indem die elementare Transformation *Class2Element* durch die Aktion *createElementaryTransformation* hinzugefügt wird. Des Weiteren wird die Plattform-spezifische Komponente angepasst, indem zum Beispiel Aktion (5) *addElementToComplexType* angewendet wird.

Durch die Anwendung der Regel (5) der Abbildung 3.11 wird die *Korrespondenzregel addRootChildElement* des Plattform-spezifischen Subalgorithmus ausgelöst, welche in Abbildung 3.12 dargestellt ist. Unter den Voraussetzungen, dass das spe-

Name	addRootChildElement
Event	addElementToComplexType(elem, ct)
Condition	the element elem is a leaf node but it is not a child of the root and the complex type ct is the complex type of the root element
Action	(1) `name = getName(elem)` (2) `elemType = getType(elem)` (3) `XML_sch_addRootChildElement(name, elemType)` (4) `XML_doc_addRootChildElements(name, elemType)`

Abbildung 3.12.: Beispielregel des physikalischen Subalgorithmus aus [DLP+11]

zifizierte Element nicht das Wurzelelement ist, nicht als Kindselement der Wurzel auftritt und der gegebene komplexe Typ der des Wurzelelements ist, werden Folgeaktionen gestartet. Diese passen die textuelle Struktur des XML-Schemas und der XML-Dokumente an. "Um dieses zu tun wird jede Korrespondenzregel unterteilt in Prozeduren zur XML-Schema- (*XML_sch**) und XML-Dokumentanpassung (*XML_doc**)."[99] [DLP+11] "Diese Prozeduren erhalten die Konsistenz zwischen

[99] "In order to do this, each correspondence rule determines two types of procedures: (1) those that modify the XML schema code and (2) those that change the XML documents. To distinguish between them, the name of the procedures of the first kind begins with XML_sch* and the name of those of the second kind starts with XML_doc*." [DLP+11]

XML-Dokumenten und dem entsprechenden XML-Schema."[100] [DLP+11]

Die zur Anpassung der textuellen Struktur notwendigen Transformationen werden als *XSLT-Stylesheets* bereitgestellt, welche in Abhängigkeit der angewendeten XML_sch*- bzw. XML_doc*-Prozeduren erschaffen werden. In [DLRZ05] wird ein durchgängiges Beispiel für solche Stylesheets gegeben.[101]

Validierung von XML-Dokumenten

In [DLRZ06] werden unterschiedliche *Typen von Constraints* des konzeptuellen Modells aufgelistet. Dazu zählen zum Beispiel die Typen *isa* (Ist-Beziehung), *exists* (Existenz), *exclusive* (Teilnehmerbeschränkung) etc.. "Neben diesen Bedingungen werden weitere, nicht durch das konzeptuelle Modell bereitgestellte, ergänzt. Dies sind Schlüssel- (*key constraints*) und Nullbedingungen (*null constraints*), welche als *Transformationsbedingungen* zusammengefasst werden."[102] [DLRZ06]

In einem *Constraints Generation Algorithm* werden aus den Typen von Bedingungen *XSLT-Templates* erzeugt, welche deren Einhaltung in XML-Dokumenten überprüfen. Das heißt, wenn zum Beispiel laut XML-Schema ein Element einen Nullwert annehmen soll, dann wird ein entsprechendes XSLT-Template erzeugt. Dieses "prüft für ein vorhandenes oder gegebenenfalls neues XML-Dokument"[103] [DLRZ06], ob diese Nullbedingung erfüllt ist oder nicht. "Liefert das Template bei Anwendung den Wert *false*, verletzt das untersuchte Dokument die Bedingung"[104] [DLRZ06] und ist demnach nicht schemakonform.[105]

Zusammenfassung

Das **GEA-Framework** bietet die Möglichkeit eine XML-Schemaevolution durchzuführen. Ein konzeptuelles, Plattform-unabhängiges Modell (*UML-Klassendiagramm*) wird in ein Plattform-spezifisches XML-Schema durch Anwendung von *Transformationsregeln* innerhalb eines *Transformationsalgorithmus* abgebildet.

Ändert ein Anwender das konzeptuelle Modell durch die Formulierung entsprechender *elementarer Transformationen*, wird unter Berücksichtigung einer *Transformationskomponente* diese Änderung durch einen *Propagierungsalgorithmus* innerhalb des Frameworks propagiert. Auf diese Art und Weise werden in einem

[100] "Every procedure has been designed to maintain the consistency between the XML documents and the XML schema." [DLP+11]

[101] Abbildung A.4 illustriert die XSLT-Stylesheets, welche durch die elementare Transformation attribToClass mit anschließender Anwendung der Korrespondenzregel addRootChildElement entstehen.

[102] "We distinguish two kinds of information not available at the conceptual level and provided during the structure generation algorithm execution: key constraints and null constraints (these constraints are referred to as transformation constraints)." [DLRZ06]

[103] "[..] insert a new document into the table or to modify previously existing ones [..]" [DLRZ06]

[104] "If the value false is returned, the document violates some constraints [..]" [DLRZ06]

[105] Die entsprechende Adaption des Dokuments ist nicht Bestandteil der Betrachtungen in [DLRZ06].

mehrstufigen Prozess notwendige *Plattform-spezifische Transformations-* und *Korrespondenzregeln* angewendet, insofern bestimmte *Nebenbedingungen* erfüllt sind.

Als Ergebnis werden *XSLT-Stylesheets* zur Überprüfung und gegebenenfalls zur Anpassung der *textuellen Struktur* von XML-Schema und gegebenen XML-Dokumenten erzeugt. Für die Umsetzung der XML-Schemaevolution wird die Funktionalität einer *Oracle-Datenbank* benötigt, in welcher die Stylesheets bei der *Copy-Based Schema Evolution* angewendet werden können.

Bewertung

Der Ansatz der Forschungsgruppe zeichnet sich dadurch aus, dass ein konzeptuelles Modell, speziell *UML-Klassendiagramme mit Profilen und Stereotypen* [OMG11, Fak11], als Grundlage für die XML-Schemaevolution verwendet wurde. Auf diesem können unterschiedliche Primitive angewendet werden, um in einem entsprechenden, mehrstufigen Prozess XML-Dokumente anzupassen.

Die dafür notwendigen Transformationsregeln (speziell für die Plattform-spezifischen Komponente) werden allerdings in *keiner vollständigen Auflistung* bereitgestellt, sondern nur auszugsweise.[106] Dies hat zur Folge, dass der Umfang der Möglichkeiten des Frameworks nur erahnt werden kann. Die zur Illustration des Ansatzes verwendeten Beispiele sind gut nachvollziehbar, was für Publikationen verständlich ist.[107] Diese können nicht zur Klärung des Umfangs beitragen.

Es steht ein Forschungsprototyp zur Verfügung, bei welchem ein nicht näher bestimmter Teil von Transformationen mit Hilfe einer *Texteingabeschnittstelle* ausgeführt wurde.[108] In diesem Zusammenhang wird ebenso erwähnt, dass eine *kleine Anzahl von einfachen Primitiven* sinnvoll sei.[109] Welche dies allerdings im Kontext der XML-Schemaevolution sind, kann nicht vollständig geklärt werden. Zu den Primitiven gehören allerdings mindestens das Einfügen und Löschen von Elementen in XML-Dokumenten, welche während einer Performance Untersuchung des Gesamtkonzepts angewendet wurden.[110] Bezüglich des Einfügens wird nicht auf die *Generierung von Elementinhalten* eingegangen. Das heißt, dass an dieser Stelle entweder generell solche Informationen durch den Nutzer gegeben werden müssen, nicht näher spezifizierte Defaultwerte zum Einsatz kommen, oder nur optionale Elemente verwendet werden. Die letzte Möglichkeit erscheint im Zusammenhang mit dem Einfügen von Elementen mit komplexen Typ als am wahrscheinlichsten.

[106] "Excerpt of XML in-place transformations." [DLP+11]

[107] Aus Sicht von XML-Schema in [DLP+11]: Verschieben des optionalen Elements *department* und Hinzufügen des optionalen Elements *operate*; in [DLRZ05]: Umwandlung des Attributs *department* in ein optionales Element mit anschließender Verschiebung

[108] "[..] a laboratory prototype has been developed as a proof-of-concept of our proposal, where we have implemented a subset of the evolution transformations with a textual user interface." [DLP+11]

[109] "[..] a reduced number of simple, carefully designed primitives, as those presented in this paper, can express a high percentage of all the evolution cases." [DLP+11]

[110] "For studying the performance of our proposal, we have made an experiment [..] the application of two evolution transformations for adding/deleting an element [..]" [DLP+11]

Ein weitere Unklarheit betrifft die *XSLT-Stylesheets* zur Anpassung der textuellen Struktur von XML-Schemata bzw. XML-Dokumenten. In den Publikationen wird suggeriert, dass diese in den *Korrespondenzregeln* erzeugt werden bzw. zur Verfügung stehen.[111] Details dazu sind allerdings nicht zu finden, in [DLP+11] wird auf diese Thematik sogar komplett verzichtet. Da allerdings die *Copy-Based Schema Evolution*[112] bei ungültigen XML-Dokumenten diese Skripte zwingend zur Adaption benötigt, ist dies elementar für die XML-Schemaevolution. Dies kann einerseits bedeuten, dass *nur kapazitätserweiternde Schemaoperationen* zulässig sind, oder andererseits nur optionale Änderungen möglich sind, wie in den Beispielen in [DLRZ05, DLP+11] vorgestellt.

In [DLRZ06] werden *XSLT-Stylesheets zur Überprüfung von Constraints* eingeführt. Verletzt ein neues bzw. vorhandenes XML-Dokument eine Constraint, das heißt das Dokument ist nicht schemakonform, wird der Nutzer darüber informiert und die Änderung wird innerhalb der Datenbank rückgängig gemacht.[113] Weitergehende Schritte, zum Beispiel eine Anpassung zur Herstellung der Gültigkeit werden in diesem Zusammenhang allerdings nicht thematisiert, sondern es wird auf die Möglichkeiten der Oracle-Datenbank hingewiesen.[114]

Da nun allerdings [DLP+11] nicht auf die Stylesheets eingeht, muss [DLRZ05] somit als Quelle verwendet werden. Würden komplexere Adaptionen der XML-Dokumente möglich sein, würde dies mit Sicherheit zu mindestens ansatzweise in der zeitlich nachfolgenden, von den gleichen Autoren wie in [DLRZ05] veröffentlichten Publikation [DLRZ06] erwähnt werden. Dies ist allerdings nicht der Fall, sodass die Problematik der XSLT-Stylesheets entweder noch nicht gelöst oder zufällig noch nicht thematisiert wurde.

Bezüglich des *Standards von XML-Schema* werden verschiedene Konzepte umgesetzt. Dafür wurde in [DLP+11] eine grafische Repräsentation der XML-Komponenten des XML-Schemas eingefügt, welche in Abbildung A.5 dargestellt ist. Im Vergleich zu *X-Evolution* fehlt hier auf den ersten Blick ebenso die Möglichkeit *Wildcards* zu definieren. *Constraints* in Form von Integritätsbedingungen sind enthalten. *Facetten zur Restriktion einfacher Typen*, sowie *Vereinigungs-* und *Listentypen* fehlen. Externe Schemata können mit berücksichtigt werden, deren Komponenten sind aber nicht näher thematisiert.[115] Da die Abbildung allerdings nur die für den *UML-XML-Ansatz* notwendigen Konstrukte enthält[116], kann keine detailliertere Aussage über den Umfang der Umsetzung des Standards durch das Framework getätigt werden.

[111] "[..] change of the logical XML schema triggers one or more XSLT stylesheets [..]" [DLRZ05]
[112] siehe auch: Kapitel 3.2.1 (XML-Schema in Datenbanksystemen)
[113] "[..] the user is informed of the violated constraints. A rollback on the database is done." [DLRZ06]
[114] "The DBMS XMLSCHEMA package has been a handy tool for this task." [DLRZ06]
[115] "[..] we consider that schemas can be classified into at least two subtypes, which include other schemas as part of their definition and those which do not." [DLP+11]
[116] "This metamodel conceptualizes those XML Schema elements [FW04] that have been used in our UML-to-XML evolution proposal." [DLP+11]

3.2.6. XCase

Das **XCase** [KKLM09] Werkzeug ist eine Implementierung des konzeptuellen Modells für XML *XSEM* (XML Semantic Modeling) [Nec09]. "XSEM verwendet *UML-Klassendiagramme* um in einer *MDA* (Model-Driven Architecture) XML Daten auf den folgenden zwei Ebenen zu modellieren: *PIM* und *PSM*."[117] [KKLM09]

"Das *PIM* (Platform-Independent Model) ermöglicht die Erschaffung von konzeptuellen Diagrammen, die ein Modell unabhängig von dessen angedachten Repräsentation in unterschiedlichen *XML-Formaten* beschreiben."[118] [KKLM09] "Die Konstrukte der PIM-Ebene sind die gleichen wie in UML-Klassendiagrammen."[119] [KKLM09]

"Ein *PSM* (Platform-Specific Model) Diagramm ist eine visuelle Repräsentation einer XML-Dokumentstruktur."[120] [KKLM09] "Alle Komponenten eines PSM sind von deren konzeptuellen Gegenstücken im PIM abgeleitet. Diese Verbindung wird beibehalten, damit Änderungen am PIM zu den betroffenen Komponenten propagiert werden können."[121] [KKLM09]

"Ein *Übersetzungsalgorithmus* kann auf Grundlage des Algorithmus aus [Nec09] automatisch jedes mögliche PSM in XML-Schema übersetzen, wobei dieses Schema dem *Venetian-Blind-Modellierungsstil*[122] entspricht."[123] [KKLM09]

Fünf-Ebenen-Architektur

In [MN09] wird eine allgemeinere *Fünf-Ebenen-Architektur* eingeführt, welche in Abbildung 3.13 dargestellt ist.[124] Zusätzlich zu den Ebenen von PIM und PSM, "welche als *konzeptuelle* Ebenen bezeichnet werden"[125] [KN10a], existieren die *logische*, *operationale* und *extensionale* Ebene.

"Das Plattform-unabhängige Level enthält ein *konzeptuelles Schema*, welches das Informationsmodell des Systems beschreibt und die Semantik aller XML-Formate einheitlich abbildet."[126] [MNM12] "Obwohl es nur ein PIM pro Projekt gibt, wird aus Gründen der Lesbarkeit eine Unterteilung in mehrere PIM-Diagramme er-

[117] "It (conceptual model XSEM, Anm. d. Autors) utilizes UML class diagrams to apply MDA to model XML data on two levels: PIM and PSM." [KKLM09]
[118] "PIM enables one to design conceptual diagrams describing the model independently of the intended representation in various XML formats." [KKLM09]
[119] "The constructs [..] at the PIM level are the same as defined in the UML class diagrams." [KKLM09]
[120] "A PSM diagram is a visual representation of an XML document structure [..]" [KKLM09]
[121] "PSM components have been derived from their conceptual counterparts, maintaining this connection for further use. This includes changes, that can be propagated to all affected components." [KKLM09]
[122] siehe auch: Kapitel 2.1.3 (Modellierungsstile von XML-Schema)
[123] "The translation algorithm is automatic and is based on the algorithm proposed in [Nec09] [..] elaborated to cover all possible PSM diagrams. It uses Venetian Blind design for resulting XSD." [KKLM09]
[124] Die inhaltlich identische Grafik aus [MNM12] wird verwendet, da diese in besserer Qualität vorliegt.
[125] "The platform-independent and platform-specific levels are called conceptual levels." [KN10a]
[126] "The platform-independent level contains a conceptual schema which describes the information model of the system and covers the semantics of all XML formats in the family in a uniform way." [MNM12]

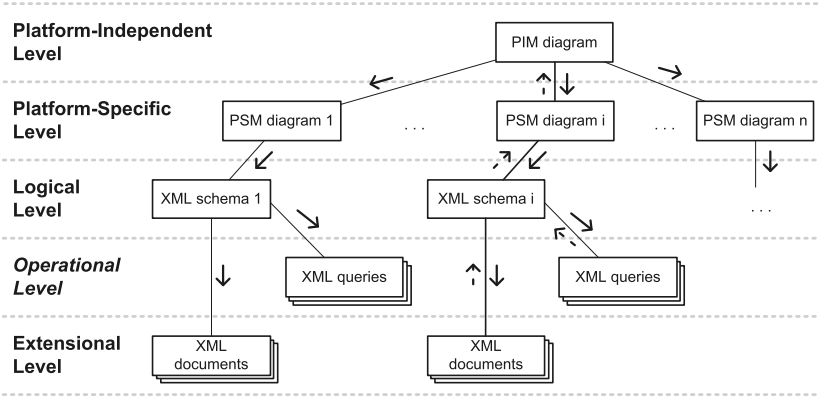


Abbildung 3.13.: Fünf-Ebenen-Architektur von XCase aus [MNM12]

laubt."[127] [KKLM09] Ein PIM enthält *Klassen* zur Modellierung von Konzepten und *binäre Assoziationen* zwischen den entsprechenden Konzepten.

"Ein PSM enthält ausgewählte Klassen des PIM-Diagramms, ordnet diese allerdings in eine *hierarchische Struktur* des modellierten XML-Formats unter Verwendung von Assoziationen ein."[128] [MN09] Neben den Klassen beinhaltet ein PSM *Element Label*, *Attributcontainer* und eine sich gegenseitig *ausschließende Inhaltsauswahl* ("content choice"). "Ein PSM enthält ein Schema, das die *Semantik* der darunter liegenden Ebenen spezifiziert."[129] [MNM12]

Das logische Level enthält für jedes XML-Format ein XML Schema, welches die *Struktur erlaubter XML-Dokumente* beschreibt. "Es wird mittels einer ausgewählten XML-Schemasprache die *Syntax* des XML-Formats ausgedrückt."[130] [MNM12] Eine Schemasprache kann eine "DTD [BPSM+08], Relax-NG [CM01], Schematron [JTC06], ein XML-Schema [FW04] etc." [MNM12] sein. Für ein XML-Schema werden in [MN09] nur die einfachen und komplexen Typen, sowie Attribut- und Elementdeklarationen thematisiert.[131]

Das operationale Level beinhaltet *Anfragen* (bzw. allgemein Operationen) über XML-Dokumente, dies können unter anderem Instanzen der extensionalen Ebene sein. "Die Dokumente der untersten Ebene bestehen aus Elementen, Attributen und Textknoten."[132] [MNM12]

[127] "Although there is only one PIM in the project, we allow the user to divide it into multiple PIM diagrams to increase readability." [KKLM09]

[128] "A PSM diagram contains classes from the PIM diagram and organizes them into the hierarchical structure of the modeled XML format by associations." [MN09]

[129] "The platform-specific level contains a schema which specifies the semantics of the XML format in terms of the level above." [MNM12]

[130] "[..] contains a logical XML schema which specifies the syntax of the XML format." [MNM12]

[131] "We consider only the basic ones in this paper. Namely, simple data types (simpleType), complex data types (complexType), elements (element) and attributes (attribute)." [MN09]

[132] "XML documents are composed of XML elements that can contain text values and/or nested XML elements. An XML element can also have XML attributes." [MNM12]

Transformationstypen

"In [MN09] wurden neben der Fünf-Ebenen-Architektur ebenso eine Beispielmenge von Operationen eingeführt, sowie deren Propagierung innerhalb der obersten beiden Ebenen vorgestellt."[133] [MNM12] "Des Weiteren erfolgt sowohl eine formale Definition der Ebenen in [NMKM12], als auch die der Änderungsoperationen der konzeptuellen Ebenen und deren Propagierung zu Nachbarlevels in [NKMM11]."[134] [MNM12]

In [MN09] werden *strukturelle* ("structural"), *ortsgebundene* ("sedentary") und *ortsändernde* ("migratory") *Operationen* eingeführt. Eine Übersicht ist in Abbildung 3.14 dargestellt. "Nicht alle Typen sind auf jeder Ebene zulässig. Zum Beispiel

- Structural:
  - *Adding* – adds a new item
  - *Removal* – removes a new item
- Sedentary:
  - *Extension* – adds a new item that does not change structure
  - *Renaming* – renames an item
  - *Renumbering* – changes the cardinality of an item
  - *Retyping* – changes the data type of an item
  - *Resetting* – changes the value of an item
  - *Mapping* – maps an item to an item from another level
  - *Unmapping* – removes a mapping between levels
- Migratory:
  - *Moving* – moves an item
  - *Reordering* – changes the order of a set of items
  - *Transformation* – transforms an item to an item of a different type

Abbildung 3.14.: Transformationstypen aus [MN09]

ist ein *Retyping* im PIM nicht vorgesehen, da im vorgestellten Ansatz Datentypen dort nicht existieren."[135] [MN09]

In [MNM12] wird diese Auflistung leicht angepasst, indem analog zu [Mal10] strukturelle Änderungen in das *Hinzufügen* ("Addition") und *Entfernen* ("Removal") aufgesplittet werden.[136] Des Weiteren wird der *Gültigkeitsbereich von Änderungen* ("scope of change") eingeführt, sodass jede der vorherigen Operationen auf *Klassen*, *Attribute*, *Assoziationen* oder *Inhaltsmodelle* angewendet werden kann. Abbildung A.7 beinhaltet die Gesamtübersicht aus [MNM12].

[133] "In [MN09] we proposed the five-level XML evolution framework and provided a sample set of operations and propagation between the highest two levels." [MNM12]

[134] "The levels and their mutual relations were formally defined in [NMKM12], the edit operations at the conceptual levels and their propagation to neighboring levels in [NKMM11]." [MNM12]

[135] "Note that not all types of the transformations exist at all five levels we are dealing with. For instance, retyping does not occur at the platform-indepdendent level, since we restrict ourselves only to classes, attributes and associations." [MN09]

[136] Es werden die vorherigen Untertypen somit als eigenständige Transformationstypen behandelt.

Propagierung von Transformationen

Die Operationen werden zwischen den Ebenen propagiert, um die Konsistenz der unterschiedlichen Bestandteile (d.h. Klassen, Deklarationen etc.) zu gewährleisten. In Abbildung 3.15 wird ein Auszug der in [MN09] vorgestellten Klasse von Adding-Operationen dargestellt. Wird zum Beispiel auf der logischen Ebene (*Logical*, d.h. XML-Schema) ein Element hinzugefügt, dann wird dies optional (*(O)*) aufwärts (*Up Propagation*) bzw. abwärts (*Down Propagation*) propagiert. "Schemaspezifische Aspekte werden an dieser Stelle nicht thematisiert. Das heißt, dass unter anderem eine Unterscheidung zwischen lokalen und globalen Deklarationen nicht stattfindet."[137] [MN09]

Level	Operations	↑ Up Propagation	↓ Down Propagation
PIM	add class, attribute or association		
PSM	add class, attribute, association, content choice or content container		add simple type, complex type, element, attribute, choice operator or sequence operator
Logical	add element (+ simple or complex type), attribute (+ simple type), choice operator or sequence operator	add a class, attribute, association, content choice or content container (O)	add element or attribute (O)
Extensional	add element or attribute	create element or attribute, change cardinality (O)	

Abbildung 3.15.: Auszug der Adding-Operation mit Propagierung aus [MN09]

In [MKMN11] wird das konzeptuelle Modell XSEM dahingehend erweitert, dass unterschiedliche *Versionen* unterstützt werden.[138] "Dadurch kann eine *Menge von Änderungen*[139] definiert werden"[140] [MKMN11], die zur Propagierung und somit zur *Revalidierung* existierender XML-Dokumente genutzt werden kann. Dieses Konzept wird in [MMN11] noch detaillierter und um die Berücksichtigung des *Inhaltsmodells* ("content model") erweitert beschrieben. Des Weiteren wird ein Algorithmus vorgestellt, mit dem ein *XSL-Stylesheet* (Extensible Stylesheet Language) aus der erfassten Menge von Änderungen generiert werden kann. "Mit diesem *Revalidierungsskript* können XML-Dokumente, die gegenüber der alten Version gültig waren, angepasst werden, sodass diese bezüglich der neuen Version des Schemas gültig sind."[141] [MN13]

[137] "Note that for the sake of simplicity we do not deal with XML schema specific aspects such as possibilities of creating a global or local element or equivalent content model." [MN09]
[138] Grundlage ist das "XCase evolution framework (XSem-Evo)", welches in [Mal10] eingeführt wurde.
[139] Die Menge der Änderungen entspricht somit einem *Mapping* zwischen zwei Versionen eines Schemas.
[140] "[..] extension, it is possible to define a set of changes between two versions of a schema." [MKMN11]
[141] "[..] we proposed an algorithm for generating an adaptation script to transform documents valid against one version of a schema into documents valid against other version of the same schema." [MN13]

Weiterentwicklung eXolutio

In [KMN12] wird **eXolutio** als Werkzeug für die Evolution und Änderungspropagierung von XML-Anwendungen vorgestellt.[142] "Dieses ist eine neue Version des konzeptuellen Modells und deren Implementierung *XCase*, welche als Vorgänger gilt."[143] [KMN12] Das *Model-View-Controller* (MVC) Design Pattern wird eingeführt. Eine allgemeine Übersicht der Architektur ist in 3.16 dargestellt. Durch das

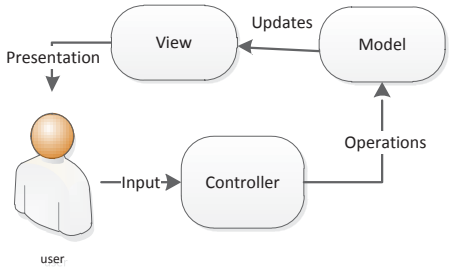


Abbildung 3.16.: Architektur der MVC-Komponenten von eXolutio aus [KMN12]

MVC Pattern wird eine logische Trennung der Funktionalitäten des Frameworks ermöglicht. Zum Beispiel sind die unterschiedlichen Modelle der Fünf-Ebenen-Architektur ein Bestandteil der *Modellkomponente*, während durch einen Nutzer ausgeführte Operationen von der *Controller-Komponente* behandelt werden. "Des Weiteren sind durch die Trennung der *Sichtkomponente* unterschiedliche Visualisierungen möglich. Dazu zählen sowohl eine Desktop-, als auch Web- und Nicht-Visualisierungsanwendung (d.h. Konsole), welche das gleiche Modell und Controller teilen."[144] [KMN12]

"In [NMKM12] wird ein formales Modell eingeführt, mit der die logische Ebene und somit die XML-Schemasprachen einheitlich als *Reguläre Baumgrammatiken* ("regular tree grammars") [MLMK05][145] dargestellt werden."[146] [NMKM12] "eXolutio unterstützt die Normalisierung und Optimierung von Schemata des PSM, sowie die Übersetzung von diesen in eine reguläre Baumgrammatik."[147]

[142] "[..] we describe a tool for evolution and change propagation of XML applications [..]" [KMN12]

[143] "There exists also an older version of our conceptual model and its implementation called XCase [KKLM09], which is the predecessor of eXolutio." [KMN12]

[144] "[..] we have a Windows Presentation Foundation visualization (a desktop application) a Silverlight visualization (a web application) and a no-visualization (a console application) versions of eXolutio which all share the same model and the same controller." [KMN12]

[145] Es wird in [MLMK05] ein Framework vorgestellt, welches drei Klassen von Baumsprachen (local, single-type und regular) einführt und basierend darauf Schemasprachen (DTD, XML-Schema und Relax-NG) klassifiziert und charakterisiert, sowie die Validierung von Dokumenten ermöglicht.

[146] "[..] we work with the formalism of regular tree grammars instead of a particular XML schema language." [NMKM12]

[147] "eXolutio also supports normalization and optimization of PSM schemas [..]. It also implements the algorithms for translating PSM schemas to regular tree grammars [..]" [NMKM12]

[NMKM12] "Des Weiteren sind Algorithmen zur Wartung und Erweiterung von PIM- und PSM-Schemata und deren Interpretationen enthalten. Diese werden bei der Adaption der konzeptuellen Modelle durch Nutzeränderungen angewendet."[148] [NMKM12]

Weitere Arbeiten

Neben den obigen Publikationen existieren weitere Arbeiten, die im Umfeld des Forschungsprototypen relevant sind. So wird in [FMN11] der *KeyMiner* [Faj10] vorgestellt, welcher ausgehend von XML-Dokumenten Integritätsbedingungen in Form von Schlüssel-/Fremdschlüsselbeziehungen in XML-Schema entdeckt.

In [KN10b] wird ein Überblick über Arbeiten gegeben, die aus einem XML Schema[149] ein konzeptuelles Modell mittels *Reverse-engineering* erzeugen. Dabei werden Kriterien zum Vergleich der Ansätze eingeführt. Diese untersuchen das eingesetzte Modell, die Anzahl gleichzeitig unterstützter Schemata, die unterstützten XML-Schemasprachen, die Möglichkeit der Abbildung zu einem existierenden Modell, den Grad der Nutzereinbindung, sowie allgemein die Evolutionsunterstützung.

Wie bereits erwähnt wird in [Mal10] das *XCase evolution framework (XSemEvo)* eingeführt. Dieses ist eine Grundlage für das Promotionsvorhaben, welches in [MN12] vorgestellt wird. In diesem geht es unter anderem um die Einführung von Constraints im konzeptuellen Modell, das heißt auf der Ebene der Plattformunabhängigen und -spezifischen Modelle. Die Möglichkeit der Verwendung der *Object Constraint Language (OCL)* zur Integration von Constraints wird in [MN13] thematisiert. Schematron wird in dem Ansatz zur Prüfung von diesen eingesetzt.

In [KN12] wird auf "die *strukturelle und konzeptuelle Vererbung* zwischen PSMs eingegangen. Das Ziel ist es, die mehrfache Wiederholung gemeinsamer Attribute und/oder Teile des Inhalts zu verhindern."[150] "Eine Implementierung der Methodik wurde als Erweiterung in eXolutio integriert."[151] [KN12]

Zusammenfassung

XCase und der offizielle Nachfolger **eXolutio** bieten die Möglichkeit eine XML-Schemaevolution durchzuführen. Grundlage ist das konzeptuelle Modell für XML *XSEM* (XML Semantic Modeling), welches in einer *Fünf-Ebenen-Architektur* als *Plattform-unabhängiges* (PIM) und *Plattform-spezifisches* (PSM) Modell integriert

[148] "This includes mainly maintaining and augmenting PIM and PSM schemas and interpretations when a designer performs various changes. We call this process adaptation of PIM and PSM [..]" [NMKM12]

[149] Es werden folgende XML Schema betrachtet: Document Type Definition (DTD) [BPSM+08], Relax-NG [CM01, JTC08], Schematron [JTC06] und XML Schema Definition (XSD) [FW04].

[150] "Sometimes, classes in one or more PSM schemas may share the same attributes and/or part of their content. Instead of repeating them at several places, we can use inheritance. We need to be able to specify that a class can reuse an already modeled part of a PSM schema. We distinguish two types of inheritance, the structural and the conceptual." [KN12]

[151] "We implemented the inheritance extension in our tool eXolutio [KMN12] [..]" [KN12]

wurde. Das konzeptuelle Modell ist ein *UML-Klassendiagramm*. Ein PSM kann unter anderem in ein XML-Schema überführt werden.

Werden strukturelle (bzw. hinzufügende und/oder löschende), ortsgebundene oder ortsändernde *Operationen* von einem Nutzer durchgeführt, bzw. werden durch den Vergleich zweier *Versionen* eines Schemas eine *Menge von Änderungen* (Mapping) erkannt, können diese Informationen *Ebenen-übergreifend propagiert* werden. Dadurch wird unter anderem die logische Ebene angepasst werden, welche neben XML-Schemata auch andere *XML-Schemasprachen* enthalten kann.

Die Adaption der Instanzen, das heißt die Änderung der extensionalen Ebene, erfolgt durch ein aus den Operationen bzw. dem versionsbedingten Mapping hergeleiteten *XSL-Stylesheet* (Extensible Stylesheet Language). Mit dem *Revalidierungsskript* kann die Gültigkeit vorhandener Dokumente hergestellt werden.

Bewertung

Der Ansatz der Forschungsgruppe zeichnet sich durch die *Fünf-Ebenen-Architektur* aus, besonders die konzeptuelle Ebene mit dem Plattform-unabhängigen und Plattform-spezifischen Modell sind charakteristisch. Obwohl die fünf Ebenen nicht zuletzt aufgrund des *MDA* (Model-Driven Architecture) Ansatzes gewählt wurden, sind allgemein nur drei entscheidend für die XML-Schemaevolution. Dies sind die konzeptuelle, logische (XML-Schema) und extensionale (XML-Dokumente) Ebene. Die Behandlung der operationalen Ebene ist eine offene Problematik[152], während die obigen beiden Ebenen als konzeptuelle zusammengefasst werden[153].

Ein Problem ergibt sich bei der Formulierung von *Änderungen auf tieferen Ebenen*, zum Beispiel bei Anpassungen von XML-Schema. Diese werden zur Erhaltung der Konsistenz aufwärts propagiert, insofern dies erwünscht ist. Dieses *Reverse-Engineering* führt zur automatischen Transformation eines Plattform-spezifischen Modells. Die Überführung in das allgemeine PIM ist allerdings problematisch und nicht automatisiert, zur Zeit existiert hier nur eine partielle Lösung.[154]

Eine weitere Unklarheit ergibt sich aus der Möglichkeit *unterschiedliche XML-Schemasprachen* auf logischer Ebene in Kombination mit der automatischen Propagierung zu verwenden. Wird nun ein XML-Schema verwendet und zum Beispiel eine ungeordnete Menge (<all>) eingeführt, wie soll diese in eine DTD propagiert werden, oder ist dies standardkonform ausgeschlossen? Die vorliegenden Arbeiten gehen auf diese Problematik nicht ein. Es wird zwar generell die Unterstützung von XML-Schema, DTD, Relax-NG und Schematron hervorgehoben, allerdings beinhalten Beispiele und Anwendungen im Allgemeinen XML-Schema.

[152] "The last but not least related open problem is the propagation of changes from data structures to respective operations, i.e. XPath/XQuery queries, XSLT scripts etc." [MNM12]

[153] "The platform-independent and platform-specific levels are called conceptual levels." [KN10a]

[154] "[..] we have proposed a partial solution. However, the issue has not yet been fully solved, since the interpretation cannot be found automatically and efficiently." [NMKM12]

Mit dem Prototypen können Änderungen erfasst bzw. aus dem Vergleich von Versionen eines Schemas hergeleitet werden.[155] Im zweiten Fall ist der *Zeitpunkt zur Erzeugung von Revalidierungsskripten* klar, bei der direkten Anwendung ist dies allerdings nicht eindeutig. Ein Revalidierungsskript wächst mit der Anzahl der Operationen.[156] Aussagen, wann ein Skript erzeugt wird, ob eine bestimmte Operationsanzahl ratsam wäre oder ob es einen besonders günstigen Zeitpunkt gibt, existieren nicht. Bezüglich der Adaption der Instanzen besteht noch Forschungsbedarf.[157]

Der *Vergleich von Versionen* ist das Hauptprinzip zur Erfassung von Änderungen, wobei die Beziehungen bzw. Verknüpfung von Konstrukten durch die Anwendung von Operationen zwischen Versionen für die Schemaevolution entscheidend sind.[158] Werden nun beide Versionen importiert, existieren die Verknüpfungen nicht und ein *manuelles Mapping* ist notwendig.[159] Es wird demnach vorausgesetzt, dass Versionen innerhalb des Prototyps erzeugt wurden.[160] Inwieweit diese Versionen vorgehalten oder verwaltet werden, und wie dieser Ansatz auf die vollzogenen Änderungen an den unterschiedlichen Versionen übertragen werden kann, wird nicht beschrieben. Diese Thematik ist ein offenes Problem [NMKM12].

Bezüglich des *Standards von XML-Schema* werden verschiedene Konzepte umgesetzt. Auf *schemaspezifische Aspekte* wird allerdings nicht eingegangen.[161] Eine detaillierte Beschreibung der Propagierung von Operationen mit speziellen Schemaelementen soll in einem Technischen Bericht von [MN09] enthalten sein.[162] Dieser Bericht existiert zum jetzigen Zeitpunkt nicht (mehr) und wird weder in den Publikationslisten der Autoren ([Mlý15], [Nec15a] und [Nec15b]), noch in der allgemeinen Liste der gesamten Gruppe in [XML15] aufgelistet. Eine entsprechende Websuche war ebenso erfolglos. Für eine detailliertere Aussage über den Umfang der Umsetzung des Standards wäre der Bericht allerdings zwingend notwendig.

[155] "There are two possible ways to recognize changes - recording of the changes as they are conducted during the design process and comparing the two versions of the schema." [MMN11]

[156] "The stylesheet grows (counting the number of top-level templates) with the amount of changes made in the schema, not with the complexity of the schema." [MMN11]

[157] "We also want to follow up our research in the area of document adaptation ([MMN11]), where we proposed an algorithm for generating an adaptation script to transform documents valid against one version of a schema into documents valid against other version of the same schema." [MN13]

[158] "The main principle is to compare the two versions, but during the users evolution operations, each construct stays linked to its other versions." [MMN11]

[159] "However, when both old and new version of the schema is imported to the system, the mapping needs to be defined." [MMN11]

[160] "Our approach expects that there may exists several versions (which can be edited separately) in the system and can produce revalidation script between any selected pair (including translation from a new version to the old one)." [MMN11]

[161] "Note that for the sake of simplicity we do not deal with XML schema specific aspects such as possibilities of creating a global or local element or equivalent content model." [MN09]

[162] "The detailed description of propagation of operations with particular schema items can be found in [13] (Five-Level Multi-Application Schema Evolution. Technical Report 2008/7, Department of Software Enginneedring, Charles University, Prague, Czech Republic, 2008., Anm. d. Autors)." [MN09]

Bestimmte schemaspezifische Konstrukte werden nach den Angaben der Forschungsgruppe nicht unterstützt. Dazu zählen Elemente mit gemischtem Inhalt (*mixed-content*), Elemente mit einfachem Inhalt (*simple content*) und Attributen, sowie Wurzelelemente mit einfachem Inhalt.[163] Des Weiteren wird die *Vererbung von komplexen Typen* mit deren Möglichkeiten der Erweiterung und Einschränkung nicht unterstützt.[164] Aus der syntaktischen Sicht wird dies allerdings als *Syntaktischer Zucker* beschrieben.[165]

Aktuell werden keine *Constraints* unterstützt, allerdings wird in [MN13] eine Möglichkeit unter Verwendung von Schematron thematisiert. Die in [MLMK05] beschriebene reguläre Baumgrammatik als Grundlage für die logische Ebene in eXolutio und somit als einheitliches Modell der XML-Schemasprachen unterstützt Integritätsbedingungen nicht.[166]

Inwieweit *Wildcards* und *externe Deklarationen und Definitionen* unterstützt werden, kann nicht abschließend eingeschätzt werden. Da diese nicht thematisiert werden, die Forschungsgruppe allerdings signifikante Beiträge für den Prozess der XML-Schemaevolution normalerweise publiziert, kann von einer nicht vorhandenen Realisierung bzw. Beachtung ausgegangen werden.

Ein letzter Aspekt ist die *Generierung von Elementinhalten*. Aktuell wird vorausgesetzt, dass notwendige Daten im Dokument vorliegen.[167] Diese Daten können zwar bereits in einem Revalidierungsskript aggregiert werden[168], weitergehende Ansätze sind allerdings noch nicht realisiert[169]. Es wird unter anderem angedacht, das PIM so zu erweitert, dass es Verknüpfungen zu Datenspeichern geben soll.[170]

[163] "[..] we did not cover all the constructs of XML schemas. From among the less important aspects let us mention mixed-content elements, elements with simple content and attributes or a root element with a simple content." [NMKM12]

[164] "A more important aspect is the wide support of the inference of complex types in XML Schema using restriction or extension, i.e. type inheritance." [NMKM12]

[165] "From the syntactic point of view it can be considered as a 'syntactic sugar'." [NMKM12]

[166] "Except for integrity constraints (such as key, unique and keyref constraints, Anm. d. Autors), the rest of the these features can be described in our framework." [MLMK05]

[167] "To date, XSEM-Evo (XCase evolution framework, Anm. d. Autors) is able to deal with changes that modify the structure and data present in the document." [MN12]

[168] "Several aggregation functions can be offered (e.g. sum, count, avg, max, min known from relational databases or concat inlining the respective values)." [MKMN11]

[169] "Since new data are often required for new versions, we will focus our future work on obtaining this data for the revalidated documents." [MMN11]

[170] "For this purpose, we will utilize the existing connection between PIM and PSM and a new similar connection between PIM and the model of a data storage [..]" [MKMN11]

3.2.7. Weitere Arbeiten

In [RB06] wird die "Notwendigkeit für die Schemaevolution motiviert. Diese ist notwendig, wenn neue oder geänderte Anforderungen auftreten, Defizite im aktuellen Schema korrigiert oder eine Migration auf eine neue Plattform erfolgt."[171] Des Weiteren wird eine *umfangreiche Literatursammlung* vorgestellt, in welcher auf die verschiedenen Publikationen in den Kategorien der Evolution (u.a. Datenbanken, XML-Schema und Ontologien) hingewiesen wird.

Eine Kategorie ist die Evolution von Ontologien, welche in [Har11] vorgestellt wird. In diesem Zusammenhang wird in Kapitel 2 ein detaillierter "Überblick über verwandte Arbeiten auf dem Gebiet der Schema- und Ontologieevolution gegeben" [Har11]. In [HTR11] wird der Überblick um tabellarische Auflistungen ergänzt, in denen der Fokus der untersuchten Systeme, die Typen von Änderungen, das Evolutionsmapping, die Propagierung von Änderungen, der Support der Versionierung und die Infrastruktur (d.h. Implementierungsdetails und GUI) dargestellt werden.

Auswirkungen von Operationen

Die *Auswirkungen von Basisoperationen* auf Kontextknoten in einem XML-Schema werden in [KMH05] thematisiert. Dort werden Möglichkeiten definiert, mit denen auf die Verletzung von Schemabedingungen (*schema constraints*) durch die Anwendung von Operationen reagiert werden kann. Dies sind die Ablehnung der Operation oder die Änderung des Schemas mit anschließender Wiederholung der Operation. Die zweite Möglichkeit kann manuell durch einen Nutzer, oder automatisiert durch einen Algorithmus erfolgen.

In [CRP09] werden weitere Auswirkungen von Modifikationen klassifiziert. Dies sind nicht-verletzende (*non-breaking changes*), verletzende aber lösbare (*breaking and resolvable changes*), sowie verletzende aber nicht lösbare (*breaking and unresolvable changes*) Typen von Änderungen. Des Weiteren werden additive, subtraktive und updatende Operationen auf Metamodellen eingeführt.

Typen von Operationen

Einfache Operationen auf den Komponenten eines konventionellen Schemas (d.h. XML-Schema) werden in [BBGO12] beschrieben. "Dort werden insgesamt 126 Operationen eingeführt."[172] [BBGO12] Dies sind jeweils Operationen zum Einfügen, Löschen und Updaten von XML-Schemakomponenten.

Komplexe Operationen im selben Framework werden in [BGOB14] vorgestellt. "Dies sind 39 höhere Basisoperationen zur Behandlung von Elementen, Attributen

[171] "Obviously, the need for schema evolution occurs very often in order to deal with new or changed requirements, to correct deficiencies in the current schemas or to migrate to a new platform." [RB06]

[172] "[..] we propose the set of primitives for changing a conventional schema (their total number is one hundred and twenty-six)." [BBGO12]

und Constraints, sowie zehn höhere Operationen zum Umgang mit kompletten, konventionellen Schemata bzw. mit Teilen daraus."[173] [BGOB14]

In [Har07] wird "im Speziellen die Verschiebung (*move*) von Elementen innerhalb von XML-Schemas analysiert". Es wird darüber hinaus eine Klassifikation von Änderungsoperatoren für XML-Schema eingeführt. "Allgemein können in einem Schema Komponenten hinzugefügt (*Add*), gelöscht (*Delete*) oder verändert (*Change*) werden [..]. Auf der anderen Seite können die Operatoren bzgl. der Änderung der Informationskapazität, das heißt nach Auswirkungen auf mögliche Instanzdaten, gruppiert werden." [Har07] Abbildung A.6 illustriert den Zusammenhang zwischen den Operatoren und der Informationskapazität.

Auswirkungen auf Anfragen

In [MML07] werden ebenso XML-Schemaänderungen als Basisoperationen und komplexe Operationen vorgestellt. Zusätzlich dazu wird eine weitere Problematik thematisiert, die *Auswirkung der Evolution auf XML-Anfragen.* Ausgehend von der Taxonomie der Änderungen wird geklärt, welchen Einfluss die entsprechenden Operationen haben und wie unter Beachtung bestimmter Richtlinien[174] der negative Einfluss auf Anfragen verhindert werden kann.

Ein Framework für die Sicherstellung der Kompatibilität von Anfragen wird in [GLQ08] vorgestellt. "Das System kann verwendet werden um zu prüfen, ob die Anpassung einer bestimmten Anfrage durch eine Schemaevolution notwendig ist."[175] [GLQ08] "Des Weiteren wird bei Bedarf die Reformulierung von betroffenen Anfragen erleichtert."[176] [GLQ08]

3.3. Zusammenfassung der vorgestellten Ansätze

In diesem Kapitel wurden klassische und aktuelle Ansätze der Schemaevolution vorgestellt. Es wurden sowohl das *Relationenmodell*, als auch *objektorientierte Schemata* und *DTDs* in **Abschnitt 3.1** thematisiert.

Die XML-Schema-fremden Ansätzen wurden primär bezüglich deren Möglichkeiten zur Evolution untersucht. Dabei wurden sowohl *Defaultwerte*, als auch die Notwendigkeit einer vollständigen *Historie* von Transformationsschritten und die eindeutige *Identifikation* von sich ändernden Objekten thematisiert. Des Weiteren wurden *Schemaoperationen* und deren *Klassifikationen*, sowie die *Objektidentität*

[173] "We have defined thirty-nine basic high-level operations. They deal with XML Schema elements, attributes, and constraints. We have also defined ten complex high-level operations which deal with entire conventional schema and portions of conventional schema (or subschema)." [BGOB14]

[174] Beispiel einer Richtlinie: "Do not change element names when they are referred in a query." [MML07]

[175] "The system can be used for checking whether schema evolutions require a particular query to be updated." [GLQ08]

[176] "With this tool designers can examine precisely the impact of schema changes over queries, therefore facilitating their reformulation." [GLQ08]

vorgestellt. Diese Aspekte und ebenso die Anwendung einer *Schemaevolutionssprache* werden nachfolgend im eigenen Ansatz integriert.

Im Zusammenhang mit DTDs wurde *XEM* vorgestellt. Die in XEM integrierten, *komponentenabhängigen Schemaoperationen* und *Lokationspfade* sind ebenso in der XML-Schemaevolution notwendig. Des Weiteren wurde mit dem Algorithmus *DTD-Diff* die Möglichkeit der Anwendung von Operationen auf ein *abstraktes Datenmodell* (*DTD Data Model*) erläutert. Dieser Aspekt wurde durch die Einführung eines konzeptuellen Modells im eigenen Ansatz integriert.

Die aktuellen Ansätze des **Abschnitts 3.2** beinhalten die XML-Schema-spezifischen Möglichkeiten sowohl von Datenbankherstellern (*Oracle*, *IBM* und *Microsoft*), als auch von *Altova* und XML-Datenbanken (*Tamino*, *Sedna* und *eXist-db*).

Diesen Ansätzen ist gemein, dass die angestrebte XML-Schemaevolution nicht umgesetzt wird. Dies liegt entweder an einer stark *eingeschränkten Realisierung von Änderungsoperationen*, die keine Gültigkeitsverletzungen ermöglichen, oder an einer Konzentration auf Teilaspekte, sodass die XML-Schemaevolution nur mit *erheblichem, manuellem Zusatzaufwand* möglich ist. In den entsprechenden Abschnitten wurden für jeden XML-spezifischen Ansatz jeweils die Realisierung der XML-Schemaevolution in einem kurzen Absatz abschließend zusammengefasst.

Wesentlich umfangreicher wurden mit *X-Evolution*, dem *GEA-Framework* und *XCase* die Arbeiten anderer Forschungsgruppen zusammengefasst und bewertet. Auf eine detaillierte Wiederholung wird an dieser Stelle daher verzichtet.

Die Ansätze realisieren eine XML-Schemaevolution, allerdings nicht in dem *Umfang und Detaillierungsgrad* der vorliegenden Arbeit. Es werden ebenso *konzeptuelle Modelle* angewendet, *Schemaoperationen* beschrieben und *Änderungen auf XML-Dokumenten* hergeleitet. Dies geschieht zumeist in Hinblick auf den Umfang von XML-Schema auf kleinen Teilmengen, sodass Bestandteile wie *Wildcards*, *Constraints* oder *externe Deklarationen und Definitionen* nicht beachtet werden. Teilweise werden sogar elementare, primäre Schemakomponenten wie *einfache, abgeleitete Datentypen* nicht berücksichtigt (X-Evolution und GEA), oder als *syntaktischer Zucker* als unwesentlich klassifiziert (XCase).

Die *automatisierte Erstellung von Transformationsschritten* aus erfassten Änderungsoperationen wird durch keinen der Ansätze thematisiert, sodass dies neben dem Umfang und Detaillierungsgrad ein weiteres Alleinstellungsmerkmal der vorliegenden Arbeit ist. Es werden bei den obigen Ansätzen zwar Möglichkeiten zur Erzeugung von Skripten zur Überprüfung der Gültigkeit vorgestellt, allerdings fehlen teilweise unter anderem Aussagen zur *Generierung von Elementinhalten* bei eingefügten, zwingenden Strukturen (XCase). Beschränkungen diesbezüglich auf kapazitätserweiternde Schemaoperationen oder auf optionale Änderungen sind für die angestrebte XML-Schemaevolution wenig hilfreich (GEA). Dies gilt ebenso für datentypspezifische *Defaultwerte*, insofern numerische Datentypen benötigt werden (X-Evolution).

Der Abschnitt 3.2 wird mit *weiteren Arbeiten* beendet, welche zusätzlich zu

den bis dahin vorgestellten Ansätzen existieren, allerdings nicht in die bisherige Struktur des Kapitels integriert wurden. Es wurden sowohl *Überblicksarbeiten*, als auch Arbeiten zu *Auswirkungen und Typen von Operationen* vorgestellt. Ein letzter Aspekt war der Effekt von *Schemaänderungen auf existierende Anfragen*.

Abschließende Betrachtung

In dem Kapitel wurden klassische, teils XML-Schema-fremde und aktuelle, XML-Schema-spezifische Ansätze thematisiert und abschließend zusammengefasst. Es wurde erläutert, dass keiner der thematisch ähnlichen Ansätze die in der vorliegenden Arbeit angestrebten XML-Schemaevolution zufriedenstellend löst. Dies gilt insbesondere unter Beachtung der in Kapitel 1.1 beschriebenen Problemstellung, sowie den daraus hergeleiteten Zielsetzungen und Schwerpunkten der Arbeit.

Somit ist die Notwendigkeit gegeben, einen neuen, alternativen Lösungsansatz zu entwickeln und vorzustellen. Im nächsten Kapitel wird daher der erste Schwerpunkt thematisiert (d.h. *Änderungen*[177]). Es werden die konzeptuelle Modellierung von XML-Schema, sowie die Verwaltung und Speicherung von Modellen erläutert.

[177] siehe auch: Kapitel 1.1.2 (Schwerpunkte der Arbeit)

4. Lösungsansatz

An der Universität Rostock wurde ein konzeptuelles Modell für XML-Schema entwickelt: **EMX** (Entity Model for XML-Schema) [Ste06]. *EMX* wurde im Forschungsprototypen **CoDEX** (Conceptual Design and Evolution of XML schemas) umgesetzt und in [Kle07a] und [Kle07b] publiziert.

Dieses Modell bildet die Grundlage für den eigenen Lösungsansatz. Wesentliche Teile dieses Kapitels wurden in [NKH12] und [NKH13b] veröffentlicht. Auf eine Markierung von jeder wörtlichen Übernahme aus diesen Publikationen wird in der vorliegenden Promotionsschrift aufgrund der verminderten Lesbarkeit verzichtet.

4.1. Konzeptuelle Modellierung

Die Verwendung bzw. Definition eines konzeptuellen Modells ist bei der XML-Schemaevolution ein sinnvoller Schritt. XML-Schema ist nach einhelliger Meinung

- "sehr komplex und nicht leicht zu durchschauen" [Lau05],
- "groß, komplex und voller Fehler"[1] [Bra02] und
- "zweifelsohne die am Schwersten zu verstehende Spezifikation, die ich (J. Clark, Anm. d. Autors) je gelesen habe"[2] [Cla02]

Die Vorteile der Schemaevolution auf einem konzeptuellen Modell sind gemäß [Kle07b] unter anderem der *Entwurf auf abstrakterem Niveau*, das *Verstecken von Details* und die *Konzentration auf das Wesentliche*. Die Modellierung und Evolution von XML-Schema wird durch ein konzeptuelles Modell signifikant erleichtert.[3]

4.1.1. Konzeptuelles Modell

EMX (Entity Model for XML-Schema) [Ste06] ist formal ein *gemischter Graph*, der durch das Tripel $G = (V, E, A)$ beschrieben wird.

Ein Graph (G) besteht aus einer Menge von Knoten (V), ungerichteten Kanten (E), sowie gerichteten Kanten (A). Die Knotenmenge besteht aus disjunkten Teilmengen, wobei jedes Element einem der nachfolgenden Typen zugeordnet wird:

[1] Übersetzung aus [Sch03] übernommen: "XML Schema is large, complex, and buggy." [Bra02]

[2] Übersetzung und Original aus [Sch03] übernommen (Original ist online nicht mehr auffindbar): " ... it is without doubt the hardest to understand specification that I have ever read." [Cla02]

[3] siehe auch: These 1

Element, komplexer Typ, einfacher Typ (built-in, benutzerdefiniert, list oder union), Gruppe, Attributbox, Annotation, externe Entität (einfacher Typ, komplexer Typ oder Element) oder Modul. Einfache Typen und externe Entitäten werden zusätzlich in die eingeklammerten Varianten unterteilt, sodass eine externe Entität zum Beispiel einen einfachen Typen darstellen kann.

Eine Kante wird jeweils durch (un-)geordnete Knotenpaare beschrieben, deren Elemente aus der Menge der Knoten entnommen werden. Des Weiteren existieren Regeln, die sowohl für gerichtete als auch ungerichtete Kanten gültige Kombinationen festlegen. Das formale Modell von EMX gemäß [Ste06] ist in A.8, die erlaubten Kantenkombinationen in Abbildung A.9 dargestellt.

Anpassung von EMX

In [NKH12] wurde das formale Modell angepasst (*angepasstes EMX*). Dieses beinhaltet weiterhin unterschiedliche *Entitätstypen* als Knoten (N - Node)[4] und *gerichtete Kanten* (E - Edge)[5] zwischen diesen, allerdings wurden die ungerichteten Kanten entfernt.[6] Eine weitere Menge wurde abschließend hinzugefügt, welche *zusätzliche Informationen* (F - Feature)[7] des Modells beinhaltet. Die Features sind für ein EMX im Allgemeinen und für die Evolution notwendig.

Es ergibt sich für das angepasste EMX das Tripel

$$EMX = (N_M,\ E_M,\ F_M)\ ,$$

wobei das tiefgestellte M ($\_{}_M$) als Bezeichner für das konzeptuelle Modell steht.

Die ungerichteten Kanten dienen der Angabe von Beziehungen, bei denen die Hierarchie der beteiligten Knoten noch nicht feststeht [Kle07b]. Sie sind nur während des Entwurfs eines EMX zulässig und können in den meisten Fälle automatisch aufgelöst werden [Ste06]. Diese Kanten tragen die Semantik, dass die Beziehung in beide Richtungen relevant bzw. unbekannt ist [Kle07b]. Im ersten Fall können diese durch zwei gerichtete Kanten ersetzt werden [Kle07b]. Der zweite Fall kann mit Hinblick auf ein zu modellierendes XML-Schema nicht auftreten, da hier laut Standard jeder Komponente eine bestimmte Eltern-Kind-Beziehung in der hierarchischen Struktur zugeordnet werden kann. Ungerichtete Kanten sind somit ein nicht notwendiges Artefakt, das im angepassten EMX nicht übernommen wird.

Das angepasste EMX wird nachfolgend vereinfacht als EMX bezeichnet, das Akronym bleibt unverändert bestehen. Insofern eine Unterscheidung notwendig sein sollte, wird dieses an der entsprechenden Stelle explizit kommuniziert werden.

[4] siehe auch: Kapitel 4.1.1 (Entitätstypen von EMX)
[5] siehe auch: Kapitel 4.1.1 (Kanten von EMX)
[6] Im angepassten EMX ist E die Menge der gerichteten, in EMX allerdings die der ungerichteten Kanten.
[7] siehe auch: Kapitel 4.1.1 (Features von EMX)

Entitätstypen von EMX

Die Knotenmenge des angepassten EMX besteht weiterhin aus Entitätstypen, allerdings wurden einerseits *Constraints* hinzugefügt, andererseits wurden Vereinfachungen und Umbenennungen vorgenommen, die nachfolgend beschrieben werden.

Eine Attributbox wird als *Attributgruppe* bezeichnet, und ist somit namentlich dem entsprechenden Konstrukt eines XML-Schemas ähnlicher. Eine Umbenennung in Attribut wäre ebenso möglich gewesen, die Idee wurde aber verworfen.

Des Weiteren wurden *externe Entitäten* entfernt. Deren Untertypen (einfacher Typ, komplexer Typ bzw. Element) sind in die entsprechenden nicht-externen Entitätstypen integriert worden. Dies ist aus Sicht eines XML-Schemas sinnvoll, da bei der Anwendung nicht unterschieden wird, ob ein Typ zum Beispiel importiert wurde oder nicht. Eine Zuordnung per qualifiziertem Namen muss möglich sein.

Eine explizite Unterscheidung zwischen den Untertypen eines *einfachen Typs* (built-in, benutzerdefiniert, list bzw. union) wurde ebenso abgeschafft. Dies ist mit dem Umstand zu begründen, dass ein eingeschränkter Typ (d.h. im EMX benutzerdefiniert) eine Liste repräsentieren kann. Eine disjunkte Einordnung einer entsprechenden Entität wäre schwierig und wenig intuitiv.

Es ergibt sich somit die disjunkte und veränderte Menge von Entitätstypen:

$$N_M = \textit{elements} \cup \textit{attribute-groups} \cup \textit{simple-types} \cup \textit{complex-types} \cup \textit{groups} \cup \textit{modules} \cup \textit{annotations} \cup \textit{constraints}$$

Jedes Element eines Entitätstyps besitzt unter Beachtung des *Element Information Items* (EII)[8] bestimmte Attribute. Ein Überblick über EIIs der Entitätstypen ist in Abbildung 4.1 gegeben. Ein Element vom Entitätstyp *elements* besitzt gemäß

Entitätstyp	Element Information Item
elements	<element>
attribute-groups	<attribute>, <attributeGroup>
simple-types	<simpleType>
complex-types	<complexType>
groups	<all>, <choice>, <sequence>, <any>, <anyAttribute>
modules	<include>, <import>, <redefine>, <overwrite>
annotations	<annotation>
constraints	<key>, <unique>, <keyref>, <assert>, <assertion>

Abbildung 4.1.: Überblick Entitätstypen mit zugeordneten Element Information Items

EII-Beispiel 2.3 unter anderem einen Name (*name*), eine Typinformation (*type*), Häufigkeitsangaben (*minOccurs* und *maxOccurs*), Defaultwerte (*default*), sowie

[8] siehe auch: Kapitel 2.1.1 (Strukturbeschreibung des XML-Schemas)

Attribute zur Darstellung, welche nicht im EII enthalten sind. Ein Element vom Entitätstyp *attribute-groups* besitzt die Vereinigung der Attribute der Element Information Items von *<attribute>* und *<attributeGroup>*.

Identifikation von Entitäten

Ein weiteres Attribut, welches nicht in den Element Information Items enthalten ist, dient der eindeutigen Identifikation einer jeden Entität. Dieses Attribut ist die **EID** (EMX ID), welche eine ganze Zahl darstellt (EID $\in \mathbb{Z}$).

Die EID ist eineindeutig in jedem EMX und dient nicht nur der Identifikation, sondern auch der Lokalisierung innerhalb des konzeptuellen Modells, als Hilfsmittel zur Speicherung, zur Auswertung und Optimierung von Operationen auf einem EMX-Knoten, etc.. Details zur Erzeugung, Anwendung und Verwaltung der EIDs werden in den entsprechenden Abschnitten und Kapiteln erläutert.

Kanten von EMX

Die Kanten des angepassten EMX sind ein Tupel, bestehend aus den EIDs beteiligter Entitäten von Entitätstypen.

$$\forall\ e \in E_M\colon\ e = (X,\ Y) \text{ mit } X,\ Y \in N_M$$

In Abbildung 4.2 werden die gerichteten Kanten dargestellt, welche gemäß des Standards von XML-Schema definiert wurden. Ein x im Schnittpunkt bedeutet,

Kante(X,Y) von X / zu Y	element	attribute-group	group	complex-type	simple-type	annotation	constraint	module	schema
element			x					x	x
attribute-group		x		x				x	x
group		x	x	x					
complex-type	x							x	x
simple-type	x	x						x	x
annotation	x	x	x	x	x		x	x	x
constraint	x			x	x				
module									x

Abbildung 4.2.: Überblick gerichteter Kanten zwischen Entitätstypen im EMX

dass eine Verbindung zwischen den beteiligten Knoten erlaubt ist. Es existiert zum Beispiel ein Element des Entitätstyps *elements* mit der EID = 1 und ein Element des Typs *simple-type* mit der EID = 2. Eine Kante (1, 2) bedeutet, dass die Entität

mit der EID = 1 (*von X*) den einfachen Typen mit der EID = 2 (*zu Y*) enthält. Die Variante (2, 1) hingegen würde signalisieren, dass ein einfacher Typ ein Element enthält, was laut Standard und ebenso laut Abbildung 4.2 nicht möglich ist.

Features von EMX

Die Features (F_M) beinhalten zusätzliche Informationen des EMX. Dazu zählen unter anderem die Attribute des Element Information Items von *<schema>*. Ein Schema bekommt ebenso eine EID, sodass Kanten zwischen diesem und den Knoten möglich sind. Die erlaubten Kombinationen sind in Abbildung 4.2 dargestellt. Ein Schema kann im EMX demnach in keinem Knoten enthalten sein. Zeitgleich existiert im zusammenhängenden Graphen des konzeptuellen Modells mindestens eine Kante mit der EID des Schemas. Das Schema ist die Wurzel des EMX.

Alle Knoten, die in einer Kante mit dem Schema enthalten sind, sind aus Sicht des XML-Schemas und unter Beachtung der Gültigkeitsbereiche[9] somit globale Komponenten. Im Gegensatz dazu sind Komponenten ohne eine solche Kante lokal.

Da ein XML-Schema durch ein EMX repräsentiert wird und Informationen wie Dateinamen, Versionsnummern, etc. nicht verloren gehen sollen, werden diese in den Features inkludiert. Dazu zählen ebenso nutzerabhängige Informationen wie zum Beispiel Defaultwerte bzw. dem Verfahren zur Generierung von solchen oder generelle Konfigurationen zum Umgang mit Operationen auf dem EMX. Es wäre unter anderem denkbar die Pragmatik von Evolutionsoperationen zu definieren, sodass nur solche Operationen auf dem EMX zugelassen werden, die keinen Informationsverlust hervorrufen (d.h. Verbot instanzreduzierender Operationen).

Die Features sind Bestandteil des Kapitels 7 und werden dort näher thematisiert. An dieser Stelle wären zu viele Vorgriffe einschließlich Erklärungen notwendig.

4.1.2. Visualisierung

Für die Komponenten des konzeptuellen Modells existiert eine grafische Repräsentation, die in Abbildung 4.3 dargestellt wird. Ausgehend vom *Abstract Data Model* (ADM) des XML-Schemas wird das *Element Information Item* (EII) mit entsprechendem EMX-Knoten dargestellt.[10] Des Weiteren wird die grafische Repräsentation vorgestellt, wobei zur Verbesserung der Lesbarkeit der dargestellte Buchstabe zusätzlich neben der Grafik ergänzt wurde. Zusätzliche Hinweise über nicht visualisierte Knoten sind ebenso enthalten.

Eine Deklaration (*declaration*) des ADM ist demnach unter anderem durch die XML-Repräsentation *<element>* im EII realisiert. Diese entspricht einer Entität vom EMX-Knoten *elements* (vgl. auch Entitätstypen in Abbildung 4.1), welche grafisch durch ein *blau umrandetes E mit weißer Schriftfarbe* symbolisiert wird.

[9] siehe auch: Kapitel 2.1.1 (Gültigkeitsbereiche in XML-Schema)
[10] siehe auch: Kapitel 2.1.1 (Strukturbeschreibung des XML-Schemas)

Abstract Data Model	Element Information Item	Knoten in EMX	Repräsentation
declaration	<element>	element	E
	<attribute>	attribute-group	@
group-definition	<attributeGroup>		
model-group	<all>, <choice>, <sequence>	group	G
	<any>, <anyAttribute>		GW @W
type-definition	<complexType>	complex-type	Implizit und hergeleitet
	<simpleType>	simple-type	Implizit und spezifizierbar
annotation	<annotations>	annotation	#
constraint	<key>, <unique>, <keyref>	constraint	K
	<assert>	Implizit im complex-type	
	<assertion>	Restriction im simple-type	
N.N.	<schema>	EMX-Datei selber	
N.N.	<include>, <import>, <redefine>, <overwrite>	module	M

Abbildung 4.3.: Abbildung und Visualisierung von EMX-Knoten

Schemata und Module sind nicht Bestandteil des ADM, sodass diese den Eintrag *N.N.* (Not Named) erhalten. Des Weiteren werden das Schema, einfache und komplexe Typen, sowie bestimmte Bedingungen (*<assert>* und *<assertion>*) nicht visualisiert. Während die Bedingungen entweder in einem komplexen Typen implizit oder als Facette im Restriktionstyp[11] enthalten sind, ist ein Schema das EMX selber und wird nicht gesondert dargestellt.

Der Verzicht der direkten Visualisierung von Typinformationen ist eine weitere Veränderung beim angepassten EMX. Dieser ist durch die **Dokument-zentrierte Darstellungsweise** zu begründen, in welcher ein konzeptuelles Modell strukturell möglichst ähnlich zum dargestellten Dokument sein soll. In Abbildung 4.4 wird das XML-Schema aus XML-Beispiel 1.2 als EMX dargestellt. Dieses wurde allerdings

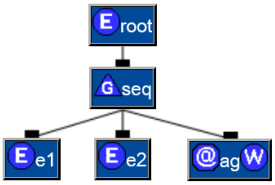


Abbildung 4.4.: EMX des XML-Schemas aus XML-Beispiel 1.2

um eine Attributwildcard ergänzt. Die Abbildung wurde mit Hilfe des Forschungsprototypen aus Kapitel 7 erstellt, sodass Details zur Realisierung der Knoten als *blaue Rechtecke* an dieser Stelle noch nicht gegeben werden.

Im Beispiel werden nur Elemente mit Namen (*root*, *e1*, *e2*), eine Gruppe mit Inhaltsmodell (*seq* - Sequenz)[12], eine Attributgruppe mit Namen (*ag*) und eine Wildcard (in *ag*) dargestellt. Dies sind die **visualisierten EMX-Knoten**. Zusätzlich dazu können Annotationen, Constraints und Module visualisiert werden.

[11] siehe auch: Kapitel 2.1.1 (Einfache Datentypen)
[12] Weitere Inhaltsmodelle der Gruppe sind laut Standard: *ch* (Choice), *all* (Menge) und *empty* (leer).

Die Attribute *a1* und *a2* sind in der Attributgruppe *ag* enthalten. Die einfachen Typen *xs:string* und *xs:decimal* sind implizit in den Elementen und Attributen gegeben und werden von diesen referenziert. Der komplexe Typ *roottype* befindet sich innerhalb der Gruppe *seq* und wird durch diese repräsentiert. Das Schema, die Typen und Attribute sind somit **nicht-visualisierte EMX-Knoten**

Ein Beispieldokument, welches zum XML-Schema des XML-Beispiels 1.2 gültig ist, wird im XML-Beispiel 4.1 dargestellt.

```
<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="example.xsd"
    a1="1" a2="2">
    <e1>1</e1>
    <e1>1</e1>
    <e2>2</e2>
    <e2>2</e2>
</root>
```

XML-Beispiel 4.1: Gültiges XML-Dokument für XML-Schema des XML-Beispiels 1.2

Es wird in diesem eine *Sequenz* von den Elementen *e1* und *e2* dargestellt, welche als Kinderelemente unterhalb eines Elements *root* angeordnet sind. Zudem sind die Attribute *a1* und *a2* enthalten, die zusammen (als *Gruppe*) das Element *root* näher beschreiben. Das EMX aus Abbildung 4.4 kommt dieser Beschreibung und somit dem XML-Dokument des XML-Beispiels 4.1 strukturell sehr nahe.

Visualisierung von Kanten

Die erlaubten Kanten des visualisierten EMX werden ausgehend von Abbildung 4.2 entsprechend der Dokument-zentrierten Darstellung angepasst. Die Anpassungen sind in Abbildung 4.5 dargestellt. Es werden Verschiebungen entlang der *gelben*

Kante(X,Y) von X / zu Y	element	attribute-group	group	complex-type	simple-type	annotation	constraint	module	schema
element			x					x	x
attribute-group		x		x				x	x
group		x	x	x					
complex-type	x							x	x
simple-type	x	x						x	x
annotation	x	x	x	x	x		x	x	x
constraint	x			x	x				
module									x

Abbildung 4.5.: Überblick der Anpassungen gerichteter Kanten zwischen Entitätstypen im visualisierten EMX (ausgehend von Abbildung 4.2)

Pfeile vorgenommen, sodass zum Beispiel Attributgruppen eines komplexen Typs unterhalb einer Gruppe angeordnet werden. Gleichzeitig sind alle Kanten, die mit einem *gelben X* markiert sind, implizit gegeben und werden daher nicht visualisiert. Nach der Anpassung verbleiben nur die Kanten mit einem *schwarzen X*. Das bereinigte Ergebnis ist in Abbildung A.10 dargestellt. Das *zu Y* Element einer Kante wird im EMX durch ein *schwarzes Rechteck* am Ende einer Kante symbolisiert.

4.2. Drei-Ebenen-Architektur

Die Einführung des konzeptuellen Modells führt zu einer **Drei-Ebenen-Architektur**, welche als Erweiterung der Abbildung 1.1 in Kapitel 1.1 (Problemstellung) gilt. Die Architektur ist in Abbildung 4.6 dargestellt.

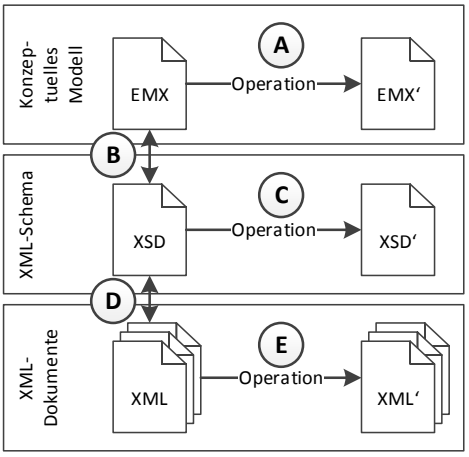


Abbildung 4.6.: Drei-Ebenen-Architektur durch Hinzunahme von EMX

Es existieren die Ebenen des konzeptuellen Modells (**Modellebene**), sowie eine **Schemaebene** und **Dokumentebene** (bzw. *Instanzebene*). Die Ebene des XML-Schemas wird in anderen Ansätzen ebenso als *logische Ebene*, die der XML-Dokumente als *extensionale Ebene* bezeichnet.

Zwischen den Ebenen existieren Abbildungen (**Korrespondenzen**), die in Abbildung 4.6 durch die Buchstaben *B* und *D* dargestellt sind. Die Korrespondenz zwischen einem XML-Schema und einem EMX ist eindeutig, sodass hier von einer *1-zu-1-Abbildung* gesprochen werden kann. Die Korrespondenz zwischen einem XML-Schema und dazu gültigen XML-Dokumenten ist eine *1-zu-n-Abbildung*. Diese Mehrdeutigkeit entsteht unter anderem durch die Optionalität von Komponenten, oder die Heterogenität von Inhaltsmodellen.

Es existieren auf Grundlage der Korrespondenzen im Zusammenspiel mit den Ebenen-spezifischen Operationen (*A*, *C* und *E*) Grundannahmen, welche als Theoreme in [NKH12] beschrieben und hier (angepasst) wiederholt werden.

Theorem 1 *Ein konzeptuelles Modell EMX wird durch die Operation* ***A*** *eines Nutzers verändert zu EMX'. Sei* ***B*** *eine Beschreibungsvorschrift zur Abbildung (Korrespondenz) des konzeptuellen Modells EMX auf ein XML-Schema XSD.* ***C*** *sei die Änderungsoperation zur Überführung von XSD zu XSD'. Dann gilt:*

$$\boldsymbol{A} \ + \ \boldsymbol{B} \ \Rightarrow \ \boldsymbol{C}$$

Theorem 1 besagt: Sind sowohl die Operation des Nutzers auf dem EMX als auch die Korrespondenzen zwischen dem konzeptuellen Modell und dem XML-Schema bekannt, so kann die Operation zum Anpassen des XML-Schemas hergeleitet werden.[13]

Theorem 2 *Ein konzeptuelles Modell EMX wird durch die Operation* ***A*** *eines Nutzers verändert zu EMX'. Sei* ***B*** *eine Korrespondenz zwischen dem konzeptuellen Modell EMX und einem XML-Schema XSD und sei* ***D*** *eine Korrespondenz zwischen dem XML-Schema XSD und XML.* ***E*** *sei die Änderungsoperation zur Überführung von XML-Instanzen XML zu XML-Instanzen XML', die bezüglich XSD' gültig sind. Dann gilt:*

$$\boldsymbol{A} \ + \ \boldsymbol{B} \ + \ \boldsymbol{D} \ \Rightarrow \ \boldsymbol{E}$$

Theorem 2 besagt: Sind sowohl die Operation des Nutzers auf dem EMX als auch die Korrespondenzen zwischen dem konzeptuellen Modell, dem XML-Schema und den XML-Instanzen bekannt, so kann die Operation zum Anpassen der XML-Instanzen hergeleitet werden.[14]

4.2.1. Ebenen-spezifische Operationen

Die Operationen auf den Ebenen sind im Allgemeinen Kombinationen von **add** (Hinzufügen), **delete** (Löschen) und **update** (Ändern) auf den Knoten der Modelle. Eine detaillierte Übersicht wird im nachfolgenden Kapitel 5 gegeben.

Auf Schemaebene sind gemäß des *Abstract Data Models* (ADM) unter anderem folgende Knoten möglich: Definitionen (*type-definitions*), Deklarationen (*declarations*), Annotationen (*annotations*), Gruppendefinitionen (*group-definitions*), Modellgruppen (*model-group-components*), und Bedingungen (*constraints*).[15]

Die Knoten einer XML-Instanz sind Dokumente (*documents*), Attribute (*attributes*), Elemente (*elements*), Prozessanweisungen (*processing-instructions*), Namensräume (*namespaces*), Texte (*texts*) oder Kommentare (*comments*) [BFM+10].

In [NKH12] wurde die Ebenen-spezifische *renameElement*-Operation[16] zur Veranschaulichung der Operationen A, C und E der Abbildung 4.6 vorgestellt. Diese Umbenennung wird nachfolgend (angepasst) wiederholt.

[13] siehe auch: These 2
[14] siehe auch: These 3
[15] siehe auch: Kapitel 2.1.1 (Strukturbeschreibung des XML-Schemas)
[16] In späteren Kapiteln wird *renameElement* als *updelementdef* eingeführt (siehe auch: Kapitel 5).

A: $renameElement_M$ (EID, oldValue, newValue)

$\forall\ n \in N_M \wedge n.EID = EID$:

if $n \in elements_M \wedge oldValue \neq newValue$

then $n.name := newValue$

Erklärung Operation $renameElement_M$: Wird eine Umbenennung auf einem Elementknoten des konzeptuellen Modells durchgeführt (signalisiert durch das tiefgestellte M ($\_M$)), dann wird das Attribut *name* geändert. Dabei wird vorher geprüft, ob es sich um eine sinnvolle Operation handelt ($oldValue \neq newValue$). Die Identifikation des Elements wird durch die eindeutige EID gewährleistet.

C: $renameElement_S$(context, oldValue, newValue)

$\forall\ n, m, k \in N_S \wedge n \neq m \wedge n \neq k \wedge m \neq k \wedge context(n)$:

// Globales Element mit neuem Namen existiert?

if $n, m \in elements_S \wedge n.scope.variety = 'global' \wedge m.scope.variety = 'global'$
$\wedge\ n.name = oldValue \wedge m.name = newValue$

then $newValue := uniqueName(newValue) \wedge$
$n.name := newValue$

if $\exists\ k \in elements_S \wedge k.scope.variety = 'local' \wedge k.ref = oldValue$

then $\forall\ k$: $k.ref := newValue$

// Lokales Element mit neuem Namen, aber anderem Typ existiert?

elseif $n, m \in elements_S \wedge n.scope.variety = 'local' \wedge m.scope.variety = 'local'$
$\wedge\ n.name = oldValue \wedge m.name = newValue \wedge n.type \neq m.type \wedge$
$n.parent.name = m.parent.name$

then $newValue := uniqueName(newValue) \wedge$
$n.name := newValue$

// Lokales Element, kein Namen-Typ-Konflikt?

elseif $n \in elements_S \wedge n.scope.variety = 'local' \wedge n.name = oldValue$

then $n.name := newValue$

// Globales Element, kein Namen-Konflikt?

elseif $n \in elements_S \wedge n.scope.variety = 'global' \wedge n.name = oldValue$

then $n.name := newValue$

if $\exists\ k \in elements_S \wedge k.scope.variety = 'local' \wedge k.ref = oldValue$

then $\forall\ k$: $k.ref := newValue$

Erklärung Operation $renameElement_S$: Wird eine Umbenennung auf einem Elementknoten des XML-Schemas durchgeführt (signalisiert durch das tiefgestellte S ($\_S$)), dann müssen unterschiedliche Nebenbedingungen geprüft werden. Zuerst muss der richtige Elementknoten mit Hilfe des Namens (*oldValue*) identifiziert werden, bei dem zusätzlich der Kontext stimmt (*context(n)*). Zum Kontext zählen unter anderem Angaben zum Gültigkeitsbereich, zu vorhandenen Referenzen und zum direkten Knotenumfeld (Position in einem Inhaltsmodell etc.).[17] Ist das richtige Element gefunden, wird in Abhängigkeit des Gültigkeitsbereichs geprüft ob Namenskonflikte auftreten (diese werden beseitigt durch *uniqueName()*)[18]. Sind die Nebenbedingungen erfüllt, wird dem Attribut *name* ein neuer Wert (*newValue*) zugewiesen. Existieren laut Kontext noch Elementreferenzen zum umbenannten Elementknoten, dann wird das Attribut *ref* der Referenz entsprechend angepasst.

E: $renameElement_D$(context, oldValue, newValue)

$\forall\ n \in N_D$:
  if $n \in elements_D \wedge n.node\text{-}name = oldValue \wedge context(n)$
  then $n.node\text{-}name := newValue$

Erklärung Operation $renameElement_D$: Wird eine Umbenennung auf einem Elementknoten eines Dokuments durchgeführt (signalisiert durch das tiefgestellte D ($\_D$)), dann müssen alle entsprechenden Knoten durch den Namen (*oldValue*), sowie den richtigen Kontext (*context(n)*) identifiziert werden. Aufgrund der Korrespondenz zwischen Schema- und Dokumentebene kann dies mehrere Elementknoten betreffen. Das Attribut *name* von allen identifizierten Knoten bekommt einen neuen Wert (*newValue*) zugewiesen.

4.2.2. Anwendung ebenen-spezifischer Operationen

Nachfolgend wird die *renameElement*-Operation als Beispiel vereinfacht beschrieben. Die notwendigen Hintergründe folgen in späteren Kapitel (5, 6 und 7), sodass hier nur die Idee, allerdings nicht die detaillierte Realisierung vorgestellt wird.

Angenommen es wird das konzeptuelle Modell aus Abbildung 4.4 dahingehend verändert, dass Element *e1* umbenannt wird in *eX*. Diese Operation ist ohne Namenskonflikt möglich und wird daher auf dem EMX durchgeführt und registriert.

Aufgrund der EID des veränderten EMX-Knotens, der eindeutigen Korrespondenz zur Schemaebene, sowie dem *Garden-of-Eden-Modellierungsstil* des XML-Schemas aus XML-Beispiel 1.2 ist bekannt, dass ein globales Element umbenannt wurde. Diese Deklaration ist im Schema durch die Elementreferenz *e1* des komplexen Typs *roottype* innerhalb einer Sequenz referenziert, sodass auch diese geändert

[17] Der Kontext wird abgeleitet aus der EID und dem EMX allgemein (siehe Kapitel 7).

[18] Namenskonflikte werden bei Umbenennungsoperationen auf dem EMX bereits vermieden (siehe auch: Kapitel 7), hier wird eine allgemeine Form von $renameElement_S$ auf dem XML-Schema präsentiert.

werden muss. In $renameElement_S$ entspricht dies dem letzten Fall (*Globales Element, kein Namen-Konflikt?*). Das heißt, dass das Attribut *ref* entsprechend den neuen Wert *eX* erhält. Zeitgleich wird der Kontext ermittelt, in welchem Anpassungen vorgenommen wurden (u.a. EID, Element Knoten, globale Deklaration, lokale Referenz an erster Position in Sequenz, keine Optionalität, neuer und alter Wert).

Gültige XML-Dokumente müssen nun ebenso angepasst werden, insofern diese ein Element mit dem Tagnamen *e1* an den laut Kontext ermittelten Positionen enthalten. Dies kann entweder das Wurzelelement selber sein, da die globale Deklaration direkt im XML-Dokument referenziert werden kann. In diesem Fall sind weitergehende Betrachtungen nicht nötig, da *e1* einen einfachen Typen besitzt und somit keine Verschachtelung möglich ist. Alternativ kann das Element *root* das Wurzelelement sein, welches den *roottype* und somit die Sequenz mit der Elementreferenz enthält. Eine Umbenennung aller Kinderelemente von *root* mit dem Namen *e1* ist zwingend erforderlich, da *e1* eine nicht verbotene Elementreferenz ist. Es müssen in diesem Fall mindestens eine Umbenennung (*minOccurs-Werte* der Sequenz und Elementreferenz multipliziert), maximal allerdings vier Umbenennungen (Multiplikation der *maxOccurs-Werte*) vorgenommen werden.

4.3. Speicherung und Verwaltung von Modellen

Das konzeptuelle Modell sowie dafür notwendige Verwaltungsinformationen werden in relationalen Strukturen gespeichert, da diese wohl vertraut und Jahrzehnte erprobt sind. Des Weiteren bieten entsprechende relationale Datenbankverwaltungssysteme in Hinblick auf die Verfügbarkeit und bezüglich der Umsetzung in Kapitel 7 Vorteile.[19] Dies bezieht sich nicht nur auf die umfangreiche Dokumentation und Kompatibilität sowohl im lokalen, als auch in Server-Client-Umgebungen, sondern auch auf die Möglichkeiten der standardisierten Konnektivität.

4.3.1. Speicherung des konzeptuellen Modells

In [NKH13b] wurde die **logische Struktur** des konzeptuellen Modells eingeführt. Diese beinhaltet eine Übersicht der für die Speicherung notwendigen **Relationen**, sowie deren Assoziationen untereinander. In Abbildung 4.7 ist die logische Struktur mit Legende dargestellt. Insgesamt existieren 18 Relationen (*Rechtecke*), in denen das konzeptuelle Modell gespeichert wird. Es wird dabei unterschieden zwischen visualisierten (*gelb umrandet*) und nicht visualisierten Knoten (*schwarz umrandet*).[20] Des Weiteren werden die EMX-Knoten gemäß des konzeptuellen Modells gesondert markiert (*dick umrandet*). In Abbildung A.11 wird eine Anpassung vorgenommen, welche die unterschiedlichen Relationen farblich den EMX-Knoten

[19] Es wird das relationale Open-Source-Datenbankverwaltungssystem MySQL Version 5.5.25a verwendet.
[20] siehe auch: Kapitel 4.1.2 (Visualisierung)

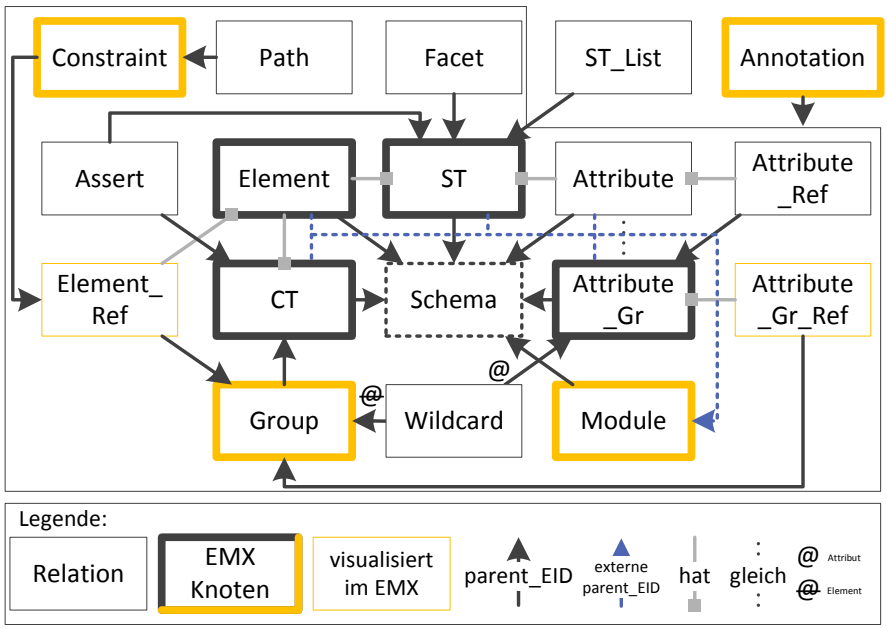


Abbildung 4.7.: Logische Struktur des konzeptuellen Modells

zuordnet. Das heißt, dass die logische Struktur aus deren Perspektive visualisiert wird. Das *Schema* repräsentiert das EMX selber, sodass dieses gesondert (*gestrichelt umrandet*) und zentral dargestellt wird.

Neben den Relationen existieren unterschiedliche **Assoziationen**, zum Beispiel werden auf EIDs basierende Referenzen angegeben (*parent_EID*). Die EID ist eineindeutig in jedem EMX[21], sodass diese als Primärschlüssel in jeder Relation verwendet werden. Annotationen können Entitäten aller Relationen referenzieren, mit Ausnahme der *Annotation* und der *ST_List*. Diese Relation dient der normalisierten Speicherung von EIDs der Teilnehmer eines Vereinigungstyps.[22] Die Relation *Path* beinhaltet die Selektoren (<*selector*>) und Feldwerte (<*field*>) von Constraints. Wildcards werden unterteilt in Attribut- (<*anyAttribute*>) und Elementwildcards (<*any*>). Erstere werden Attributgruppen zugeordnet (@), während Letztere zu den Gruppen gehören (@). Ein Element *hat* einen einfachen (*ST*) oder komplexen Typen (*CT*), ein Attribut kann nur einen einfachen Typen besitzen. Jede Element-, Attribut- oder Attributgruppenreferenz *hat* eine entsprechende Deklaration, die referenziert werden muss.

Da alle Deklarationen und Definitionen entweder als Elternelement das *Schema* oder ein externes *Modul* besitzen, wird von einem globalen Modellierungsstil ausgegangen. Das heißt, dass im Gegensatz zum *Venetian-Blind-Stil* aus [Ste06] der

[21] siehe auch: Kapitel 4.1.1 (Identifikation von Entitäten)
[22] siehe auch: Kapitel 2.1.1 (Einfache Datentypen)

Garden-of-Eden-Modellierungsstil im angepassten EMX verwendet wird.[23] Dieser zeichnet sich durch einen hohen Grad der Wiederverwendbarkeit von Strukturen zur Lasten der Lesbarkeit aus. Der Nachteil wird allerdings durch das konzeptuelle Modell amortisiert. Eine Transformation zwischen den unterschiedlichen Modellierungsstilen ist gemäß [Kap13] möglich, indem lokale Deklarationen und Definitionen global spezifiziert und nachfolgend lokal referenziert werden. Durch eine Erweiterung der logischen Struktur sind alle Modellierungsstile integrierbar, was intern schon möglich ist.[24] Dafür müssen für lokale Deklarationen und Definitionen folgende Assoziationen (*parent_EID*) eingeführt werden: Element zu Group, Attribut zu CT, CT zu Element, ST zu Attribut und ST zu Element.

In Abbildung 4.8 werden die Relationsschemata zur Speicherung des EMX-Knotens *elements* dargestellt. Dies ist ein Ausschnitt der kompletten Übersicht

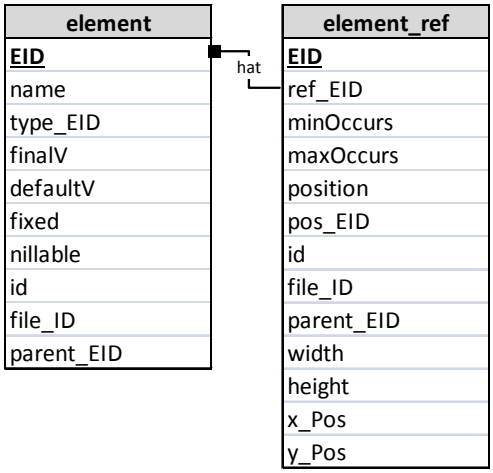


Abbildung 4.8.: Relationsschemata des EMX-Knotens *elements*

aller Schemata in Abbildung A.12. Eine Elementreferenz kann demnach standardkonform unter anderem Häufigkeitsangaben (*minOccurs* und *maxOccurs*), eine Referenz zur Deklaration (*ref_EID*), sowie Angaben zum Elternelement (*parent_EID*) besitzen. Eine Positionsangabe mittels des Attributs *position*[25] und einer Bezugsentität (*pos_EID*) ist möglich. Da Elementreferenzen visualisiert werden, sind Attribute des Erscheinungsbilds ebenso enthalten (*width*, *height*, *x_Pos* und *y_Pos*).

Solche Angaben fehlen hingegen bei der Elementdeklaration. Ein Element besitzt Attribute für den Namen (*name*), für Standardwerte (*defaultV* und *fixed*), für die Abgeschlossenheit (*finalV*), die Nullwertfähigkeit (*nillable*) oder zur Angabe des referenzierten Typs (*type_EID*). Die *parent_EID* gibt an, ob die Deklaration

[23] siehe auch: Kapitel 2.1.3 (Modellierungsstile von XML-Schema)
[24] Konsistente EID Referenzen werden nicht in den relationalen Strukturen, sondern im EMX kontrolliert.
[25] siehe auch: Kapitel 5 (Transformationssprache)

global im Schema spezifiziert oder aus einem Modul importiert wurde.

Eine XML-Schema-ID (*id*) kann ebenso sowohl bei Elementen, als auch bei deren Referenzen gespeichert werden. Auf die Bedeutung von *file_ID* wird in Abschnitt 4.3.3 eingegangen, da dies mit der Verwaltung der Modelle zusammenhängt.

4.3.2. Anwendung der Speicherung des konzeptuellen Modells

In Abbildung 4.9 ist die Speicherung eines konzeptuellen Modells innerhalb der relationalen Strukturen dargestellt. Es wurde das konzeptuelle Modell aus Abbildung

Element

EID	name	type_EID	parent_EID
2	root	10	1
3	e1	8	1
4	e2	9	1

Schema

EID	xmlns_xs
1	http://www.w3.org/2001/XMLSchema

Element_Ref

EID	ref_EID	minOccurs	maxOccurs	parent_EID	x_Pos	y_Pos
5	2			-1	50	25
6	3	1	2	11	25	75
7	4	0	2	11	50	75

ST

EID	name	mode	builtInType	parent_EID
8	decimal	built-in	xs:decimal	1
9	string	built-in	xs:string	1

CT

EID	name	parent_EID
10	roottype	1

Facet

EID	type	value	fixed	st_EID
19	whiteSpace	collapse	1	8
20	whiteSpace	preserve	0	9

Wildcard

EID	parent_EID
18	16

Group

EID	minOccurs	maxOccurs	mode	parent_EID	x_Pos	y_Pos
11	1	2	sequence	10	50	50

Attribute

EID	name	type_EID	parent_EID
12	a1	8	1
13	a2	9	1

Attribute_Ref

EID	ref_EID	useV	parent_EID
14	12	required	16
15	13	optional	16

Attribute_Gr

EID	name	parent_EID
16	ag	1

Attribute_Gr_Ref

EID	ref_EID	parent_EID	x_Pos	y_Pos
17	20	11	75	75

Abbildung 4.9.: Speicherung des konzeptuellen Modells aus Abbildung 4.4

4.4 gespeichert, welches dem XML-Schema aus XML-Beispiel 1.2 entspricht (mit Ausnahme der Wildcard). Die Speicherung wurde dahingehend vereinfacht, dass nur gefüllte Relationen und Attribute dargestellt werden. Es fehlen bei den präsentierten Relationsschemata unter anderem Angaben zur Verwaltung (*file_ID*) und unterschiedliche weitere Attribute, die Abbildung A.12 entnommen werden können. Des Weiteren wurden EIDs aus Gründen der Kompaktheit verkürzt und aufsteigend vergeben.

Neben der Modell-bedingten Einführung einer Attributgruppe (*ag*) als Container

für die Attributreferenzen, ist die Elementreferenz mit der *EID = 5* interessant. Diese besitzt keine Häufigkeitsangaben, was in EMX auch nicht zwingend notwendig ist. Allerdings wird hier ein Elternelement mit *negativer EID* verwendet. Dies ist damit zu begründen, dass keine Elementreferenz des Elements *root* existiert, diese aber visualisiert wird. Negative EIDs dienen im Modell und ebenso in der Umsetzung in Kapitel 7 generell als Platzhalter mit unterschiedlicher Semantik. Dies soll an dieser Stelle noch nicht vorgreifend thematisiert werden.

Das gilt ebenso für die dargestellten Facetten der *built-in-Typen*. Diese sind im Standard des XML-Schemas spezifiziert und werden nachfolgend für den Aufbau von Typhierarchien einfacher Typen benötigt. Dies wird in Kapitel 7 beschrieben.

4.3.3. Verwaltung von Modellen

Zur Verwaltung der konzeptuellen Modelle wurden die in Abbildung 4.10 dargestellten Strukturen entwickelt. Ein Nutzer (*user*) kann innerhalb von Projekten

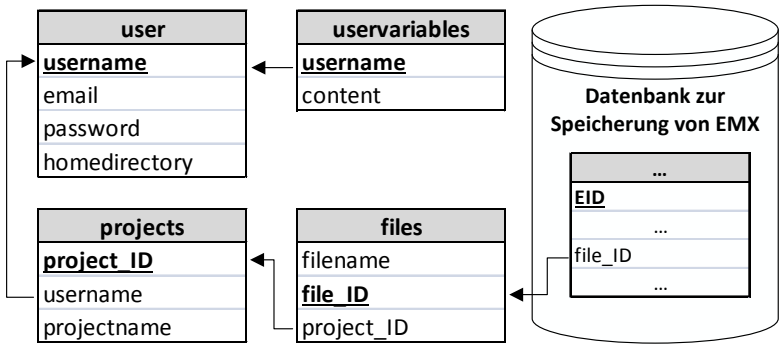


Abbildung 4.10.: Relationsschemata zur Verwaltung des konzeptuellen Modells

(*projects*) unterschiedliche Dateien (*files*) speichern. Dabei wird automatisch im erzeugten Homeverzeichnis (*homedirectory*) eine nutzerabhängige Ordnerstruktur angelegt. Gleich benannte Nutzer und Projekte eines Nutzers werden abgelehnt. Erstere sind unter anderem für die Benennung der Datenbankschemata notwendig, welche als Container der Relationsschemata der logischen Struktur dienen. Als allgemeiner Dateipfad wird folgender Aufbau realisiert:

$$\backslash\backslash homedirectory\backslash username\backslash projectname\backslash filename$$

Innerhalb von Projekten können unterschiedliche Dateien gespeichert werden, wobei auch hier eine Namensdopplung verhindert wird. Dateien können XML-Schemata (**.xsd*) oder XML-Dokumente (**.xml*) sein, die auf Dateiebene in das entsprechende Projekt des Nutzers einsortiert werden. Die Redundanzvermeidung gilt hier pro Dateiendung, sodass ein *file.xsd* ein *file.xml* nicht ausschließt.

EMX-Dateien sind nicht auf Dateiebene enthalten, sondern existieren in deren Entitäten zerlegt innerhalb des relationalen Datenbanksystems. Jede Entität be-

sitzt eine Referenz zur entsprechenden EMX-Datei (*file_ID*). Dies ist durch die *Datenbanktonne* mit abstraktem Relationsschema (Name: ...) symbolisiert. Eine Konsequenz ist, dass zum Beispiel alle Elementreferenzen unabhängig von der Zugehörigkeit zu einem speziellen EMX innerhalb einer Relation gespeichert werden.

In Abschnitt 4.1.1 wurden die Features des konzeptuellen Modells vorgestellt. Diese beinhalten nicht nur die Attribute des Schemas (<*schema*>), sondern auch zusätzliche nutzerabhängige Informationen über Defaultwerte, Verfahren zur Generierung, etc.. Diese sind für jeden Nutzer projektunabhängig im Attribut *content* der Nutzervariablen (*uservariables*) gespeichert.

Da weitergehende Informationen zusätzliche Details der nachfolgenden Kapitel benötigen, wird hier erneut zur Vermeidung von Vorgriffen auf diese verwiesen. Dies gilt insbesondere für die in Kapitel 7 thematisierte Umsetzung.

Abschließende Betrachtung

In diesem Kapitel wurde *EMX* als konzeptuelles Modell vorgestellt. Mit *EMX* wird eine Modellebene definiert, wodurch die in Kapitel 1.1 eingeführte Architektur ergänzt wird. Zum Abschluss wurden die Speicherung und Verwaltung des konzeptuellen Modells innerhalb relationaler Strukturen erläutert.

Im nächsten Kapitel wird der zweite Schwerpunkt thematisiert (d.h. *Bestimmung*[26]). Sowohl die Spezifikation und Umsetzung von Änderungsoperationen, als auch die Definition einer Updatesprache und deren Optimierung wird beschrieben. Das Logging von Nutzeraktionen und deren Auswertung wird ebenso erläutert.

[26] siehe auch: Kapitel 1.1.2 (Schwerpunkte der Arbeit)

5. Transformationssprache

Die Erfassung von Änderungen am XML-Schema bzw. am in Kapitel 4 eingeführten konzeptuellen Modell *EMX* (Entity Model for XML-Schema) ist eine Notwendigkeit zur Analyse, Bereinigung und Auswertung von Evolutionsschritten. Die domainspezifische Transformationssprache **ELaX** (Evolution Language for XML-Schema) und deren Optimierung mittels des regelbasierten Algorithmus **ROfEL** (Rule-based Optimizer for ELaX) werden nachfolgend vorgestellt.

Wesentliche Teile dieses Kapitels wurden in [NKH13c][1] und [NKH14] veröffentlicht. Auf eine Markierung von jeder wörtlichen Übernahme wird in der vorliegenden Promotionsschrift aufgrund der verminderten Lesbarkeit verzichtet.

5.1. Kriterien der Transformationssprache

Die Transformationssprache **ELaX** (Evolution Language for XML-Schema) ist aus der Notwendigkeit entstanden, Anpassungen an XML-Schema vornehmen und formal darstellen zu können. Dabei sollten Änderungen sowohl auf einfache, als auch leicht verständliche und eindeutige Art und Weise beschrieben werden können. Eine Transformationssprache wie ELaX ist notwendig, um Änderungen auf einem XML-Schema durchführen und formal ausdrücken zu können.[2] Es wurden in [NKH13c] vier Kriterien definiert, welche maßgeblich zur Entwicklung beitrugen:

1. Beachtung des Datenmodells von XML-Schema (*Abstract Data Model* (ADM) und *Element Information Item* (EII)) und des konzeptuellen Modells (*Entity Model for XML-Schema* (EMX) aus Kapitel 4.1.1)

2. Adäquate und vollständige Realisierung der Operationen *add* (Hinzufügen), *delete* (Löschen) und *update* (Ändern), sowie deren Kombinationen

3. Definition einer deskriptiven und lesbaren Schnittstelle zur Erzeugung, Änderung und Entfernung von XML-Schema

4. Intuitive und einfache Syntax zur Formulierung der Operationsschritte

Das *erste Kriterium* beinhaltet, dass die unterschiedlichen Komponenten von XML-Schema (d.h. ADM und EII) in der Transformationssprache enthalten sein

[1] Eine ausführlichere Erläuterung der Konzepte aus [NKH13c] erfolgt in [NKH13a] und [NKH13d].
[2] siehe auch: These 4

müssen, sowie deren Entsprechungen in EMX gemäß Abbildung 4.3. Das bedeutet unter anderem, dass eine Unterscheidung von EMX-Knoten erfolgen muss. Des Weiteren sollte zwischen Deklarationen und deren Referenzen unterschieden werden, damit der *Garden-of-Eden-Modellierungsstil*[3] mit dessen Anforderungen abgebildet wird.

Das *zweite Kriterium* folgt inhaltlich den Kategorisierungen von Operationen aus Kapitel 3. Kategorisierungen wurden unter anderem in [TG04], [MCSG06], [DLP+11], [CRP09], [BBGO12], [BGOB14], [Har07], [MML07], [MN09], [Mal10] und [MNM12] vorgenommen. Im Allgemeinen werden atomare Operationen thematisiert, welche auf die drei obigen reduziert werden können. Zum Beispiel ist ein *renameElement* vergleichbar mit einem Update des Namens einer Elementdeklaration.[4] Komplexere Operationen werden als Folge bzw. Kombination atomarer Operationen angesehen, sodass hier keine explizite Differenzierung erfolgt.

Das *dritte Kriterium* ist notwendig, damit mit Hilfe der Transformationssprache ein XML-Schema allgemein angepasst werden kann. Es soll nicht nur möglich sein einzelne Komponenten zu verändern, sondern ein Schema von Grund auf aufzubauen. Dabei ist die geforderte Lesbarkeit direkt mit *Kriterium vier* assoziiert. Das heißt, dass die Semantik einer Änderung direkt aus einer Operation erschlossen werden kann. Zeitgleich sollte im Sinne der Abstraktion auf unnötige Konstrukte verzichtet werden, damit Operationsschritte intuitiv formuliert werden können.

5.2. Spezifikation und Umsetzung von Änderungen

Das konzeptuelle Modell *EMX* (Entity Model for XML-Schema) enthält die folgenden Knoten: annotations, attribute-groups, groups, simple-types (st), complex-types (ct), elements, modules und constraints. Diese Knoten werden entweder hinzugefügt (*add*), gelöscht (*delete*) oder geändert (*update*).

Dabei sind Kombinationen von diesen Operationen möglich, sodass sich die nachfolgende, an die *EBNF* (Erweiterte Backus-Naur-Form) angelehnte Definition der Transformationssprache *ELaX* (Evolution Language for XML-Schema) ergibt.[5]

$$elax ::= ((< add > | < delete > | < update >) \text{ ";"})+ \; ; \tag{E1}$$

$$\begin{aligned} add ::= \text{"}add\text{"} \; (&< addannotation > | < addattributegroup > \\ &| < addgroup > | < addst > | < addct > | < addelement > \\ &| < addmodule > | < addconstraint >) \; ; \end{aligned} \tag{E2}$$

$$\begin{aligned} delete ::= \text{"}delete\text{"} \; (&< delannotation > | < delattributegroup > \\ &| < delgroup > | < delst > | < delct > | < delelement > \\ &| < delmodule > | < delconstraint >) \; ; \end{aligned} \tag{E3}$$

[3] siehe auch: Kapitel 2.1.3 (Modellierungsstile von XML-Schema)
[4] siehe auch: Kapitel 4.2.1 (Ebenen-spezifische Operationen)
[5] Die gesamte Spezifikation von ELaX ist in [Nös15b] bzw. ebenso als Überblick in Anhang B enthalten.

$$\begin{aligned} update ::= &\text{"}update\text{"} (< updannotation > | < updattributegroup > \\ &| < updgroup > | < updst > | < updct > | < updelement > \\ &| < updmodule > | < updconstraint > | < updschema >) ; \end{aligned} \tag{E4}$$

Ein Ausdruck bzw. Statement von ELaX besteht gemäß Regel E1 aus der alternativen (|) Anwendung von *<add>* (E2), *<delete>* (E3) oder *<update>* (E4). Diese Regeln können durch Semikolon (*";"*) getrennt kombiniert werden, wobei eine Häufigkeitsangabe integriert ist. Die Häufigkeiten sind definiert durch *+* (1 bis n), *?* (0 bis 1) oder *** (0 bis n), wobei eine Nichtangabe genau einmal bedeutet. In Anführungszeichen ("") gesetzte Zeichenketten dienen als Terminalsymbole, während in <> gesetzte Zeichenketten Nichtterminalsymbole repräsentieren. Des Weiteren werden Datentypen eines XML-Schemas groß geschrieben (*QNAME*, *NCNAME*, *STRING*, *INT* und *ID*)[6]. Die *emxid* ist die EID des konzeptuellen Modells.[7]

5.2.1. Hinzufügen von Elementen

Die nachfolgenden Betrachtungen sind [NKH13a] bzw. [NKH13d] entnommen, es erfolgt keine gesonderte Markierung jeder wörtlichen Übernahme.

Ausgehend von der Regel (E2) können Elemente hinzugefügt werden (*<addelement>*). Diese können gemäß des *Garden-of-Eden-Modellierungsstils* entweder Deklarationen mit globalem Gültigkeitsbereich, oder lokale Referenzen auf diese Deklarationen sein. Des Weiteren können Wildcards definiert werden, die einen Teil der hohen Erweiterbarkeit von XML begründen. Die spezifischen Regeln für Wildcards sind in Anhang B aufgelistet. Es ergibt sich die folgende Definition:

$$\begin{aligned} addelement ::= &< addelementdef > | < addelementref > | \\ &< addelementwildard > ; \end{aligned} \tag{E5}$$

$$\begin{aligned} addelementdef ::= &\text{"}element\text{"} \text{"}name\text{"} \text{ NCNAME } \text{"}type\text{"} \ < eid > \\ &((\text{"}default\text{"}|\text{"}fixed\text{"}) \text{ STRING })? \\ &(\text{"}final\text{"} (\text{"}\#all\text{"}|\text{"}restriction\text{"}|\text{"}extension\text{"}))? \\ &(\text{"}nillable\text{"} (\text{"}true\text{"}|\text{"}false\text{"}))? (\text{"}id\text{"} \text{ ID })? ; \end{aligned} \tag{E6}$$

$$\begin{aligned} addelementref ::= &\text{"}elementref\text{"} \ < eid > \ (\text{"}minoccurs\text{"} \text{ INT })? \\ &(\text{"}maxoccurs\text{"} \text{ STRING })? (\text{"}id\text{"} \text{ ID })? \\ &< position > \ (\text{"}xPos\text{"} \text{ INT } \text{"}yPos\text{"} \text{ INT })? ; \end{aligned} \tag{E7}$$

Für die Lokalisierung und Identifizierung von Elementen, sowie Knoten allgemein, werden weitere Regeln benötigt. Dies ist einerseits die Positionsbestimmung in Inhaltsmodellen unter Beachtung des Knotenumfelds (E9), andererseits die Identifizierung über die absolute Adressierung mittels einer Teilmenge von *XPath*[8]

[6] siehe auch: Kapitel 2.1.1 (Einfache Datentypen)
[7] siehe auch: Kapitel 4.1.1 (Identifikation von Entitäten)
[8] siehe auch: Kapitel 2.2 (XPath)

(E11). Des Weiteren ist eine Identifizierung von Komponenten mittels der Angabe des qualifizierten Namens (E8) möglich. Die eineindeutige EID des konzeptuellen Modells dient jeweils als äquivalente Abkürzung (*emxid* in (E8) und (E10)).

$$eid ::= \text{QNAME} \mid \text{emxid} ; \tag{E8}$$

$$position ::= (\text{"}after\text{"}|\text{"}before\text{"}| (\text{"}as\text{"}(\text{"}first\text{"}|\text{"}last\text{"}) \text{ "}into\text{"}) |\text{"}in\text{"}) < locator > ; \tag{E9}$$

$$locator ::= < xpathexpr > \mid \text{emxid} ; \tag{E10}$$

$$xpathexpr ::= (\text{"/"} (\text{"."} \mid (\text{"}node()\text{"} \mid (\text{"}node()[@name = \text{'"} \text{ NCNAME } \text{"']"})) (\text{"[" INT "]"})?))+ ; \tag{E11}$$

Eine Elementreferenz-Regel beginnt laut (E7) mit *"elementref"*, gefolgt vom Namen der referenzierten Elementdeklaration (*QNAME* oder *emxid* laut (E8)) und weiteren, optionalen Angaben über die Häufigkeit des Auftretens (*"minoccurs"* und *"maxoccurs"*) bzw. die Zuweisung einer XML-Schema-ID (*"id"*).

Die optionale Angabe zum Erscheinungsbild (*"xPos"* und *"yPos"*) der Referenz wird im Kapitel 7 näher thematisiert. Diese dient der grafischen Spezifikation von Referenzreihenfolgen innerhalb eines Inhaltsmodells.[9]

Die Position einer Elementreferenz kann wie in Regel (E9) dargestellt, vor (*"before"*), nach (*"after"*), als erstes (*"as first into"*), als letztes (*"as last into"*) oder in (*"in"*) ein Inhaltsmodell unter Beachtung der Nachbarknoten erfolgen. Die Identifizierung von Knoten findet unter Verwendung der eindeutigen Identifikatoren des konzeptuellen Modells statt (*emxid*), alternativ kann ein absoluter Pfad unter Verwendung einer Teilmenge von XPath (*<xpathexpr>*) angegeben werden (E11).

Von XPath werden die Navigationsschritte Kind (*child::node()* bzw. *"/"*) und eigener Knoten (*self::node()* bzw. *"."*), sowie die allgemeine Navigation ohne Prädikat (*"node()"*), mit Angabe eines spezifizierten Namens innerhalb eines Prädikats (*"node()[...]"*) bzw. die genaue Angabe einer Position (*"[" INT "]"*) unterstützt. Die Angabe einer Position muss unter Kenntnis des XML-Schemas immer dann erfolgen, wenn ein XPath-Ausdruck statt eines Knotens eine Menge von Knoten liefert und somit nicht eindeutig ist. Die gewählte Teilmenge von XPath ist ausreichend, um auf einfache Art und Weise Knoten im XML-Schema zu identifizieren bzw. zu lokalisieren. Dies ist durch die Verwendung des *Garden-of-Eden-Stils* möglich, da alle Deklarationen und Definitionen global gültig sind.

Elementdeklarationen (E6) benötigen im Gegensatz zu Elementreferenzen keine spezifische Lokalisierung, da diese generell unter dem *Element Information Item* des Schemas (*<schema>*) deklariert werden. Die Reihenfolge von diesen hat darüber hinaus keinen Einfluss auf ein XML-Schema. Deklarationen besitzen zwingend einen Namen (*"name"*) und ein Typen (*"type"*). Optional können Defaultwerte (*"default"* oder *"fixed"*), Abgeschlossenheitsangaben (*"final"*), die Nullwertfähigkeit (*"nillable"*) und eine XML-Schema-ID (*"id"*) spezifiziert werden.

[9] siehe auch: Kapitel 6.4.4 (Elementreferenzen) Abbildung 6.24

5.2.2. Löschen von Elementen

Elementdeklarationen und Referenzen können durch die Regel (E5) hinzugefügt werden. Der nächste zu realisierende Schritt ist das Entfernen dieser Knoten (E12). Der entscheidende Unterschied im Vergleich ist, dass nur Informationen zur Identifikation benötigt werden. Dies sind auf der einen Seite der qualifizierte Namen bzw. eine entsprechende EID (E13), aber auch im Fall von Elementreferenzen die Position innerhalb eines XML-Schemas. Die folgenden Regeln wurden definiert:

$$\begin{aligned} delelement ::= &< delelementdef > | < delelementref > \\ &| < delelementwildcard > ; \end{aligned} \tag{E12}$$

$$delelementdef ::= \text{"}element\text{"}\ \text{"}name\text{"}\ < eid > ; \tag{E13}$$

$$delelementref ::= \text{"}elementref\text{"}\ \text{"}at\text{"}\ (< locator > | < refposition >) ; \tag{E14}$$

$$\begin{aligned} refposition ::= &((\text{"}first\text{"}|\text{"}last\text{"}|\text{"}all\text{"}|\ (\text{"}position\text{"}\ \text{INT}\ )) \\ &\text{"}in\text{"} < xpathexpr >) \mid \text{emxid} ; \end{aligned} \tag{E15}$$

Die Elementreferenz-Regel (E14) beginnt mit *"elementref"*, bevor die Anweisung nach *"at"* mit einer Positionsangabe schließt. Es kann entweder die Lokalisierung mittels <*locator*> (E10) erfolgen, oder, beim mehrfachen Vorhandensein der gleichen Referenz innerhalb eines Inhaltsmodells, die Lokalisierung unter Zuhilfenahme der Regel (E15) stattfinden.

Die Lokalisierung mittels (E15) ermöglicht unter Verwendung von XPath (E11) die Adressierung der ersten (*"first"*), der letzten (*"last"*), aller (*"all"*) oder die an einer bestimmten Position befindliche Elementreferenz (*"position"*). Ist der Identifikator (*emxid*) bekannt, kann dieser alternativ und abkürzend verwendet werden.

5.2.3. Ändern von Elementen

Die Änderung von vorhandenen Knoten wird ausgehend von (E4) durch die Regel (E16) realisiert. Grundlegend können alle vorher durch (E5) hinzugefügten Elemente nachträglich geändert werden. Es werden dazu die entsprechenden qualifizierten Namen bzw. eine entsprechende EID (E8), die zu ändernden Werte der entsprechenden Knoten, sowie Angaben zur Positionierung benötigt. Die nachfolgenden Regeln wurden definiert:

$$\begin{aligned} updelement ::= &< updelementdef > | < updelementref > | \\ &< updelementwildcard > ; \end{aligned} \tag{E16}$$

$$\begin{aligned} updelementdef ::= &\text{"}element\text{"}\ \text{"}name\text{"}\ < eid >\ \text{"}change\text{"} \\ &(\text{"}name\text{"}\ \text{NCNAME}\ )?\ (\text{"}type\text{"}\ < eid >\ )? \\ &((\text{"}default\text{"}|\text{"}fixed\text{"})\ \text{STRING}\ )? \\ &(\text{"}final\text{"}\ (\text{"}\#all\text{"}|\text{"}restriction\text{"}|\text{"}extension\text{"}))? \\ &(\text{"}nillable\text{"}\ (\text{"}true\text{"}|\text{"}false\text{"}))?\ (\text{"}id\text{"}\ \text{ID}\ )? ; \end{aligned} \tag{E17}$$

$$
\begin{aligned}
updelementref ::= \; & \text{"}elementref\text{"} \; < eid > \\
& \text{"}at\text{"} \; (< locator > \; | < refposition >) \\
& \text{"}change\text{"} \; (\text{"}ref\text{"} \; < eid >)? \\
& (\text{"}minoccurs\text{"} \; \text{INT} \;)? \; (\text{"}maxoccurs\text{"} \; \text{INT} \;)? \\
& (\text{"}id\text{"} \; \text{ID} \;)? \; (\text{"}move\text{"} \; \text{"}to\text{"} < position >)? \\
& (\text{"}xPos\text{"} \; \text{INT} \; \text{"}yPos\text{"} \; \text{INT} \;)? \; ;
\end{aligned}
\tag{E18}
$$

Elementreferenzen werden mit Hilfe der Regel (E18) angepasst. Beginnend mit *"elementref"*, einem Identifikator (*<eid>*), sowie der Positionierungsinformation (nach *"at"*) kann eine Referenz geändert werden. Die Positionsbestimmung wird durch (E10) oder (E15) beschrieben.

Die entsprechende Änderungsoperation wird danach durch *"change"*, gefolgt von dem entsprechenden Bezeichner eines Wertes und dessen zu ändernden Wert ergänzt (*Attribut-Wert-Paare*), bevor die Möglichkeit zum Verschieben (*"move to"*) gegeben wird. Das Verschieben entspricht dem kompletten Entfernen und Hinzufügen einer Elementreferenz, kann aber durch die verkürzte Regel leichter vorgenommen und formal dargestellt werden. Die Angabe zum Erscheinungsbild (*"xPos"* und *"yPos"*) kann optional ebenso verändert werden.

Eine Elementdeklaration wird durch die Regel (E17) angepasst. Es erfolgt analog zur Referenz eine Identifizierung (*<eid>*), gefolgt von (*"change"*) und einer Liste von zu ändernden Attribut-Wert-Paaren. Alle durch Regel (E6) eingefügten Attribute können durch (E17) nachträglich verändert werden.

Eine Sonderrolle spielt beim Ändern wiederum das Schema (*<updschema>*). Dieses kann weder hinzugefügt noch gelöscht werden, da ein EMX das Schema selber repräsentiert. Dessen Eigenschaften bzw. *Features*[10] können allerdings angepasst werden. Eine explizite Identifizierung wie bei den EMX-Knoten erfolgt in diesem Fall nicht, sodass ein ELaX-Statement zur Anpassung des Schemas mit *"update schema change"* beginnt, gefolgt von optionalen Attribut-Wert-Paaren.

5.2.4. Anwendung der Transformationssprache

Das XML-Schema aus XML-Beispiel 5.1 soll angepasst werden. Es werden exemplarisch drei Änderungen durchgeführt, die nachfolgend unter Verwendung der oben eingeführten ELaX-Operationen schrittweise umgesetzt werden:

1. Einführung einer Elementdeklaration *e3* mit dem Typen *xs:string*.
2. Anpassung der Elementreferenz *e1* durch Änderung der Häufigkeiten.
3. Löschen der Elementreferenz *e2*.

[10] siehe auch: Kapitel 4.1.1 (Features von EMX)

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="root" type="roottype"/>
    <xs:element name="e1" type="xs:decimal"/>
    <xs:element name="e2" type="xs:string"/>
    <xs:complexType name="roottype">
        <xs:sequence minOccurs="1" maxOccurs="2">
            <xs:element ref="e1" minOccurs="1" maxOccurs="2"/>
            <xs:element ref="e2" minOccurs="0" maxOccurs="2"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

XML-Beispiel 5.1: XML-Schema zur Modifikation mittels ELaX

Schritt 1: Es soll eine Elementdeklaration eingeführt werden. Dies ist durch die Regel (E6) möglich, sodass die entsprechende ELaX-Operation wie folgt lautet:

add element name **e3** *type* **xs:string** *;* (S1)

Reihenfolge der Regeln: (E1), (E2), (E5), (E6), (E8)

Die Reihenfolge der angewendeten Regeln ist unter der jeweiligen Operation aufgelistet und dient der besseren Nachvollziehbarkeit. Die fett hervorgehobenen Bestandteile sind einerseits die eingesetzten Werte der jeweiligen Datentypen (u.a. *QNAME* oder *NCNAME*), andererseits die entsprechenden XPath-Ausdrücke zur Identifikation bzw. Lokalisierung der Knoten im XML-Schema.

Schritt 2: Es soll eine Elementreferenz verändert werden. Dies ist durch die Regel (E18) möglich, sodass die entsprechende ELaX-Operation wie folgt lautet:

update elementref **e1**
at **/node()/node()[@name='roottype']/node()**
change minoccurs **0** *maxoccurs* **42** *;* (S2)

Reihenfolge der Regeln: (E1), (E4), (E16), (E18), (E8), (E10), (E11)

Nachdem die Elementreferenz durch (E8) und (E10) identifiziert wurde, werden nach (*change*) das minimale und maximale Auftreten verändert. Die Elementreferenz ist demnach zukünftig optional und kann bis zu 42-mal wiederholt werden. Der verwendete XPath-Ausdruck kann unter Kenntnis des *Garden-of-Eden-Modellierungsstils* wie in Tabelle 5.1 dargestellt von links beginnend interpretiert werden.

Schritt 3: Es soll eine Elementreferenz gelöscht werden. Dies ist durch die Regel (E14) möglich, sodass die entsprechende ELaX-Operation wie folgt lautet:

delete elementref
at last in **/node()/node()[@name='roottype']/node()** *;* (S3)

Reihenfolge der Regeln: (E1), (E3), (E12), (E14), (E15), (E11)

XPath-Ausdruck	Erklärung
/node()	Im *Garden-of-Eden-Modellierungsstil* sind alle Deklarationen und Definitionen aufgrund des globalen Gültigkeitsbereichs direkt unter *<schema>*, sodass jeder absolute XPath-Ausdruck entsprechend beginnt.
/node()[@name='roottype']	Es soll direkt im komplexen Typen mit dem entsprechenden Namen eine Elementreferenz geändert werden. Der Name ist in diesem Fall eindeutig, somit auch der XPath-Ausdruck (keine Positionsangabe notwendig).
/node()	Die Elementreferenz wird im gegebenen Inhaltsmodell des komplexen Typs geändert (hier *sequence*), dabei existieren keine weiteren, gleich benannten Nachbarknoten (keine Positionsangabe notwendig).

Tabelle 5.1.: Lokalisierungsschritte des XPath-Ausdrucks der Operationen (S2) und (S3)

Beim Löschen einer Elementreferenz ist nur deren Identifizierung notwendig. Dies geschieht durch die Angabe, dass die letzte Referenz (*last in*) im Inhaltsmodell des komplexen Typs *roottype* entfernt werden soll. Der verwendete XPath-Ausdruck entspricht dem des 2. Schritts, dessen Interpretation ist in Tabelle 5.1 dargestellt.

Bewertung bezüglich der Kriterien

Durch die Anwendung der Operationen (S1), (S2) und (S3) wird das XML-Schema des XML-Beispiels 5.1 angepasst. Das Ergebnis nach der Änderung durch die Transformationssprache wird im XML-Beispiel A.4 dargestellt.

Es ist unter Anwendung der ELaX-Operationen möglich, Anpassungen an einem vorhandenen XML-Schema vorzunehmen und formal darzustellen. Dabei wurde sowohl auf die zugrunde liegenden Datenmodelle (*Abstract Data Model*, *Element Information Item* und *EMX*) geachtet, als auch auf die adäquate Umsetzung der Operationen *add*, *delete* und *update*, sowie deren Kombinationen.

Die vorgestellte Transformationssprache ermöglicht darüber hinaus die intuitive Formulierung von Operationsschritten und bietet zeitgleich eine deskriptive und lesbare Schnittstelle. Ein ELaX-Statement startet immer mit *"add"*, *"delete"* oder *"update"*, gefolgt von einer der alternativen Komponenten (*"element"*, *"elementref"*, etc.) und einem Identifikator der aktuellen Komponente. Anschließend werden im Allgemeinen optionale Attribut-Wert-Paare mit den Informationen der Komponente ergänzt, wobei beim Ändern ein *"change"* als Trennung dient.

Die Anwendbarkeit wurde an einem einfachen Beispiel gezeigt. Umfangreichere Beispiele folgen im Zusammenhang mit der Optimierung in Abschnitt 5.4 bzw. in den nachfolgenden Kapiteln. Die in Abschnitt 5.1 formulierten Kriterien, die die Entwicklung von ELaX maßgeblich beeinflussten, sind erfüllt.

5.3. Erfassung und Auswertung von Änderungen

"Es existieren zwei Möglichkeiten Änderungen zu erfassen. Dies ist einerseits die Aufzeichnung von diesen während des Designprozesses, andererseits der Vergleich zweier Versionen eines Schemas."[11] [MMN11]

Die Erfassung von ELaX-Operationen durch ein Mapping- und Matchingverfahren wurde in [Def13] umgesetzt. Dort wurden ausgehend von zwei Versionen eines XML-Schemas die zur Anpassung notwendigen ELaX-Operationen ermittelt. Obwohl die Transformationssprache seit Erstellung von [Def13] geändert wurde, ist die zweite Möglichkeit zur Erfassung von Änderungen grundsätzlich möglich.

In der vorliegenden Arbeit wird die Aufzeichnung von Anpassungen (**Logging**) bevorzugt, da diese entscheidende Vorteile bieten. Es ist unter anderem möglich, während der Erstellung auf fehlerhafte Ausführungen zu verweisen bzw. automatisch Korrekturen vorzunehmen. Falls Operationen zu einem Informationsverlust führen (d.h. instanzreduzierend sind), kann dies ebenso sofort angezeigt und gegebenenfalls vermieden werden. Ist es andererseits notwendig auf Instanzebene durch nicht optionale Komponenten Wissen zu generieren (d.h. durch instanzerweiternde Operationen), können entsprechende Defaultwerte umgehend erfragt bzw. auch hier auf die Gefahren verwiesen werden. Des Weiteren ist es beim Vergleich zweier Versionen nicht möglich, eine Komponente generell und unmissverständlich versionsübergreifend zu identifizieren, da diese umbenannt, umsortiert oder strukturell stark verändert werden kann. Es würde somit im Allgemeinen ein Hinzufügen und Löschen, statt einer Änderung ermittelt werden. Dies kann zu einem unnötigen Informationsverlust auf Instanzebene führen. Durch die Aufzeichnung (*Logging*) von ELaX-Operationen wird eine feingranulare **Historie** aufgebaut, aus der evolutionsrelevante Informationen automatisch ermittelt werden können.[12]

5.3.1. Speicherung von Änderungen

Die Speicherung von Änderungen erfolgt analog zum konzeptuellen Modell und der Verwaltung in relationalen Strukturen. In Abbildung 5.1 wird das Relationsschema für die Erfassung der Änderungen dargestellt. Das Schema *logging* beinhaltet Attribute für die Modellzuordnung (*file_ID*), die zeitliche Analyse (*time*), die Komponentenzuordnung (*EID*) und für den Inhalt des Logeintrags (*content*). Des Weiteren erfolgt eine Differenzierung zwischen Nachrichten- (*msgType*) und Operationstypen (*opType*). Als Schlüssel ist in dem Schema die EID ungeeignet, da eine Komponente mehrfach geändert werden kann. Daher wurde der zusammengesetzte Schlüssel bestehend aus *file_ID* und *time* gewählt. Als Konsequenz kann pro Zeitpunkt[13] nur eine Operation durchgeführt werden.

[11] "There are two possible ways to recognize changes - recording of the changes as they are conducted during the design process and comparing the two versions of the schema." [MMN11]

[12] siehe auch: These 5

[13] Ein Zeitpunkt wird in Millisekunden erfasst (primitiver Datentyp: *long*), siehe auch: Kapitel 7.

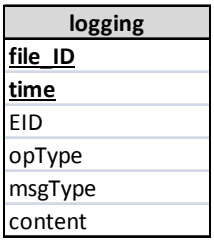


Abbildung 5.1.: Relationsschema für die Änderungen des konzeptuellen Modells

Es wird unterschieden zwischen den Operationstypen *add* (0), *delete* (1) und *update* (2). Die Zahlenwerte in den Klammern werden entsprechend gespeichert. Nachrichtentypen sind hingegen entweder "normale" ELaX-Statements (0, blau), automatisch erzeugte ELaX-Statements (1, hellblau), "normale" Aktionsmeldungen (2, schwarz) oder Fehlermeldungen (3, rot). Die farbliche Gestaltung, sowie Aktions- und Fehlermeldungen stehen im direkten Zusammenhang mit der Umsetzung in Kapitel 7 und sind an dieser Stelle nur der Vollständigkeit wegen aufgenommen. Die Unterscheidung der ELaX-Statements ist deswegen notwendig, um zwischen explizit ausgeführten und zusätzlichen, ergänzenden oder gegebenenfalls korrigierenden Operationen unterscheiden zu können.

Der Inhalt (*content*) besteht bei ELaX-Statements ($msgType \in \{0,1\}$) aus den in Abschnitt 5.2 erfassten Regeln. Somit entsteht eine gewollte Redundanz, da die *EID* und der *opType* aus dem Inhalt hergeleitet werden könnten. Eine Auswertung der Änderungen kann durch die Redundanz allerdings effizienter gestaltet werden. Statt eine komplette Zeichenkette zu laden und zu analysieren, wird mittels eines Attribut-Wert-Vergleichs die vollzogene Änderung sowie die betroffene Komponente ermittelt. Des Weiteren können alle Änderungen einer Komponente bzw. nur bestimmte Operationen auf dem EMX (z.B. nur *delete*) ohne komplexe Bedingungen ausgegeben werden. Dies wäre ohne die Redundanz nicht möglich.

Ist eine EID und deren Operationstyp nicht gegeben, wird ein Standardwert (-1) gespeichert. Dies ist zum Beispiel bei Aktions- und Fehlermeldungen notwendig.

5.3.2. Anwendung des Loggings

Das XML-Schema aus XML-Beispiel 5.1 wurde im folgenden Schema des XML-Beispiels 5.2 übernommen und um Informationen zur EID ergänzt. Es wurde zur Visualisierung der EID jeder Komponente eine XML-Schema-ID zugewiesen. Zum Beispiel besitzt die Elementdeklaration *e1* die *EID = 1*, sodass konform zum ID-Datentyp der Wert *EID1* vergeben wurde.

Das Log zur Erzeugung des XML-Schemas ist in Abbildung 5.2 dargestellt. Dieses wurde allerdings dahingehend vereinfacht, dass weder Informationen zum *msgType*, noch zur *file_ID* übernommen wurden. Da alle Operationen auf einem

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="root" type="roottype" id="EID7"/>
    <xs:element name="e1" type="xs:decimal" id="EID1"/>
    <xs:element name="e2" type="xs:string" id="EID2"/>
    <xs:complexType name="roottype" id="EID3">
        <xs:sequence minOccurs="1" maxOccurs="2" id="EID4">
            <xs:element ref="e1" minOccurs="1" maxOccurs="2" id="EID5"/>
            <xs:element ref="e2" minOccurs="0" maxOccurs="2" id="EID6"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

XML-Beispiel 5.2: XML-Schema zur Erklärung der Speicherung von ELaX-Operationen

time	EID	opType	content
1	1	0	add element name 'e1' type 'xs:string' id 'EID1' ;
2	1	2	update element name 'e1' change type 'xs:decimal' ;
3	2	0	add element name 'e2' type 'xs:string' id 'EID2' ;
4	42	0	add element name 'e3' type 'xs:decimal' id 'EID42' ;
5	3	0	add complextype name 'roottype' id 'EID3' ;
6	4	0	add group mode sequence minoccurs '1' maxoccurs '2' id 'EID4' in '3' ;
7	42	2	update element name 'e3' change type 'xs:string' ;
8	5	0	add elementref 'e1' minoccurs '1' maxoccurs '2' id 'EID5' in '4' ;
9	6	0	add elementref 'e2' minoccurs '0' maxoccurs '2' id 'EID6' in '4' ;
10	42	1	delete element name 'e3' ;
11	7	0	add element name 'root' type '3' id 'EID7' ;
12	42	1	delete element name 'e3' ;

Abbildung 5.2.: Log zur Erzeugung des XML-Schemas des XML-Beispiels 5.2

EMX durch Anwendung von ”normalen” ELaX-Statements durchgeführt werden ($msgType = 0$), sind die *file_ID* und die *msgType* jeweils identisch. Zudem sind die *EID* und *time* zur Verbesserung der Lesbarkeit aufsteigend vergeben.

Das Log aus Abbildung 5.2 enthält zusätzlich zu den Komponenten des XML-Schemas aus XML-Beispiel 5.2 weitere Einträge ($EID = 42$). Diese betreffen die nicht existente Elementdeklaration *e3*, welche im Abschnitt 5.4.2 thematisiert wird.

5.4. Optimierung der Transformationssprache

Im Kontext der vorliegenden Arbeit besteht ein Log unter anderem aus einer **Sequenz von Operationen**, die zeitlich sortiert werden kann. Dabei ist bei der Speicherung über einen längeren Zeitraum bzw. bei der durch die intensive Nutzung entstehenden Menge von Operationen die Möglichkeit gegeben, dass Logeinträge erfasst werden, die *unnötige* oder *redundante* Modifikationen enthalten. Es ist unter anderem vorstellbar, dass ein Element umbenannt, diese Änderung allerdings rückgängig gemacht wird. Des Weiteren könnte eine Elementreferenz in ein

Inhaltsmodell eingefügt werden, nur um diese anschließend wieder zu löschen. Es können diesbezüglich im Allgemeinen Operationen durch das "Herumprobieren" geloggt werden, die später zurück genommen werden.

Ein weiteres Problem sind *ungültige* Operationen. Es wäre denkbar, dass aufgrund von Übertragungsanomalien (z.B. Netzwerkprobleme) inkorrekte Logeinträge entstehen. Zum Beispiel könnte die Erzeugung unterschiedlicher Komponenten mit derselben EID gespeichert werden (d.h. zweimal die Operation *add(EID)*).

Mit dem regelbasierten Algorithmus **ROfEL** (Rule-based Optimizer for ELaX) werden unnötige, redundante und ungültige ELaX-Operationen in einem Log erkannt und beseitigt.[14] Das Ziel ist die **Minimierung der Operationsanzahl**.

Die folgenden Betrachtungen sind [NKH14] entnommen, es erfolgt analog zu den vorherigen Abschnitten keine gesonderte Markierung jeder wörtlichen Übernahme.

5.4.1. Regelbasierter Optimierer

ROfEL besteht aus unterschiedlichen Ersetzungsregeln, die in Abstimmung mit ELaX entweder *add*, *del* (delete) oder *upd* (update) behandeln. Sollte eine explizite Unterscheidung nicht notwendig sein, wird die allgemeine Operation *op(EID)* oder die Variable (_) verwendet. *Empty* bezeichnet eine nicht existente Operation.

Die Operationen wurden nach deren Zweck zur Behandlung von redundanten (R), unnötigen (U - unnecessary) oder ungültigen (I - invalid) Operationen klassifiziert. ROfEL terminiert (S - stop), wenn keine weitere Operation auf einer Komponente mit gleicher EID gegeben ist. Die folgenden Regeln wurden definiert:

$$\text{S: } empty \rightarrow \textbf{\textit{op(EID)}} \Rightarrow op(EID) \tag{R1}$$

// ↓ jüngste Operation: delete (**del**) ↓

$$\text{R: } del(EID) \rightarrow \textbf{\textit{del(EID)}} \Rightarrow del(EID) \tag{R2}$$

$$\text{U: } add(EID, content) \rightarrow \textbf{\textit{del(EID)}} \Rightarrow empty \tag{R3}$$

$$\text{U: } upd(EID, content) \rightarrow \textbf{\textit{del(EID)}} \Rightarrow del(EID) \tag{R4}$$

mit *time(del(EID))* := **TIME***(del(EID), upd(EID, content))*

// ↓ jüngste Operation: **add** ↓

$$\text{U: } op(EID) \rightarrow del(EID) \rightarrow \textbf{\textit{add(EID, content)}} \Rightarrow op(EID) \rightarrow \textbf{\textit{add(EID, content)}} \tag{R5}$$

$$\text{I: } add(EID, \_) \rightarrow \textbf{\textit{add(EID, content)}} \Rightarrow add(EID, content) \tag{R6}$$

$$\text{I: } upd(EID, \_) \rightarrow \textbf{\textit{add(EID, content)}} \Rightarrow upd(EID, content) \tag{R7}$$

[14] siehe auch: These 6

// ↓ jüngste Operation: update (**upd**) ↓

$$\text{I: } op(EID) \rightarrow del(EID) \rightarrow \textbf{\textit{upd(EID, content)}} \Rightarrow op(EID) \rightarrow \textbf{\textit{upd(EID, content)}} \quad \text{(R8)}$$

$$\text{U: } add(EID, content) \rightarrow \textbf{\textit{upd(EID, content)}} \Rightarrow add(EID, content) \quad \text{(R9)}$$

$$\text{U: } add(EID, content) \rightarrow \textbf{\textit{upd(EID, content')}} \Rightarrow add(EID, \textbf{MERGE}(content', content)) \quad \text{(R10)}$$

$$\text{R: } upd(EID, content) \rightarrow \textbf{\textit{upd(EID, content)}} \Rightarrow upd(EID, content) \quad \text{(R11)}$$

$$\text{U: } upd(EID, content) \rightarrow \textbf{\textit{upd(EID, content')}} \Rightarrow upd(EID, \textbf{MERGE}(content', content)) \quad \text{(R12)}$$

Die Regeln werden sequentiell von links nach rechts analysiert ($\rightarrow$), wobei die linke Operation zeitlich vor der rechten kommt (d.h. *time(links) < time(rechts)*).[15] Um zu garantieren, dass die Operationen auf der gleichen Komponente angewendet werden, muss die EID jeweils übereinstimmen.

Wenn zwei Operationen im Log existieren und eine Regel ist auf diese anwendbar, dann ist das Ergebnis der Ersetzung auf der rechten Seite von $\Rightarrow$ gegeben. Die Zeit des Ergebnisses ist die der linken Operation, außer weitergehende Untersuchungen sind explizit notwendig bzw. die Zeit ist nicht bekannt (z.B. bei *empty*).

Funktion TIME() des Optimierers

Regel R4 ist ein Beispiel für weitergehende Untersuchungen. In dieser wird die Situation behandelt, dass eine Komponente gelöscht wird (*del()*), vorher allerdings geändert wurde (*upd()*). Das Ergebnis der Ersetzungsregel ist folgerichtig, dass das Update keine Relevanz hat (U). Dies ist damit zu begründen, dass die Komponente am Ende nicht im Modell enthalten ist. Da gemäß des Loggings jede Operation einen Zeitwert besitzt (*time*), muss das Ergebnis entweder die Zeit des Vorgängers (links) oder des Nachfolgers (rechts) erhalten. Dies wird mit der **TIME()** Funktion entschieden, welche als Pseudocode in Abbildung 5.3 dargestellt ist.

Die Funktion hat zwei Übergabeparameter, die zu untersuchenden Operationen (*op* und *op'*). Zusätzliche Informationen und Nebenbedingungen sind in Kommentaren gegeben (*//*). Zum Beispiel bezeichnet die Variable t den Zeitwert von *op* (*time(op) = t*). Als Nebenbedingung muss unter anderem gelten, dass die EID beider Operationen gleich sein muss (*op.EID == op'.EID*). Die Einrückung symbolisiert die Schachtelung und somit logische Zusammengehörigkeit von Codefragmenten.

In Abhängigkeit einer weiteren Operation *opx* (*op.EID != opx.EID*) wird im Funktionskörper entschieden, ob als Ergebnis entweder die Zeit von *op* oder *op'*

[15] Die linke Operation ist demnach die ältere, da die Zeit kleiner ist, die rechte ist somit die jüngere.

```
TIME(op, op'):
// time(op) = t; time(op') = t'; time(opx) = tx;
// op.EID == op'.EID; op.EID != opx.EID; t > t';
   begin
     if ((t > tx > t') AND (op.EID in opx.content))
       then return t;
     return t';
   end.
```

Abbildung 5.3.: Funktion TIME() des Optimierers

zurückgegeben wird. Liegt *opx* zeitlich zwischen den Eingabeoperationen und enthält im *content* eine Referenz zu deren EID, dann wird die Zeit der jüngeren Operation (*time(op)*) zurückgegeben, sonst die der älteren (*time(op')*).

Die Idee hinter der Betrachtung ist, dass wenn eine Operation (*opx*) mit der zu löschenden Komponente assoziiert ist, diese zum Zeitpunkt der Assoziation noch nicht gelöscht sein darf. Dies wäre die Situation, wenn die Funktion *TIME()* die Zeit der Vorgängeroperation als Ergebnis zurück geben würde. Ist eine solche Zwischenoperation nicht vorhanden, dann existieren selbige Assoziationen nicht.

Funktion MERGE() des Optimierers

Die Regeln R10 und R12 referenzieren die Funktion **MERGE()**, welche als Pseudocode in Abbildung 5.4 dargestellt ist. Beide Regeln ändern (*upd*) eine Kom-

```
MERGE(content, content'):
// content = (A1 = 'a1', A2 = 'a2', A3 = '', A4 = 'a4');
// content' = (A1 = 'a1', A2 = '', A3 = 'a3', A5 = 'a5');
   begin
     result := {};
     count := 1;
     while (count <= content.size())
       result.add(content.get(count));
       if (content.get(count) in content')
         then
           content'.remove(content.get(count));
       count := count + 1;
     count := 1;
     while (count <= content'.size())
       result.add(content'.get(count));
       count := count + 1;
// result = (A1 = 'a1', A2 = 'a2', A3 = '', A4 = 'a4', A5 = 'a5');
     return result;
   end.
```

Abbildung 5.4.: Funktion MERGE() des Optimierers

ponente, nachdem diese hinzugefügt (R10) bzw. vorher bereits angepasst wurde

(R12). In beiden Fällen kann der Inhalt der Operationen vermischt bzw. kombiniert werden, sodass jeweils eine unnötige Operation entfernt werden kann.

Es werden in der Funktion als Übergabeparameter die Inhalte (*content* und *content'*) der beteiligten Operationen übergeben. Diese werden gemäß der Spezifikation von ELaX als Sequenz von Attribut-Wert-Paaren angesehen (*A1 = 'a1'*, etc.). Als Ergebnis (*result*) wird eine kombinierte Menge zurückgegeben, wobei die Attribute der jüngeren Operation die der älteren überdecken. Diese werden in der ersten Schleife (*while()*) in das Ergebnis übernommen und, insofern vorhanden, aus dem Inhalt (*content'*) der Vorgängeroperation entfernt. Anschließend werden alle verbleibenden Attribute in einer zweiten Schleife aus dem *content'* in das Ergebnis ergänzt. Dieses Vorgehen ist im letzten Kommentar exemplarisch zu erkennen.

Die Idee ist, dass die zuletzt gültigen Attribute der entsprechenden Komponente im Modell enthalten sind. Wurden allerdings im Vorfeld bereits Attribute spezifiziert, dann dürfen diese nicht fehlen und müssen mit im Ergebnis enthalten sein.

Ternäre Ersetzungsregeln des Optimierers

Zusätzlich zu den binären Ersetzungsregeln, die zwei Operationen vergleichen, existieren mit R5 und R8 ternäre Regeln. Ausgehend davon, dass nach einer Löschoperation (*del*) auf der gleichen Komponente weitere Operationen durchgeführt werden, wird die Löschoperation entfernt. Das heißt, dass zum Beispiel im Gegensatz zu [CGMO11] das Löschen keine dominante Operation ist. Dies ist damit zu begründen, dass die Entfernung einer Komponente im Allgemeinen mit dem Informationsverlust einhergeht und daher eine untergeordnete Rolle spielen sollte.

Die entstehende binäre Ersetzungsregel wird danach analysiert, wobei *op()* auch *empty* sein kann. Ein weiterer Vorteil ist, dass hier durch die ternäre Struktur die übliche Zuordnung der Zeit nicht möglich ist. Im Normalfall wird die Zeit der linken Operation verwendet. In den ternären Regeln wird diese Entscheidung hingegen auf die anschließende Analyse der binären Ergebnisregel verschoben. Würde allerdings zum Beispiel die Regel R5 umgeschrieben werden in "*del(EID)* $\rightarrow$ *upd(EID, content)* $\Rightarrow$ *upd(EID, content)*", dann müsste *upd* die Zeit von *del* erhalten. Wäre in der ternären Regel allerdings *op()* entsprechend *empty*, dann wäre diese Zeitzuordnung nicht korrekt und die binäre Variante müsste mit komplexen Fallunterscheidungen ergänzt werden. Dies ist ternär nicht notwendig.

Eine weitere Feinheit ist die Klassifikation in eine unnötige (R5 - *U*) und ungültige (R8 - *I*) Operation. Der Unterschied ist, dass im ersten Fall eine EID aus dem EMX entfernt wurde und danach neu eingefügt wird. Im zweiten Fall wird allerdings eine gelöschte EID für nachfolgende Änderungsoperationen verwendet. Dies widerspricht dem konzeptuellen Modell, in welchem nicht existente EIDs nicht referenziert werden dürfen. Daher wurde Regel R8 als ungültig klassifiziert.

Hauptfunktion ROFEL() des Optimierers

Die vorgestellten Ersetzungsregeln, sowie die Funktionen TIME() und MERGE() sind Bestandteil der Hauptfunktion des regelbasierten Optimierers. Diese Funktion heißt **ROFEL()** und ist in Abbildung 5.5 dargestellt. Es werden die Regeln R2,

```
ROFEL(log):
// log = ((t1,op1), (t2,op2), ...); t1 < t2 < ...;
 begin
   for (i := log.size(); i >= 2; i := i - 1)
     for (k := i - 1; k >= 1 ; k := k - 1)
       if(!(log.get(k).EID == log.get(i).EID AND log.get(k).time != log.get(i).time))
         then continue;
// R: del(EID) -> del(EID) => del(EID) (R2)
       if (log.get(k).opType == 1 AND log.get(i).opType == 1)
         then
           log.remove(i);
           return ROFEL(log);
// U: add(EID,content) -> del(EID) => empty (R3)
       if (log.get(k).opType == 0 AND log.get(i).opType == 1)
         then
           log.remove(i);
           log.remove(k);
           return ROFEL(log);
// U: upd(EID,content) -> del(EID) => del(EID) (R4)
       if (log.get(k).opType == 2 AND log.get(i).opType == 1)
         then
           temp := TIME(log.get(i), log.get(k));
           if (temp == log.get(i).time)
             then
               log.remove(k);
               return ROFEL(log);
           log.get(k) := log.get(i);
           log.remove(i);
           return ROFEL(log);
       [..]
// U: add(EID,content) -> upd(EID,content') => add(EID, MERGE(content',content)) (R10)
       if (log.get(k).opType == 0 AND log.get(i).opType == 2)
         then
           temp := MERGE(log.get(i).content, log.get(k).content);
           log.get(k).content := temp;
           log.remove(i);
           return ROFEL(log);
   return log;
 end.
```

Abbildung 5.5.: Hauptfunktion ROFEL() des Optimierers

R3, R4 und R10 abgebildet, welche in Abschnitt 5.4.2 angewendet werden. Die übrigen Regeln wurden ausgelassen (*[..]*), können aber ebenso formuliert werden.

Die Funktion bekommt das Log (*log*) übergeben, welches zeitlich aufsteigend

sortiert ist ($t1 < t2 < ...$) und die Änderungsoperationen (*op1*, *op2*, etc.) enthält. Innerhalb von zwei verschachtelten Schleifen (*for()*) wird das Log rückwärts analysiert. Das heißt, dass eine Operation (i) fixiert und mit allen vorherigen (k) verglichen wird. Dabei muss die *EID* übereinstimmen und die Zeit (*time*) unterschiedlich sein. Die zweite Bedingung wird durch die Schlüsselwahl des Relationsschemas *logging*[16] sicher gestellt. Ist eine Regel anwendbar, das heißt die Operationstypen (*opType*) stimmen, dann werden in Abhängigkeit dieser Regel Optimierungen vollzogen. Zum Beispiel wird in der Regel R2 eine redundante Löschoperation entfernt, indem die jüngere Operation mit dem Index i gelöscht wird.

Die Übereinstimmung der Regeln mit den aktuell fixierten Operationen wird in der festen Reihenfolge der obigen Definition vollzogen. Das heißt, dass zuerst versucht wird die *delete-*, gefolgt von den *add-* und *update-Operationen* anzuwenden.

Im Anschluss an eine Regelanwendung wird die Funktion rekursiv mit dem veränderten, reduzierten Log aufgerufen (*return ROFEL(log)*). ROFEL() terminiert, wenn keine weitere Regel mehr anwendbar ist (d.h. $i = 2$ und $k = 1$).[17]

5.4.2. Anwendung des regelbasierten Optimierers

In Abschnitt 5.3.2 wurde das XML-Schema im XML-Beispiel 5.2 durch die Anwendung von ELaX-Operationen erzeugt. Dabei wurde auf die nicht existente Elementdeklaration *e3* verwiesen, welche im zugeordneten Log der Abbildung 5.2 enthalten ist. Dieses Log wird in Abbildung 5.6 übernommen, allerdings um eine *ROfEL-Spalte* ergänzt. In dieser Spalte sind die Ersetzungsregeln eingetragen,

time	ROfEL	EID	opType	content
1	R10	1	0	add element name 'e1' type 'xs:string' id 'EID1' ;
2		1	2	update element name 'e1' change type 'xs:decimal' ;
3		2	0	add element name 'e2' type 'xs:string' id 'EID2' ;
4		42	0	add element name 'e3' type 'xs:decimal' id 'EID42' ;
5	R3	3	0	add complextype name 'roottype' id 'EID3' ;
6		4	0	add group mode sequence minoccurs '1' maxoccurs '2' id 'EID4' in '3' ;
7		42	2	update element name 'e3' change type 'xs:string' ;
8	R4	5	0	add elementref 'e1' minoccurs '1' maxoccurs '2' id 'EID5' in '4' ;
9		6	0	add elementref 'e2' minoccurs '0' maxoccurs '2' id 'EID6' in '4' ;
10		42	1	delete element name 'e3' ;
11	R2	7	0	add element name 'root' type '3' id 'EID7' ;
12		42	1	delete element name 'e3' ;

Abbildung 5.6.: Log der Abbildung 5.2 ergänzt um ROfEL-Regeln

mit deren Hilfe *ROFEL()* die unterschiedlichen Logeinträge nachfolgend optimiert. Das Ergebnis der Optimierung ist das in Abbildung A.13 dargestellte Log.

Das zeitlich sortierte Log wird rückwärts analysiert, sodass die Operation mit der Zeitmarke *12* fixiert und mit Zeiteintrag *11* verglichen wird. Da keine Übereinstim-

[16] siehe auch: Kapitel 5.3.1 (Speicherung von Änderungen)

[17] In Kapitel 5.4.3 wird gezeigt, dass der regelbasierte Algorithmus ROfEL immer terminiert.

mung der EIDs vorliegt, wird die nächste Operation mit der Zeit *10* ausgewählt. Beide Operationen löschen die gleiche Komponente (*opType == 1*), die Regel *R2* kann angewendet werden. Die redundante Operation bzw. der Eintrag mit der Zeit *12* wird daher gelöscht. *ROFEL()* wird mit dem angepassten Log erneut gestartet.

Die Regel *R4* kann als nächstes angewendet werden, eine Komponente wird geändert, allerdings später gelöscht. Diese Regel ruft die *TIME()* Funktion auf, um den Zeiteintrag des Ergebnisses zu ermitteln. Dem Ergebnis (*del(EID)*) wird entweder die Zeit *10* oder *7* zugeordnet. Keine andere Operation, die zeitlich zwischen beiden Einträgen liegt (d.h. Einträge *8* und *9*), referenziert die EID *42*. Daher gibt *TIME()* den Zeitwert *7* zurück. Der Zeiteintrag *7* wird daher dahingehend geändert, dass der Inhalt (*content*) durch "*delete element name 'e3';*" ersetzt wird. Des Weiteren wird der *opType* auf *1* gesetzt, bevor der Logeintrag *10* gelöscht wird.

Nachdem *ROFEL()* erneut aufgerufen wurde, kann die Regel *R3* zwischen dem neuen Eintrag *7* (vormals *10*) und der Operation mit der Zeit *4* angewendet werden. Eine Komponente wurde eingefügt, allerdings später gelöscht. Somit sind die Modifikationen der betreffenden Komponente im Allgemeinen unnötig. Beide Einträge werden gelöscht, sodass die entsprechende Komponente (d.h. Elementdeklaration *e3*) nicht im XML-Schema des XML-Beispiels 5.2 auftaucht.

Die letzte anwendbare Regel ist *R10*, eine Elementdeklaration wird eingefügt (Zeit *1*) und sofort geändert (Zeit *2*). In diesem Fall wird die Funktion *MERGE()* aufgerufen, um die Attribut-Werte-Paare zu kombinieren. Der ELaX-Spezifikation folgend besitzt der Inhalt (*content*) der *update-Operation* das Attribut *type* mit dem Wert *xs:decimal* (d.h. alles nach *change*), während *add* die Attribute *name* mit *e1*, *type* mit *xs:string* und *id* mit *EID1* enthält. In das Ergebnis von *MERGE()* werden alle Attribute mit Wert der *update-Operation* übernommen (d.h. *type = 'xs:decimal'*) und zeitgleich in *add* gelöscht (d.h. *type = 'xs:string'*). Anschließend werden alle verbleibenden Attribut-Wert-Paare von *add* ergänzt (d.h. *name = 'e1'* und *id = 'EID1'*). Zum Abschluss wird der Zeiteintrag *1* dahingehend verändert, dass dessen Inhalt durch "*add element name 'e1' type 'xs:decimal' id 'EID1';*" ersetzt wird. Der Logeintrag mit der Zeit *2* wird vollständig gelöscht.

Durch die Anwendung des regelbasierten Algorithmus ROfEL wurde das vorgestellte Log aus Abbildung 5.2 bzw. 5.6 optimiert. Das resultierende Log aus Abbildung A.13 enthält keine unnötigen, redundanten oder ungültigen Operationen. Des Weiteren wurde dem Ziel von ROfEL entsprechend die Operationsanzahl des Logs reduziert, sodass in den folgenden Evolutionsschritten weniger Analysen notwendig sind. Die Minimierung der Anzahl von notwendigen Änderungen an einem XML-Schema ist eine Voraussetzung für die effiziente XML-Schemaevolution.[18]

[18] siehe auch: These 7

5.4.3. Korrektheit des regelbasierten Optimierers

Die Korrektheit von ROfEL kann durch das Herleiten der *Terminierung* und der *Konfluenz* gezeigt werden [SS12]. Die Begrifflichkeiten sollen nachfolgend kurz beschrieben werden, ebenso wie die Idee der Herleitung dieser Eigenschaften. Das heißt, dass in diesem Zusammenhang in der vorliegenden Arbeit kein formaler Beweis geführt, sondern nur eine hinreichende Beweisskizze gegeben wird.

Terminierung von ROfEL

Die Terminierung fordert, dass ROfEL bei jeder beliebigen Eingabe nach endlich vielen Schritten beendet wird. In diesem Kontext spielt die Monotonie eine entscheidende Rolle. Das heißt, dass nach endlich vielen Schritten ein Grenzwert erreicht wird, bei dem keine weitere Optimierung möglich ist. Damit wäre zeitgleich die Terminierung gegeben. In ROfEL wird dieser Grenzwert erreicht, wenn entweder das Log komplett leer bzw. keine weitere Regel mehr anwendbar ist.

In Abbildung 5.7 werden alle Ersetzungsregeln in einer Matrix dargestellt. Diese

Operation time(links) < time(rechts)		**linke**			
		add	**delete**	**update**	**empty**
rechte	**add**	R6	R5	R7	R1
	delete	R3	R2	R4	R1
	update	R9, R10	R8	R11, R12	R1

Abbildung 5.7.: Operationsmatrix der Ersetzungsregeln von ROfEL

veranschaulicht, dass in jeder möglichen Konstellation eine anwendbare Regel existiert. Die Voraussetzung ist die Übereinstimmung der EID, sodass beliebige Optimierungen zwischen unterschiedlichen Komponenten nicht möglich sind.

Durch die Anwendung der Regeln kommt es im Allgemeinen zur Ersetzung zweier Operationen durch jeweils eine neue (rechts von $\Rightarrow$). Als Ausnahme gilt R1, bei der ROfEL durch das Fehlen weiterer Operationen auf der gleichen Komponenten terminiert. Eine weitere Ausnahme bilden die ternären Regeln R5 und R8, bei denen eine löschende Zwischenoperation entfernt wird. Das Ergebnis dieser Regeln ist eine binäre Operation, sodass es wiederum zur obigen Ersetzung kommt.

Ausgehend von einer endlichen Anzahl von Logeinträgen, wird somit das Log soweit reduziert, bis entweder keine Regel oder die Stoppregel R1 für alle vorhandenen EIDs anwendbar ist. Als Problemfälle gelten diesbezüglich Schleifen, welche die Monotonie verhindern, oder der Extremfall, dass keine Regel anwendbar ist.

Durch das Ersetzen von Operationen können in ROfEL keine Schleifen auftreten, da durch die erfolgreiche Anwendung einer Regel Logeinträge entfernt werden. Dadurch kann es nicht zu der Situation kommen, dass eine Regel sich selber bedingt

und die Anzahl der Logeinträge trotz Regelanwendung nicht reduziert wird. Falls der Extremfall auftritt, dass keine Regel anwendbar ist, terminiert der Algorithmus sofort. Dies wäre unter anderem denkbar beim alleinigen Vorhandensein von Operationen auf Komponenten mit vollständig unterschiedlichen EIDs.

Die Monotonie ist somit gegeben, es wird ein Grenzwert erreicht. Zeitgleich ist die Terminierung bei einer endlichen Anzahl von Logeinträgen gegeben.

Konfluenz von ROfEL

Die Konfluenz besagt, dass bei einer beliebigen Ausführung der Regeln immer dasselbe Ergebnis geliefert werden soll. Bei terminierenden Regelsystemen wie ROfEL "reicht die lokale Konfluenz für die Konfluenz aus" [SS12]. Problematisch wären demnach besonders Logeinträge, welche die Anwendung unterschiedlicher Regeln ermöglichen. Es würden somit gemäß [SS12] *Kritische Paare* existieren, die allerdings kein Problem darstellen, wenn sie als harmlos eingestuft werden können.

Diese Paare können in ROfEL aufgrund der Reihenfolge der Regeln nicht auftreten. Ein zeitlich sortiertes Log wird rückwärts analysiert. Dabei werden zuerst die Regeln der Löschoperationen, und anschließend die Einfüge- und Änderungsoperationen angewandt. Das heißt, dass die Ersetzungsregeln, insofern die EIDs übereinstimmen, in einer festen Reihenfolge ausgeführt werden (d.h. R2, R3, etc.). Es erfolgt somit eine Gewichtung bzw. Bevorzugung von Regeln, sodass es immer genau ein definiertes Nachfolgelog gibt. Die Konsequenz daraus ist allerdings auch, dass eine beliebige Ausführung der Regeln in ROfEL nicht möglich ist.

Nach der Anwendung einer Regel wird ROfEL mit dem weiterhin sortierten, angepassten Log neu gestartet. Insofern eine weitere Regel anwendbar ist und ROfEL somit nicht terminiert, wird erneut genau ein definiertes Nachfolgelog erzeugt.

Es entsteht daher eine eindeutig bestimmbare Sequenz von Ersetzungsregeln, die letztendlich bei der Terminierung von ROfEL dasselbe Ergebnis bzw. optimierte Log liefern. Die Konfluenz ist somit ebenso gegeben. Der regelbasierte Algorithmus ROfEL ist korrekt, da er ein konfluentes, terminierendes Regelsystem ist.[19]

Abschließende Betrachtung

In diesem Kapitel wurde *ELaX* als Transformationssprache vorgestellt. Mit *ELaX* werden Änderungen an dem konzeptuellen Modell und/oder XML-Schema beschrieben, wodurch eine anschließende Auswertung der Nutzeraktionen möglich ist. Zum Abschluss wurde mit *ROfEL* ein regelbasierter Optimierer eingeführt, mit welchem die Anzahl von geloggten ELaX-Operationen verringert werden kann.

Im nächsten Kapitel wird der letzte Schwerpunkt thematisiert (d.h. *Adaptionen*[20]). Die automatisierte Erzeugung von Transformationsschritten zur Wahrung und/oder Wiederherstellung der Gültigkeit einer Datenbasis wird erläutert.

[19] siehe auch: These 8

[20] siehe auch: Kapitel 1.1.2 (Schwerpunkte der Arbeit)

6. Adaption der Instanzen

Die Transformationssprache *ELaX* (Evolution Language for XML-Schema) wird zur Erfassung von Änderungen an einem XML-Schema bzw. dem entsprechenden konzeptuellen Modell *EMX* (Entity Model for XML-Schema) genutzt. Dabei werden verschiedene atomare *add-*, *delete-* und *update-Operationen* angewendet.

In **Abschnitt 6.1** wird eine Klassifikation von diesen durchgeführt. Diese Charakterisierung dient als Grundlage für die in **6.2** thematisierten Auswirkungen von ELaX-Operationen auf die Instanzebene. Des Weiteren werden in **Abschnitt 6.3** die Vorgehensweise zur Ermittlung der von den Operationen betroffenen Komponenten vorgestellt, sowie in **6.4** Verfahren präsentiert, mit denen eventuell fehlende Informationen generiert werden. Abschließend wird in **Abschnitt 6.5** erläutert, inwiefern das *DOM (Document Object Model)* [HHW+04] eines XML-Dokuments unter Verwendung der vorher ermittelten Informationen adaptiert werden muss, um die Gültigkeit von dieser Instanz gegebenenfalls wieder herzustellen.

6.1. Klassifikation der Operationen

Die **Klassifikation** von ELaX-Operationen erfolgt durch die Charakterisierung von *add*, *delete* und *update* gemäß deren *Kapazität* und *Informationsgehalt*.[1]

Es wird mit einem kurzen Exkurs in die Definitionen von Informationskapazität und Informationsgehalt im Kontext von Datenbankschemata begonnen. Die Informationskapazität eines Datenbankschemas ist gemäß [Hul86] die Menge aller möglichen Instanzen dieses Datenbankschemas. In [SSH13] wurde diese Kapazität eines Wertebereichs W mit $\mu(W)$ oder die Kapazität eines Datenbankschemas S mit $DAT(S)$ dargestellt.[2] Die in W oder S aktuell in der Datenbank gespeicherte Instanz wurde mit $\sigma(W)$ oder im Falle eines Datenbankschemas mit $d(S)$ bezeichnet. Die aktuell gespeicherte Instanz muss aus der Menge der möglichen Instanzen stammen.

Die Kapazität des Schemas kann zum Beispiel durch eine Schema-Evolution verringert werden, wobei die aktuelle Instanz trotzdem im neuen Schema dargestellt werden kann. In diesem Fall wäre die Schematransformation kapazitätsvermindernd, bewahrt aber den Informationsgehalt der aktuellen Datenbankinstanz. Ist

[1] In den XML-Beispielen A.5, A.6, A.7 und A.8 werden ergänzend zur folgenden Beschreibung Operationen bzgl. der Kapazität und in A.9, A.10, A.11 und A.12 bzgl. des Informationsgehalts präsentiert.

[2] Dies geschieht in [SSH13] hauptsächlich in den Kapiteln 3 und 4, sowie im Abschnitt 5.5.1.

nach der kapazitätsvermindernden Evolutionsoperation die aktuell in der Datenbank vorhandene Instanz aber nicht mehr darstellbar, so ist die Schematransformation nicht nur kapazitätsvermindernd, sondern bewahrt auch nicht den Informationsgehalt der Datenbank. Die Informationskapazität ist somit eine Eigenschaft des Schemas, der Informationsgehalt eine Eigenschaft der aktuellen Instanz.

Nach diesem kurzen Exkurs in die Definitionen von Informationskapazität und Informationsgehalt werden diese Begriffe nun im Kontext eines XML-Schemas angewendet. Die **Kapazität** eines XML-Schemas beschreibt in der vorliegenden Arbeit den Umfang bzw. die Möglichkeiten, Informationen darzustellen. Die Kapazität ist somit eine Kenngröße der *Schema-* und ebenso *Modellebene*.[3]

Um Informationen darstellen zu können, werden unter anderem Deklarationen und Definitionen benötigt, welche im *Garden-of-Eden-Modellierungsstil*[4] global spezifiziert werden. Wird zum Beispiel eine neue, einfache Definition hinzugefügt, dann kann diese anschließend von Deklarationen mittels *type-Attribut* referenziert werden. Die Kapazität des XML-Schemas ist daher erweitert worden, es wurde somit eine *kapazitätserweiternde* Operation angewendet.

Wird allerdings zum Beispiel eine einfache Typdefinition gelöscht, dann kann diese nicht mehr referenziert bzw. verwendet werden. Als Konsequenz können weniger Informationen dargestellt werden, da die soeben gelöschte Komponente fehlt. Das Löschen ist in diesem Fall eine *kapazitätsreduzierende* Operation.

Die *kapazitätserhaltenden* Operationen ermöglichen im Gegensatz zu den erweiternden und reduzierenden keine Erhöhung oder Verringerung der Möglichkeit, Informationen zu modellieren. Im Allgemeinen sind dies Operationen auf Komponenten des XML-Schemas, welche entweder nicht referenzierbar oder benannt sind. Das Hinzufügen, Löschen und Ändern von Annotationen ist kapazitätserhaltend.

Das Ändern einer Komponente ist im Allgemeinen *kapazitätsverändernd*. Diese vierte Möglichkeit wird zur Charakterisierung immer dann verwendet, wenn eine Operation eine Kombination aus kapazitätserhaltend, -reduzierend und/oder -erweiternd ist. Zum Beispiel ist das Ändern einer einfachen Typdefinition reduzierend, wenn ein *final = '#all'* Attribut-Wert-Paar eingefügt wird. Dadurch kann dieser einfache Typ unter anderem nicht mehr als Basistyp eines Restriktionstyps[5] verwendet werden, die Kapazität des XML-Schemas wird verringert. Das Löschen desselben *final-Attributs* ist hingegen kapazitätserweiternd, da der betroffene Typ anschließend als Basistyp im Schema zur Verfügung steht. Wird nur der Name (*name*) verändert, dann ist dies kapazitätserhaltend. Das Ändern einer einfachen Typdefinition ist somit folgerichtig kapazitätsverändernd.

Der **Informationsgehalt** beschreibt in der vorliegenden Arbeit die Auswirkung einer Operation auf die gespeicherten Informationen auf *Instanzebene*. Das heißt, dass Daten entweder durch eine Operation verloren (*instanzreduzierend*), er-

[3] siehe auch: Kapitel 4.2 (Drei-Ebenen-Architektur)
[4] siehe auch: Kapitel 2.1.3 (Modellierungsstile von XML-Schema)
[5] siehe auch: Kapitel 2.1.1 (Einfache Datentypen)

halten (*instanzerhaltend*), vermehrt (*instanzerweiternd*) oder allgemein verändert (*instanzverändernd*) werden. Operationen sind im Allgemeinen instanzreduzierend, wenn zwingende Komponenten gelöscht werden (z.B. $minOccurs > 0$).

Werden allerdings Komponenten entfernt, die von vornherein auf Instanzebene verboten waren (z.B. $maxOccurs = 0$), sind dies instanzerhaltende Operationen. Das Gleiche gilt für das Einfügen optionaler Komponenten (z.B. $minOccurs = 0$), da diese in einem gültigen Dokument nicht zusätzlich ergänzt werden müssen.

Das Hinzufügen von zwingenden Komponenten ist im Allgemeinen instanzerweiternd. Das Löschen von optionalen Elementen ist hingegen instanzverändernd. Dies ist dadurch zu begründen, dass Daten entweder gelöscht werden müssen, oder aufgrund der Optionalität auf der Instanzebene nicht enthalten sind.

6.1.1. Kapazität und Informationsgehalt von ELaX

Ein Überblick aller ELaX-Operationen mit deren *Kapazität* und *Informationsgehalt* ist in Abbildung 6.1 aufgelistet. Dies ist ein Ausschnitt der Abbildung A.14.

add	Kap	Inf	delete	Kap	Inf	update	Kap	Inf
addannotation	=	=	delannotation	=	=	updannotation	=	=
addattributegroupdef	>	=	delattributegroupdef	<	< =	updattributegroupdef	=	=
addattribute	>	=	delattribute	<	< =	updattribute	< = >	< = >
addattributeref	=	= >	delattributeref	=	< =	updattributeref	=	< = >
addattributegroupref	=	= >	delattributegroupref	=	< =	updattributegroupref	=	< = >
addattributewildcard	=	=	delattributewildcard	=	< =	updattributewildcard	=	< =
addgroup	=	=	delgroup	=	< =	updgroup	=	< = >
addct	>	=	delct	<	< =	updct	< = >	< = >
addst	>	=	delst	<	< =	updst	< = >	< = >
addelementdef	>	=	delelementdef	<	< =	updelementdef	< = >	< = >
addelementref	=	= >	delelementref	=	< =	updelementref	=	< = >
addelementwildcard	=	= >	delelementwildcard	=	< =	updelementwildcard	=	< = >
addmodule	= >	=	delmodule	< =	< =	updmodule	< = >	< = >
addconstraint	= >	< =	delconstraint	< =	=	updconstraint	< = >	< =
						updschema	< = >	< = >

Legende: **Kap** Kapazität ; **Inf** Informationsgehalt ; **<** reduzierend ; **=** erhaltend ; **>** erweiternd ; **< = >** verändernd

Abbildung 6.1.: Klassifikation von ELaX durch Kapazität und Informationsgehalt

Es sind alle Operationen der Transformationssprache mit deren Kapazität (*Kap*) und Informationsgehalt (*Inf*) enthalten. Dabei erfolgt eine Sortierung gemäß der betroffenen Komponente, sodass die Auswirkung des Hinzufügens, Löschens und Änderns in einer Zeile enthalten sind. Ein Schema kann nur geändert, nicht allerdings hinzugefügt oder gelöscht werden, da es das EMX selber repräsentiert.[6]

Die ELaX-Operationen werden kategorisiert, je nachdem ob diese reduzierende ($<$), erhöhende ($>$), erhaltende ($=$) oder verändernde Charakteristika besitzen. Eine kapazitätsverändernde Operation hat zum Beispiel in der Spalte *Kap* eine

[6] siehe auch: Kapitel 5.2.3 (Ändern von Elementen)

Kombination aus mindestens zwei Symbolen von $<$, $>$ und/oder $=$. Die Operation *updattribute* ist unter anderem sowohl kapazitäts- als auch instanzverändernd.

Hinzufügen von Komponenten

Im *Garden-of-Eden-Modellierungsstil* werden alle Definitionen und Deklarationen global spezifiziert, sodass das Hinzufügen (*addattributegroupdef*, *addattribute*, *addst*, *addct* und *addelementdef*) kapazitätserweiternd ist. Die eingefügten Komponenten können erst nach einer Referenzierung auf Instanzebene den Informationsgehalt beeinflussen. Werden Definitionen und Deklarationen neu hinzugefügt, ist dies noch nicht möglich, sodass die Operationen instanzerhaltend sind.

Dies gilt ebenso für das instanzerhaltende Einbinden von externen Komponenten unter Anwendung von *addmodule*. Module können mehrere oder keine globalen Deklarationen oder Definitionen enthalten, sodass eine Kapazitätserweiterung oder -erhaltung möglich ist. Somit ist das Hinzufügen kapazitätsverändernd.

Deklarationsreferenzen sind kapazitätserhaltend und instanzverändernd, da diese lokal in eine vorhandene Struktur eingefügt werden. Das heißt, dass eine Instanz durch das Hinzufügen einer Referenz mit zwingender Häufigkeit erweitert werden muss. Dies gilt für die Referenzen von Attributen (*addattributeref* mit *use = 'required'*), Attributgruppen mit zwingenden Attributreferenzen (*addattributegroupref*), Elementen (*addelementref* mit *minOccurs > 0*) und ebenso Elementwildcards (*addelementwildcard*). Sind dieselben Komponenten allerdings optional (*use = 'optional'* oder *minOccurs = 0*), wird der Informationsgehalt erhalten. Attributwildcards besitzen laut Standard kein *use-Attribut*, sodass ein *addattributewildcard* instanzerhaltend ist. Da die Komponenten im XML-Schema nicht referenziert werden können, erhalten die entsprechenden Operationen die Kapazität.

Constraints definieren Schlüssel-/Fremdschlüsselbeziehungen an Strukturen im XML-Schema, die bei einer gültigen Instanziierung erfüllt sein müssen. Das heißt, dass beim nachträglichen Hinzufügen (*addconstraint*) im schlechtesten Fall Daten auf Instanzebene gelöscht werden müssen, wenn keine entsprechenden Fremdschlüsselbeziehungen existieren. Sind die Beziehungen allerdings vorhanden oder die betroffene Komponente ist nicht im XML-Dokument enthalten, ist das Hinzufügen instanzerhaltend. Die Operation *addconstraint* ist insgesamt instanzverändernd. Das Hinzufügen ermöglicht die zusätzliche Referenzierung mittels *<keyref>*, falls ein *<key>* oder *<unique>* eingefügt wurde (d.h. kapazitätserweiternd), bzw. keine Referenzierung, falls eine neue *<keyref>* spezifiziert wurde (d.h. kapazitätserhaltend). Die Operation *addconstraint* ist somit kapazitätsverändernd.

Das Einfügen von Gruppen (*addgroup*) und Annotationen (*addannotation*) ist sowohl kapazitäts- als auch instanzerhaltend, da diese Komponenten weder auf Schemaebene referenziert noch auf Instanzebene instanziiert werden.

Löschen von Komponenten

Das Löschen von Komponenten ist im Allgemeinen instanzverändernd. Die Operationen *delannotation* und *delconstraint* sind als Ausnahme instanzerhaltend.

Das Entfernen von Referenzen (*delattributeref*, *delattributegroupref* und *delelementref*) und Wildcards (*delattributewildcard* und *delelementwildcard*) ist instanzverändernd, da es entweder reduzierend ist, wenn diese zwingend oder optional vorliegen, oder erhaltend ist, wenn diese verboten oder wiederum optional sind. Die Optionalität ist in beiden Varianten möglich, da im Allgemeinen erst bei der direkten Auswertung einer Instanz das Vorhandensein analysiert werden kann. Die Referenz- und Wildcardoperationen sind kapazitätserhaltend.

Das Löschen von Deklarationen (*delattribute*, *delattributegroupdef* und *delelementdef*) und Definitionen (*delst* und *delct*) ist hingegen kapazitätsreduzierend, da diese Komponenten anschließend nicht mehr zur Referenzierung zur Verfügung stehen. Durch die Entfernung von diesen Komponenten entstehen allerdings zeitgleich Inkonsistenzen im Schema, da vorhandene Referenzen nicht länger gültig sind. Im schlimmsten Fall kommt es zur *kaskadierenden Löschung*[7], das heißt, dass zum Beispiel durch das Löschen einer Deklaration ebenso alle zugehörigen, zwingenden Referenzen gelöscht werden (d.h. instanzreduzierend). Da alternativ keine Referenz gegeben sein kann, ist die Löschung einer Deklaration ebenso instanzerhaltend. Allgemein sind dies somit instanzverändernde Operationen.

Die Operation *delmodule* ist in Abhängigkeit der Anzahl und Referenzierung eingebundener, externer Entitäten kapazitätsverändernd und analog zum Löschen von im Schema spezifizierten Deklarationen und Definitionen instanzverändernd.

Das Löschen von Constraints mittels *delconstraint* ist kapazitätsverändernd, je nachdem ob ein *<key>* bzw. *<unique>* (d.h. kapazitätsreduzierend) oder *<keyref>* (d.h. kapazitätserhaltend) betroffen sind. Da Constraints nur Bedingungen an die in Instanzen realisierten Strukturen stellen, ist deren Entfernung instanzerhaltend.

Die Entfernung einer Gruppe (*delgroup*) hat keinen Einfluss auf die Kapazität, allerdings können im durch die Gruppe repräsentierten Inhaltsmodell Referenzen enthalten sein. Diese werden ebenso entfernt, eine kaskadierende Löschung ist die Konsequenz. In Abhängigkeit der Referenzen ist die Operation instanzverändernd.

Ändern von Komponenten

Die *update-Operationen* sind im Allgemeinen instanzverändernd. Um Wiederholungen zu vermeiden, sollen im Gegensatz zum Hinzufügen und Löschen nur die Abweichler der $< = >$ Klassifikation aus Abbildung 6.1 erläutert werden.

Das Ändern einer Attributgruppe (*updattributegroupdef*) ist sowohl kapazitäts- als auch instanzerhaltend. Bei Attributgruppen kann nur das Attribut *name* verändert werden, das für die Referenzierung innerhalb des Schemas verwendet wird.

[7] siehe auch: Kapitel 7.2.3 (Umsetzung des konzeptuellen Modells)

Daher ist die Attributgruppe sowohl vor als auch nach einer Änderung vorhanden (d.h. kapazitätserhaltend). In einer Instanz ist der Name nicht enthalten, sodass die Instanzebene von der Umbenennung der Komponente nicht betroffen ist.

Die Änderung einer Referenz (*updattributeref*, *updattributegroupref* und *updelementref*), einer Gruppe (*updgroup*) und einer Wildcard (*updattributewildcard* und *updelementwildcard*) sind analog zum Einfügen und Löschen kapazitätserhaltend.

Die Änderung einer Definition (*updst* und *updct*) oder Deklaration (*updattribute* und *updelementdef*), sowie einer Constraint (*updconstraint*), des Schemas (*updschema*) und der externen Entitäten (*updmodule*) sind kapazitätsverändernd. Dies ist im Allgemeinen durch die unterschiedlichen Modifikationen des *final-Attributs* bzw. bei Attributdeklarationen durch *inheritable* zu begründen. Bei Constraints sind der Wechsel zwischen <*key*>, <*unique*> und <*keyref*> ausschlaggebend, während dies bei Modulen die Änderung des Schemastandorts (*schemaLocation*) und beim Schema des Zielnamensraums (*targetNamespace*) sind.

Der Informationsgehalt kann weder bei Anpassungen von Attributwildcards (*updattributewildcard*) noch bei Constraints erweitert werden. Eine Constraint spezifiziert wie bereits beim Hinzufügen und Löschen erwähnt eine Bedingung, die nicht zur Erweiterung einer Instanz führt. Durch die Operation *updconstraint* müssen entweder Daten gelöscht werden (d.h. unerfüllte Fremdschlüsselbedingung) oder unverändert bestehen bleiben (d.h. erfüllte oder nicht vorhandene Fremdschlüsselbedingung). Die Attributwildcard besitzt kein *use-Attribut* und ist somit optional in der Instanz gegeben. Wird diese nun angepasst, dann sind die gegebenen Attribute entweder noch gültig (d.h. instanzerhaltend) oder müssen entfernt werden (d.h. instanzreduzierend). Das instanzerweiternde Hinzufügen ist nicht notwendig.

6.1.2. Herleitung der Anpassung der Instanzebene

Durch die Klassifikation der ELaX-Operationen kann die Notwendigkeit einer Instanzanpassung hergeleitet werden. Dafür wurde in Abbildung 6.2 die Klassifikation aus Abbildung 6.1 übernommen und erweitert. Es wurden die Spalten für die Instanzkosten (*Ins*) und Folgekosten (*Folge*) ergänzt. Die Instanzkosten präsentieren kein Kostenmodell, sondern geben an, ob eine Instanz angepasst (*1*) oder nicht angepasst (*0*) werden muss. Dabei ist es allerdings trotz positiver Bewertung möglich, dass die entsprechenden Komponenten durch die Ausnutzung der Optionalität nicht auf Instanzebene gegeben sind. Instanzkosten würden aber dennoch vorliegen, da eine Analyse der entsprechenden Instanzen notwendig ist.

Die Folgekosten können immer dann auftreten, wenn durch eine Operation vorerst keine Instanzkosten entstehen. Dies ist unter anderem der Fall, wenn Definitionen, Deklarationen, Gruppen oder Module entfernt werden. In diesem Fall entstehen bei einer vorliegenden Referenzierung durch andere Komponenten (z.B. Elementreferenzen) Inkonsistenzen im XML-Schema. Dadurch kann es zum kaskadierenden Löschen der referenzierenden Komponenten kommen, es entstehen

add	Kap	Inf	Ins	Folge	delete	Kap	Inf	Ins	Folge	update	Kap	Inf	Ins	Folge
addannotation	=	=	0		delannotation	=	=	0		updannotation	=	=	0	
addattributegroupdef	>	=	0		delattributegroupdef	<	< =	0	X	updattributegroupdef	=	=	0	
addattribute	>	=	0		delattribute	<	< =	0	X	updattribute	< = >	< = >	1	
addattributeref	=	= >	1		delattributeref	=	< =	1		updattributeref	=	< = >	1	
addattributegroupref	=	= >	1		delattributegroupref	=	< =	1		updattributegroupref	=	< = >	1	
addattributewildcard	=	=	0		delattributewildcard	=	< =	1		updattributewildcard	=	< =	1	
addgroup	=	=	0		delgroup	=	< =	0	X	updgroup	=	< = >	1	
addct	>	=	0		delct	<	< =	0	X	updct	< = >	< = >	1	
addst	>	=	0		delst	<	< =	0	X	updst	< = >	< = >	1	
addelementdef	>	=	0		delelementdef	<	< =	0	X	updelementdef	< = >	< = >	1	
addelementref	=	= >	1		delelementref	=	< =	1		updelementref	=	< = >	1	
addelementwildcard	=	= >	1		delelementwildcard	=	< =	1		updelementwildcard	=	< = >	1	
addmodule	= >	=	0		delmodule	< =	< =	0	X	updmodule	< = >	< = >	0	X
addconstraint	= >	< =	1		delconstraint	< =	=	0		updconstraint	< = >	< =	1	
										updschema	< = >	< = >	1	

Legende: **Kap** Kapazität ; **Inf** Informationsgehalt ; **<** reduzierend ; **=** erhaltend ; **>** erweiternd ; **< = >** verändernd ; **Ins** Instanzkosten ; **Folge** Folgekosten

Abbildung 6.2.: Klassifikation von ELaX erweitert um Instanz- und Folgekosten

Folgekosten. Dies ist durch ein *X* dargestellt. Die Modifikation eines Moduls (*updmodule*) kann dem Löschen von externen Deklarationen und Definitionen gleichgesetzt werden, sodass wie beim Löschen Folgekosten entstehen.

Analog zu den Instanzkosten gilt, dass die Folgekosten nicht zwingend auftreten müssen. Das heißt, dass wenn keine Referenzierung vorliegt, entstehen keine Folgekosten. Dies kann auf Schema- oder Modellebene geprüft werden. Liegt allerdings eine Referenzierung vor, muss wiederum die Instanzebene analysiert werden.

Zusammenhang Klassifikation und Instanzanpassung

Der entscheidende Zusammenhang besteht zwischen dem Informationsgehalt und den Instanz- und Folgekosten. Ändert sich der Informationsgehalt durch instanzreduzierende, -erweiternde oder -verändernde Operationen, dann müssen die Instanzen analysiert werden. Dadurch entstehen gegebenenfalls Folgekosten, wenn die Kapazität eines XML-Schemas durch eine kapazitätsreduzierende Operation verändert wird. Daher ist die Klassifikation der Kapazität ebenso notwendig.

Der *Garden-of-Eden-Modellierungsstil* ermöglicht, dass nicht jedes Einfügen einer Komponente zwangsläufig zur Anpassung der Instanzebene führt. Dies ist damit zu begründen, dass globale Deklarationen und Definitionen noch nicht lokal referenziert sind und somit keine Instanzkosten entstehen. Die Konzentration auf einen Modellierungsstil von XML-Schema vereinfacht die Analyse von Anpassungen auf Schemaebene und trägt somit zur effizienten XML-Schemaevolution bei.[8]

6.2. Analyse der Auswirkungen auf die Instanzen

Die ELaX-Operationen werden nach deren *Logging* und der Anwendung des regelbasierten Algorithmus *ROfEL* (Rule-based Optimizer for ELaX) weitergehend

[8] siehe auch: These 9

analysiert. Auf Grundlage der Zusammenhänge zwischen der *Klassifikation* und der Herleitung der Notwendigkeit von Instanzanpassungen werden nachfolgend unterschiedliche **Programmablaufpläne** (**PAP** - ProgrammAblaufPlan) vorgestellt. Der Programmablaufplan zur ELaX-Analyse ist in Abbildung 6.3 dargestellt. Jeder

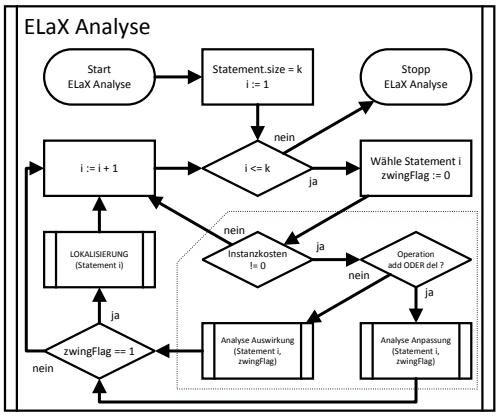


Abbildung 6.3.: PAP - ELaX Analyse

Ablaufplan besitzt einen *Start-* und *Endpunkt*, welche als *ovale Strukturen* enthalten sind. Des Weiteren existieren Operationen (*Rechteck*), Bedingungen (*Raute*) und Unterprogramme (*Rechteck mit doppelten, vertikalen Linien*). Der Ablauf des Plans wird durch die *Pfeilrichtung* bestimmt, wobei diese ausgehend von einer Bedingung *Kantenlabel* als Entscheidung enthalten können. Zusätzlich existieren Sprungmarken (*Kreis mit Label*), die in Abbildung 6.3 noch nicht enthalten sind, allerdings nachfolgend zur Erhöhung der Lesbarkeit verwendet werden.

In Abbildung 6.3 wird ausgehend von einer Menge von ELaX-Operationen analysiert (*Statement.size = k*), ob das gerade fixierte *i-te ELaX-Statement* eine Instanzanpassung hervorruft. Zur Symbolisierung, dass dieses Wissen aus der Analyse des in Abbildung A.14 dargestellten Dokumentausschnitts[9] extrahiert wird, wurde der betroffene Bereich innerhalb des Programmablaufplans *gestrichelt umrandet*. In Abhängigkeit der Operation entstehen gemäß Abschnitt 6.1.2 Instanzkosten. Ist dies nicht der Fall, dann wird entsprechend das nächste Statement (*i := i + 1*) analysiert. Sobald kein weiteres Statement ($i > k$) mehr vorliegt, endet der PAP durch das Erreichen des Endpunkts *Stopp ELaX Analyse*.

Ist eine Instanzanpassung eventuell notwendig, wird in Abhängigkeit der Operation entweder das Unterprogramm *Analyse Auswirkung* (*update*) oder *Analyse Anpassung* (*add* oder *delete*) aufgerufen. Dabei wird zusätzlich zum Statement eine Variable (*zwingFlag*) übergeben, mit deren Hilfe anschließend entschieden wird, ob das Unterprogramm *LOKALISIERUNG* aufgerufen wird oder nicht. Dieses wird in Abschnitt 6.3 thematisiert. Die Variable wird für jeden Durchlauf neu initialisiert.

[9]Das komplette Dokument mit sämtlichen Hinweisen und Kommentaren ist in [Nös15a] enthalten.

6.2.1. Hinzufügen und Löschen von Komponenten

In der Spalte *Dokumentanpassung Kriterien* der Abbildung A.14 sind die für die Analyse der Anpassungen notwendigen Bedingungen formuliert. Diese wurde in Abbildung 6.4 in einem weiteren Programmablaufplan integriert. In diesem Plan

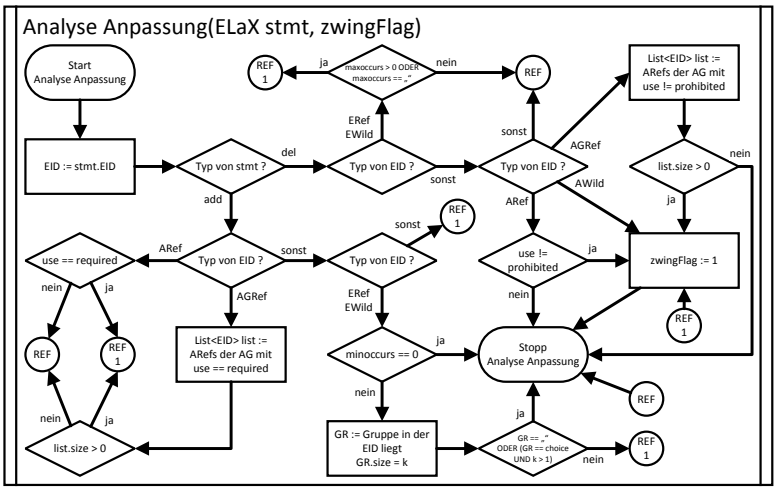


Abbildung 6.4.: PAP - Analyse Anpassung

kommen zur Erhöhung der Lesbarkeit *nummerierte Sprungmarken* zum Einsatz. Die Semantik entspricht einem *gerichteten Pfeil*, sodass der Ablaufplan an der gegebenen Referenz (*REF*) fortgesetzt wird. Des Weiteren werden an diesen PAP *Referenzparameter* übergeben, die an Variablen gebunden werden (z.B. *stmt := ELaX Statement*). Das heißt, dass nach dem Ablauf dieses Unterprogramms der *zwingFlag* von der *ELaX Analyse* der Abbildung 6.3 weiter verwendet wird.

Hinzufügen von Komponenten

Werden Attributreferenzen (*ARef*), Elementwildcards (*EWild*), Elementreferenzen (*ERef*) oder Attributgruppenreferenzen (*AGRef*) hinzugefügt, dann haben die entsprechenden ELaX-Operationen nur dann instanzerweiternde Eigenschaften, wenn diese zwingend in den Instanzen ergänzt werden müssen. In diesem Fall wird der *zwingFlag* gesetzt (*zwingFlag := 1*) und das Unterprogramm beendet.

Dies ist der Fall, wenn das *use-Attribut* einer Attributreferenz den Wert *required* besitzt bzw. eine Attributgruppenreferenz mindestens eine solche Referenz in deren Attributgruppendeklaration (*AG*) hat. Die zwingende Häufigkeit bei Elementwildcards und -referenzen wird durch das Attribut *minOccurs > 0* spezifiziert.

Ist dieses gegeben, dann sind Anpassungen in der Instanz dennoch nicht notwendig, falls einerseits keine Gruppe gegeben ist (*GR == ""*), oder andererseits das Inhaltsmodell einer solchen eine Auswahl (*choice*) darstellt. Der erste Fall be-

deutet, dass ein *visualisierter Elementknoten* im EMX eingefügt wurde, welcher eine negative *parent_EID* besitzt.[10] Das Einfügen einer solchen Elementreferenz ist generell instanzerhaltend. Ist das Inhaltsmodell eine Auswahl und die Gruppe enthält bereits mindestens eine weitere Komponente ($k > 0$), dann ist das Hinzufügen wiederum eine instanzerhaltende Operation. Wird eine Constraint eingefügt, dann wird der *zwingFlag* ebenso gesetzt (Ablauf: *add* → *sonst* → *sonst*).

Löschen von Komponenten

Werden obige Komponenten mit Ausnahme von Constraints gelöscht, dann ist nicht die zwingende Häufigkeit entscheidend, sondern ob die entsprechende Komponente verboten war. Ist dies der Fall, dann ist die Operation instanzerhaltend.

Eine Attributreferenz ist verboten, wenn das *use-Attribut* den Wert *prohibited* besitzt. Sind alle Attributreferenzen einer Attributgruppe verboten, und eine solche Attributgruppenreferenz wird gelöscht, dann ist diese Operation ebenso instanzerhaltend. Ist im Gegensatz dazu mindestens eine Attributreferenz in der Attributgruppenreferenz optional oder zwingend (d.h. *use != prohibited*), muss die Instanz analysiert werden. Dies gilt auch für Attributwildcards (*AWild*), da diese kein *use-Attribut* besitzen und daher immer optional sind.

Das Entfernen von Elementwildcards und -referenzen kann nur dann instanzreduzierend sein, wenn die maximale Häufigkeit (*maxOccurs*) größer als 0 ist. Ist das Attribut nicht gegeben (*maxOccurs == ""*), ist der Defaultwert laut Standard 1. Somit muss die Instanz ebenso analysiert werden, der *zwingFlag* wird gesetzt.

Operationen mit Folgekosten

Das Entfernen von globalen Definitionen und Deklarationen verursacht keine direkten Instanzkosten, sondern es entstehen Folgekosten.[11] Das heißt, dass zum Beispiel beim Löschen einer einfachen Typdefinition alle Deklarationen angepasst werden müssen, die die entfernte Komponente im Attribut *type* referenziert haben. Dies kann im schlimmsten Fall zum *kaskadierenden Löschen* der Deklarationen und anschließend zur Entfernung aller beteiligten Referenzen führen. Ein Beispiel ist in XML-Beispiel 6.1 gegeben, in welchem der einfache Restriktionstyp *loeschen* entfernt wird (*durchgestrichen*) und somit ebenso die Entfernung der *rot* markierten Komponenten (Deklarationen *e1* und *a1*, sowie deren Referenzen) folgen würde.

In [Kap14] wird dieses Konzept aufgegriffen und um die Möglichkeit der **Kompensation** erweitert. Das heißt, dass unter anderem versucht wird unter Ausnutzung der Typhierarchie *Kompensationstypen* im konzeptuellen Modell zu ermitteln. Ein Kompensationstyp ist im Allgemeinen ein *Obertyp*, der im Falle eines einfachen Typen den Wertebereich von diesem überdeckt. Im XML-Beispiel 6.1 ist der

[10] siehe auch: Kapitel 4.3.1 (Speicherung des konzeptuellen Modells)
[11] siehe auch: Kapitel 6.1.2 (Herleitung der Anpassung der Instanzebene)

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="root" type="roottype"/>
    <xs:simpleType name="loeschen">
        <xs:restriction base="xs:string"></xs:restriction>
    </xs:simpleType>
    <xs:element name="e1" type="loeschen"/>
    <xs:attribute name="a1" type="loeschen"/>
    <xs:complexType name="roottype">
        <xs:sequence>
            <xs:element ref="e1"/>
        </xs:sequence>
        <xs:attribute ref="a1"/>
    </xs:complexType>
</xs:schema>
```

XML-Beispiel 6.1: Beispiel des kaskadierenden Löschens

Kompensationstyp des Restriktionstyps *loeschen* der Basistyp (*base = "xs:string"*). Wird ein komplexer Typ betrachtet, dann ist ein Kompensationstyp dadurch charakterisiert, dass dieser sowohl das Inhaltsmodell als auch die optionalen und zwingenden Attribute vollständig beinhaltet.

Wird ein solcher Kompensationstyp gefunden, dann wird durch die Ausführung einer Löschung die entsprechende Deklaration nicht kaskadierend entfernt, sondern ein *updelementdef* oder *updattribute* ausgeführt und im Log ergänzt. Da das Ändern des Typs einer Deklaration in diesem Fall instanzerhaltend ist, ist eine Instanzanpassung nicht notwendig. Ist es nicht möglich einen Kompensationstyp zu ermitteln, dann wird zur Erhaltung der Konsistenz des XML-Schemas bzw. des konzeptuellen Modells das kaskadierende Löschen ausgeführt.

Durch das kaskadierende Löschen werden somit zusätzliche Löschoperationen von Deklarationen (z.B. *delelementdef*) und deren Referenzen (z.B. *delelementref*) automatisch im Log ergänzt und wiederum analysiert. Es können daher Folgekosten durch die Löschung einer globalen Definition entstehen. Das Prinzip der Ergänzung von Löschoperationen wird ebenso beim Entfernen von Deklarationen angewendet, sodass die referenzierenden Komponenten entfernt werden.

6.2.2. Ändern von Komponenten

Die Änderung von Komponenten ist im Allgemeinen immer mit einer Instanzanpassung verbunden, da die entsprechenden Operationen instanzverändernd sind. Die Abbildung 6.5 enthält ein Programmablaufplan zur Überprüfung der in der Spalte *Dokument Auswirkung* der Abbildung A.14 enthaltenen Bedingungen. Eine Beschreibung des Plans folgt nach der Vorstellung der involvierten *Extralisten*.

Ein ELaX-Statement zur Änderung einer Komponente beinhaltet nach dem

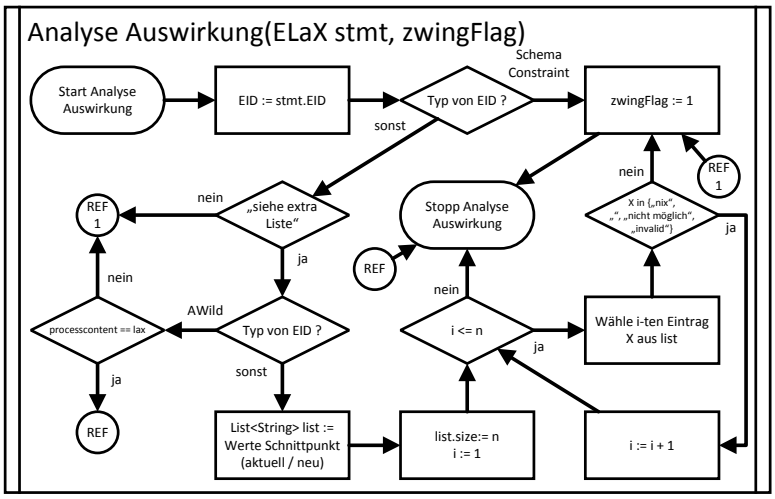


Abbildung 6.5.: PAP - Analyse Auswirkung

"change" eine Sequenz von optionalen Attribut-Wert-Paaren[12], sodass die Analyse im Vergleich zum Hinzufügen oder Löschen facettenreicher ist. Daher wurden für die Operationen *updattribute* (A.15), *updattributeref* (A.16), *updattributewildcard* (A.17), *updgroup* (A.18), *updst* (A.19), *updct* (A.20), *updelementdef* (A.21), *updelementref* und *updelementwildcard* (A.22) zusätzliche Listen eingefügt.

Diese Listen thematisieren die unterschiedlichen Attribut-Wert-Paare und deren informell beschriebenen Auswirkungen auf eine Instanz. Im Anhang sind in den Abbildungen A.15 bis A.22 diese *Extralisten* aus [Nös15a] übernommen worden.

Beispiel einer Extraliste

Eine *Extraliste* für Elementreferenzen ist in Abbildung 6.6 dargestellt. Diese ent-

	neu ->		
aktuell	**minoccurs**	**min--**	**min++**
	0	nicht möglich	add value
	1	nix	immer WP: add value?
	n	nix	immer WP: add value?

	neu ->		
aktuell	**maxoccurs**	**max--**	**max++**
	0	nicht möglich	nix
	1	immer WP: del value?	nix
	n	immer WP: del value?	nix

Abbildung 6.6.: Extraliste der ELaX-Operation updelementref aus [Nös15a]

hält zwei Tabellen für die Änderung der minimalen (*minOccurs*) und maximalen (*maxOccurs*) Häufigkeit. Horizontal (*neu*) werden die ausgeführten Operationen (*min--*, *min++*, *max--* und *max++*) aufgelistet, während vertikal (*aktuell*) die

[12] siehe auch: Kapitel 5.2.3 (Ändern von Elementen)

momentane Wertbelegung angezeigt wird (*0*, *1* und *n*). Eine Reduzierung der Häufigkeit (*min--*) von 1 verursacht zum Beispiel keine Instanzanpassung (*nix*), während eine Erhöhung (*min++*) von 1 generell eine Wertprüfung (*immer WP*) verlangt. Als Konsequenz der Wertprüfung kann optional (*?*) das Hinzufügen neuer Elemente (*add value*) notwendig werden. Dies ist allerdings von der Realisierung der betrachteten Elementreferenz innerhalb einer Instanz abhängig.

Die Änderungen der Positionierung (*xPos* und *yPos*) und der Referenz (*ref*) sind in der Abbildung 6.6 nicht enthalten, da diese laut Abbildung A.14 generell detaillierter analysiert werden müssen. Bei einer Referenz ist dies die Prüfung des Namens der umschließenden Tags, sowie deren Inhalts (*mod value / Markup ?*), während dies bei der Positionierung die Umsortierung (*reorder Markup ?*) ist.

Analyse der Auswirkungen

Der PAP aus Abbildung 6.5 beschreibt, inwieweit die zusätzlichen Listen ausgewertet werden. Ausgehend von einer Operation wird unterschieden, ob das Schema oder Constraints betroffen sind. Ist dies der Fall, wird der *zwingFlag* gesetzt (*zwingFlag := 1*) und das Unterprogramm wird beendet (*Stopp Analyse Auswirkung*).

Sind andere Komponenten betroffen, dann wird nach der Existenz von Extralisten geschaut (*siehe extra Liste*). Liegen diese nicht vor, dann wird der *zwingFlag* gesetzt. Dies gilt auch für Attributwildcards, insofern diese das Attribut *processcontent* mit der Wertbelegung *strict* oder *skip* besitzen. Sind zusätzliche Listen vorhanden, dann werden alle Änderungen der Operation in der zugehörigen Tabelle geprüft und entsprechend in einer Liste von Werten gespeichert (*Werte Schnittpunkt*). Werden bei der anschließenden Prüfung der Liste nur Änderungen gefunden, die entweder keine Instanzanpassungen (*nix*) benötigen, oder *nicht möglich* bzw. ungültig (*invalid*) sind, dann ist die Operation instanzerhaltend und der *zwingFlag* wird nicht gesetzt. Ist nur eine instanzverändernde Änderung enthalten (u.a. *immer WP*), muss die gesamte Operation nachfolgend analysiert werden.

Die Operation (S2)[13] würde zum Beispiel gemäß Abbildung 6.6 die Schnittpunktliste "(nix, nix)" erzeugen. Dies ist damit zu begründen, dass die minimale Häufigkeit von 1 auf 0 reduziert wird (*min--*) und zeitgleich die maximale Häufigkeit von 2 auf 42 erhöht wird (*max++*). Beide Operationen sind instanzerhaltend, sodass eine Instanzanpassung im ausgewählten Beispiel nicht notwendig ist.

6.3. Lokalisierung von Komponenten

Die Auswirkungen von ELaX-Operationen wurden im vorherigen Abschnitt analysiert, sodass an dieser Stelle jedes Statement potentiell Instanzanpassungen hervorrufen wird. Bevor nun allerdings die Instanzebene ausgewertet und gegebenenfalls

[13] update elementref e1 at /node()/node()[@name='roottype']/node() change minoccurs 0 maxoccurs 42

adaptiert wird, muss die entsprechende Komponente innerhalb einer Instanz lokalisiert werden. Diese Lokalisierung ist auf Modell- und Schemaebene möglich, wobei als Ergebnis für jedes ELaX-Statement eine Liste von **Lokalisierungspfaden** ermittelt wird. Ist diese Liste leer, dann ist die durch das Statement betroffene Komponente nicht in der Instanzebene gegeben. Die Konsequenz ist, dass die entsprechende Operation wiederum keine Instanzanpassung bedingt.

Das Vorgehen zur Ermittlung der Lokalisierungspfade ist im Programmablaufplan in Abbildung A.23 gegeben. Ausgehend vom ELaX-Statement werden die betroffenen EIDs der Komponente identifiziert, was in **Abschnitt 6.3.1** beschrieben wird. Anschließend wird in **Abschnitt 6.3.2** für jedes *Wurzelelement* eine Liste von **EID-Ketten** zur identifizierten Komponente erzeugt, aus welchen die *Lokalisierungspfade* in Form von absoluten XPath-Ausdrücken generiert werden.

Die *Wurzelelemente* sind gemäß des konzeptuellen Modells alle Elementreferenzen mit negativer *parent_EID*. Da allerdings im *Garden-of-Eden-Modellierungsstil* jede Elementdeklaration als Wurzel verwendet werden kann, ist es möglich, jede visualisierte Elementreferenz als Ausgangspunkt der EID-Ketten zu spezifizieren. Der PAP in Abbildung A.26 stellt die Entscheidungsfindung dar. Die beteiligte **Nutzerkonfiguration** (*Nutzer Config*) ist ein weiteres *Feature*[14] vom konzeptuellen Modell und kann entsprechend konfiguriert werden. Die Verwendung von Elementreferenzen mit negativer *parent_EID* ist die Defaultkonfiguration.

6.3.1. Identifizierung von Komponenten

Die Identifizierung der durch ein ELaX-Statement betroffenen Komponenten wird im Programmablaufplan A.24 beschrieben. Es werden durch den Plan EIDs ermittelt, welche vom Wurzelelement ausgehend das Ziel der EID-Ketten bilden. In der *Dokument-zentrierten Darstellungsweise*[15] sind das die visualisierten EMX-Knoten der Elemente und Gruppen. In Abbildung 6.7 wird das Ergebnis zusammengefasst dargestellt. Die betroffenen Komponenten (z.B. *ADef*, *ARef*, etc.) sind diejenigen,

CONNODES		ADef	ARef	AWild	AGRef	Group	ST	CT	EDef	ERef	EWild	Schema	Constraint
PAP ATT		X	X	X			X						
PAP ELEM		X	X	X			X	X	X			X	
EID						X				X			
EID der nächsten parent_EID ist	ERef	X	X	X			X	X	X			X	X
	Group	X	X	X	X		X	X	X		X	X	

Abbildung 6.7.: Übersicht der ermittelten Ziel-EIDs durch den PAP der Abbildung A.24

welche gemäß der Operationen aus Abbildung 6.2 unabhängig von *add*, *delete* und *update* Instanzkosten verursachen. Das heißt, dass unter anderem Attributgruppen nicht betrachtet werden, während Attributdeklarationen aufgrund der Instanzkosten der Operation *updattribute* thematisiert werden.

[14] siehe auch: Kapitel 4.1.1 (Features von EMX)
[15] siehe auch: Kapitel 4.1.2 (Visualisierung)

Ist eine Attributgruppenreferenz (*AGRef*) betroffen, dann wird die *parent_EID* der nächsten Gruppe als Ziel-EID gespeichert. Elementwildcards (*EWild*) speichern ebenso deren *parent_EID* (*Group*), während Constraints die nächste EID einer *ERef* ermitteln. Ist eine Elementreferenz (*ERef*) oder Gruppe (*Group*) übergeben worden, wird deren eigene *EID* gespeichert.

Die weiteren Komponenten werden gesondert analysiert. Die dafür notwendigen PAPs sind *ELEM* und *ATT*, die je nach übergebener Komponente die EIDs der nächsten Gruppe oder Elementreferenz liefern. Diese werden nachfolgend erklärt.

Komponenten mit Elementbezug

In Abbildung 6.8 wird der Programmablaufplan *ELEM* dargestellt. Bekommt die-

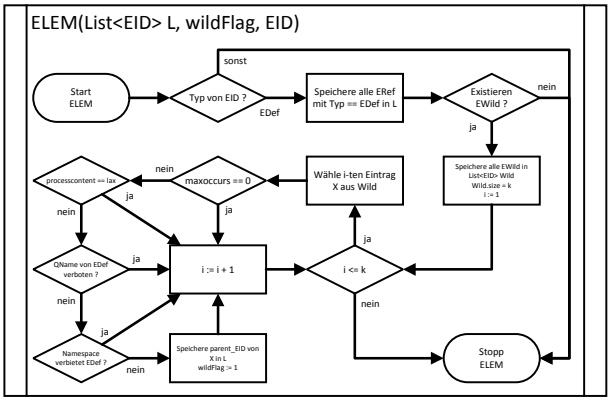


Abbildung 6.8.: PAP - ELEM aus [Nös15c]

ser eine Elementdeklaration übergeben, dann werden alle dazugehörigen Elementreferenzen ermittelt und deren EID gespeichert.

Dies kann sowohl direkt geschehen (*EDef*), als auch bei der Ermittlung der Deklarationen, die einen bestimmten einfachen (*ST*) oder komplexen Typen (*CT*) haben. Des Weiteren können komplexe Typen sowohl durch die Komponenten *ADef*, *ARef* und *AWild*, als auch durch die Möglichkeit der Defaultattributgruppe des *Schemas* betroffen sein. Diese Typen können wiederum durch Elementdeklarationen referenziert werden, sodass diese ebenso berücksichtigt werden.

In einem nächsten Schritt werden die im konzeptuellen Modell vorhandenen Elementwildcards analysiert (*Existieren EWild?*). Diese können die übergebene Elementdeklaration auf der Instanzebene integrieren. Daher werden alle Elementwildcards geprüft, ob diese möglicherweise betroffen sind. Allerdings können auch hier im Vorfeld bereits Charakteristika (d.h. Attribute) untersucht werden, die eine Instanzanpassung an der Position der Elementwildcard vermeiden.

Dazu zählen sowohl die verbotene Häufigkeit (*maxOccurs == 0*), als auch die Angabe der laxen Überprüfung (*processcontent == lax*), das Vorhandensein von

Beschränkungen der erlaubten Deklarationen (*QName*) und des Namensraums (*Namespace*). Ist kein Ausschlusskriterium gegeben, dann wird die *parent_EID* (*Group*) der Elementwildcard entsprechend zusätzlich als Ziel-EID gespeichert. Des Weiteren wird der *wildFlag* gesetzt, damit nachfolgend die Existenz bzw. Involvierung von Elementwildcards nicht erneut geprüft werden muss.

Komponenten mit Attributbezug

In Abbildung 6.9 wird der Programmablaufplan *ATT* dargestellt. Dieser ermit-

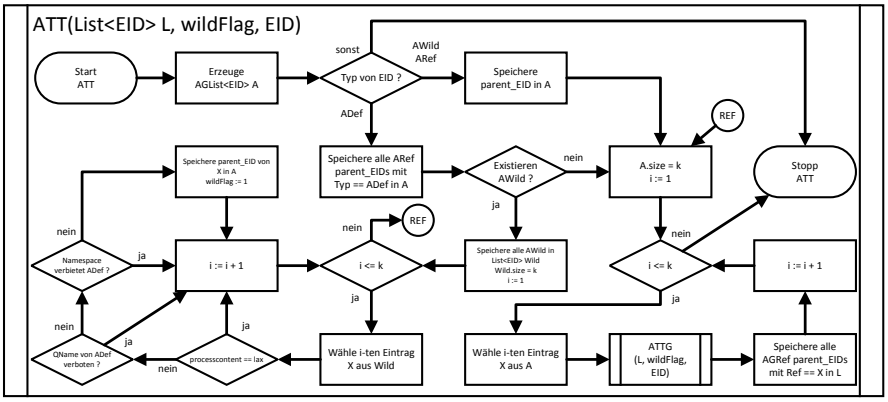


Abbildung 6.9.: PAP - ATT aus [Nös15c]

telt alle Attributgruppen, die bestimmte Attributreferenzen (*ARef*), Wildcards (*AWild*), sowie die Referenzen von Attributdeklarationen (*ADef*) beinhalten. Die *parent_EIDs* der Attributgruppenreferenzen (*AGRef*) dieser Attributgruppen werden als Ziel-EID gespeichert. Analog zu *ELEM* werden bei den Attributdeklarationen ebenso alle betroffenen Attributwildcards analysiert. Da allerdings keine Häufigkeitsangabe möglich ist, wird diese Überprüfung nicht vollzogen. Werden solche Attributwildcards gefunden, dann wird der *wildFlag* gesetzt.

Wurden alle Attributgruppen ermittelt, dann muss zusätzlich geprüft werden, ob eine dieser Gruppen als Defaultattributgruppe des Schemas spezifiziert wurde. Dies wird durch das Unterprogramm *ATTG* realisiert, welches in Abbildung A.25 dargestellt ist. Es werden bei der Übereinstimmung der aktuell analysierten Attributgruppe mit der Defaultattributgruppe alle Elementdeklarationen ermittelt, welche eine komplexen Typen mit *defaultattributesapply != false* besitzen. Existieren solche Deklarationen, dann wird der obige Programmablaufplan *ELEM* ausgeführt und die Liste der Ziel-EIDs wird gegebenenfalls entsprechend erweitert.

Beispiel Ziel-EID

Das XML-Schema aus XML-Beispiel 6.2 enthält unterschiedliche Komponenten, die nachfolgend zur Darstellung der Ziel-EID Ermittlung verwendet werden.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" id="EID1">
    <xs:element name="root" type="roottype" id="EID2"/>
    <xs:element name="e1" type="ctype" id="EID3"/>
    <xs:element name="e2" type="xs:string" id="EID4"/>
    <xs:attribute name="a1" type="xs:string" id="EID5"/>
    <xs:attribute name="a2" type="xs:string" id="EID6"/>
    <xs:attributeGroup name="ag1" id="EID7">
        <xs:attribute ref="a1" id="EID8"/>
        <xs:anyAttribute id="EID9"/>
    </xs:attributeGroup>
    <xs:complexType name="roottype" id="EID10">
        <xs:sequence minOccurs="1" maxOccurs="2" id="EID11">
            <xs:element ref="e1" minOccurs="1" maxOccurs="2" id="EID12"/>
            <xs:element ref="e2" minOccurs="0" maxOccurs="2" id="EID13"/>
        </xs:sequence>
        <xs:attributeGroup ref="ag1" id="EID14"/>
    </xs:complexType>
    <xs:element name="e3" type="xs:string" id="EID15"/>
    <xs:complexType name="ctype" id="EID16">
        <xs:sequence minOccurs="2" maxOccurs="2" id="EID17">
            <xs:element ref="e3" minOccurs="0" maxOccurs="1" id="EID18"/>
        </xs:sequence>
        <xs:attributeGroup ref="ag1" id="EID19"/>
    </xs:complexType>
</xs:schema>
```

XML-Beispiel 6.2: XML-Schema zur Darstellung der Lokalisierung

Zur Verbesserung der Lesbarkeit wurden erneut die XML-Schema-IDs zur Andeutung der EIDs verwendet. Das Schema ist als EMX in Abbildung 6.10 dargestellt. Wird nun zum Beispiel die Elementreferenz *e2* (EID *13*) verändert, indem der

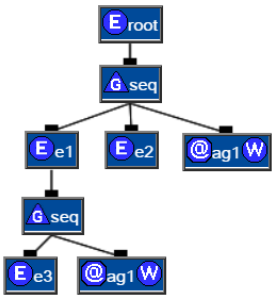


Abbildung 6.10.: EMX des XML-Schemas aus XML-Beispiel 6.2

minOccurs-Wert erhöht wird, dann ist dies eine instanzverändernde Operation.[16]

[16]update elementref e2 at 13 change minoccurs 1 ; *bzw.* update elementref 4 at 13 change minoccurs 1 ;

Die Ziel-EID wäre gemäß Abbildung 6.7 *13*, die eigene EID der Elementreferenz.

Eine weitere ELaX-Operation könnte einen *fixed-Wert* in der Attributdeklaration *a2* einführen.[17] Diese Operation hat gemäß der Extraliste der Abbildung A.15 die Eigenschaft, dass vorhandene Werte geprüft und gegebenenfalls auf Instanzebene verändert werden müssen (*immer WP: add/mod value*). Das heißt, dass die betroffenen Komponenten und somit die Ziel-EIDs ermittelt werden müssen.

Das Unterprogramm *ATT* aus Abbildung 6.9 wird aufgerufen. Da keine Attributreferenz existiert, wird nach der Existenz von Attributwildcards geschaut. Die AWild mit der EID *9* ist gegeben. Diese beinhaltet keine Einschränkungen, welche die Nutzung der Attributdeklaration *a2* auf Instanzebene verhindern. Somit wird die *parent_EID 7* (d.h. Attributgruppe *ag1*) zwischen gespeichert und der *wildFlag* gesetzt. Diese Attributgruppe ist nicht die Defaultattributgruppe des Schemas, sodass keine zusätzlichen Ziel-EIDs durch den PAP *ATTG* ergänzt werden. Im Anschluss wird nach einer Attributgruppenreferenz gesucht, die *ag1* referenziert. Dies sind die Attributgruppenreferenzen mit den EIDs *14* und *19*, welche als *parent_EID* die Gruppen mit den EIDs *11* und *17* besitzen. Die Ziel-EIDs dieses Beispiels sind somit sowohl *11* als auch *17*.

6.3.2. Konstruktion von Lokalisierungspfaden

Durch die Identifizierung der Komponenten, die durch eine ELaX-Operation betroffen sind, liegen Listen von Ziel-EIDs vor. Diese werden verwendet, um EID-Ketten zu konstruieren, welche in XPath-Ausdrücke umgeformt werden.

Konstruktion von EID-Ketten

Eine EID-Kette besteht aus einer Sequenz von EIDs, welche ausgehend von einem Wurzelelement die visualisierten EMX-Knoten zur Ziel-EID beinhalten. Es werden dabei sowohl die EIDs der Elementreferenzen, als auch der Gruppen erfasst. Der Programmablaufplan *LOK* in Abbildung 6.11 stellt dieses Vorgehen dar. Dieser Algorithmus ermittelt alle Kinderelemente der aktuell analysierten Komponente (*PosEID*), insofern die Position nicht die gesuchte *ZielEID* ist und solche Kinderelemente vorliegen. Die ermittelten Elemente werden nacheinander untersucht, ob diese Attributgruppenreferenzen (*AGRef*), Gruppen (*Group*) oder Elementreferenzen (*ERef*) sind. Für die letzten beiden Varianten wird zusätzlich geprüft, ob die Komponente erlaubt ist (*maxOccurs == 0*). Ist dies nicht der Fall, dann wird der PAP beendet (*Stopp LOK*). Dies gilt ebenso, wenn entweder das Kindelement eine *Annotation*, *Constraint* oder *Modul* ist, oder keine weiteren Elemente existieren.

LOK wird rekursiv mit der EID-Kette, dem aktuellen Kinderelement (*X*), der Ziel-EID und dem Statement erneut aufgerufen, insofern Attributgruppenreferenzen, Gruppen oder Elementreferenzen gerade ausgewählt wurden. Dabei wird die

[17] update attribute name a2 change fixed XYZ ; *bzw.* update attribute name 6 change fixed XYZ ;

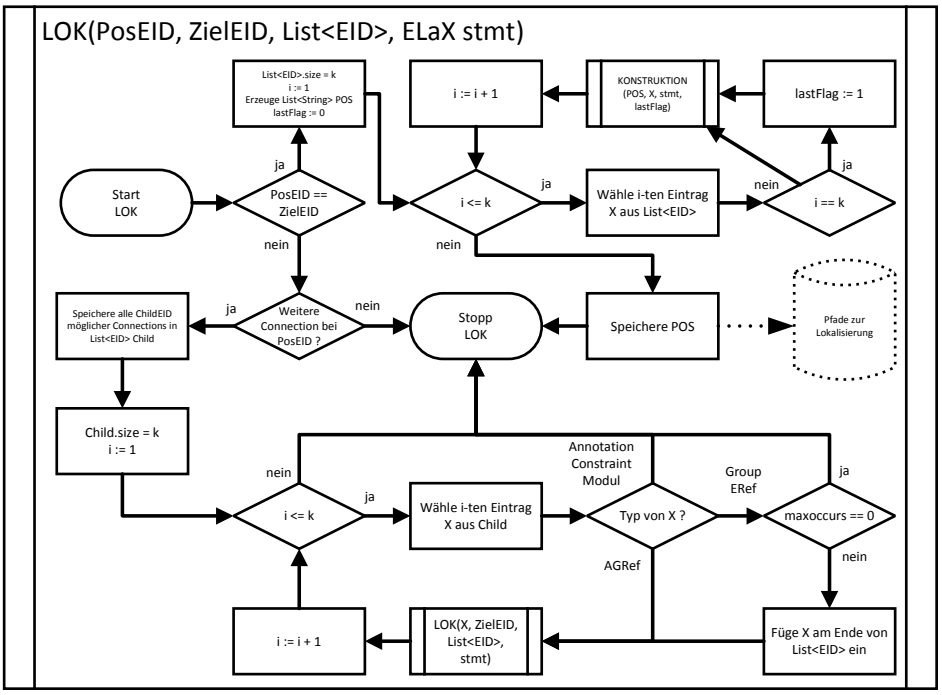


Abbildung 6.11.: PAP - LOK aus [Nös15c]

EID der letzten beiden Komponenten in der EID-Kette ergänzt (*ERef* und *Group*).

Als Resultat von *LOK* entsteht eine per Tiefensuche konstruierte EID-Kette aus Gruppen und Elementreferenzen. Entspricht die aktuelle analysierte Komponente der Ziel-EID (*PosEID == ZielEID*), dann werden die EID-Ketten sequentiell vom Anfang (Wurzel) zum Ende (Ziel-EID) zur Konstruktion von XPath-Ausdrücken verwendet. Dies wird nach dem folgenden Beispiel detaillierter erläutert.

Die Lokalisierung von Komponenten endet mit der Speicherung aller Pfade (*POS*) in einer *Statement und Pfad Tabelle*. Dies ist in dem allgemeinen Lokalisierungsablaufplan in Abbildung A.23 dargestellt. Das heißt, dass für jede instanzverändernde Operation sowohl eine Liste von Lokalisierungspfaden, als auch der *wildFlag* und das ELaX-Statement gespeichert werden. Damit endet ebenso die ELaX-Analyse des aktuellen Statements, welche in Abbildung 6.3 initiiert wurde.

Beispiel EID-Kette

Das XML-Beispiel 6.2 wurde zur Identifizierung von Komponenten eingeführt und anschließend durch zwei ELaX-Operationen adaptiert. Dabei wurde die Elementreferenz *e2* durch eine Operation *op1* verändert und die Ziel-EID *13* ermittelt. Des Weiteren wurde die Attributdeklaration *a2* durch eine Operation *op2* angepasst. Es wurden diesbezüglich die Ziel-EIDs *11* und *17* identifiziert.

Ausgehend von der Wurzel (*root*) des konzeptuellen Modells aus Abbildung 6.10, werden durch *LOK* die EID-Ketten konstruiert. Dieses EMX wurde in Abbildung 6.12 verändert dargestellt, da dies für die Nachvollziehbarkeit hilfreich ist. Die

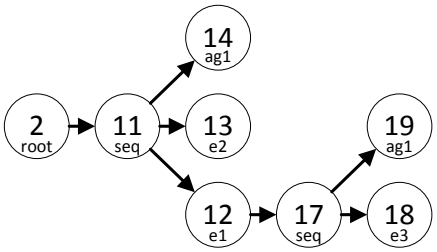


Abbildung 6.12.: Veränderte Darstellung des EMX aus Abbildung 6.10

Abbildung stellt die EIDs der visualisierten EMX-Knoten dar, ergänzt um Hinweise zum Rückschluss auf die Komponente. Das Element *2* ist demnach die Elementreferenz *root*, während *11* eine Gruppe mit dem Inhaltsmodell Sequenz ist. Den gerichteten Pfeilen folgend wird somit für die Ziel-EID *13* die EID-Kette *(2, 11, 13)* konstruiert. Für die Operation *op2* wurden zwei Ziel-EIDs identifiziert, somit werden die EID-Ketten *(2, 11)* und *(2, 11, 12, 17)* ermittelt.

Konstruktion von XPath-Ausdrücken

Die EID-Ketten werden in XPath-Ausdrücke umgewandelt. Dabei wird der PAP *KONSTRUKTION* der Abbildung 6.13 angewendet. Dieser wird von *LOK* aufgerufen, falls in diesem die aktuelle Komponente der Ziel-EID entspricht. Es erfolgt eine sequentielle Abarbeitung der EID-Ketten. Das heißt, dass die erste EID genommen und damit ein Teilpfad konstruiert wird. Dieser wird in der Liste *POS* gespeichert und im Durchlauf mit der nächsten EID ergänzt. Somit wird ein Lokalisierungspfad schrittweise aufgebaut, bis die Ziel-EID und somit das letzte Element der EID-Kette (*lastFlag*) erreicht wird.

Bevor ein Teilpfad allerdings konstruiert wird, sind unterschiedliche Vorbetrachtungen notwendig. Zum Einen werden die Häufigkeiten der aktuell untersuchten Gruppe oder Elementreferenz betrachtet. Diese werden verwendet, um auf Instanzebene die vorhandenen Elemente zu referenzieren. Es muss dabei beachtet werden, dass diese Häufigkeiten möglicherweise verändert wurden (*upd maxoccurs*). Da eine Instanz allerdings gültig bezüglich der alten Werte ist, werden diese verwendet.

Des Weiteren erfolgt eine Reduzierung der maximalen Häufigkeit durch den PAP *LIMITMAX*, welcher in Abbildung A.27 dargestellt ist. In Abhängigkeit der minimalen Häufigkeit wird der *unbounded-Wert* des *maxOccurs-Attributs* in eine algorithmisch auswertbare Form gebracht. Dies kann zum Beispiel ein sehr großer Wert sein, der allerdings immer größer oder gleich *minOccurs* sein muss. Die beteiligte

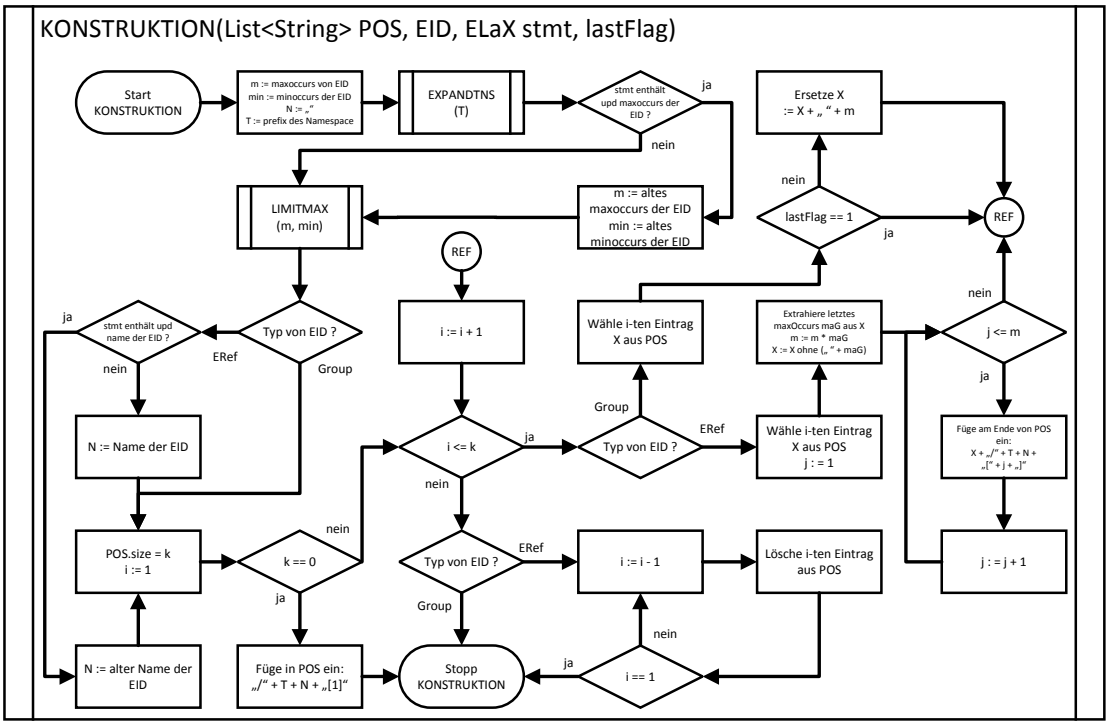


Abbildung 6.13.: PAP - KONSTRUKTION aus [Nös15c]

Nutzerkonfiguration (*Nutzer Config*) in *LIMITMAX* ist ein weiteres *Feature*[18] vom konzeptuellen Modell und kann entsprechend konfiguriert werden. Die Verwendung von *42* statt *unbounded* ist die Defaultkonfiguration.

Eine weitere Vorbetrachtung betrifft den Namen (N) der Elementreferenz. Wurde dieser verändert (*upd name*), muss ebenso zur Lokalisierung der alte Wert auf Instanzebene verwendet werden. Des Weiteren wird der *Targetnamespace* beachtet. Ist dieser gegeben, dann wird der entsprechende Präfix (T) bei der Konstruktion von Teilpfaden dem Namen voran gestellt. Der PAP *EXPANDTNS* der Abbildung A.28 ergänzt den Präfix des Targetnamespaces jeweils um einen Doppelpunkt.

In Abbildung 6.14 wird die schrittweise Konstruktion von Lokalisierungspfaden dargestellt, welche die Arbeitsweise des PAPs *KONSTRUKTION* exemplarisch erklärt. Es wird eine EID-Kette abgearbeitet, welche aus fünf Elementen besteht. Dies sind abwechselnd Elementreferenzen (*ERef*) und Gruppen (*Group*), welche entsprechend in fünf Durchläufen behandelt werden. Wird ein Wurzelelement mit dem Namen *root* dem PAP übergeben, dann wird der Pfad "*/root[1]*" konstruiert und in *POS* gespeichert. Das nächste Element der EID-Kette muss entsprechend eine Gruppe sein. In der Abbildung ist das eine Gruppe mit der maximalen Häufigkeit *2* (rot markiert), welche an die bereits vorhandenen Einträge der Liste *POS*

[18] siehe auch: Kapitel 4.1.1 (Features von EMX)

Durchlauf	1.	2.	3.	4.	5.
Element der EID-Kette	root	Group maxOccurs 2	ERef maxOccurs 2	Group maxOccurs 1	ERef1 maxOccurs 2
POS	/root[1]	/root[1] 2	/root[1]/eref[1]	/root[1]/eref[1] 1	/root[1]/eref[1]/eref1[1] /root[1]/eref[1]/eref1[2]
			/root[1]/eref[2]	/root[1]/eref[2] 1	/root[1]/eref[2]/eref1[1] /root[1]/eref[2]/eref1[2]
			/root[1]/eref[3]	/root[1]/eref[3] 1	/root[1]/eref[3]/eref1[1] /root[1]/eref[3]/eref1[2]
			/root[1]/eref[4]	/root[1]/eref[4] 1	/root[1]/eref[4]/eref1[1] /root[1]/eref[4]/eref1[2]

Abbildung 6.14.: Schrittweise Erweiterung von POS durch PAP KONSTRUKTION

angehängt wird. Dieses Anhängen wird nur vollzogen, insofern der *lastFlag* nicht gesetzt wurde, da ansonsten ungültige XPath-Ausdrücke gespeichert würden.

Dieser angehängte Wert wird im nächsten Durchlauf mit der maximalen Häufigkeit der Elementreferenz multipliziert, sodass jedes Element von *POS* viermal erweitert wird. Diese Erweiterung erfolgt, indem der Name der Referenz mit Position (d.h. *eref[1]* bis *eref[4]*) an das Element von *POS* angehängt wird. Der vorher ergänzte *maxOccurs-Wert* der Gruppe wird entfernt, ein Schrägstrich / zur Trennung der XPath-Ausdrücke wird ebenso eingefügt.

Dieses Vorgehen wird solange wiederholt, bis die Ziel-EID einer EID-Kette erreicht wird. Die Liste von absoluten Lokalisierungspfaden beinhaltet alle Positionen, an denen eine durch die Anwendung einer ELaX-Operation auf Schema- und Modellebene betroffene Komponente in der Instanzebene vorhanden sein kann.[19] Diese Liste *POS* wird nur durch die Ausnutzung der Informationen auf Schema- und Modellebene aufgebaut, sodass auf Instanzebene nicht jede Position gegeben sein muss. Nachfolgend wird das obige Beispiel abgeschlossen.

Beispiel Lokalisierungspfad

Es wurden für die Operationen *op1* die EID-Kette *(2, 11, 13)* und für *op2* die EID-Ketten *(2, 11)* und *(2, 11, 12, 17)* ermittelt. Aus diesen Ketten werden durch den PAP *KONSTRUKTION* die Lokalisierungspfade konstruiert. Die dafür notwendigen Informationen sind in Abbildung 6.15 dargestellt. Diese wurden aus dem

EID	2	11	13	12	17
Details der EID für Lokalisierungspfad	root	Group maxOccurs 2	e2 maxOccurs 2	e1 maxOccurs 2	Group maxOccurs 2

Abbildung 6.15.: Überblick der Elemente der EID-Ketten des Beispiels

XML-Beispiel 6.2 extrahiert und beinhalten den Namen der Elementreferenzen

[19] siehe auch: These 10

(*root*, *e1* und *e2*), die *EIDs* der Komponenten, sowie deren *maxOccurs-Werte*.

Ausgehend von der EID-Kette *(2, 11, 13)*, wird demnach der Teilpfad "*/root[1]*" konstruiert, der im zweiten Durchlauf aufgrund des *maxOccurs-Werts 2* (EID *11*) erweitert wird zu "*/root[1] 2*". Dieser angehängte Wert wird im dritten und letzten Durchlauf mit dem *maxOccurs-Wert 2* (EID *13*) multipliziert und entfernt, sodass alle bisherigen Teilpfade viermal ergänzt werden. Es werden somit für Operation *op1* die Lokalisierungspfade "*/root[1]/e2[1]*", "*/root[1]/e2[2]*", "*/root[1]/e2[3]*" und "*/root[1]/e2[4]*" konstruiert und in *POS* gespeichert.

Die EID-Kette *(2, 11)* wird ebenso behandelt. Es wird der Teilpfad "*/root[1]*" konstruiert, der im zweiten Durchlauf durch den *maxOccurs-Wert 2* (EID *11*) ergänzt werden müsste. Da im zweiten Durchlauf bereits die Ziel-EID erreicht wird, wird das Anhängen nicht vollzogen. Der Lokalisierungspfad "*/root[1]*" wird in *POS* für die Operation *op2* gespeichert.

Die letzte EID-Kette ist *(2, 11, 12, 17)*. Diese beinhaltet die vorherige Kette, allerdings wird die letzte EID-Kette komplett neu von der Wurzel ausgehend konstruiert. Es werden vier Durchläufe vollzogen, wobei der letzte erneut eine Gruppe (EID *17*) behandelt. Somit wird das Anhängen des *maxOccurs-Werts 2* wiederum verhindert. Es werden die folgenden Pfade für diese EID-Kette konstruiert: "*/root[1]/e1[1]*", "*/root[1]/e1[2]*", "*/root[1]/e1[3]*" und "*/root[1]/e1[4]*".

In Abbildung 6.16 wird abschließend für das Beispiel gezeigt, welche Informationen in der *Statement und Pfad Tabelle* gespeichert werden. Es werden zum

ELaX	wildFlag	POS
op1	0	/root[1]/e2[1], /root[1]/e2[2], /root[1]/e2[3], /root[1]/e2[4]
op2	1	/root[1]
	1	/root[1]/e1[1], /root[1]/e1[2], /root[1]/e1[3], /root[1]/e1[4]

Abbildung 6.16.: Inhalt der Statement und Pfad Tabelle des Beispiels

Abschluss der Lokalisierung eines ELaX-Statements die instanzverändernde Operation (*ELaX*), der *wildFlag* und die Lokalisierungspfade (*POS*) gespeichert.

6.4. Generierung von Informationen

Instanzerweiternde ELaX-Operationen benötigen Element- und Attributinhalte, welche beim Einfügen von Komponenten auf Instanzebene verwendet werden. Diese Informationen können ebenso wie die *Lokalisierungspfade* des vorherigen Abschnitts unter Verwendung der Schema- und Modellebene generiert werden. Dabei wird unterschieden sowohl zwischen dem einfachen und komplexen Inhalt, als auch dem Wildcardinhalt. Im Allgemeinen ist letzterer bei Elementwildcards notwendig, insofern diese nicht optional sind (d.h. *minOccurs != 0*).

Die Generierung des *einfachen Inhalts* wird in **Abschnitt 6.4.1** erläutert. Dabei werden unteren Anderem auch die unterschiedlichen Methoden der Nutzerinterak-

tion vorgestellt. Diese werden angewendet, insofern eine automatische Generierung von Inhalten auf der Schema- und Modellebene nicht möglich ist. Der *komplexe Inhalt* wird in **6.4.2** thematisiert, bevor der *Wildcardinhalt* in **Abschnitt 6.4.3** folgt. Abschließend wird exemplarisch in **6.4.4** die Generierung von Informationen beim *Einfügen, Löschen und Ändern von Elementreferenzen* erläutert.

In diesem Zusammenhang werden ebenso *Bedingungen* konstruiert, welche auf Instanzebene vor einer Adaption gelten müssen. Diese Bedingungen verhindern im Allgemeinen nicht, dass eine Instanzanalyse notwendig ist. Dennoch können diese zur Vermeidung von Modifikationen der Instanzen im Vorfeld geprüft werden.

6.4.1. Einfacher Inhalt

Die Generierung von Informationen ist in unterschiedlichen *Programmablaufplänen* (*PAP* - ProgrammAblaufPlan) beschrieben, die nachfolgend ausschnittsweise vorgestellt werden. Der PAP *GEN* ist in Abbildung 6.17 dargestellt. Dieser wird zur

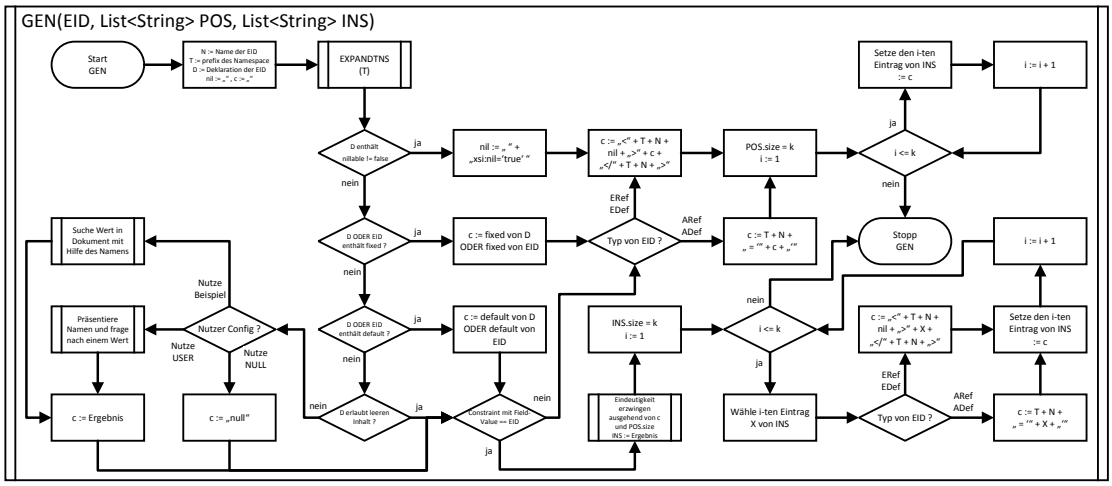


Abbildung 6.17.: PAP - GEN aus [Nös15c]

Generierung von einfachen Element- und Attributinhalten verwendet. Es werden sowohl die *EID* einer Komponente, als auch die im letzten Abschnitt ermittelten Lokalisierungspfade *POS* und eine Liste *INS* übergeben. Als Ergebnis des Plans wird beim Beenden (*Stopp GEN*) für jeden Eintrag in *POS* ein einfacher Inhalt an der entsprechenden Position in *INS* hinzugefügt. Diese können als *Referenzparameter* anschließend in den aufrufenden Strukturen verwendet werden.

Die EID bzw. die Deklaration (*D*) wird bezüglich deren Nullwertfähigkeit (*nillable*), *fixed-Wert* oder Defaultwert (*default*) nacheinander untersucht. Ist eine dieser Varianten möglich, wird entsprechend eine vollständige Zeichenkette zum Einfügen konstruiert. Diese beinhaltet die umschließenden Tags (z.B. *<e1></e1>*) mit notwendigen Attributen (z.B. *xsi:nil*) und gegebenenfalls einen Wert. Attribute

werden mit deren Namen statt Elementtags erzeugt. Ist ein leerer Inhalt im einfachen Datentyp möglich (z.B. *xs:string*), dann wird dieser verwendet. Anschließend wird die Liste *INS* unter Beachtung von *POS* entsprechend gefüllt.

Wird ein Defaultwert oder leerer Inhalt verwendet, und auf den einzufügenden Komponenten sind Constraints definiert, dann muss dies beachtet werden. *GEN* würde unter Verwendung der in [Kap13] eingeführten Methoden zur Sicherstellung von Namenseindeutigkeiten im globalen Gültigkeitsbereich eines XML-Schemas eindeutige Werte liefern (*Eindeutigkeit erzwingen [..]*), allerdings existieren Einschränkungen in der Behandlung von Constraints. Referentielle Integritäten auf Instanzebene sind auf Schema- und Modellebene nicht analysierbar und können daher in der XML-Schemaevolution nicht automatisiert behandelt werden.[20] Die Constraint-basierte Wissensgenerierung wird daher hier nur der Vollständigkeit halber aufgenommen. Diese bietet genügend Potential für weitere Arbeiten.[21]

Nutzerinteraktion

Sind obige Varianten zur Generierung von Werten nicht möglich, dann ist eine Nutzerbeteiligung im Allgemeinen kaum zu vermeiden. Drei Möglichkeiten wurden konzipiert, welche als *Nutzerkonfiguration* (*Nutzer Config*) ein weiteres *Feature*[22] des konzeptuellen Modells bilden und entsprechend konfiguriert werden können.

Die einfachste Methode ist das Präsentieren der betroffenen Komponenten und dem damit verbundenen *Nachfragen von Werten* (*Nutze USER*). Eine weitere Möglichkeit ist das *Einlesen von Dokumenten*, in welchen mit entsprechenden Suchmethoden (z.B. Namenssuche) notwendige Werte ermittelt werden können (*Nutze Beispiel*). Beide Vorgehen sind beim Vorhandensein einer großen Anzahl von Lokalisierungspfaden aufwendig und garantieren kein zufriedenstellendes Ergebnis.

Es ist denkbar, dass die übergebenen Dokumente die erhofften Werte nicht enthalten bzw. nicht effizient durchsucht werden können (z.B. unstrukturierte Textdateien). Im Allgemeinen können darüber hinaus ermittelte oder übergebene Werte im aktuellen Kontext auch fehlerhaft sein bzw. dem laut XML-Schema geforderten, einfachen Datentyp nicht entsprechen. Daher wurde die dritte Möglichkeit (*Nutze NULL*) als Defaultkonfiguration dieses Features spezifiziert.

Diese Methode nutzt die Möglichkeiten des Einbindens von Modulen (*addmodule*), eine gemäß Abbildung 6.2 kapazitätsverändernde, instanzerhaltende ELaX-Operation. Das heißt, dass das Hinzufügen keine Instanzkosten verursacht, allerdings einen wichtigen Beitrag zur Adaption der Instanzen liefert. Des Weiteren ist bei einer entsprechenden Konfiguration keine Nutzerinteraktion notwendig, da die entsprechende ELaX-Operation bei Bedarf automatisch ausgeführt werden kann.

[20] siehe auch: These 11
[21] siehe auch: Kapitel 8.2 (Ausblick)
[22] siehe auch: Kapitel 4.1.1 (Features von EMX)

Nutzung von Nullwerten

Es wurde das in XML-Beispiel A.13 dargestellte XML-Schema erzeugt, welches öffentlich unter `http://www.ls-dbis.de/codex` bereitgestellt wird und als Modul eingebunden werden kann. XML-Schema kennt keine Nullwerte im Sinne des relationalen Modells, sondern ermöglicht nur die Nullwertfähigkeit bei Elementen durch das *nillable-Attribut*. Daher können Nullwerte nicht verwendet werden.

Mit dem bereitgestellten, obigen Schema ist es allerdings unter Verwendung der Spezifikation von XML-Schema möglich, Datentypen so zu erweitern, dass diese unabhängig vom Datentyp bestimmte **Nullwerte** akzeptieren. Dafür wurde der einfache Typ *null* spezifiziert, welcher ein Restriktionstyp mit Basis *xs:string* ist. Dieser erlaubt aktuell vier Aufzählungswerte (*enumeration*), wobei die angedachte Semantik in Klammern ergänzt wurde: *null* (no information null), *exist* (existent but unknown), *never* (no applicable null) und die leere Zeichenkette.

Des Weiteren wurden alle built-in-Datentypen des Standards in eigene Vereinigungstypen aufgenommen, die Benennung erfolgte analog zum involvierten Membertyp. Das heißt, dass unter anderem *xs:string* den Namen *string* erhält. Jedem Vereinigungstypen wurde als zweiter Teilnehmer der obige *null* Datentyp zugeordnet. Wird nun die Typkonformität kontrolliert, dann wird gemäß des Standards zuerst der Wertebereich des ersten Membertyps und dann jedes weiteren geprüft.

Abschließend muss die Komponente angepasst werden, für welche die Generierung von Werten (d.h. *GEN*) ausgeführt wurde. Das heißt, dass entweder ein *updelementdef* oder ein *updattribute* ausgeführt wird. In beiden Fälle wird der entsprechende Typ geändert, sodass die EID der externen Typdefinition referenziert wird. Dieser alternative Typ ist gemäß [Kap14] ein Obertyp (*OT*). Daraus folgt, dass die notwendige ELaX-Operation wiederum keine Instanzkosten verursacht, da gemäß der *Extralisten* in den Abbildungen A.15 und A.21 der *Werte Schnittpunkt*[23] keine Instanzanpassungen benötigt (*nix*). Die Operationen sind instanzerhaltend.

6.4.2. Komplexer Inhalt

Zusätzlich zu einfachen Elementinhalten wird mit dem PAP in Abbildung 6.18 ein Ablauf beschrieben, mit dem komplexe Elementinhalte erzeugt werden können. Die Übergabeparameter entsprechen denen von *GEN*, allerdings referenziert die übergebene EID eine komplexe Typdefinition (*CT*) mit einer Gruppe (*GR*).

Das Ziel von *GENCT* ist es, einen möglichst kompakten, wenig Informationen enthaltenden Elementinhalt zu erzeugen, der als *Referenzparameter* in *INS* für jeden Eintrag in *POS* gespeichert wird. Daher sind in dem PAP verschiedene Bedingungen formuliert, um dieses Ziel zu erreichen. Das heißt, dass zum Beispiel getestet wird ob die Gruppe (*GR*) des übergebenen Elements verboten ist (*minOccurs == 0*) und daher keine Kinderelemente benötigt werden.

[23] siehe auch: Kapitel 6.2.2 (Ändern von Komponenten)

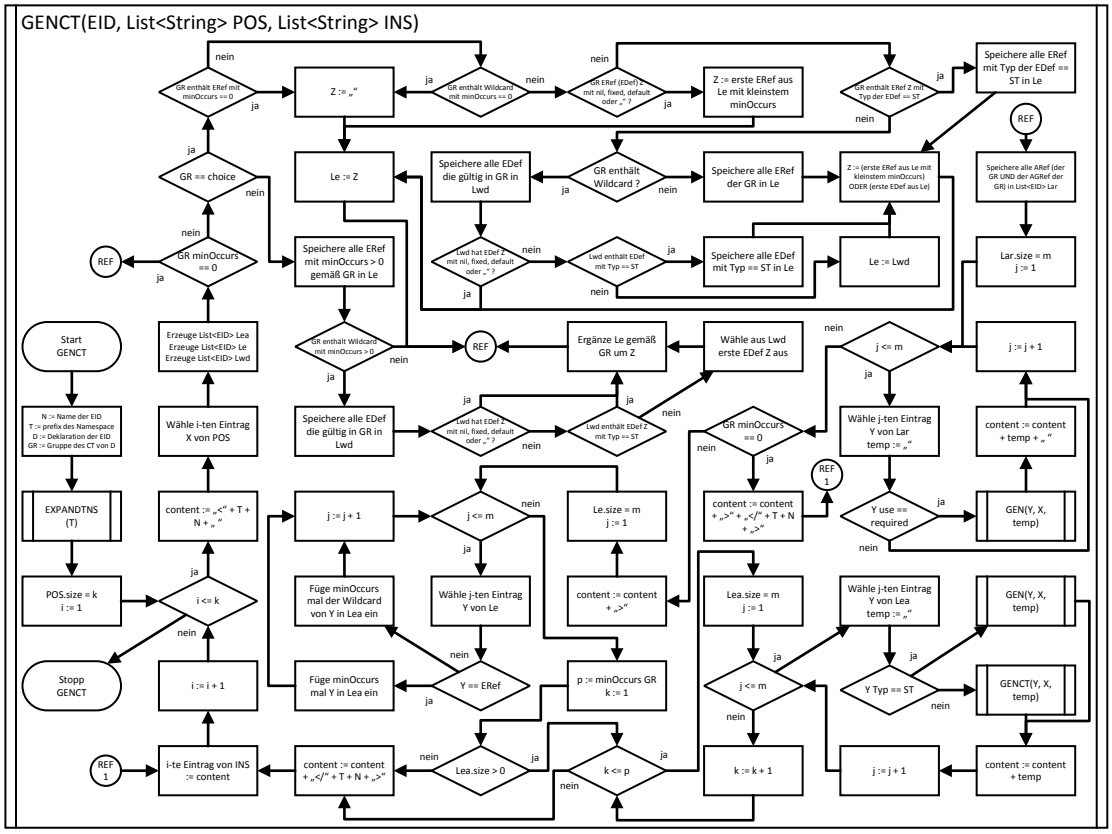


Abbildung 6.18.: PAP - GENCT aus [Nös15c]

Es wird ebenso untersucht, ob die Gruppe eine Auswahl (*GR == choice*) mit einer optionalen Referenz oder Wildcard ist. In diesem Fall werden ebenfalls keine Kinderelemente generiert, da die entsprechende, optionale Komponente in jedem Gruppendurchlauf gewählt wird. Des Weiteren werden nur solche Elemente und Attribute erzeugt, welche zwingend erforderlich sind (*minOccurs > 0* und *use == required*). Bei Elementen wird generell die minimale Häufigkeit verwendet (*Füge minOccurs mal [..]*), ebenso bei den Gruppendurchläufen (*p := minOccurs GR*).

Der PAP *GENCT* enthält vier Listen von EIDs, welche nacheinander gefüllt werden. Dies sind die Listen zum temporären Sammeln sowohl aller zwingenden Elementreferenzen und Wildcards (*Le*), als auch aller gültigen Deklarationen einer Wildcard (*Lwd*) und zwingender Attributreferenzen (*Lar*). Die vierte Liste ist die Gesamtliste aller Elementinhalte eines Gruppendurchlaufs (*Lea*).

Die Funktionsweise des Programmablaufplans *GENCT* mit den Listeninhalten sowie dem erzeugten Ergebnis (*INS*) wird nachfolgend an einem Beispiel erläutert.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" id="EID1">
    <xs:element name="root" type="roottype" id="EID2"/>
    <xs:element name="e1" type="ctype" id="EID3"/>
    <xs:element name="e2" type="xs:string" id="EID4"/>
    <xs:attribute name="a1" type="xs:string" id="EID5"/>
    <xs:attribute name="a2" type="xs:string" id="EID6"/>
    <xs:attributeGroup name="ag1" id="EID7">
        <xs:attribute ref="a1" id="EID8" use="required"/>
        <xs:anyAttribute id="EID9"/>
    </xs:attributeGroup>
    <xs:complexType name="roottype" id="EID10">
        <xs:sequence minOccurs="1" maxOccurs="2" id="EID11">
            <xs:element ref="e1" minOccurs="1" maxOccurs="2" id="EID12"/>
            <xs:element ref="e2" minOccurs="1" maxOccurs="2" id="EID13"/>
        </xs:sequence>
        <xs:attributeGroup ref="ag1" id="EID14"/>
    </xs:complexType>
    <xs:element name="e3" type="xs:string" id="EID15"/>
    <xs:complexType name="ctype" id="EID16">
        <xs:sequence minOccurs="2" maxOccurs="2" id="EID17">
            <xs:element ref="e3" minOccurs="1" maxOccurs="1" id="EID18"/>
            <xs:any minOccurs="1" id="EID20"/>
        </xs:sequence>
        <xs:attributeGroup ref="ag1" id="EID19"/>
    </xs:complexType>
</xs:schema>
```

XML-Beispiel 6.3: Erweiterung des XML-Schemas aus XML-Beispiel 6.2

Beispiel GENCT

Ausgehend vom XML-Beispiel 6.2 wurde das XML-Schema aus XML-Beispiel 6.3 spezifiziert. Die Änderungen sind jeweils rot hervorgehoben. Das heißt, dass im XML-Schema 6.3 sowohl die Attributreferenz *a1* (EID *8*), als auch die Elementreferenzen e2 (EID *13*) und e3 (EID *18*) zwingend sind. Eine zwingende Elementwildcard (EID *20*) wurde ebenso in einer Gruppe (EID *17*) ergänzt.

Die durch *GENCT* erzeugten Strukturen sind in Abbildung 6.19 dargestellt, wobei aus Gründen der Lesbarkeit die Namen und nicht die EIDs verwendet werden. Es sind vier Zeilen (#) enthalten, welche durch die Übergabe eines komplexen *Elements* aus XML-Beispiel *1* (6.2) oder *2* (6.3) an *GENCT* erzeugt werden. In der

#	Beispiel	Element	Le	Lwd	Lar	Lea	INS
1	1	root	e1	-	-	e1	<root><e1></e1></root>
2	2	root	e1, e2	-	a1	e1, e2	<root a1="">**#4.INS**<e2></e2></root>
3	1	e1	-	-	-	-	<e1></e1>
4	2	e1	e3, e2	e2, e1, root	a1	e3, e2	<e1 a1=""><e3></e3><e2></e2><e3></e3><e2></e2></e1>

Abbildung 6.19.: Beispiel zur Generierung komplexer Elementinhalte durch GENCT

Spalte *INS* wird jeweils das generierte Ergebnis dargestellt. Der Eintrag *#4.INS* in Zeile *2* ist eine verkürzte Schreibweise und bedeutet, dass aus Zeile *4* der Inhalt von *INS* einmal eingefügt werden muss.

Wird das Element *e1* aus dem XML-Schema 6.2 (*Beispiel 1*) übergeben (*#3*), dann sind alle beteiligten Listen leer (-) und es wird der einfache Elementinhalt *<e1></e1>* in *INS* gespeichert. Dies ist damit zu begründen, dass die einzige Elementreferenz *e3* des komplexen Typs *ctype* optional ist und keine Elementwildcards existieren. Somit sind die Listen *Le*, *Lwd* und *Lea* jeweils leer.

Die referenzierte Attributgruppe *ag1* enthält eine Attributreferenz *a1* und eine Attributwildcard (EID *9*). Da im ersten Fall kein *use-Attribut* vorhanden ist, ist diese Referenz optional. Attributwildcards sind standardkonform immer optional, sodass diese niemals erzeugt werden. Daher ist die Liste *Lar* ebenfalls leer.

In *#4* wird das Element *e1* aus dem XML-Schema des XML-Beispiels 6.3 analysiert (*Beispiel 2*). Die Elementreferenz *e3* ist nun zwingend und die nicht optionale Elementwildcard (EID *20*) wurde hinzugefügt. Somit wird *e3* in *Le* ergänzt. *Lwd* wird mit den Deklarationen gefüllt, die für die Wildcard gültig sind.

Da die Elementwildcard keine Einschränkungen vornimmt, können dort alle Elementdeklarationen verwendet werden (*e2, e1, root*). Diese Einträge werden nach deren Datentypen sortiert, sodass *e2* als erstes Element aufgelistet wird. Dies ist damit zu begründen, dass *e2* den einfachen Datentyp *xs:string* besitzt, der eine leere Zeichenkette als Wertinhalt erlaubt. Element *e2* wird in *Le* hinzugefügt.

Anschließend werden alle Attributreferenzen in *Lar* gespeichert, die ein *use-Attribut* mit dem Wert *required* besitzen. Dies ist im XML-Beispiel 6.3 die Referenz *a1*. Für diesen Eintrag wird der obige PAP *GEN* aufgerufen, um einen einfachen Attributinhalt zu erzeugen. Dieser liefert aufgrund des Datentyps *xs:string* das Ergebnis *a1=""*, sodass diese Zeichenkette im Elementtag von *e1* ergänzt wird.

Im nächsten Schritt wird die Gesamtliste *Lea* erzeugt. Es sind zwei Gruppendurchläufe notwendig, da dies die minimale Häufigkeit der Gruppe (EID *17*) des komplexen Typs *ctype* spezifiziert (*minOccurs = 2*). Das heißt, dass *Lea* mit allen Elementen aus *Le* mit deren minimalen Auftreten (jeweils *minOccurs = 1*) insgesamt zweimal wiederholt wird (Ergebnis: *e3, e2, e3, e2*). Für jedes Element aus *Lea* wird anschließend in Abhängigkeit des Datentyps entweder *GEN* oder *GENCT* aufgerufen, um dessen einfachen oder komplexen Elementinhalt zu generieren. Dieser wird dem aktuell konstruierten Inhalt (*content*) von *e1* angehängt. Im Anschluss wird der Elementtag von *e1* geschlossen (*</e1>*) und das Ergebnis wird jeweils in *INS* ergänzt.

6.4.3. Wildcard Inhalt

Im vorherigen Abschnitt wurden unter anderem Wildcards thematisiert, die bei der Erzeugung von komplexen Elementinhalten berücksichtigt werden müssen. In diesem Zusammenhang wurde erläutert, dass bei instanzerweiternden Operationen

möglichst eine Deklaration mit einfachem Datentyp verwendet werden sollte.

Ist dies allerdings nicht möglich oder es liegen mehrere Deklarationen mit einfachem Datentyp vor, dann muss ein anderer bzw. ein bestimmter, gültiger Kandidat ausgewählt werden. Der PAP *SORTDEF* aus Abbildung 6.20 wurde für die Problematik der Deklarationsauswahl realisiert. Dieser Programmablaufplan bekommt

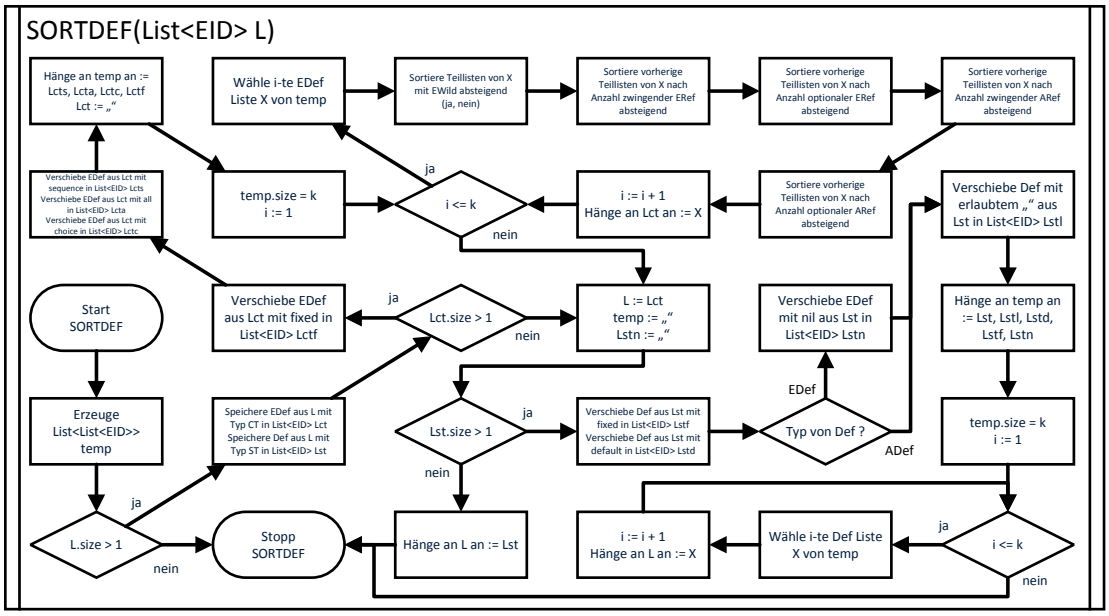


Abbildung 6.20.: PAP - SORTDEF aus [Nös15c]

eine Liste von gültigen Deklarationen übergeben und sortiert diese nach unterschiedlichen Kriterien. Es können sowohl Listen von Attribut- als auch Elementwildcards behandelt werden. Letztere sind dadurch charakterisiert, dass zusätzlich komplexe Datentypen und die Nullwertfähigkeit analysiert werden.

Es werden unterschiedliche Listen erzeugt, welche schrittweise bearbeitet werden. Das Ziel ist es, eine sortierte Liste zu konstruieren, welche am Anfang die Deklarationen mit den komplexen und am Ende die mit den einfachen Datentypen hat. Die Teillisten werden im Allgemeinen zusätzlich sortiert, sodass zum Beispiel bei den komplexen Datentypen zuerst nach dem Vorhandensein von Elementwildcards geschaut wird, bevor nach der Anzahl der zwingenden und optionalen Elementreferenzen sortiert wird. Anschließend werden bei gleicher Anzahl die zwingenden und optionalen Attributreferenzen betrachtet.

Die Reihenfolge der Deklarationen in Abhängigkeit der Datentypen ist in Abbildung 6.21 dargestellt. Die komplexen Typen mit dem Inhaltsmodell *sequence* sind demnach in der Liste *Lcts* gespeichert. Innerhalb dieser Liste wird, wie oben erwähnt, nach Wildcards, Element- und Attributreferenzen absteigend sortiert.

Am Ende der Ergebnisliste werden die einfachen Datentypen ergänzt, wobei

Datentyp	komplexe Typen mit				einfache Typen mit				
Kriterium	sequence	all	choice	fixed	sonstige	leere Zeichenkette	default	fixed	nillable
	EWild, ERef (zwingend), ERef (optional), ARef (zwingend), ARef (optional)								
Listenname	Lcts	Lcta	Lctc	Lctf	Lst	Lstl	Lstd	Lstf	Lstn

Abbildung 6.21.: Deklarationsreihenfolge in Abhängigkeit der Datentypen

sonstige alle Typen enthält, die weder eine leere Zeichenkette (*""*), noch die Attribute *default*, *fixed* oder *nillable* spezifizieren. Die Ergebnisliste kann anschließend als Referenzparameter in den aufrufenden Strukturen verwendet werden.

Die Reihenfolge wurde unter Beachtung der *Generierung von Werten* definiert. Dabei wird die Vermutung zugrunde gelegt, dass Sequenzen komplizierter zu erzeugen sind als Mengen. Eine Sequenz erlaubt eine minimale Häufigkeit größer als *1*, was bei Mengen nicht möglich ist. Es werden demnach im Allgemeinen mehr Gruppendurchläufe benötigt. Eine Auswahl verwendet nur ein Kinderelement, was eine Vereinfachung zu den beiden vorherigen Inhaltsmodellen darstellt.

Die einfachen Typen wurden in der umgekehrten Reihenfolge definiert, in welcher *GEN* diese prüft. Die in Abschnitt 6.4.1 eingeführten Nullwerte durch den Datentyp *null* sind nicht gesondert dargestellt, sondern in der Liste *Lst* enthalten.

Wird nun ein gültiger Kandidat für eine Wildcard gesucht, dann kann nach der Anwendung von *SORTDEF* unter anderem eine im Allgemeinen einfach zu erzeugende Deklaration gewählt werden. Dies ist der letzte Eintrag der Ergebnisliste.

Die Generierung von sowohl einfachen und komplexen Inhalten, als auch von Wildcardinhalten bei vorhandenen, gültigen Deklarationen, ist unter Verwendung der vorgestellten Mechanismen auf Schema- und Modellebene in der XML-Schemaevolution automatisierbar.[24] *GEN* und *GENCT* werden diesbezüglich angewendet.

6.4.4. Elementreferenzen

Das Einfügen, Löschen und Ändern von Elementreferenzen wird nachfolgend im Zusammenhang mit der Generierung von Informationen thematisiert. Dabei soll exemplarisch erläutert werden, inwiefern instanzverändernde Operationen noch detaillierter als in Abschnitt 6.2 analysiert werden können. Das Ziel ist die Formulierung von Bedingungen (*CON*), welche auf Instanzebene vor einer Adaption geprüft werden. Des Weiteren werden die Programmablaufpläne *GEN* und *GENCT* angewendet, allerdings nur insofern dies notwendig ist.

Der allgemeine PAP zur *Generierung von Werten* ist in Abbildung 6.22 dargestellt. Dieser enthält zusätzliche Programmablaufpläne für alle instanzverändernden Operationen der Abbildung 6.2, welche *Instanzkosten* verursachen. Die *add-* und *delete-Operationen* wurden jeweils zusammengefasst und sind in den Abbildungen A.30 und A.31 dargestellt. Die *update-Operationen* sind aufgrund deren

[24] siehe auch: These 12

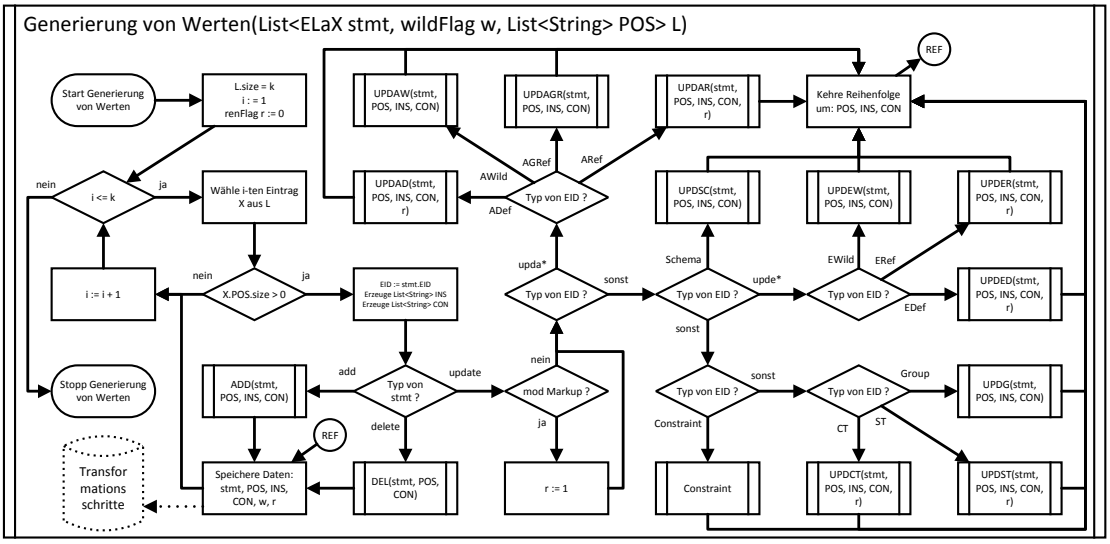


Abbildung 6.22.: PAP - Generierung von Werten aus [Nös15c]

Umfangs jeweils einzeln mit den unterschiedlichen Hilfsmethoden und gegebenenfalls mit Hinweisen in den Abbildungen A.32 bis A.55 aufgenommen worden.

Die Betrachtung von *Constraints* bildet an dieser Stelle erneut eine Ausnahme. Es wurde in Abschnitt 6.4.1 dargelegt, dass referentielle Integritäten auf Instanzebene nicht auf Schema- und Modellebene analysierbar sind und demnach in der XML-Schemaevolution nicht automatisierbar behandelt werden können.

Eine weitere Problematik beim Ändern von Constraints ergibt sich aus der Verwendung von XPath zur Definition von *Selektor*- und *Fieldwerten*.[25] Würde eine Constraint verändert werden, dann müsste unter anderem geprüft werden, ob diese Ausdrücke übereinstimmen (*Containment*). Die Knotenäquivalenz in Verbindung mit beliebigen Nachfolgern (*//*) und Wildcards (***) ist allerdings bereits bei einem einfachen Schema (*DTD*) unentscheidbar [Sch04].[26] In [NS03][27] und [DT01][28] wird dies ebenso bestätigt, wobei in [DT01] Wildcards bei einfachen XPath-Ausdrücken noch nicht berücksichtigt wurden. Das Ändern von Constraints ist in der XML-Schemaevolution nicht automatisierbar, es wurde daher hier nur der Vollständigkeit halber aufgenommen. Diese Problematik bietet genügend Potential für weitere Arbeiten[29], ein möglicher Ansatz wird in Kapitel 7.2.3 skizziert.

[25] siehe auch: http://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/#sec-c-selector-xpath

[26] "XPath containment: Undecidable for XP(/, //,[],*,|) + nodeset equality + simple DTDs" [Sch04]

[27] "DTD, /, //, [], *: undecidable with nodeset comparisons" [NS03]

[28] "Containment of simple XPath expressions under unbounded SXICs (Simple XPath Integrity Constraints, Anm. d. Autors) is undecidable." [DT01]

[29] siehe auch: Kapitel 8.2 (Ausblick)

Hinzufügen und Löschen von Elementreferenzen

Die Entscheidung zur Anwendung eines PAPs wird durch den Typ der ELaX-Operation festgelegt. Wird eine Elementreferenz hinzugefügt, dann wird der PAP *ADD* aus Abbildung A.30 angewendet. Es werden sowohl die absoluten *Lokalisierungspfade* (*POS*), als auch leere Strukturen für die *Generierung von Werten* (*INS*) und Bedingungen zur Anwendung (*CON*) als Referenzparameter übergeben.

Analog zur Abbildung 6.4 werden an dieser Stelle die Tests bezüglich der Optionalität (*minOccurs > 0*) und dem zwingenden Auftritt der Gruppe (*GR maxOccurs == 0*) durchgeführt. Dies stellt eine Redundanz dar, allerdings sind die Tests im konzeptuellen Modell als Attribut-Wert-Vergleiche leicht anzuwenden. Wird aufgrund der Analyse eine Adaption der Instanzebene überflüssig, dann werden endgültig alle Einträge aus POS entfernt (*Entferne alle Zeilen aus POS*).

Eine zusätzliche Überprüfung betrifft die Lokalisierungspfade. Elementreferenzen werden gemäß Abbildung 6.7 durch deren eigene EID (*Ziel-EID*) identifiziert. Das heißt, dass der letzte Bestandteil des XPath-Ausdrucks (*Lokationspfad*) die Referenz selbst adressiert. Wird dieser Bestandteil entfernt (*temp*) und der verbleibende Ausdruck ist leer (*temp.length > 0*), dann wäre die Elementreferenz auf Instanzebene ein Wurzelknoten. Das Hinzufügen von solchen Komponenten bedeutet, dass eine Instanz im Vorfeld nicht gültig bezüglich des XML-Schemas war. Dies würde dem dargestellten Szenario der XML-Schemaevolution widersprechen, sodass die entsprechenden Lokalisierungspfade entfernt werden.

Ist der verbleibende Ausdruck (*Teilpfad*) allerdings nicht leer, dann wird in Abhängigkeit der referenzierten Deklaration (*Y*) entweder ein einfacher (*GEN*) oder komplexer Elementinhalt (*GENCT*) in *INS* gespeichert.[30] Des Weiteren wird eine **Existenzbedingung** formuliert und in *CON* eingefügt (d.h. *boolean(Lokationspfad)*). Diese Bedingung wird auf Instanzebene vor einer Adaption geprüft. Das heißt, dass vor einer Modifikation getestet wird, ob die Gruppe der Elementreferenz entsprechend in der Instanz gegeben ist oder nicht.

Existenzbedingungen werden auch beim Löschen von Elementreferenzen formuliert. Der PAP *DEL* aus Abbildung A.31 beinhaltet dies. Im Allgemeinen wird für jede instanzverändernde ELaX-Operation eine solche Bedingung gespeichert.

Beim Löschen einer Elementreferenz wird zusätzlich geprüft, ob diese verboten ist (*maxOccurs == 0*). Ist dies der Fall, dann werden analog zum Einfügen die Lokalisierungspfade aus *POS* entfernt (*Entferne alle Zeilen aus POS*). Da gemäß Abbildung 6.2 keine zusätzlichen Elementinhalte aufgrund des *Informationsgehalts* benötigt werden, wird weder *INS* an *DEL* übergeben, noch *GEN* bzw. *GENCT* aufgerufen. Das Löschen von Komponenten ist nicht instanzerweiternd.

[30] siehe auch: Kapitel 6.4 (Generierung von Informationen)

Ändern von Elementreferenzen

Das Ändern einer Elementreferenz wird durch den PAP in Abbildung 6.23 beschrieben. Es werden bei jeder Änderungsoperation zwei Phasen unterschieden, eine *Analyse-* und *Generierungsphase*. Die Analysephase ist eine Spezialisierung

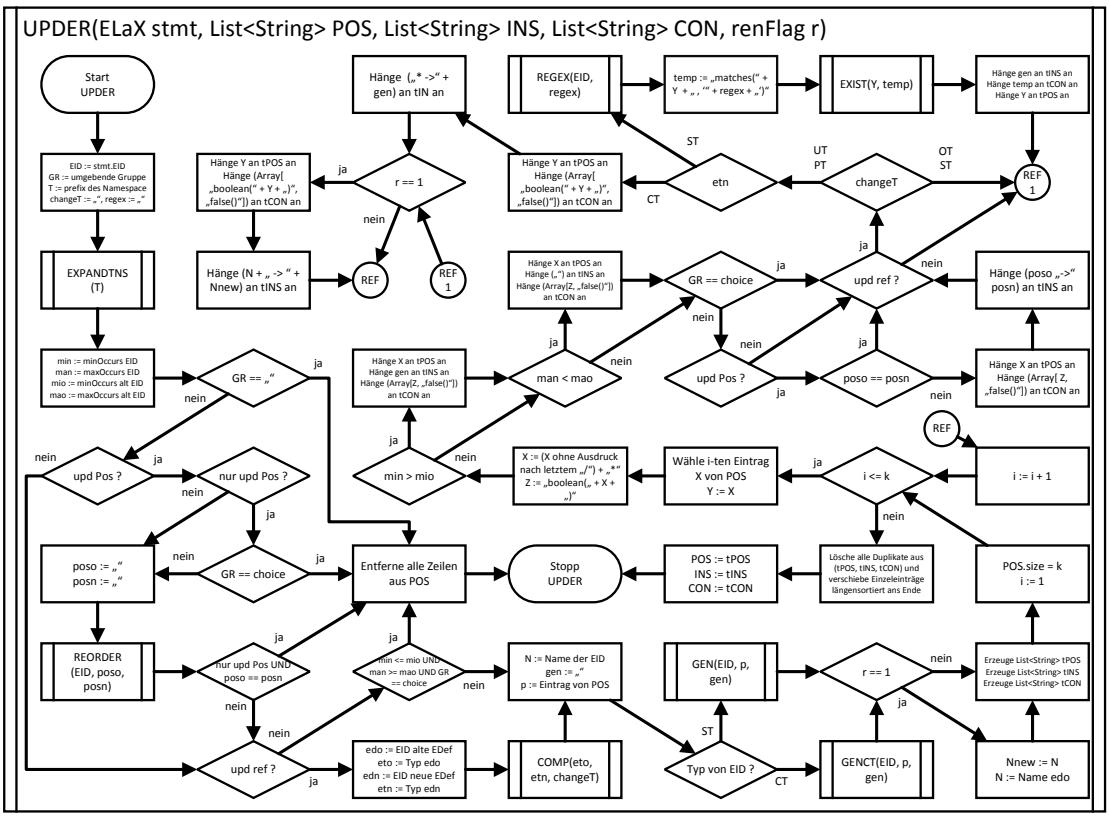


Abbildung 6.23.: PAP - UPDER aus [Nös15c]

der in Abbildung 6.5 thematisierten Auswirkung auf ein Dokument. Im Allgemeinen läuft diese bis zur Initialisierung der Lokalisierungspfadgröße ($POS.size = k$).

Es wird detaillierter geprüft, ob die vorliegende Änderung eventuell keinen Einfluss auf die Instanzebene besitzt. Dazu zählt unter anderem, dass wenn die umgebene Gruppe leer ist ($GR = ""$), die Operation analog zum Einfügen nicht berücksichtigt wird (s.o. Elementreferenz auf Instanzebene ist ein Wurzelknoten).

Des Weiteren wird analysiert, ob Änderungen der Position ($xPos$ und/oder $yPos$) vollzogen wurden. Wird eine Elementreferenz im EMX grafisch verschoben, dann werden die entsprechenden Positionen angepasst. Solche Änderungen sind allerdings nur dann entscheidend, wenn das Inhaltsmodell instanzverändernd betroffen ist. Um dies zu entscheiden wurde der PAP *REORDER* aus Abbildung A.41 umgesetzt. Dieser untersucht die Referenzreihenfolge. In Abbildung 6.24 wird dieses

anhand von zwei Elementreferenzen (A und B) illustriert. Es werden alle Kombi-

		xPos	yPos	EID	xPos	yPos	EID	xPos	yPos	EID
ERef	A	1	_	_	1	1	_	1	1	1
	B	2	_	_	1	2	_	1	1	2
Reihenfolge		A, B			A, B			A, B		

Abbildung 6.24.: Beispiel der Ermittlung der Reihenfolge von Elementreferenzen

nationen von Positionierungen (*xPos* und *yPos*) mit der resultierenden *Reihenfolge* dargestellt. Insofern eine Belegung nicht analysiert werden muss, wird _ verwendet. Zum Beispiel ist die Referenz A im Inhaltsmodell vor B (A, B), falls *xPos* kleiner ist. Ist dieser Wert allerdings gleich, wird *yPos* und bei erneuter Gleichheit die *EID* ausgewertet. Da die EID eindeutig ist, kann die Reihenfolge zwischen Elementreferenzen trotz gleicher Positionierung zweifelsfrei bestimmt werden.

Ändert sich die Reihenfolge nicht instanzverändernd, und weitere Test bezüglich der *minOccurs-* und *maxOccurs-Werte* führen zu keiner Entfernung der Lokalisierungspfade (*Entferne alle Zeilen aus POS*), dann wird unter anderem die Referenz untersucht (*upd ref ?*). Es kommt der PAP *COMP* zum Einsatz, der aufbauend auf den Mechanismen aus [Kap14] die Beziehung zwischen dem alten (*eto*) und neuen (*etn*) Typen analysiert. Gemäß [Kap14] sind die folgenden disjunkten Typbeziehungen möglich, die in *changeT* für spätere Analysen gespeichert werden: Paralleltyp (PT), Obertyp (OT), Untertyp (UT) oder selber Typ (ST).

Anschließend wird *GEN* bzw. *GENCT* aufgerufen, sodass entweder ein einfacher oder komplexer Elementinhalt generiert und in *gen* gespeichert wird. Die Analysephase endet mit der Erzeugung von Strukturen, in denen nachfolgend für jeden Lokalisierungspfad aus *POS* sowohl temporäre Pfade (*tPOS*), als auch Bedingungen (*tCON*) und einzufügende Elementinhalte (*tINS*) gesammelt werden.

Generierungsphase von Elementreferenzen

Die Generierungsphase ermittelt alle für eine Instanzanpassung notwendigen Informationen und speichert diese in den übergebenen Referenzparametern des PAPs *UPDER*. Als temporäre Strukturen werden obige Listen (*tPOS*, *tCON* und *tINS*) verwendet, die für jeden Lokalisierungspfad aus *POS* schrittweise ergänzt werden.

Es werden in Abhängigkeit der Attribut-Wert-Paare des ELaX-Ausdrucks (E18), sowie unter Beachtung der Klassifikation in Abbildung A.14 und der Extraliste in Abbildung 6.6 Informationen gespeichert. Zum Beispiel wird bei einer Erhöhung der minimalen Häufigkeit ($min > mio$) der in der Analysephase erzeugte Elementinhalt *gen* mit einer *Existenzbedingung* (d.h. *boolean(X)*) gespeichert. Als Lokalisierungspfad (X) wird bei Häufigkeitsänderungen ein modifizierter Teilpfad verwendet, da Elementreferenzen von deren Kontext und direktem Knotenumfeld abhängig sind. Die Konsequenz ist, dass Duplikate in den temporären Struktu-

ren entstehen können, welche allerdings abschließend beseitigt werden (*Lösche alle Duplikate*).[31] Bei Verringerung der maximalen Häufigkeit ($man < mao$) wird im Unterschied zu *minOccurs* statt *gen* ein leerer String (*""*) in tINS ergänzt.

Anschließend wird die Positionsänderungen behandelt (*upd Pos ?*), wobei sowohl die alte (*poso*), als auch neue Position (*posn*) in *tINS* gespeichert werden ($poso \rightarrow posn$). Diese Angaben wurden in der Analysephase durch den PAP *REORDER* ermittelt und können somit gegebenenfalls verwendet werden. Als Vorbedingung für die Positionsanalyse darf das Inhaltsmodell der umgebenen Gruppe keine Auswahl sein ($GR == choice$), da in einem solchen Inhaltsmodell die Reihenfolge von Elementreferenzen die Gültigkeit der Instanzebene nicht beeinflusst.

Die Analyse einer Referenzänderung wird im Anschluss vollzogen (*upd ref ?*). Die Typbeziehung der alten und neuen Elementdeklaration, welche in der Analysephase in *changeT* gespeichert wurden, sind dabei entscheidend. Ist der Datentyp der neu referenzierten Deklaration ein Obertyp (OT) oder sogar der gleiche Typ (ST), so muss gegebenenfalls nur eine Umbenennung vollzogen werden. Liegt allerdings ein Unter- (UT) oder Paralleltyp (PT) vor, müssen weitergehende Analysen in Abhängigkeit eines einfachen (ST) oder komplexen Typs (CT) vollzogen werden.

Ist ein komplexer Typ durch die referenzierte Elementdeklaration gegeben, dann wird die entsprechende Elementreferenz komplett ersetzt ($* \rightarrow gen$). Dies ist damit zu begründen, dass ein Vergleich einer Definition auf Schema- und Modellebene syntaktisch möglich wäre, allerdings semantisch nicht sinnvoll ist. Zum Beispiel müssten alle Kinderelemente (u.a. *ERefs*, *ARefs* und *AGRefs*) untersucht werden, ob diese im neuen Typen ebenso gegeben sind oder nicht. Ein Mapping und Matching wäre notwendig, was gemäß [Def13] besonders bei Paralleltypen schwierig zu realisieren wäre. Es wäre unter anderem zusätzliches Wissen notwendig, um entscheiden zu können, dass verschiedene, unterschiedlich benannte Elemente einander entsprechen. Dieses Wissen (d.h. die Semantik) müsste neben dem XML-Schema versionsübergreifend gepflegt und aktuell gehalten werden, ein nicht zumutbarer Overhead an Prozessen würde entstehen. Des Weiteren kann aufgrund der *1-zu-n-Abbildung* der Schema- und Instanzebene nicht gewährleistet werden, dass das obige Mapping automatisiert in einem Dokument anwendbar ist. Es wäre somit ein hohes Maß an Nutzerinteraktion notwendig, was der Zielsetzung der Arbeit widersprechen würde. In der XML-Schemaevolution wird daher bei komplexen Typen der automatisierbare Ansatz der Ersetzung bevorzugt.

Ist ein einfacher Typ (ST) gegeben, dann wird eine **Matchbedingung** konstruiert (d.h. *matches(Lokalisierungspfad, Regex)*). Der dabei notwendige reguläre Ausdruck *Regex*, mit welchem der einfache Elementinhalt bezüglich dessen Gültigkeit geprüft wird, wird durch den PAP *REGEX* in Abbildung A.37 ermittelt.[32]

[31] Duplikate: *min++* mit POS = {/a/b[1], /a/b[2], /a/b[3]} resultiert in tPOS = {/a/*, /a/*, /a/*}, tINS = {, , }, tCON = {boolean(/a/*), boolean(/a/*), boolean(/a/*)}

[32] Die Basis für *Regex* sind im Allgemeinen die *Pattern* der Abbildung A.1, welche gemäß der in [Kap14] beschriebenen Typhierarchien erweitert werden müssen (z.B. das Einfügen von Aufzählungswerten).

Die Grundlagen für diese *Wertebereichsintegrität* wurden in [Gru13] geschaffen. Die *Matchbedingung* wird anschließend durch den PAP *EXIST* der Abbildung A.29 um eine *Existenzbedingung* erweitert, sodass es nur zur Prüfung der *Wertebereichsintegrität* kommt, insofern das entsprechende Element in der Instanz gegeben ist.

Zum Abschluss der Generierungsphase wird geprüft, ob eine Umbenennung vorgenommen wurde. Insofern der übergebene *renFlag* dies angibt ($r == 1$), wird in die temporären Strukturen ein Hinweis ergänzt (u.a. in *tINS*: $N \rightarrow Nnew$).

Die Generierungsphase für den aktuellen Lokalisierungspfad ist abgeschlossen und der nächste wird analysiert ($i := i + 1$). Der PAP *UPDER* der Abbildung 6.23 endet mit der Ersetzung der Referenzparameter durch die temporären Strukturen. Diese werden anschließend im Programmablaufplan *Generierung von Werten* in Abbildung 6.22 als *Transformationsschritte* gespeichert und zur Anpassung der Instanzebene verwendet. Ein ausführliches Beispiel folgt im nächsten Abschnitt.

6.5. Anwendung der Transformationsschritte

In Abbildung 6.25 wird die zeitliche Reihenfolge der XML-Schemaevolution dargestellt. Diese beginnt mit der Änderung des konzeptuellen Modells EMX durch

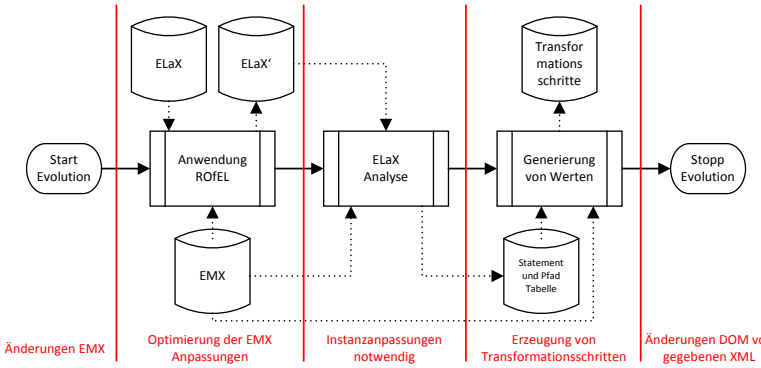


Abbildung 6.25.: Überblick der zeitlichen Reihenfolge der XML-Schemaevolution

Anwendung von ELaX-Operationen (*Änderungen EMX*), welche entsprechend geloggt werden. Das entstehende Log wird unter Verwendung des regelbasierten Algorithmus ROfEL optimiert (*Optimierung der EMX Anpassungen*). Anschließend kommt es zur Analyse der Auswirkungen auf die Instanzen (*Instanzanpassungen notwendig*) und zur *Erzeugung von Transformationsschritten*. Diese Transformationsschritte werden verwendet, um das *Document Object Model (DOM)* [HHW+04] eines XML-Dokuments gegebenenfalls so zu verändern, dass die Gültigkeit bezüglich des veränderten XML-Schemas wieder hergestellt wird.

Die in Abbildung 4.6 enthaltende *Drei-Ebenen-Architektur* kann im Kontext der vorliegenden Arbeit folgerichtig ebenso verfeinert werden. Abbildung 6.26 stellt das

Ergebnis dieser Verfeinerung dar. Es wurden die Ebenen-spezifischen Operationen

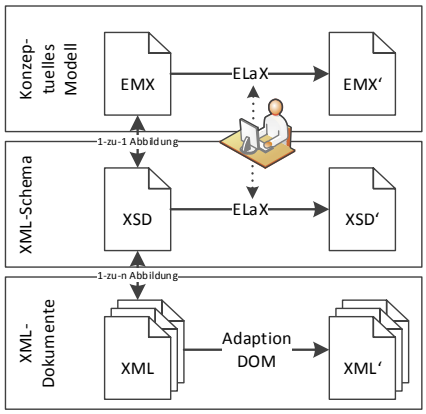


Abbildung 6.26.: Drei-Ebenen-Architektur aus 4.6 mit angepassten Operationen

(A, C und E) nebst Korrespondenzen (B und D) explizit benannt. Des Weiteren wurde ein Nutzer innerhalb der Modell- und Schemaebene ergänzt, um dessen primären Interaktionsort in der XML-Schemaevolution zu visualisieren.

Diese letzte Phase der Abbildung 6.25 wird nachfolgend mit Hilfe eines ausführlichen Beispiels beschrieben. In **Abschnitt 6.5.1** wird ein Beispielszenario für die XML-Schemaevolution eingeführt. Das dabei spezifizierte XML-Schema wird anschließend in **6.5.2** durch unterschiedliche Operationen angepasst. Es werden dabei ebenso die notwendigen Transformationsschritte vorgestellt. Abschließend wird in **Abschnitt 6.5.3** die Adaption der Instanzen beschrieben, sowie das angepasste Beispielszenario als Ergebnis der Evolution präsentiert.

6.5.1. Einführung eines Beispielszenarios

Das XML-Schema des XML-Beispiels 6.4 wird nachfolgend als Ausgangspunkt für eine XML-Schemaevolution verwendet. Es ist eine Erweiterung des im XML-Beispiel 6.2 dargestellten Schemas, welches bei der *Lokalisierung von Komponenten* in Abschnitt 6.3.1 eingeführt wurde.

Das XML-Schema ist im *Garden-of-Eden-Modellierungsstil* spezifiziert, sodass nur globale Element- (*root*, *e1*, *e2*, *e3* und *e4*) und Attributdeklarationen (*a1*, *a2* und *a3*), sowie komplexe Typdefinitionen (*roottype* und *ctype*) vorhanden sind. Diese Komponenten werden jeweils lokal referenziert. Des Weiteren wurde die Attributgruppe *ag1* zur Kapselung von Attributreferenzen der obigen Deklarationen spezifiziert, wobei zusätzlich eine Wildcard enthalten ist. Diese Attributwildcard enthält das einschränkende Attribut *notQName*, wodurch die Verwendung von *a3* als möglicher Kandidat verboten ist. Um diesen Mechanismus anwenden zu können, wurde ein *XML-Schema Version 1.1* verwendet. Dies ist im Markup der Schemakomponente (<*schema*>) durch das Attribut *vc:minVersion="1.1"* ersichtlich.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:vc="http://www.w3.org/2007/XMLSchema-versioning"
  vc:minVersion="1.1" id="EID1">
    <xs:element name="root" type="roottype" id="EID2"/>
    <xs:element name="e1" type="ctype" id="EID3"/>
    <xs:element name="e2" type="xs:string" id="EID4"/>
    <xs:element name="e3" type="xs:string" id="EID5"/>
    <xs:element name="e4" type="xs:string" id="EID6"/>
    <xs:attribute name="a1" type="xs:string" id="EID7"/>
    <xs:attribute name="a2" type="xs:string" id="EID8"/>
    <xs:attribute name="a3" type="xs:string" id="EID9"/>
    <xs:attributeGroup name="ag1" id="EID10">
        <xs:attribute ref="a1" use="required" id="EID11"/>
        <xs:attribute ref="a3" use="optional" id="EID12"/>
        <xs:anyAttribute notQName="a3" id="EID13"/>
    </xs:attributeGroup>
    <xs:complexType name="roottype" id="EID14">
        <xs:sequence minOccurs="1" maxOccurs="1" id="EID15">
            <xs:element ref="e1" minOccurs="1" maxOccurs="2" id="EID16"/>
            <xs:element ref="e2" minOccurs="1" maxOccurs="2" id="EID17"/>
        </xs:sequence>
        <xs:attributeGroup ref="ag1" id="EID18"/>
    </xs:complexType>
    <xs:complexType name="ctype" id="EID19">
        <xs:sequence minOccurs="2" maxOccurs="2" id="EID20">
            <xs:element ref="e3" minOccurs="1" maxOccurs="2" id="EID21"/>
        </xs:sequence>
        <xs:attributeGroup ref="ag1" id="EID22"/>
    </xs:complexType>
</xs:schema>
```

XML-Beispiel 6.4: XML-Schema des Beispielszenarios

Alle Komponenten besitzen eine eindeutige *EID*, welche aufgrund der verbesserten Nachvollziehbarkeit als XML-Schema-ID (*id*) explizit dargestellt ist. Die EID wurde für die Lesbarkeit wiederum stark vereinfacht und fortlaufend vergeben.

Konzeptuelles Modell

Das konzeptuelle Modell des XML-Schemas aus XML-Beispiel 6.4 ist in Abbildung 6.27 dargestellt. Es sind gemäß Abschnitt 4.1.2 nur bestimmte Komponenten enthalten (*visualisierte EMX-Knoten*), welche in 6.27 aufgrund der *Dokumentzentrierten Darstellungsweise* Elementreferenzen (*e1*, *e2* und *e3*), Attributgruppenreferenzen mit Wildcard (*ag1* mit *W*) und Inhaltsmodelle (*seq*) sind. Die Elementdeklaration *root* ist im Beispielszenario die Wurzel eines XML-Dokuments.

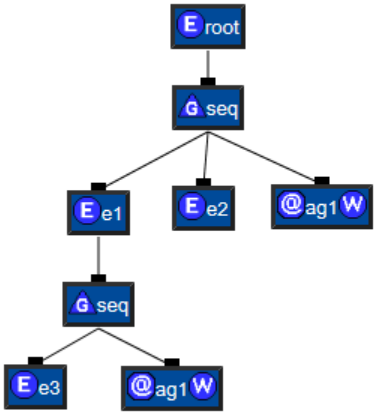


Abbildung 6.27.: Konzeptuelles Modell des XML-Schemas des XML-Beispiels 6.4

Gültige Instanzen des Ausgangsschemas

In den XML-Beispielen 6.5, 6.6 und 6.7 sind bezüglich des XML-Schemas des XML-Beispiels 6.4 gültige XML-Dokumente enthalten. Diese unterscheiden sich lediglich durch den Umfang der realisierten, optionalen Komponenten des Schemas. Alle Textinhalte sind gemäß der einfachen Datentypen gewählt und zur Unterscheidung aufsteigend vergeben. Somit können Änderungen leichter nachvollzogen werden.

```
<?xml version="1.0" encoding="UTF-8"?>
<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="beispielszenario.xsd" a1="a11">
    <e1 a1="a12">
        <e3>e31</e3>
        <e3>e32</e3>
    </e1>
    <e2>e21</e2>
</root>
```

XML-Beispiel 6.5: Gültiges XML-Dokument des XML-Schemas aus XML-Beispiel 6.4 (minimale Realisierung des Inhaltsmodells des Schemas)

Es wird ausgehend von dem Wurzelelement *root* in 6.5 eine minimale Realisierung des Inhaltsmodells präsentiert. Das heißt, dass nur die minimale Auftrittshäufigkeit von Elementen (*minOccurs*) und zwingende Attribute (*use = "required"*) enthalten sind. Des Weiteren wird die minimale Häufigkeit einer Gruppe verwendet.

Im Gegensatz dazu wird im XML-Dokument des XML-Beispiels 6.6 die maximale Realisierung dargestellt. Für jede Gruppe und Elementreferenz wird somit die maximale Häufigkeit (*maxOccurs*) verwendet. Alle nicht verbotenen Attribute (*use* $\neq$ *"prohibited"*) sind enthalten. Die Attributwildcard der Attributgruppe *ag1* (*EID = 13*), welche laut Standard kein *use-Attribut* besitzt und optional ist, wird durch das Attribut *a2* an den jeweils zulässigen Positionen realisiert.

```
<?xml version="1.0" encoding="UTF-8"?>
<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="beispielszenario.xsd"
    a1="a11" a2="a21" a3="a31" xsi:type="roottype">
    <e1 a1="a12" a2="a22" a3="a32" xsi:type="ctype">
        <e3>e31</e3>
        <e3>e32</e3>
        <e3>e33</e3>
        <e3>e34</e3>
    </e1>
    <e1 a1="a13" a2="a23" a3="a33" xsi:type="ctype">
        <e3>e35</e3>
        <e3>e36</e3>
        <e3>e37</e3>
        <e3>e38</e3>
    </e1>
    <e2>e21</e2>
    <e2>e22</e2>
</root>
```

XML-Beispiel 6.6: Gültiges XML-Dokument des XML-Schemas aus XML-Beispiel 6.4 (maximale Realisierung des Inhaltsmodells des Schemas)

Das XML-Dokument des XML-Beispiels 6.7 ist eine Mischung aus den obigen Dokumenten. Es stellt vom Umfang her eine durchschnittliche Realisierung dar.

```
<?xml version="1.0" encoding="UTF-8"?>
<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="beispielszenario.xsd" a1="a11">
    <e1 a1="a12" a2="a22" a3="a32" xsi:type="ctype">
        <e3>e31</e3>
        <e3>e32</e3>
        <e3>e33</e3>
    </e1>
    <e2>e21</e2>
</root>
```

XML-Beispiel 6.7: Gültiges XML-Dokument des XML-Schemas aus XML-Beispiel 6.4

Mit diesem XML-Dokument soll verdeutlicht werden, dass nicht vorausgesetzt werden kann, dass entweder eine minimale oder maximale Realisierung vorliegt. Das heißt, dass für die XML-Schemaevolution alle Möglichkeiten berücksichtigt werden müssen. Die durch die Optionalität von Strukturen auf Schemaebene bedingte Heterogenität der Instanzebene ist ein wesentlicher Grund für die hohe Komplexität und feingranulare Betrachtungsweise der XML-Schemaevolution.[33]

[33] siehe auch: These 13

6.5.2. Anpassung des Beispielszenarios

Das XML-Schema des XML-Beispiels 6.4 soll angepasst werden. Dafür sind unterschiedliche Operationen ausgewählt worden, die unter Beachtung der vorherigen Abschnitte des Kapitels einen umfangreichen Überblick bieten. Dazu zählen Einfüge-, Lösch- und Änderungsoperationen auf Elementen, Attributen, sowie einfachen und komplexen Typen. Das XML-Beispiel 6.8 visualisiert die Anpassungen.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:vc="http://www.w3.org/2007/XMLSchema-versioning"
  vc:minVersion="1.1" id="EID1" xmlns:cx="file://codex-null.xsd">
    <xs:import namespace="file://codex-null.xsd"
      schemaLocation="http://www.ls-dbis.de/codex" id="EID24"/>
    <xs:element name="root" type="roottype" id="EID2"/>
    <xs:element name="e1" type="ctypexctype" id="EID3"/>
    <xs:element name="e2" type="xs:string" id="EID4"/>
    <xs:element name="exe3" type="xs:string" id="EID5"/>
    <xs:element name="e4" type="xs:string" id="EID6"/>
    <xs:attribute name="a1" type="xs:string" fixed="fixed" id="EID7"/>
    <xs:attribute name="a2" type="cxxs:integer:string" id="EID8"/>
    <xs:attribute name="a3" type="xs:string" id="EID9"/>
    <xs:attributeGroup name="ag1" id="EID10">
        <xs:attribute ref="a1" use="required" id="EID11"/>
        <xs:attribute ref="a3" use="optional" id="EID12"/>
        <xs:anyAttribute notQName="a3" id="EID13"/>
    </xs:attributeGroup>
    <xs:complexType name="roottype" id="EID14">
        <xs:sequence minOccurs="1" maxOccurs="1" id="EID15">
            <xs:element ref="e2" minOccurs="21" maxOccurs="2" id="EID17"/>
            <xs:element ref="e1" minOccurs="1" maxOccurs="2" id="EID16"/>
            <xs:element ref="e4" minOccurs="1" id="EID23"/>
        </xs:sequence>
        <xs:attributeGroup ref="ag1" id="EID18"/>
    </xs:complexType>
    <xs:complexType name="ctypexctype" id="EID19">
        <xs:sequence minOccurs="2" maxOccurs="2" id="EID20">
            <xs:element ref="exe3" minOccurs="1" maxOccurs="12" id="EID21"/>
        </xs:sequence>
        <xs:attributeGroup ref="ag1" id="EID22"/>
    </xs:complexType>
</xs:schema>
```

XML-Beispiel 6.8: Anpassung des XML-Schemas aus XML-Beispiel 6.4

Im Vergleich zum Ausgangsschema des XML-Beispiels 6.4 wurden Strukturen farblich hervorgehoben. Dabei gilt, dass *rote* und *graue* Komponenten direkte Anpassungen sind, während *blaue* semi-/automatisch im konzeptuellen Modell vollzogen werden. Zur Vermeidung des umständlichen Vergleichs mit dem Ausgangsschema wurde der alte Wert bei Änderungen nicht entfernt, sondern durchgestrichen.

Das heißt, dass die Elementdeklaration mit der *EID = 5* explizit umbenannt wurde von *e3* zu *ex*. Eine Anpassung der Elementreferenz mit der *EID = 21* ist

dadurch nötig, die automatisch vollzogen wird und somit *blau* hervorgehoben ist.

Löschoperationen von Komponenten sind generell durchgestrichen, sodass zum Beispiel die Attributdeklaration mit der *EID = 9* gelöscht wurde. Als eine Konsequenz wird die Attributreferenz in der Attributgruppe *ag1* ebenso entfernt.

Die Elementreferenz mit der *EID = 16* ist *grau* hervorgehoben. Das heißt, dass diese Komponente umsortiert wurde, wodurch die Elementreihenfolge der umgebenen Sequenz (*EID = 15*) verändert wird. Auf eine irreführende Darstellung der alten Position innerhalb der Gruppe wurde allerdings an dieser Stelle verzichtet.

Änderungsoperationen

In Abbildung 6.28 ist das mittels *ROfEL* optimierte Log der Änderungsoperationen dargestellt. Es wurden demnach neun Operationen auf dem konzeptuellen Modell

time	EID	opType	msgType	content
1	-1	-1	2	24 entities successfully loaded.
2	7	2	0	update attribute name 'a1' change fixed 'fixed' ;
3	8	2	0	update attribute name 'a2' change type 'xs:integer' ;
4	9	1	0	delete attribute name 'a3' ;
5	12	1	1	delete attributeref at '12' ;
6	16	2	0	update elementref 'e1' at '16' change xpos '2' ypos '2' ;
7	17	2	0	update elementref 'e2' at '17' change minoccurs '2' ;
8	23	0	0	add elementref 'e4' minoccurs '1' id 'EID23' in '15' ;
9	21	2	0	update elementref 'e3' at '21' change maxoccurs '1' ;
10	19	2	0	update complextype name ctype change name 'ctypex' ;
11	5	2	0	update element name 'e3' change name 'ex' ;
12	-1	-1	2	24 entities transmitted.
13	24	0	1	add module from 'http://www.ls-dbis.de/codex' mode import with namespace 'file://codex-null.xsd' prefix 'cx' id 'EID24' ;
14	8	2	1	update attribute name 'a2' change type 'cx:integer' ;

Abbildung 6.28.: Optimiertes Log der Änderungsoperationen, angewendet auf das XML-Schema des XML-Beispiels 6.4 (Ergebnis XML-Beispiel 6.9)

angewendet, welche durch die *normalen ELaX-Statements* (*msgType = 0*) zu identifizieren sind. Zusätzlich sind *semi-/automatische* Operationen (*msgType = 1*), sowie *normale Aktionsmeldungen* (*msgType = 2*) enthalten. Letztere sind Statusmeldungen über das erfolgreiche Laden und Speichern des konzeptuellen Modells, welche im Kapitel 7 detaillierter erläutert werden.

Es wurden die folgenden im Allgemeinen *instanzverändernden* Operationen angewendet, die gemäß der Abbildung 6.2 Instanzkosten verursachen:

- Einführung eines *fixed-Wertes* in der Attributdeklaration *a1*
- Änderung des einfachen Typs des Attributs *a2* (restriktiverer Typ)

- Löschen der Attributdeklaration *a3* (mit kaskadierendem Löschen)
- Umsortieren des Inhaltsmodells durch Verschiebung der Elementreferenz *e1*
- Erhöhung der minimalen Häufigkeit der Elementreferenz *e2*
- Hinzufügen der zwingenden Elementreferenz *e2* in eine nicht-leere Sequenz
- Verringerung der maximalen Häufigkeit der Elementreferenz *e3*
- Umbenennung des komplexen Typs *ctype*
- Umbenennung der Elementdeklaration *e3*

Die noch verbleibenden Operationen (*msgType = 1*) stehen im direktem Zusammenhang mit den obigen expliziten Anpassungen. Dazu zählt unter anderem das *kaskadierende Löschen* der Attributreferenz mit der *EID = 12*, sowie die Einbindung des externen Moduls (*EID = 24*) zur Integration und anschließenden Verwendung der in Abschnitt 6.4.1 eingeführten *Nullwerte* im XML-Schema.

Erzeugung von Transformationsschritten

Nach der Analyse der Auswirkungen auf Instanzen werden die Transformationsschritte erzeugt und gespeichert, insofern Adaptionen der Instanzebene notwendig sind. Dieses Vorgehen wurde für Elementreferenzen in Abschnitt 6.4.4 exemplarisch erläutert, der zugrunde liegende Programmablaufplan (PAP) ist in Abbildung 6.22 dargestellt. Die Abbildung 6.29 beinhaltet die für das Beispielszenario notwendigen Transformationsschritte. Es wurden abweichend zur obigen Beschreibung aus

Time	EID-Ketten	POS	INS	CON	w	r
2	(2, 15), (2, 15, 16, 20)	**/root[1]/@a1**, /root[1]/e1[1]/@a1, /root[1]/e1[2]/@a1	a1='fixed'	[boolean(/root[1]/@a1), boolean(/root[1][@a1='fixed'])]	1	0
3	(2, 15), (2, 15, 16, 20)	**/root[1]/@a2**, /root[1]/e1[1]/@a2, /root[1]/e1[2]/@a2	a2=''	[boolean(/root[1]/@a2), matches(/root[1]/@a2, '^(\+|-)?([0-9])+')]	1	0
5	(2, 15), (2, 15, 16, 20)	**/root[1]/@a3**, /root[1]/e1[1]/@a3, /root[1]/e1[2]/@a3		boolean(/root[1]/@a3)	0	0
6	(2, 15, 16)	/root[1]/*	1 -> 2	[boolean(/root[1]/*), false()]	0	0
7	(2, 15, 17)	/root[1]/*	<e2></e2>	[boolean(/root[1]/*), false()]	0	0
8	(2, 15)	/root[1]	<e4></e4>	boolean(/root[1])	0	0
9	(2, 15, 16, 20, 21)	**/root[1]/e1[1]/***, /root[1]/e1[2]/*	""	[boolean(/root[1]/e1[1]/*), false()]	0	0
10	(2, 15, 16)	**/root[1]/e1[1]**, /root[1]/e1[2]	ctype -> ctypex	[boolean(/root[1]/e1[1]), boolean(not(/root[1]/e1[1]/@xsi:type) or /root[1]/e1[1][@xsi:type ='ctypex'])]	0	1
11	(2, 15, 16, 20, 21)	**/root[1]/e1[2]/e3[4]**, /root[1]/e1[2]/e3[3], /root[1]/e1[2]/e3[2], /root[1]/e1[2]/e3[1] /root[1]/e1[1]/e3[4], /root[1]/e1[1]/e3[3], /root[1]/e1[1]/e3[2], /root[1]/e1[1]/e3[1],	e3 -> ex	[boolean(/root[1]/e1[2]/e3[4]), false()]	0	1

Abbildung 6.29.: Transformationsschritte nach Anwendung PAP der Abbildung 6.22

Gründen der Übersichtlichkeit allerdings einige Anpassungen vorgenommen.[34]

Statt das ELaX-Statement zu speichern, wurde der Zeiteintrag (*Time*) aus dem Log der Abbildung 6.28 übernommen. Zusätzlich wurden die *EID-Ketten* gespeichert, welche als Grundlage zur Konstruktion der in *POS* enthaltenen XPath-Ausdrücke verwendet werden. Dieses wurde in Abschnitt 6.3.2 beschrieben.

[34]Die kompletten Transformationsschritte des Beispielszenarios sind in Abbildung A.56 enthalten.

In *POS* werden die Lokalisierungsinformationen der durch die Änderungsoperation betroffenen Komponenten aufgelistet, wobei diese durch die entsprechenden PAPs bereits verändert bzw. ergänzt sind.[35] Zeitgleich wurden dabei die zum Einfügen notwendigen Strukturen in *INS*, sowie entsprechende Bedingungen in *CON* gespeichert. Die Buchstaben *w* und *r* sind zusätzlich gesammelte Informationen, ob während der Lokalisierung Wildcards identifiziert wurden (wildFlag) bzw. ob das Markup von Komponenten verändert wurde (renameFlag).

In Abbildung 6.29 sind in *POS* einige XPath-Ausdrücke hervorgehoben. Diese wurden zur Erzeugung der Einträge in *CON* verwendet. Es gilt allerdings, dass pro Eintrag in *POS* mindestens einer in *INS* und *CON* ergänzt werden muss. Auf dieses Detail wurde aus Gründen der Übersichtlichkeit verzichtet. Die Abbildung 6.30 zeigt die kompletten Transformationsschritte für den Eintrag mit *Time = 2*.

Statement	POS	INS	CON	w	r
update attribute name 'a1' change fixed 'fixed' ;	/root[1]/@a1	a1='fixed'	[boolean(/root[1]/@a1), boolean(/root[1][@a1='fixed'])]	1	0
update attribute name 'a1' change fixed 'fixed' ;	/root[1]/e1[1]/@a1	a1='fixed'	[boolean(/root[1]/e1[1]/@a1), boolean(/root[1]/e1[1][@a1='fixed'])]	1	0
update attribute name 'a1' change fixed 'fixed' ;	/root[1]/e1[2]/@a1	a1='fixed'	[boolean(/root[1]/e1[2]/@a1), boolean(/root[1]/e1[2][@a1='fixed'])]	1	0

Abbildung 6.30.: Transformationsschritte für Eintrag *Time = 2* der Abbildung 6.29

6.5.3. Adaption der Instanzen des Beispielszenarios

Die erzeugten Transformationsschritte der Abbildung 6.29 werden verwendet, um das *Document Object Model* (*DOM*) von gegebenen XML-Dokumenten zu verändern. Für die strukturierte Auswertung der Transformationen wurde der Programmablaufplan (PAP) der Abbildung 6.31 spezifiziert. Dieser illustriert die Zusam-

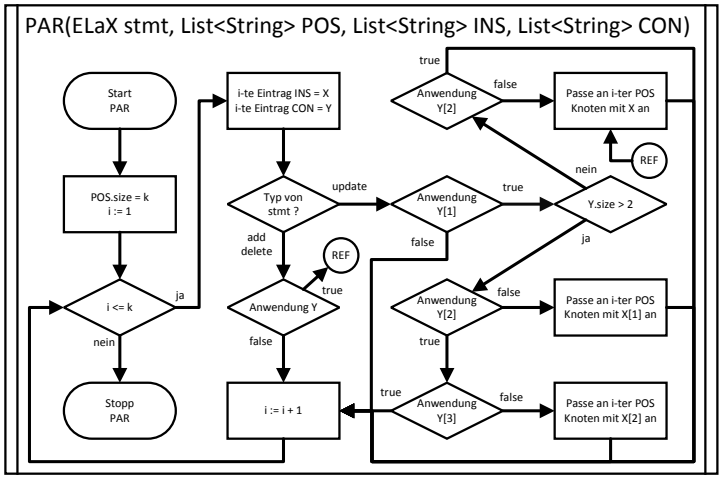


Abbildung 6.31.: PAP - PAR aus [Nös15c]

[35] Jeweils (Time; PAP): (2, 3; A.32) | (5; A.31) | (6, 7, 9; 6.23) | (8; A.30) | (10; A.44) | (11; A.45)

menhänge zwischen den Lokalisierungspfaden (*POS*), den Bedingungen (*CON*), sowie weitergehenden Informationen über die Instanzanpassungen (*INS*).

In Abhängigkeit der Operation (*add*, *delete* und *update*) werden alle betroffenen, absolut adressierten Komponenten schrittweise analysiert. Wird eine Komponente hinzugefügt oder gelöscht, dann ist in *CON* jeweils nur eine *Existenzbedingung* vorhanden (z.B. *boolean(/root[1])*). Ist diese erfüllt (*Anwendung Y*), wird entweder eine Komponente eingefügt (z.B. *<e4></e4>*) oder entfernt (*INS* ist leer).

Änderungsoperationen sind komplexer zu analysieren, wie in den vorherigen Abschnitten bereits erläutert wurde.[36] Liegt eine solche Operation vor, dann wird ebenfalls eine *Existenzprüfung* vollzogen (*Anwendung Y[1]*). Insofern diese Prüfung erfolgreich war, wird im Anschluss im Allgemeinen eine *Matchbedingung* ausgewertet (*Anwendung Y[2]*). In dieser wird eine Komponente bezüglich gegebener Einschränkungen untersucht (z.B. *boolean(/root[1][@a1='fixed']*).

Sind diese Einschränkungen nicht erfüllt, dann werden die Informationen aus *INS* angewendet. Zum Beispiel wird das Attribut *a1* durch die Operation mit der *Time = 2* ersetzt (*a1 = "fixed"*), insofern im untersuchten XML-Dokument das an der *POS* befindliche, zwingende Attribut nicht den Wert *"fixed"* hat.

Die dritte Überprüfung (*Anwendung Y[3]*) ist im Allgemeinen notwendig, falls Attributreferenzen bzw. Attributdeklarationen zusätzlich umbenannt werden. Dieser Test wird allerdings nur dann vollzogen, falls der vorherige positiv abgeschlossen ist (*Y[2] = true*). Ist dies nicht der Fall, dann wurde die Umbenennung durch *X[1]* bereits vollzogen. Dies gilt zum Beispiel, falls ein Attribut mit *fixed-Wert* ersetzt wurde, da *GEN* in diesem Fall bereits den neuen Namen verwendet hat.

Transformationen in DOM

Das *Document Object Model (DOM)* [HHW+04] ist eine hierarchische Datenstruktur, in der alle Komponenten Knoten mit Eigenschaften und gegebenenfalls mit Subelementen sind. Diese Knoten können in Übereinstimmung mit dem *Datenmodell von XML-Dokumenten* [BFM+10] unter anderem Dokumente, Elemente, Attribute, Kommentare, Prozessanweisungen etc. sein.

DOM ermöglicht die Adressierung, Änderung und Verschiebung von Knoten (*Nodes*), wobei unterschiedliche Schnittstellen zur vereinfachten Bearbeitung spezifiziert sind. Zum Beispiel können Knoten in der sortierten Dokumentordnung in einer *NodeList* enthalten sein, welche schrittweise ausgewertet werden kann. Somit ist es unter anderem möglich, das Inhaltsmodell einer Gruppe entsprechend zu analysieren (z.B. deren Kinderelemente). Eine *NamedNodeMap* bietet sich hingegen an, um eine unsortierte Menge von Knoten zu speichern (z.B. Attribute).

Die in *POS* gespeicherten Ausdrücke können verwendet werden, um entsprechende Knoten mittels XPath zu adressieren [Whi04]. Dabei ist es über konfigurierbare Schnittstellen möglich, unterschiedliche Rückgabewerte zu erhalten. Dazu zählen

[36] Eine Übersicht über die Transformationsschritte der Update-Operationen ist in A.58 und A.59 gegeben.

neben dem einfachen *Node* und der entsprechenden Menge bzw. Liste auch Wahrheitswerte (*boolean*), Nummern (*number*) oder Zeichenketten (*string*).

Wird nun die *Existenz* einer Komponente im XML-Dokument bzw. dessen DOM untersucht, dann kann dies durch die Anwendung von *CON* (*Anwendung Y* bzw. *Y[1]*) vollzogen werden. Ist dieser Test positiv und der boolesche Rückgabewert der *Existenzbedingung* entsprechend *true*, dann muss in Abhängigkeit des *Statements* im Allgemeinen eine *Matchbedingung* ausgewertet werden. Ist gemäß Abbildung 6.31 diese Bedingung nicht erfüllt, das heißt der boolesche Rückgabewert ist *false*, dann muss das analysierte XML-Dokument angepasst werden.

Die Abbildung 6.32 enthält für einen Ausschnitt der Übersicht der Transformationsschritte der Abbildung 6.29 die Rückgabewerte.[37] Die Spalten *min*, *max* und

Time	POS	INS	CON	Rückgabewert CON Existenz- / Matchbedingung	min XML	max XML	avg XML	Return Type
2	/root[1]/@a1	a1='fixed'	[boolean(/root[1]/@a1), boolean(/root[1][@a1='fixed'])]		true / false	true / false	true / false	Node
3	/root[1]/@a2	a2=''	[boolean(/root[1]/@a2), matches(/root[1]/@a2, '^(\+|-)?([0-9])+')]		false	true / false	false	Node
5	/root[1]/@a3		boolean(/root[1]/@a3)		false	true	false	Node
6	/root[1]/*	1 -> 2	[boolean(/root[1]/*), false()]		true / false	true / false	true / false	NodeList
7	/root[1]/*	<e2></e2>	[boolean(/root[1]/*), false()]		true / false	true / false	true / false	NodeList
8	/root[1]	<e4></e4>	boolean(/root[1])		true	true	true	NodeList
9	/root[1]/e1[1]/*	""	[boolean(/root[1]/e1[1]/*), false()]		true / false	true / false	true / false	NodeList
10	/root[1]/e1[1]	ctype -> ctypex	[boolean(/root[1]/e1[1]), boolean(not(/root[1]/e1[1]/@xsi:type) or /root[1]/e1[1][@xsi:type ='ctypex'])]		true / true	true / false	true / false	Node
11	/root[1]/e1[2]/e3[4]	e3 -> ex	[boolean(/root[1]/e1[2]/e3[4]), false()]		false	true / false	false	Node

Abbildung 6.32.: Ausschnitt der Rückgabewerte von CON der Abbildung 6.29

avg XML bezeichnen die XML-Beispiele zur minimalen (6.5), maximalen (6.6) und durchschnittlichen Realisierung (6.7) des Inhaltsmodells des XML-Beispiels 6.4. Ist der Rückgabewert der Existenzbedingung *true* (links vom Slash bzw. alleiniger Wert) und der von der Matchbedingung *false* (rechts vom Slash), dann wird der XPath-Ausdruck der *POS* verwendet, um den angegebenen *Return Type* zu ermitteln. Dies ist nur dann notwendig, falls der Eintrag nicht unterstrichen ist.

Das Attribut *a2* im Eintrag mit der *Time = 3* existiert zum Beispiel nicht in der minimalen Realisierung (*min XML*), sodass die Existenzbedingung *false* ist. Somit muss kein Knoten (*Node*) im XML-Dokument ermittelt werden. In der maximalen Realisierung (*max XML*) ist dieses Attribut allerdings enthalten, sodass nach der negativen Matchbedingung (insgesamt *true / false*) der entsprechende Knoten ermittelt werden muss. Im Eintrag mit der *Time = 8* muss in allen Realisierungen eine *NodeList* ermittelt werden, da die Existenzbedingung *true* ist. Dies ist das Inhaltsmodell der Gruppe, in welches eine Elementreferenz eingefügt werden soll.

Es wird nachfolgend jeweils die maximale Realisierung (*max XML*) betrachtet, da in dieser alle Komponenten enthalten sind. Die Transformationsschritte werden dabei in der zeitlichen Reihenfolge der Spalte *Time* des Logs der Abbildung 6.28 abgearbeitet, auch wenn dies durch die nachfolgende, vermischte Vorstellung der einzelnen Operationen irrtümlicherweise angenommen werden könnte.[38]

[37] Eine komplette Übersicht der Rückgabewerte des Beispielszenarios ist in Abbildung A.57 enthalten.
[38] Die Übersicht der Abbildung A.57 kann somit folgerichtig von oben nach unten abgearbeitet werden.

Ändern von Komponenten in DOM

Die Änderung von Komponenten betrifft die Einträge mit der *Time = 2*, *Time = 3* und *Time = 10*. Die Umsortierung und Umbenennung als Spezialfälle der Änderung werden nachfolgend einzeln thematisiert.

Der Eintrag mit der *Time = 2* führt einen *fixed-Wert* bei einer Attributdeklaration ein.[39] Die Existenzbedingung ist *true*, sodass anschließend die Matchbedingung geprüft wird. Diese ist *false*, da *a1* nicht den notwendigen Wert *fixed* hat (*a1 = "a11"*). Somit wird unter Nutzung von *POS* der entsprechende Knoten (*Node*) im DOM ermittelt. Der Wert des Attributknotens wird anschließend mit einer Anweisung entsprechend geändert (*node.setNodeValue("fixed")*).

Im zweiten Eintrag mit der *Time = 3* wurde der Typ einer Attributdeklaration geändert.[40] Da nach erfolgreicher Existenzprüfung die Matchbedingung *false* ist, muss wiederum der Wert des Attributknotens verändert werden. In diesem Fall kommt es allerdings zu einer semiautomatischen ELaX-Operation, es wird die *Nullwertfähigkeit* durch das Einbinden des externen Moduls `http://www.ls-dbis.de/codex` ermöglicht (Eintrag mit *Time = 13* in der Abbildung 6.28). Mit der anschließenden Änderung des Datentyps von *xs:integer* in *cx:integer* (Eintrag mit *Time = 14* in Abbildung 6.28) ist es möglich einen neutralen, nicht semantisch vorbelasteten Wert zu wählen.[41] Daher kann nach Ermittlung des entsprechenden Attributknotens im DOM jenem die leere Zeichenkette zugewiesen werden, was ohne die obige Erweiterung nicht schemakonform gewesen wäre.

Der Eintrag mit der *Time = 10* ist zu den obigen beiden anlog zu behandeln.[42] Es handelt sich zwar um eine Umbenennung (*renameFlag = 1*), sodass in *INS* eine andere Syntax verwendet wird, allerdings ist dies auf der Instanzebene wiederum eine Wertzuweisung eines Attributknotens (*node.setAttribute("xsi:type", "ctypex")*).

Löschen von Komponenten in DOM

Das Löschen von Komponenten im DOM wird durch den Eintrag mit der *Time = 5* repräsentiert.[43] Diese Operation wurde automatisch durch das Entfernen der Attributdeklaration *a3* vollzogen. Beim direkten Löschen wird keine Matchbedingung erzeugt, da die Existenzbedingung ausreichend ist. Wird nun die Komponente erfolgreich lokalisiert, wird der entsprechende Attributknoten ausgewählt (z.B. *aNode*). Anschließend muss abweichend vom Ändern der umgebene Elementknoten (z.B. *eNode*) ermittelt werden (*/root[1]*), bei dem das Attribut entfernt wird (*eNode.removeAttributeNode(aNode)*). Eine direkte Entfernung von Attributknoten ohne deren Elementknoten ist in den Schnittstellen von DOM nicht vorgesehen.

[39] Eintrag mit Time = 2 : update attribute name 'a1' change fixed 'fixed' ;
[40] Eintrag mit Time = 3 : update attribute name 'a2' change type 'xs:integer' ;
[41] siehe auch: Kapitel 6.4.1 (Einfacher Inhalt)
[42] Eintrag mit Time = 10 : update complextype name ctype change name 'ctypex' ;
[43] Eintrag mit Time = 5 : delete attributeref at '12' ;

Das Verringern der maximalen Häufigkeit des Eintrags mit der *Time = 9* führt ebenso zum Löschen von Komponenten.[44] Im Vergleich zu einer direkten Löschoperation ist allerdings in *INS* der Eintrag *""* enthalten. Des Weiteren wird hier nicht nur die Existenzbedingung geprüft, sondern aufgrund der Einheitlichkeit der Änderungsoperationen wiederum eine Matchbedingung. Der Rückgabewert von dieser ist generell *false*, weil die Funktion *false()* verwendet wird.

Analog zum Löschen muss der umgebene Elementknoten mit dessen Inhalt ermittelt (*/root[1]/e1[1]*) und in einer *NodeList* gespeichert werden. Da diese Datenstruktur alle Kinderelemente in Dokumentordnung enthält, kann sequentiell geprüft werden, wie viele Elemente von *e3* pro Gruppendurchlauf enthalten sind. Übersteigt die ermittelte Anzahl die maximale Häufigkeit (*maxOccurs*), dann müssen alle nachfolgende, überflüssigen Elemente (z.B. *eNode*) gespeichert und anschließend entfernt werden (*eNode.getParentNode().removeChild(eNode)*). Wie bei Attributknoten ist das direkte Entfernen ohne Elternknoten ebenso nicht möglich.

Hinzufügen von Komponenten in DOM

Das Hinzufügen mit Hilfe der *add-Operation* wird durch den Eintrag mit der *Time = 8* realisiert.[45] Es wird nur eine Existenzbedingung des Elternelements benötigt, eine Matchbedingung wird nicht erzeugt. Dennoch ist in *INS* der notwendige Knoten mit dessen Inhalt enthalten, welcher eingefügt werden soll. Ist die Existenz *true*, dann wird das Inhaltsmodell des Elternelements in eine *NodeList* geladen. Da es nur einen Gruppendurchlauf gibt (*maxOccurs = 1*) und der neue Knoten (*e4*) am Ende angehängt wird, kann dies mit einer direkten Anweisung beim ersten Knoten (z.B. *eNode*) der *NodeList* umgesetzt werden (*eNode.getParentNode().appendChild(e4)*). Dieses Vorgehen muss entsprechend der minimalen Häufigkeit des neuen Knotens (*minOccurs = 1*) wiederholt werden.

Sind mehrere Gruppendurchläufe möglich (*maxOccurs > 1*), dann muss jeweils das entsprechende Nachfolgeelement ermittelt werden (z.B. *nNode*). Dies ist durch die sequentielle Abarbeitung der *NodeList* möglich, welche in Dokumentordnung vorliegt. Anschließend kann der neue Knoten (*e4*) gemäß der minimalen Häufigkeit eingefügt werden (*nNode.getParentNode().insertBefore(e4, nNode)*). Im letzten Durchlauf muss dann wiederum obiges Vorgehen angewendet werden, insofern der neue Knoten am Ende eingefügt wird (*eNode.getParentNode().appendChild(e4)*).

Das Erhöhen der minimalen Häufigkeit kann ebenso zum Einfügen von neuen Knoten führen, dies wird durch den Eintrag mit der *Time = 7* gezeigt.[46] Da es sich um eine *update-Operation* handelt, wird aufgrund der Einheitlichkeit der Änderungsoperationen wiederum die generell *false* erzeugende Matchbedingung *false()* verwendet. Analog zur *add-Operation* wird das Inhaltsmodell des Elternknotens

[44] Eintrag mit Time = 9 : update elementref 'e3' at '21' change maxoccurs '1' ;
[45] Eintrag mit Time = 8 : add elementref 'e4' minoccurs '1' id 'EID23' in '15' ;
[46] Eintrag mit Time = 7 : update elementref 'e2' at '17' change minoccurs '2' ;

(*/root[1]*) in einer *NodeList* gespeichert. Diese Liste wird durchlaufen, wobei die Elementknoten (z.B. *eNode*) mit dem gleichen Namen (*eNode.getNodeName()*) wie der geänderte Knoten (*e2*) gezählt werden. Sind weniger Elemente als *minOccurs* enthalten, das heißt ein Nachfolgeelement (z.B. *nNode*) tritt vor der minimalen Anzahl auf, dann muss der Inhalt von *INS* eingefügt werden. Das Einfügen ist wiederum abhängig von der Positionierung innerhalb des Elternelements und dem aktuellen Gruppendurchlauf (analog zum Eintrag mit *Time = 8*).[47]

Umsortieren von Komponenten in DOM

Das Umsortieren von Komponenten ist ein Spezialfall der Änderung. Der Eintrag mit der *Time = 6* verschiebt den Knoten *e1* von der ersten, auf die zweite Position (*INS*).[48] Es wird die *NodeList* des Elternelements (*/root[1]*) benötigt, welche sequentiell abgearbeitet wird. Der zu verschiebende Elementknoten (*e1*) wird durch seinen Namen identifiziert (*e1.getNodeName()*) und aus der aktuellen *NodeList* entfernt (*node.getParentNode().removeChild(e1)*). Die entfernten Knoten müssen dabei in einer Zusatzstruktur in Dokumentordnung zwischengespeichert werden.

Wird nun die neue Position in der *NodeList* erreicht, dass heißt das erste Element nach jener (z.B. *nNode*), dann müssen die vorher entfernten Knoten eingefügt werden. Alle in der Zusatzstruktur befindlichen Elementknoten werden aus dieser gelöscht und in der *NodeList* hinzugefügt (*nNode.getParentNode().insertBefore(e1, nNode)*). Im nächsten Gruppendurchlauf wird dieses Vorgehen wiederholt. Analog zu den bisherigen Operationen müssen gegebenenfalls die Elementknoten der Zusatzstruktur angehängt werden, falls die neue Position von *e1* die letzte im Inhaltsmodell des Elternknotens ist (*node.getParentNode().appendChild(e1)*).

Umbenennung von Komponenten in DOM

Die Umbenennung ist eine Änderungsoperation, die im Eintrag mit der *Time = 11* dargestellt ist.[49] Da es mit den Schnittstellen von DOM nicht möglich ist, einen Knotennamen von *e3* direkt zu ändern, muss jeweils ein neuer Knoten erschaffen werden (z.B. *ex*). Dieser bekommt sowohl den neuen Namen (*createElement("ex")*), als auch alle Kinderknoten von e3 (*e3.getChildNodes()* und *ex.appendChild()*) und Attribute (*e3.getAttributes()* und *ex.setAttributeNode()*). Anschließend wird der alte Knoten durch den neuen ersetzt (*e3.getParentNode().replaceChild(ex, e3)*).

Die Umbenennung von Elementknoten sollte im vorliegenden Ansatz der XML-Schemaevolution generell als letzte Operation vollzogen werden, da ansonsten sämtliche nachfolgenden XPath-Ausdrücke zusätzlich angepasst werden müssten. Der dadurch entstehende Mehraufwand ist durch eine einfache Sortierung der Operationen zu vermeiden. Des Weiteren sollten die Umbenennungen, insofern mehr als

[47] *eNode.getParentNode().appendChild(e2)* bzw. *nNode.getParentNode().insertBefore(e2, nNode)*
[48] Eintrag mit Time = 6 : update elementref 'e1' at '16' change xpos '2' ypos '2' ;
[49] Eintrag mit Time = 11 : update element name 'e3' change name 'ex' ;

eine Komponente betroffen ist, gemäß deren *POS* sortiert werden. Das heißt, dass zuerst solche Operationen vollzogen werden, welche in der Hierarchie des XML-Dokuments am weitesten von der Wurzel entfernt sind. Somit kann gewährleistet werden, dass sämtliche Umbenennungen im XML-Dokument angewendet werden. Die Sortierung ist unter anderem in den Übersichten der Transformationsschritte der Abbildungen A.56 und A.57 daran zu erkennen, dass der Lokalisierungspfad */root[1]/e1[2]/e3[4]* vor */root[1]/e1[2]/e3[3]* behandelt wird.

Ergebnis der Evolution

Nach der Anpassung des Beispielszenarios durch die Anwendung der in Abbildung 6.28 dargestellten ELaX-Operationen ist sowohl das Ausgangsschema (XML-Schema des XML-Beispiels 6.4) als auch dessen konzeptuelles Modell (Abbildung 6.27) verändert. Das angepasste XML-Schema ist im XML-Beispiel 6.9 dargestellt.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:vc="http://www.w3.org/2007/XMLSchema-versioning"
  vc:minVersion="1.1" id="EID1" xmlns:cx="file://codex-null.xsd">
    <xs:import namespace="file://codex-null.xsd"
      schemaLocation="http://www.ls-dbis.de/codex" id="EID24"/>
    <xs:element name="root" type="roottype" id="EID2"/>
    <xs:element name="e1" type="ctypex" id="EID3"/>
    <xs:element name="e2" type="xs:string" id="EID4"/>
    <xs:element name="ex" type="xs:string" id="EID5"/>
    <xs:element name="e4" type="xs:string" id="EID6"/>
    <xs:attribute name="a1" type="xs:string" fixed="fixed" id="EID7"/>
    <xs:attribute name="a2" type="cx:integer" id="EID8"/>
    <xs:attributeGroup name="ag1" id="EID10">
        <xs:attribute ref="a1" use="required" id="EID11"/>
        <xs:anyAttribute notQName="a3" id="EID13"/>
    </xs:attributeGroup>
    <xs:complexType name="roottype" id="EID14">
        <xs:sequence minOccurs="1" maxOccurs="1" id="EID15">
            <xs:element ref="e2" minOccurs="2" maxOccurs="2" id="EID17"/>
            <xs:element ref="e1" minOccurs="1" maxOccurs="2" id="EID16"/>
            <xs:element ref="e4" minOccurs="1" id="EID23"/>
        </xs:sequence>
        <xs:attributeGroup ref="ag1" id="EID18"/>
    </xs:complexType>
    <xs:complexType name="ctypex" id="EID19">
        <xs:sequence minOccurs="2" maxOccurs="2" id="EID20">
            <xs:element ref="ex" minOccurs="1" maxOccurs="1" id="EID21"/>
        </xs:sequence>
        <xs:attributeGroup ref="ag1" id="EID22"/>
    </xs:complexType>
</xs:schema>
```

XML-Beispiel 6.9: Angepasstes XML-Schema nach Änderungsoperationen

Aus diesem Schema sind gegenüber dem XML-Schema des XML-Beispiels 6.8 die unterschiedlichen, farblich hervorgehobenen Strukturen entfernt worden.

Das angepasste, konzeptuelle Modell ist in der Abbildung 6.33 dargestellt. Dieses

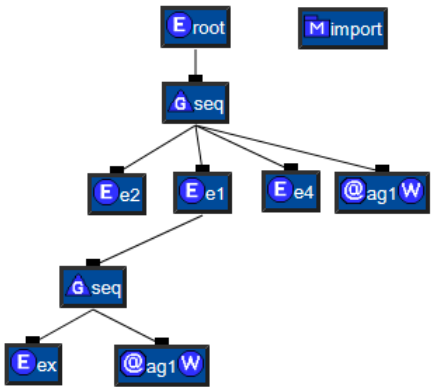


Abbildung 6.33.: Angepasstes, konzeptuelles Modell nach Änderungsoperationen

EMX ist im Vergleich zum vorherigen Modell der Abbildung 6.27 an vier Positionen in der grafischen Repräsentation verändert worden.

Dazu zählt unter anderem die umbenannte Elementdeklaration *ex* (vormals *e3*). Des Weiteren wurde die Elementreferenz *e1* an die zweite Position verschoben, sodass *e1* und *e2* im konzeptuellen Modell vertauscht sind. Die Elementreferenz *e4* wurde ebenso eingeführt. Die letzte Änderung ist die Konsequenz aus der Einbindung des externen Moduls zur Realisierung der Nullwertfähigkeit des XML-Schemas. Es wurde diesbezüglich das dargestellte Modul importiert.

Durch die Anpassung des Beispielszenarios wurden die obigen Transformationsschritte erzeugt. Als Ergebnis der Anwendung der Transformationen entstehen die in den XML-Beispielen 6.10, 6.11 und 6.12 dargestellten XML-Dokumente.

```
<?xml version="1.0" encoding="UTF-8"?>
<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="beispielszenario.xsd" a1="fixed">
    <e2>e21</e2>
    <e2></e2>
    <e1 a1="fixed">
        <e3>e31</e3>
        <e3>e32</e3>
    </e1>
    <e4></e4>
</root>
```

XML-Beispiel 6.10: Gültiges XML-Dokument des XML-Schemas aus XML-Beispiel 6.9 (ausgehend vom minimalen XML-Dokument aus XML-Beispiel 6.5)

```
<?xml version="1.0" encoding="UTF-8"?>
<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="beispielszenario.xsd"
    a1="fixed" a2="" xsi:type="roottype">
    <e2>e21</e2>
    <e2>e22</e2>
    <e1 a1="fixed" a2="" xsi:type="ctypex">
        <e3>e31</e3>
        <e3>e32</e3>
    </e1>
    <e1 a1="fixed" a2="" xsi:type="ctypex">
        <e3>e35</e3>
        <e3>e36</e3>
    </e1>
    <e4></e4>
</root>
```

XML-Beispiel 6.11: Gültiges XML-Dokument des XML-Schemas aus XML-Beispiel 6.9 (ausgehend vom maximalen XML-Dokument aus XML-Beispiel 6.6)

```
<?xml version="1.0" encoding="UTF-8"?>
<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="beispielszenario.xsd" a1="fixed">
    <e2>e21</e2>
    <e2></e2>
    <e1 a1="fixed" a2="" xsi:type="ctypex">
        <e3>e31</e3>
        <e3>e32</e3>
    </e1>
    <e4></e4>
</root>
```

XML-Beispiel 6.12: Gültiges XML-Dokument des XML-Schemas aus XML-Beispiel 6.9 (ausgehend von XML-Dokument aus XML-Beispiel 6.7)

Diese Instanzen wurden aus den XML-Beispielen des Abschnitts 6.5.1 erzeugt. Sie sind gültig bezüglich des angepassten XML-Schemas des XML-Beispiels 6.9.

Die in Kapitel 1.1 definierte Fragestellung, ”ob durch eine Charakterisierung und Erfassung (d.h. Bestimmung) der durchgeführten Änderungen am XML-Schema, die zur Wiederherstellung der Gültigkeit der XML-Dokumente notwendigen Adaptionen automatisch hergeleitet werden können”, kann daher positiv beantwortet werden. Es existiert somit die Möglichkeit, durch die Änderung eines XML-Schemas die gegebenen, assoziierten XML-Dokumente automatisiert anzupassen.

Die Gültigkeit der transformierten Instanzen bezüglich eines veränderten XML-Schemas kann durch die Anwendung des vorgestellten Ansatzes der XML-Schemaevolution weitestgehend automatisiert gewährleistet werden.[50]

[50] siehe auch: These 14

Abschließende Betrachtung

In diesem Kapitel wurde eine automatisierte Erzeugung von Transformationsschritten zur Wahrung und/oder Wiederherstellung der Gültigkeit einer Datenbasis erläutert. Es wurden die ELaX-Operationen des vorherigen Kapitels klassifiziert, bevor deren Auswirkungen auf die Instanzen analysiert wurden. Die Lokalisierung von Komponenten und die Generierung von Informationen wurde ebenso thematisiert. Zum Abschluss wurde die Anwendung der Transformationsschritte anhand eines ausführlichen Beispiels beschrieben.

Im nächsten Kapitel wird die letzte Zielsetzung der vorliegenden Arbeit behandelt. Es wird die Unterstützung von Nicht-Experten bei der hochkomplexen, fehleranfälligen Evolution durch ein geeignetes Tool und sinnvolle Abstraktionen thematisiert. Zur Veranschaulichung wird ein Prototyp beschrieben, mit welchem die unterschiedlichen Aspekte und Themen der vorherigen Kapitel weitestgehend realisiert beziehungsweise umgesetzt werden.

7. Prototypische Umsetzung

In [Ste06] wurde der Forschungsprototyp **CoDEX** (Conceptual Design and Evolution of XML schemas) vorgestellt, welcher als Desktop-Applikation unter Verwendung der integrierten Entwicklungsumgebung *Eclipse Europa* und *RCP* (Rich Client Plattform) [ML05] implementiert wurde. Dieser Prototyp ist aufgrund der verschiedenen Plugins und deren versionsbedingter Abhängigkeiten nur noch bedingt lauffähig und wartbar, sodass ein komplett neuer Prototyp entwickelt wurde.

CodeX (Conceptual design and evolution of XML schemas)[1] ist in der aktuellen Umsetzung als Webapplikation mit **GWT** (Google Web Toolkit) [HT07] implementiert und wird nachfolgend beschrieben.

Die allgemeine Architektur wird in **Abschnitt 7.1** vorgestellt. Dabei wird auf die Implementierung eingegangen, sowie ein Überblick des realisierten Umfangs unter Beachtung der vorherigen Kapitel gegeben. In **Abschnitt 7.2** werden die grafische Oberfläche und der **EMX-Editor** beschrieben, bevor die Umsetzung des konzeptuellen Modells innerhalb des Editors dargestellt wird. Anschließend wird in einem ausführlichen Beispiel die XML-Schemaevolution mit Hilfe von *CodeX* erläutert. Zum Abschluss des Kapitels wird auf Erweiterungen hingewiesen, welche in studentischen Arbeiten konzipiert und gegebenenfalls ergänzt wurden.

7.1. Architektur des Prototypen

Die allgemeine Architektur ist in Abbildung 7.1 dargestellt. Zu den Komponenten gehört die grafische Benutzeroberfläche (*GUI*), über welche ein Anwender *Schemaänderungen* und *Korrekturen* in der XML-Schemaevolution durchführen kann. Des Weiteren können über die *Import-* und *Export-Komponenten* Daten in Form von XML-Dokumenten bereitgestellt bzw. als *Ergebnis* entnommen werden. Eine Konfiguration von nutzerabhängigen Einstellungen (*Config*) ist ebenso möglich.

Die *Schemaänderungen* werden grafisch auf einer *Visualisierung* des konzeptuellen Modells (*EMX* - Entity Model for XML-Schema) durchgeführt und in der *Evolutionsengine* analysiert. Dabei werden Informationen der *Wissensbasis* angewendet, in welcher unter anderem die Korrespondenzen (*Modell Mapping*) und *Operationsspezifikationen* (*ELaX* - Evolution Language for XML-Schema) enthalten sind. Die in der *Evolutionsengine* angewandten Operationen werden im *Log*

[1] Das Akronym wurde beibehalten, lediglich die Schreibweise der Majuskeln D und E wurde angepasst. Zur Verbesserung der Unterscheidbarkeit wird gegebenenfalls das Synonym *CodeX 2* verwendet.

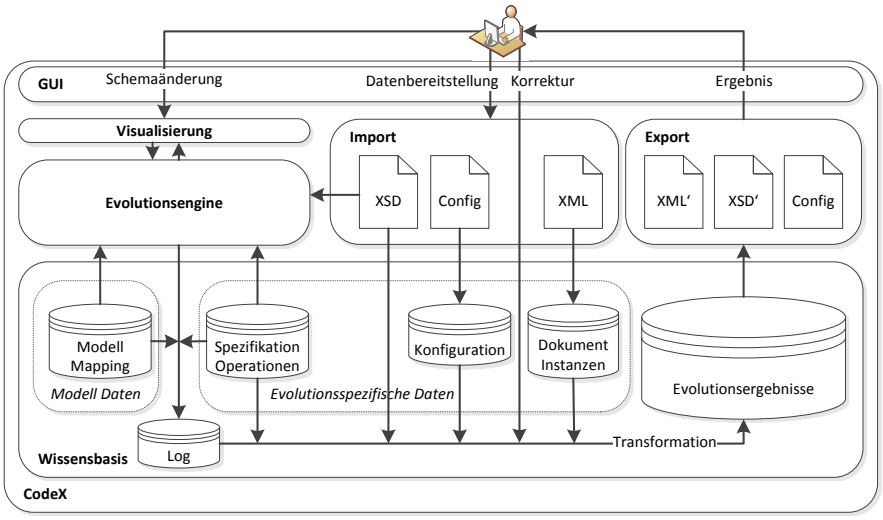


Abbildung 7.1.: Komponentenmodell der prototypischen Umsetzung

gespeichert und anschließend mittels der *Evolutionsspezifischen Daten* in Evolutionsergebnisse transformiert. Diese können korrigiert und exportiert werden.

7.1.1. Details der Implementierung

CodeX 2 wurde unter Anwendung des *GWT* (Google Web Toolkit) 2.5.1 in Java 1.7.0.21 vollständig neu implementiert, um das zeitintensive Refactoring der *CoDEX-RCP-Anwendung* (Rich Client Plattform) zu vermeiden. Das Ziel war es, eine Webapplikation zu entwickeln, welche die XML-Schemaevolution unterstützt und unkomplizierter als Software bereitgestellt werden kann.

Der Prototyp wird als *WAR*-File[2] (Web Application Archive) für Forschungszwecke und eine nicht-kommerzielle Nutzung bereitgestellt, und kann innerhalb eines Webservers als lokale oder serverseitige Anwendung betrieben werden.[3] Es wird dafür eine relationale MySQL-Datenbank benötigt, sowie ein Browser zur Darstellung und Erfassung der Ein- und Ausgaben der Applikation.

CodeX ist mit Hilfe der *IDE* (Integrated Development Environment) *Eclipse Kepler* entwickelt worden, in welchem ein *Apache-Ant-Skript* zur automatisierten Erzeugung des obigen WAR-Files angewendet wird. Eine Übersicht zur allgemeinen Verteilung des Quellcodes ist in Abbildung 7.2 dargestellt. Es wird ausgehend vom *Code* des Eclipse-Projekts unterschieden zwischen client- und serverseitigem Quellcode, wobei GWT ausgehend von *Java* das notwendige *JavaScript* automatisch erzeugt. Die Kommunikation zwischen dem *Client* und *Server* wird asynchron mittels *RPC* (Remote Procedure Call) [HT07] realisiert. Des Weiteren kön-

[2]Download des WAR-Files von *CodeX 2* unter: `http://www.noesinger.net/`
[3]CodeX lief bis März 2015 erfolgreich auf dem Server Samos der Informatik der Universität Rostock.

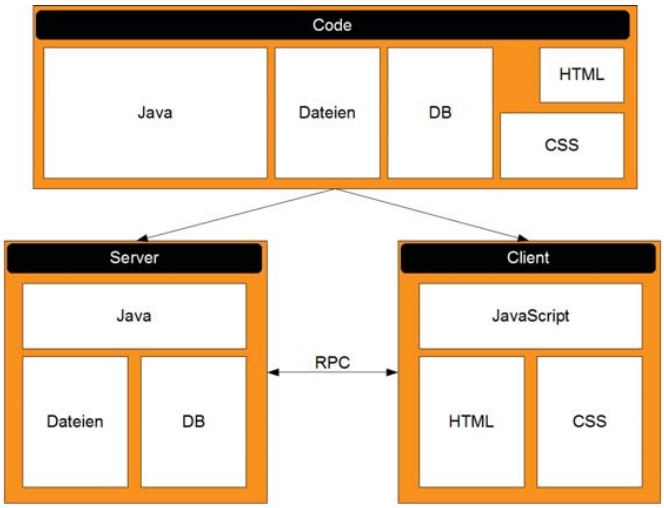


Abbildung 7.2.: Überblick der Übersetzung des GWT-Quellcodes nach [Gru13]

nen komplexe Java-Objekte als Informationscontainer serialisiert und ausgetauscht werden. Dies ist in Abbildung 7.2 allerdings nicht visualisiert.

Der Server dient sowohl der Persistierung des konzeptuellen Modells innerhalb einer Datenbank (*DB*), als auch der Speicherung von XML-Dokumenten und Schemata im Dateisystem (*Dateien*). Im Client wird hingegen das konzeptuelle Modell mittels *HTML* (Hypertext Markup Language), *CSS* (Cascading Style Sheets) und *JavaScript* visualisiert und gegebenenfalls verändert. Dabei werden vorgefertigte, kontextabhängige Dialoge angewendet, um einen Anwender bei der XML-Schemaevolution weitestgehend zu unterstützen.[4]

Die Implementierung beinhaltet mehr als 330 Java-Klassen, welche in über 75000 Code-Zeilen realisiert sind.[5]

7.1.2. Einordnung der vorgestellten Ansätze

Es wird nachfolgend in Hinblick auf die vorgestellten Ansätze der vorherigen Kapitel das Komponentenmodell der Abbildung 7.1 analysiert und ergänzt. Dabei werden die entsprechenden Komponenten aufgelistet, welche hauptsächlich den aktuell thematisierten Ansatz implementiert haben.

Die grafische Benutzeroberfläche (*GUI*) als zentrales Frontend des Prototyps, sowie die *Import-* und *Export-Komponenten* werden im nächsten Abschnitt 7.2 näher beschrieben. Dies ist damit zu begründen, dass diese Bestandteile Schnittstellen für die anderen Ansätze sind, diese allerdings nicht explizit realisieren.

[4] siehe auch: Kapitel 7.2 (Forschungsprototyp CodeX 2.0)

[5] Die Ermittlung der LOC (Lines of Code) erfolgte in Eclipse mittels des regulären Ausdrucks \n[\s]* (86839 LOC). Subtrahiert wurden die mit [*]+ ermittelten Javadoc Zeilen (11164 LOC).

In Abbildung 7.3 wird der Status der prototypischen Umsetzung im Komponentenmodell visualisiert. Es ist dargestellt, welche Komponenten realisiert (*grüner*

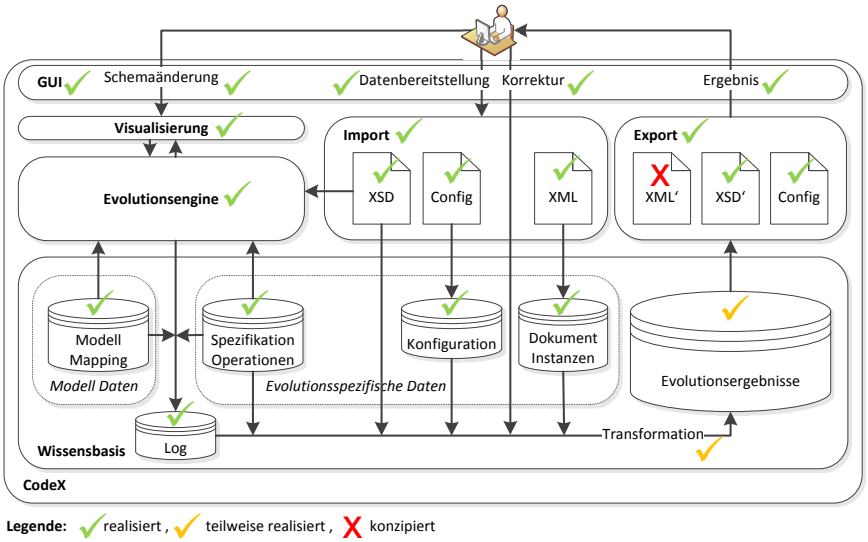


Abbildung 7.3.: Komponentenmodell mit Status der prototypischen Umsetzung

Haken), teilweise realisiert (*gelber Haken*) oder konzipiert sind (*rotes Kreuz*).

Ansätze des Kapitels 4 (Lösungsansatz)

In **Abschnitt 4.1** wurde das konzeptuelle Modell *EMX* eingeführt. Es beinhaltet sowohl visualisierte Knoten, deren Repräsentationen in Abbildung 4.3 dargestellt sind, als auch gerichtete Kanten, deren gültige Knotenkombinationen in Abbildung A.10 aufgelistet sind. Des Weiteren wurden Features eingeführt, welche zusätzliche Informationen zum XML-Schema enthalten bzw. als Nutzerkonfigurationen innerhalb der vorherigen Kapitel erläutert wurden. Zu den Konfigurationen zählen unter anderem die Möglichkeit, die unbeschränkte, maximale Häufigkeit (*maxOccurs = unbounded*) einzuschränken, oder die standardmäßige Generierung von Werten zu beeinflussen (z.B. per Nutzerinteraktion, Nullwerte, etc.). EMX ist primär in den Komponenten der *Visualisierung*, *Evolutionsengine*, sowie in der *Konfiguration* der *Wissensbasis* realisiert.

Die Speicherung und Verwaltung des konzeptuellen Modells wurde in **Abschnitt 4.3** erläutert. Das entsprechende Relationsschema ist in der Abbildung A.12 vollständig dargestellt und gehört zur *Wissensbasis* von CodeX. Speziell zählen dazu die Komponenten der *Modell Daten*, das *Log* und die *Konfiguration*, da diese ebenso in relationalen Strukturen gespeichert wird (Relation *uservariables*).

Ansätze des Kapitels 5 (Transformationssprache)

In **Abschnitt 5.2** wurde die Transformationssprache *ELaX* eingeführt, mit der das Hinzufügen, Löschen und Ändern des konzeptuellen Modells einheitlich beschrieben wird. Die Sprachspezifikation ist in Anhang B aufgelistet. Die Erfassung und Auswertung von Änderungen wurde in **Abschnitt 5.3** thematisiert. Wird eine *Schemaänderung* am EMX vollzogen, werden die entsprechenden ELaX-Ausdrücke der *Operationsspezifikation* automatisch erzeugt und im *Log* gespeichert.

Die Optimierung von ELaX durch den regelbasierten Optimierer *ROfEL* wurde in **Abschnitt 5.4** beschrieben. Die Regeln (R1) bis (R12), sowie die Funktionen TIME() (Abbildung 5.3) und MERGE() (Abbildung 5.4) wurden in der Hauptfunktion ROFEL() der Abbildung 5.5 kombiniert. Diese Funktion wird in der *Transformation* innerhalb der *Wissensbasis* aufgerufen.

Ansätze des Kapitels 6 (Adaption der Instanzen)

Die Klassifikation von Operationen zur Adaption der Instanzen wird in **Abschnitt 6.1** spezifiziert. Diesbezüglich ist die Abbildung 6.2 der Instanz- und Folgekosten von ELaX für die in **Abschnitt 6.2** erläuterte Analyse der Auswirkungen auf Instanzen entscheidend. Sind Instanz- bzw. Folgekosten möglich, dann werden die entsprechenden Komponenten identifiziert und anschließend lokalisiert. Dabei werden Lokalisierungspfade in XPath aus den vorher ermittelten EID-Ketten erzeugt. Dies wird in **Abschnitt 6.3** erläutert.

In **6.4** wird beschrieben, wie Werte für einfache und komplexe Deklarationen, sowie Wildcards erzeugt werden können. Die Nullwertfähigkeit als eine konfigurierbare Möglichkeit ist wiederum bei der Wertgenerierung enthalten. Die *Transformation* der *Wissensbasis*, sowie die *Evolutionsergebnisse* beinhalten primär die obigen Ansätze des Kapitels 6. Es sind ebenso das *Log*, die *Konfiguration*, die *Modell Daten* und *Operationsspezifikation* beteiligt.

In **Abschnitt 6.5** wird die Anpassung des *DOM* von gegebenen XML-Instanzen beschrieben. Obwohl die Bereitstellung der Dokumente in der Komponente der *Dokumentinstanzen* realisiert ist, wurde eine abschließende Realisierung der Anpassung der XML-Dokumente nur konzipiert und mit einem ausführlichen Beispiel beschrieben. Somit fehlt die Komponente *XML'* (*rotes Kreuz*) und folgerichtig Teile der *Transformation* und *Evolutionsergebnisse* (jeweils *gelber Haken*).[6]

7.2. Forschungsprototyp CodeX 2.0

Der Prototyp *CodeX* wird als WAR-File (*CodeX_2.war*) bereitgestellt und kann als lokale Webanwendung gestartet werden. Bevor dies allerdings möglich ist, müs-

[6] Es wurden hier studentische Vorarbeiten nicht abgeschlossen. Eine Kompensation mit den eigenen Ressourcen war zeitlich unmöglich, sodass auf die weitergehende Implementierung verzichtet wurde.

sen auf dem Testsystem die in Anhang C (*Vorbereitende Schritte zur Nutzung des Prototypen*) beschriebenen, initialen Schritte vollzogen werden. Die vorbereitenden Maßnahmen beinhalten unter anderem Konfigurationen, damit zum Beispiel *CodeX* die relationalen Strukturen anlegen und anschließend zur Persistierung des konzeptuellen Modells nutzen kann. Eine Liste der während der Entwicklung verwendeten Software ist im Anhang C (*Verwendete Technologien*) enthalten.

Nachdem die initialen Vorbereitungen abgeschlossen sind, kann ausgehend vom *Login*-Bildschirm (Abbildung A.60) ein neuer Anwender registriert werden. Der Registrierungsprozess (Abbildung A.61) besteht aus der Angabe von unterschiedlichen Parametern (z.B. Nutzername, Passwort, etc.) und einer Absicherung gegen automatische Bots.[7] Ist die Eingabe nicht zufriedenstellend, wird dies durch entsprechende Fehlermeldungen kommuniziert. Nach der erfolgreichen Registrierung bzw. nach dem Login startet die Hauptansicht von CodeX.

7.2.1. Grafische Benutzeroberfläche

Die *GUI* (Graphical User Interface) ist das primäre Frontend von CodeX. Dies wird in Abbildung 7.4 dargestellt. Die GUI besteht aus einer *Menüleiste*, einem

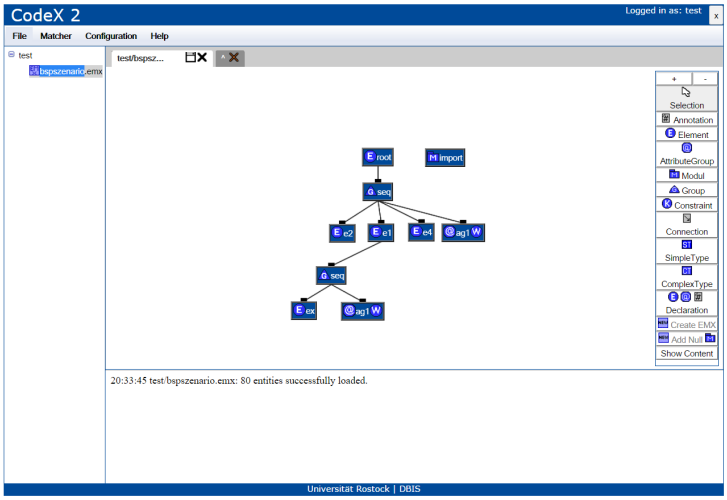


Abbildung 7.4.: Grafisches Frontend des Prototypen CodeX

Projektfenster, einem *Editorfenster*, sowie einer *Konsole*.

Die *Menüleiste* enthält grundlegende Funktionen in dem Menüeintrag *File*, in welchem sowohl neue Projekte und Dateien (*New*), als auch vorhandene Strukturen exportiert, importiert oder hochgeladen werden können. Das *Logout* beendet die aktuelle Sitzung des Anwenders, wobei ungesicherte Veränderungen verloren gehen. Auf dieses Verhalten wird mit einer *Mouseover-Information* hingewiesen.

[7] Das Passwort wird mittels MD5 verschlüsselt, ansonsten werden die Angaben nicht weiter verifiziert.

Neben dem *File* existieren noch ein *Matcher*[8], eine Hilfe (*Help*) und die Konfiguration (*Configuration*). Letzterer Menüeintrag enthält alle Nutzerkonfigurationen.

Ist ein Projekt angelegt, ist das *Projektfenster* am linken Bildschirm sichtbar und es können neue konzeptuelle Modelle (File → New → EMX) bzw. XML-Dokumente angelegt werden (File → New → XML/XSD)[9]. Alternativ können auch Dateien importiert (File → Import) oder ins Dateisystem abgelegt werden (File → Upload). Bei Projekten und Dateien innerhalb eines Projekts gilt die Namenseindeutigkeit.[10]

Jeder Eintrag im *Projektfenster* hat ein Kontextmenü, welches beim Rechtsklick geöffnet wird. Hier sind sowohl die Löschung des ausgewählten Eintrags (*Delete*) als auch das Schließen des Kontextmenüs möglich (*Close*). Das Löschen wird allerdings nicht sofort vollzogen, sondern ein *Bestätigungsdialog* wird geöffnet. Beim EMX kann im Kontextmenü zusätzlich eine Transformation (*Optimize*) gestartet werden, wodurch ein entsprechender Dialog geöffnet wird.[11]

Die *Konsole* auf der rechten, unteren Seite des Bildschirms dient der Darstellung der unterschiedlichen, farblich differenzierten Nachrichtentypen. Dies sind entweder ”normale” ELaX-Statements (blau), automatisch erzeugte ELaX-Statements (hellblau), ”normale” Aktionsmeldungen (schwarz) oder Fehlermeldungen (rot).[12]

In Abbildung 7.4 wurde zum Beispiel die ”normale” Aktionsmeldung festgehalten, dass um 20.33 Uhr (*20:33:45*) im Projekt *test* das angegebene, konzeptuelle Modell (*bspszenario.emx*) mit 80 Entitäten erfolgreich geladen wurde.

7.2.2. EMX-Editor

Der *EMX-Editor* wird durch den doppelten Linksklick auf einen entsprechen Eintrag des *Projektfensters* im *Editorfenster* als Tab bzw. Reiter geöffnet. Dieser Editor ist die wichtigste Komponente zur konzeptuellen Modellierung in CodeX.

Der EMX-Editor wird dialogbasiert bedient, das heißt, dass unterschiedliche, kontextabhängige Dialoge die Modellierung unterstützen. Die Dialoge werden bei deren Öffnung in den Vordergrund gesetzt, wodurch die restliche GUI ausgegraut wird und nicht mehr bedienbar ist. Es wird grob unterschieden zwischen *Informations-*, *Bestätigungs-*, *Übersichts-*, *Konfigurations-* und *Prozessdialogen*.

Informationsdialoge beinhalten ein Informationsfeld mit einer Nachricht, sowie einen *OK-Button* zum Schließen. In Abbildung A.62 ist ein Beispiel dargestellt.

Bestätigungsdialoge beinhalten ebenso ein Informationsfeld, allerdings muss ein Anwender in diesem Fall eine Entscheidung treffen. Wurde zum Beispiel ein Modell verändert, und im *Editorfenster* wird der entsprechende Tab durch den Klick auf das Kreuz auf demselben geschlossen, erscheint der Dialog der Abbildung A.63.

[8] siehe auch: Kapitel 7.2.5 (XSD-Matcher)
[9] siehe auch: Kapitel 7.2.5 (XML-Editor)
[10] siehe auch: Kapitel 4.3.3 (Verwaltung von Modellen)
[11] siehe auch: Kapitel 7.2.4 (Die Transformation des EMX)
[12] siehe auch: Kapitel 5.3.1 (Speicherung von Änderungen)

Es können entweder die Änderungen verworfen (*No-Button*), das Schließen abgebrochen (*Abort-Button*) oder das Modell gespeichert werden (*OK-Button*). Ein Bestätigungsdialog kann auch anders benannte Buttons besitzen.

Der *Übersichtsdialog* besitzt lediglich einen *Close-Button*, dient aber als Container für weitere Dialoge. In Abbildung A.64 wird ein Übersichtsdialog dargestellt. Es kann hier entweder eine neue Typdefinition angelegt (*New definition*)[13] oder die einfache Typhierarchie angezeigt werden (*Show current simple type hierarchy*)[14]. Eine Auswahl schließt die Übersicht und öffnet den ausgewählten Dialog.

Konfigurationsdialoge sind für die Spezifikation der Knoteneigenschaften eines EMX entwickelt worden. In Abbildung A.65 ist ein solcher Dialog für eine Annotation dargestellt. In dem *Konfigurationsdialog* wird die eindeutige *EID* (EMX ID) dargestellt und es können die knotenspezifischen Eigenschaften bzw. Attribute definiert werden. Diese sind gemäß des Datenmodells eines XML-Schemas (*Element Information Item*) eine *documentation*, *appinfo* und XML-Schema-ID (*id*).

Prozessdialoge sind komplexer, da diese aus einer mehrstufigen, festgeschriebenen und reihenfolgeabhängigen Anzahl von Dialogen bestehen. Charakteristisch ist dabei der *Continue-Button*, mit welchem die nächste Stufe in der Reihenfolge aufgerufen wird. Ein Beispiel wird bei der Transformation des EMX erläutert.

Die Toolbar des EMX-Editors

Die *Toolbar* ist eine Ansammlung von Buttons, welche im EMX-Editor auf der rechten Seite dargestellt wird. Jeder Button besitzt eine *Mouseover-Information*, welche dessen Funktion zusammenfasst. Die *Toolbar* enthält die *Zoom-Buttons* zum Vergrößern (-) und Verkleinern (+) der Arbeitsfläche des Editors (*Canvas*). Somit können auch größere Modelle entwickelt werden, insofern diese im Browser dargestellt werden können.[15] Beim erneuten Öffnen des Modells wird der *Canvas* entsprechend so vergrößert, dass alle visualisierten Entitätstypen sichtbar sind.

Die Repräsentationen von EMX-Knoten sind in der *Toolbar* ebenso enthalten. Durch eine Auswahl und den anschließenden Linksklick im *Canvas*, können entsprechend *Annotationen*, *Elemente*, *Attributgruppen*, *Module*, Inhaltsmodelle (*Group*) oder *Constraints* platziert werden. Es wird jeweils ein knotenspezifischer *Konfigurationsdialog* geöffnet, in welchem die Eigenschaften definiert werden können. Durch das Schließen des Dialogs mittels *OK-Button* wird der entsprechende Knoten dargestellt.[16] Es wird jeweils ein Rechteck mit Symbol[17] und gegebenenfalls einem Namen im Modell ergänzt. Bei Gruppen wird statt des Namens der abgekürzte Inhaltstyp angezeigt, bei Annotationen wird nur ein leerer Platzhalter

[13] siehe auch: Kapitel 7.2.3 (Umsetzung des konzeptuellen Modells)
[14] siehe auch: Kapitel 7.2.5 (Management der Typhierarchie)
[15] Hinweis: Der Browser-Zoom bei neueren Versionen verursacht Probleme in der Positionierung.
[16] Hinweis: Zur Vermeidung der Platzierung weiterer Knoten sollte der *Selection-Button* ausgewählt sein.
[17] Hinweis: Mit dem Symbol können Knoten innerhalb des *Canvas* verschoben werden.

verwendet. Der *Konfigurationsdialog* kann zu einem späteren Zeitpunkt durch das Kontextmenü des Knotens erneut aufgerufen werden (*Edit*).

Damit Kanten zwischen den Knoten zur Darstellung von Beziehungen eingefügt werden, muss der *Connection-Button* ausgewählt sein. Es wird beim Klick auf einen Knoten[18] eine entsprechende Verbindung vom Kind- zum Elternelement gemäß Abbildung A.10 ergänzt. Es werden nur gültige Verbindungen eingefügt, wobei hier der Anwender durch entsprechende Meldungen in der *Konsole* über mögliche Fehler informiert wird. Des Weiteren werden die angeklickten Knoten intern kurzfristig gespeichert, falls das Einfügen nicht sofort gelingt.[19] Zudem werden Korrekturen vorgenommen, insofern die Reihenfolge durch den Anwender nicht beachtet wurde (Kind → Eltern). Dies geschieht, wenn nur eine Kombination gültig ist. Das Entfernen von Kanten wird über das Kontextmenü des Knotens ermöglicht. In Abbildung A.66 wird der Dialog zum Löschen einer Kante gezeigt.

Zusätzlich zu den visualisierten Knoten, welche durch die vorherigen Buttons ins *Canvas* eingefügt werden, existieren nicht visualisierte Knoten. Diese sind durch die Buttons *SimpleType*, *ComplexType* und *Declaration* auswählbar.[20] Es öffnet sich jeweils ein *Übersichtsdialog*, in welchem die entsprechenden, im EMX enthaltenden Entitäten angezeigt und verändert werden können.

Der *ComplexType-Button* beinhaltet darüber hinaus die Möglichkeit, komplexe Typen im Modell zu ergänzen. Wird eine Gruppe per *Group-Button* initial eingefügt, dann ist deren Rechteck solange rot bzw. ungültig, bis ein komplexer Typ spezifiziert wurde. Dies geschieht im *Übersichtsdialog* durch den Button *Derive complex type*. Durch die Betätigung dieses Buttons wird der Knoten der Gruppe blau bzw. gültig und dessen Kontextmenü wird um die Möglichkeit der Konfiguration des komplexen Typs erweitert (*Edit ComplexType*). Zusätzlich dazu wird der entsprechende komplexe Typ allen Elementdeklarationen zugeordnet, deren visualisierte Elementreferenzen die Gruppe mittels Kante als Kindelement besitzen.[21]

Der *Create-EMX-Button* dient als Hilfestellung zum schnellen Erzeugen eines Beispiels. Dies ist allerdings nur dann möglich, wenn das *Canvas* leer ist und keine Konflikte mit bereits existierenden Entitäten auftreten. Der einfache Typ *xs:string* wird zum Beispiel durch die Betätigung des Buttons erzeugt. Ist dieser Typ als nicht visualisierter Knoten bereits vorhanden, dann ist dies ein Konflikt, der die Erzeugung des Beispiels verhindert. Sind Elemente im *Canvas* enthalten, so ist der *Create-EMX-Button* generell ausgegraut und kann nicht verwendet werden.

Ein ähnliches Verhalten ist beim *Add-Null-Button* spezifiziert. Durch diesen Eintrag der *Toolbox* wird die Nullwertfähigkeit durch das Einbinden des externen Moduls `http://www.ls-dbis.de/codex` ermöglicht. Wurde dieses Modul allerdings bereits eingebunden, dann ist der Button ebenso ausgegraut. Dies gilt unter ande-

[18] Hinweis: Der Name, Inhaltstyp oder leere Platzhalter neben dem Symbol dient als Platzierungspunkt.
[19] Hinweis: Dieser *Connection Counter* kann durch den *Selection-Button* explizit zurückgesetzt werden.
[20] Hinweis: Das Schema kann mit einem Rechtsklick auf den Tab des EMX-Editors angezeigt werden.
[21] Hinweis: Durch das Entfernen der Kante wird der Typ der Gruppe aus der Elementdeklaration entfernt.

rem, wenn der *Create-EMX-Button* benutzt wurde.

Der *Show-Content-Button* dient der Abbildung aller internen Strukturen, welche im EMX-Editor des Clients enthalten sind. Dazu zählen sowohl die Knoten des *Canvas* (*model*), als auch alle internen Entitäten (*entity*), die durch externe Module eingebundenen Entitäten (*modul*), alle Kanten (*connection*), sowie das aktuell, noch nicht serverseitig gespeicherte Log (*log*). Diese Übersicht ist in Kombination mit der Textsuche des Browsers sehr hilfreich beim Auffinden von Strukturen.

7.2.3. Umsetzung des konzeptuellen Modells

Beim erstmaligen Erzeugen eines neuen EMX beinhaltet dieses lediglich das Schema als Wurzel. Der Konfigurationsdialog des Schemas ist in Abbildung A.67 dargestellt. Damit nachfolgend Deklarationen und weitere Entitäten vollständig definiert werden können, sollten zuerst einfache Typen spezifiziert werden.

Die Erzeugung von einfachen Typen

Dies ist durch den *SimpleType-Button* der *Toolbar* möglich, durch welchen der Übersichtsdialog der Abbildung A.64 geöffnet wird. Durch den *New-definition-Button* wird anschließend ein Konfigurationsdialog für einfache Typen sichtbar und der aktuelle Übersichtsdialog wird geschlossen. Gemäß des Standards von XML-Schema können in dem Drop-Down-Menü *mode* entweder built-in-Typen (*built-in*), Listentypen (*list*), Vereinigungstypen (*union*) oder Restriktionstypen (*restriction*) definiert werden. Da erstere als Grundlage für die weiteren Typen notwendig sind (z.B. als Basistypen), sollten am Anfang built-in-Typen erzeugt werden.

Die gemäß Standard vorhandenen Typen sind im Drop-Down-Menü *built-in* des Dialogs in Abbildung 7.5 auswählbar. Neben diesem Feld ist ein *rotes Ausrufezei-*

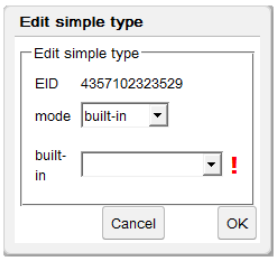


Abbildung 7.5.: Konfigurationsdialog eines einfachen built-in-Typs

chen sichtbar, welches als Symbol für notwendige, fehlende und/oder inkorrekte Spezifikationen verwendet wird. Dies dient der Unterstützung von Nicht-Experten bei der Modellierung von XML-Schema, wobei Mouseover-Informationen weiter-

gehende Hinweise beinhalten.[22] In diesem Fall kann zum Beispiel kein Typ erzeugt werden, ohne das im Drop-Down-Menü ein Eintrag gewählt wurde.[23]

Sind alle Eingaben korrekt, wird durch den OK-Button der Konfigurationsdialog geschlossen und in der *Konsole* erscheint ein entsprechendes ELaX-Statement.[24] Durch das anschließende, erneute Wählen des SimpleType-Buttons wird der erweiterte Übersichtsdialog der Abbildung 7.6 geöffnet. Dieser Dialog listet zusätz-

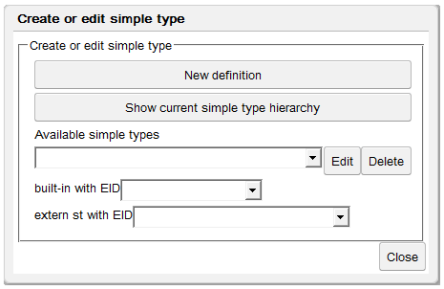


Abbildung 7.6.: Erweiterter Übersichtsdialog des SimpleType-Buttons

lich zum Übersichtsdialog der Abbildung A.64 alle erzeugten built-in-Typen auf (*built-in with EID*), sowie zum späteren Zeitpunkt alle Listen-, Vereinigungs- und Restriktionstypen. Des Weiteren werden nach einem erfolgreichen Einbinden von Modulen ebenso die externen, einfachen Entitäten angezeigt (*extern st with EID*).

In Abbildung 7.7 ist der Konfigurationsdialog von Restriktionstypen dargestellt. Dieser enthält unter anderem einen Basistypen (*baseType*), welcher aus den bisher erzeugten, einfachen Typen gewählt werden kann. Es gilt dabei, dass nur die Typen im Drop-Down-Menü aufgelistet werden, die nicht durch Beschränkungen (*finalV*) ausgeschlossen sind. Eine Selbstreferenzierung ist ebenso nicht möglich.

Wurde ein Basistyp gewählt, erscheinen sowohl die EID des Basistypen (*base_EID*) als auch der *Facet-definition-Button*. Der Konfigurationsdialog der Abbildung 7.8 wird durch diesen Button geöffnet. Die dargestellten Facetten sind abhängig von Basistypen und können Abbildung A.1 entnommen werden.

Es wird unterschieden zwischen den ausgegrauten, nicht änderbaren Facetten der einfachen Typen, welche als Vorgänger in der Typhierarchie stehen, und den eigenen, änderbaren Facetten (*Facets of the current simple type*). Es gilt dabei, dass nicht weiter einschränkbare Facetten im Allgemeinen nicht änderbar sind. Die Längenfacette (*length*) wird zum Beispiel aus der Liste der möglichen Facetten entfernt, insofern ein Vorgänger diese als nicht änderbar (*fixed*) spezifiziert hat. Dies ist im angepassten Konfigurationsdialog der Abbildung A.68 dargestellt.[25]

[22] Hinweis: Solange Konfigurationsdialoge rote Ausrufezeichen enthalten, wird durch den OK-Button ein Informationsdialog geöffnet statt eine neue Entität zu erzeugen.

[23] Hinweis: Mehrfaches Erzeugen wird verhindert: *This built-in simple type was already added.*

[24] Hinweis: Erst das finale Betätigen des OK-Buttons gilt i.A. als Bestätigung einer Erzeugung, sodass ein Ausprobieren der gegebenen Funktionalitäten keine unnötigen ELaX-Statements im Log erzeugt.

[25] Hinweis: Um bei tieferen Hierarchien die Herkunft von Facetten zu verdeutlichen, wird *of EID* ergänzt.

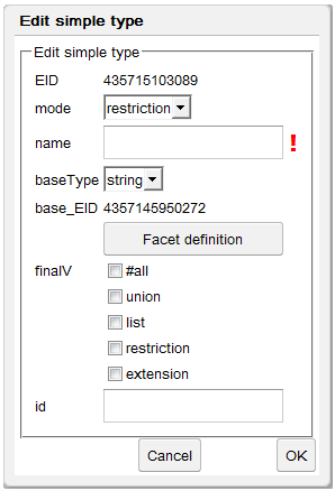


Abbildung 7.7.: Konfigurationsdialog eines Restriktionstyps

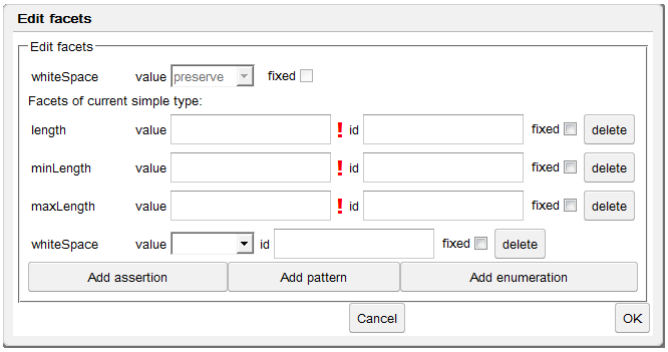


Abbildung 7.8.: Konfigurationsdialog eines Restriktionstyps - Facetten

Wurden alle Facetten spezifiziert bzw. ungewollte entfernt (*delete*), dann können durch den OK-Button die Änderungen übernommen werden. Die Facetten werden anschließend solange temporär gespeichert, bis diese entweder geändert werden oder der Konfigurationsdialog des Restriktionstyps ebenso mit dem OK-Button geschlossen wird. Ein ELaX-Statement des Restriktionstypen mit allen Facetten wird gegebenenfalls in der *Konsole* angezeigt und im Log gespeichert.

Die Konfigurationsdialoge von Listentypen sind in Abbildung A.69, die von Vereinigungstypen in Abbildung A.70 dargestellt. In beiden Dialogen können keine Facetten spezifiziert werden. Da allerdings Listentypen standardmäßig die Facette *whiteSpace* besitzen, wird diese automatisch ergänzt. Ein entsprechendes, hellblaues ELaX-Statement wird erzeugt. Bei der Validierung von Vereinigungstypen ist darüber hinaus die Reihenfolge der Teilnehmer (*types*) entscheidend. Im Konfigurationsdialog wird der oberste Typ entsprechend als erster übernommen.

Deklarationen mit einfachen Typen

Nachdem einfache Typen definiert wurden, können diese anschließend verwendet werden. Mit Hilfe des *Declaration-Buttons* der *Toolbar* wird der Übersichtsdialog der Abbildung A.71 geöffnet. In diesem können Element- und Attributdeklarationen, sowie nicht visualisierte Annotationen sowohl definiert, als auch anschließend verändert und/oder gelöscht werden.

Die *New-Buttons* öffnen jeweils die Konfigurationsdialoge der Abbildungen 7.9, A.72 und A.73[26]. In Abbildung 7.9 ist der Konfigurationsdialog einer Elementde-

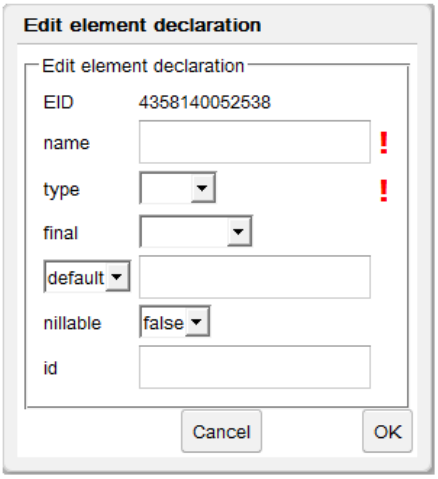


Abbildung 7.9.: Konfigurationsdialog einer Elementdeklaration

klaration dargestellt. Das Drop-Down-Menü *type* enthält in diesem Fall nur einfache Typen. Wird der Dialog allerdings über den *Element-Button* der *Toolbar* aufgerufen, dann sind ebenso komplexe Typen enthalten. Da die Typen nur mit deren Namen aufgelistet werden, wird mittels Präfix zwischen einfachen (*st*) und komplexen Typen (*ct*) unterschieden. Des Weiteren wird als Suffix *(m)* angehängt, insofern der entsprechende Typ mit Hilfe eines Moduls integriert wurde.

Neben den vordefinierten Möglichkeiten der Attribute *type*, *final* und *nillable*, muss ein Anwender entscheiden, ob ein *default-* oder *fixed-Wert* spezifiziert werden soll. Das heißt, dass standardkonform bei Deklarationen eine der beiden Varianten mit Hilfe des Drop-Down-Menüs ausgewählt werden muss. Dadurch werden wiederum Nicht-Experten bei der Modellierung von XML-Schema unterstützt.

Das Entfernen von obigen Deklarationen führt zum *kaskadierenden Löschen*[27] von abhängigen Entitäten. Das heißt, dass, falls im konzeptuellen Modell eine gelöschte Deklaration referenziert wird, die entsprechende Referenz ebenso entfernt

[26] Hinweis: Bei *to parent* muss eine Entität ausgewählt werden. Das Drop-Down-Menü enthält dabei keine visualisierten Entitäten, da diese mit dem *Annotation-Button* der Toolbar erzeugt werden.
[27] siehe auch: Kapitel 6.1.1 (Löschen von Komponenten)

wird. Dies gilt auch für nicht visualisierte Annotationen, insofern diese einer entsprechenden Deklaration zugeordnet waren. Die notwendigen ELaX-Statements werden wiederum erzeugt, allerdings als normale, blaue Statements.

Attribut- und Elementreferenzen

Die erzeugten Deklarationen können unter anderem innerhalb von Attribut- und Elementreferenzen verwendet werden, wobei erstere in Attributgruppen enthalten sind. Diese können mit dem *AttributeGroup-Button* der *Toolbar* innerhalb des *Canvas* positioniert werden, wobei der Konfigurationsdialog der Abbildung 7.10 geöffnet wird. Es sind in dem Dialog sowohl die Attribute der Gruppe, als auch der

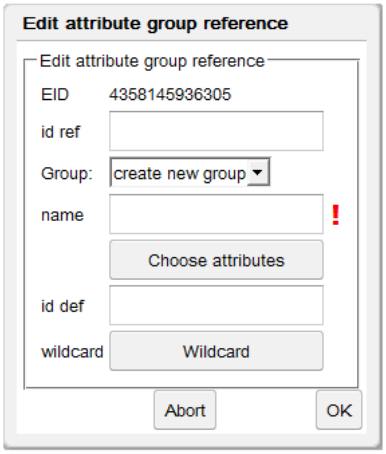


Abbildung 7.10.: Konfigurationsdialog einer Attributgruppe mit Referenz

Attributgruppenreferenz enthalten. Im Drop-Down-Menü *Group* kann entweder eine neue Attributgruppe erzeugt (*create new group*) oder eine bereits vorhandene ausgewählt werden. Im letzten Fall werden sowohl die Attributreferenzen, als auch Wildcard, Name (*name*) und XML-Schema-ID (*id def*) übernommen.

Attributreferenzen der Attributgruppe werden durch den *Choose-attributes-Button* spezifiziert. Der Konfigurationsdialog ist in Abbildung 7.11 dargestellt. Sind im konzeptuellen Modell keine Attributdeklarationen vorhanden, können diese durch den *New-attribute-declaration-Button* erzeugt werden. Es öffnet sich in diesem Fall wie oben beschrieben der Konfigurationsdialog der Abbildung A.72.

Existieren Deklarationen, dann können diese im Drop-Down-Menü einer Referenz ausgewählt werden.[28] Zusätzliche Referenzen werden mit dem *New-attribute-declaration-Button* hinzugefügt, wobei doppelte und/oder leere Referenzen einer Attributdeklaration verhindert werden. Sind keine roten Ausrufezeichen vorhanden, dann werden durch den OK-Button die Attributreferenzen solange temporär

[28]Hinweis: Eine Zeile im Konfigurationsdialog der Attributliste entspricht genau einer Attributreferenz.

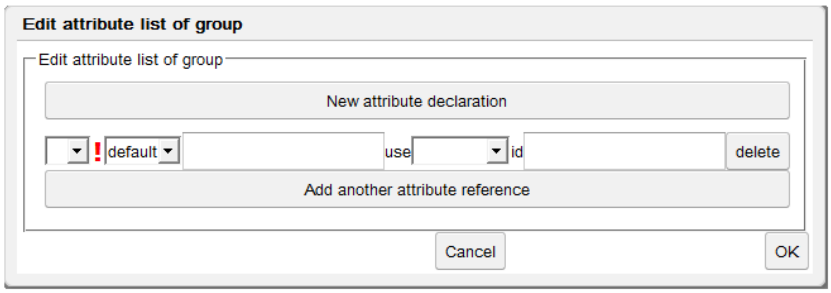


Abbildung 7.11.: Konfigurationsdialog von Attributreferenzen einer Attributgruppe

gespeichert, bis der Konfigurationsdialog der Attributgruppe ebenso mit Hilfe des OK-Buttons geschlossen wird.[29]

Elementreferenzen werden unter Anwendung des *Element-Buttons* der *Toolbar* im *Canvas* positioniert. Es wird dadurch der Konfigurationsdialog der Abbildung A.74 geöffnet. Anschließend können alle bereits im Modell vorhandenen Elementdeklarationen im Drop-Down-Menü *reference* ausgewählt werden. Ist eine Auswahl nicht möglich, so kann alternativ zum obigen Vorgehen (d.h. mit dem Declaration-Button) der *Element-Declaration-Button* zum Erzeugen von Deklarationen verwendet werden. Es wird dadurch der Konfigurationsdialog der Abbildung 7.9 geöffnet.

Wird eine neue Elementdeklaration erzeugt, wird diese automatisch der Elementreferenz zugeordnet und sofort im Modell ergänzt. Daher ist nach der initialen Erzeugung zur Konsistenzerhaltung des konzeptuellen Modells eine Änderung bis zum nächsten Aufruf der Elementreferenz nicht möglich. Das entsprechende Drop-Down-Menü der Referenz und der *Element-Declaration-Button* sind ausgegraut.

Elementdeklarationen mit komplexen Typen

Wird ein komplexer Typ im Konfigurationsdialog der Elementdeklaration ausgewählt und eine Elementreferenz anschließend mittels OK-Button hinzugefügt, dann wird automatisch eine Kante zu der Gruppe ergänzt, welche als Repräsentation des entsprechenden Typs im *Canvas* enthalten ist.

Da allerdings zum Anfang einer Modellierung weder komplexe Typen noch deren Gruppen vorliegen können, ist als temporärer Typ einer Elementdeklaration *not yet given ct* im Drop-Down-Menü *type* auswählbar. Als Konsequenz wird das Symbol des Knotens der Elementreferenz entsprechend rot gefärbt.

Gruppen können mit Hilfe des *Group-Buttons* der *Toolbar* im *Canvas* positioniert werden, wodurch der Konfigurationsdialog der Abbildung 7.12 geöffnet wird. Es können mittels *mode* die unterschiedlichen Inhaltsmodelle Reihenfolge (*sequence*), Auswahl (*choice*) und Menge (*all*) spezifiziert werden, wobei die unterstrichenen

[29] Hinweis: Attributgruppen können nach Erzeugung im Schema als *defaultAttribute* ausgewählt werden.

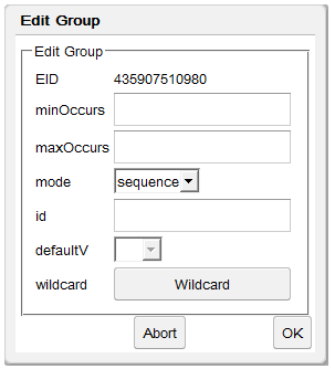


Abbildung 7.12.: Konfigurationsdialog einer Gruppe

Buchstaben im Knoten der Gruppe als Abkürzung verwendet werden. Wird der nicht benannte Eintrag gewählt, dann ist das Inhaltsmodell leer (*empty*) und die Gruppe kann keine Elementreferenzen als Kinderelement haben.[30] Des Weiteren werden im Konfigurationsdialog alle Attribute mit Ausnahme von *mode* ausgegraut, da diese standardkonform beim leeren Inhaltsmodell nicht vorhanden sind.

Wird der Konfigurationsdialog mit dem OK-Button geschlossen, dann wird beim erstmaligen Erzeugen einer Gruppe deren Knoten rot eingefärbt. Dies ist damit zu begründen, dass diese Gruppe noch keinen komplexen Typen besitzt.

Mit dem *ComplexType-Button* der *Toolbar* wird der Übersichtsdialog der Abbildung A.75 geöffnet, sodass mittels *Derive-complex-type-Button* komplexe Typen hergeleitet werden können. Wird dieser Button betätigt, dann wird der Übersichtsdialog geschlossen und alle roten Gruppen bekommen einen komplexen Typen zugeordnet. Die Gruppenknoten werden blau und alle verbundenen, roten Elementreferenzen, deren Deklaration *not yet given ct* spezifiziert hatten, werden angepasst und ebenso blau gefärbt.[31] Die notwendigen ELaX-Statements werden erzeugt.

Der komplexe Typ ist anschließend über den Kontexteintrag des Gruppenknotens editierbar (*Edit ComplexType*).[32] Es öffnet sich durch diesen Eintrag der Konfigurationsdialog der Abbildung A.76. Der durch den *Derive-complex-type-Button* erzeugte Name des Typs ist im Allgemeinen eine Konkatenation aus *ct* und der EID des komplexen Typs. Der Name sollte gegebenenfalls geändert werden.

[30] Hinweis: Vorhandene Kanten zu Elementreferenzen werden durch *empty* entsprechend gelöscht.
[31] Hinweis: Durch das Löschen der Kante zwischen einer Elementreferenz und Gruppe, wird der Elementdeklaration der temporäre Typ *not yet given ct* zugeordnet und der Referenzknoten rot eingefärbt.
[32] Hinweis: *Show Assertions* ist relevant bzgl. der Integration von Integritätsbedingungen [Gru13].

Wildcards

In den Konfigurationsdialogen von Attributgruppen (Abbildung 7.10) und Gruppen (Abbildung 7.12) sind jeweils *Wildcard-Buttons* enthalten. Deren Betätigung

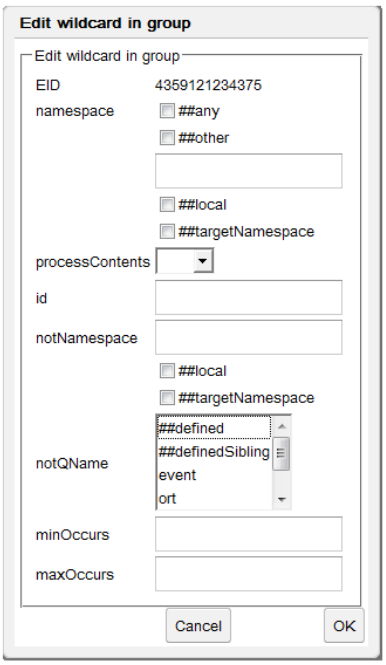


Abbildung 7.13.: Konfigurationsdialog einer Elementwildcard

öffnet entweder den Konfigurationsdialog einer Elementwildcard (Abbildung 7.13) oder einer Attributwildcard (Abbildung A.77). Der strukturelle Unterschied zwischen beiden Dialogen ist, dass Elementwildcards Attribute für Häufigkeitsangaben (*minOccurs* und *maxOccurs*) enthalten können und der Eintrag *##definedSibling* beim Attribut *notQName* ergänzt wurde.[33]

Wird eine Wildcard hinzugefügt, dann wird der entsprechende Knoten einer Gruppe oder Attributgruppe um das Wildcardsymbol gemäß Abbildung 4.3 ergänzt (*blau umrandetes W mit weißer Schriftfarbe*). Zum anschließenden Löschen der Wildcard muss der *Wildcard-Button* des Knotens erneut betätigt werden. Der nun sichtbare, ergänzte *Delete-Button* löscht die Wildcard, insofern der Konfigurationsdialog der Gruppe oder Attributgruppe mittels OK-Button beendet wird.[34]

[33] Hinweis: Durch das Halten von STRG können bei notQName mehrere Einträge ausgewählt werden.
[34] Hinweis: Der Cancel-Button der Gruppe oder Attributgruppe verwirft die Löschung der Wildcard.

Constraints

Mit Hilfe des *Constraint-Buttons* der *Toolbar* können Constraints im *Canvas* hinzugefügt werden. Durch die Anwendung wird der Konfigurationsdialog der Abbildung 7.14 geöffnet. Es kann im Drop-Down-Menü *type* der Typ der Constraint (*key*, *unique* und *keyref*) gewählt werden, wobei nur Schlüsselreferenzen vorhandene Constraints referenzieren können. Ist keine Schlüsselreferenz ausgewählt (*keyref*), dann ist das entsprechende Drop-Down-Menü *refer* ausgegraut.

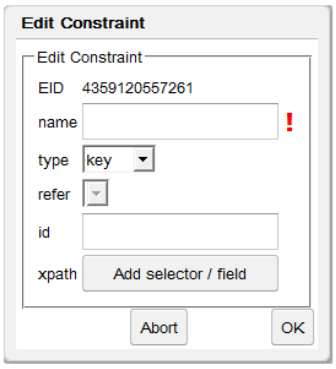


Abbildung 7.14.: Konfigurationsdialog einer Constraint

Der Konfigurationsdialog kann anschließend nur dann mittels OK-Button geschlossen werden, wenn sowohl ein entsprechender Selektor als auch mindestens ein Feldwert spezifiziert wurden.[35] Dies ist durch den *Add-selector/field-Button* möglich, der den Konfigurationsdialog der Abbildung 7.15 öffnet. In diesem Dialog

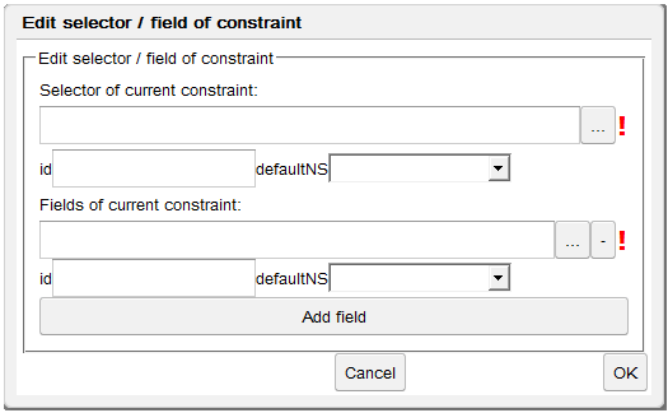


Abbildung 7.15.: Konfigurationsdialog einer Constraint - Selektor und Feldwerte

können XPath-Ausdrücke des Selektors und der Feldwerte spezifiziert werden. Der

[35] Hinweis: Ein Informationsdialog weist gegebenenfalls auf das Fehlen des Selektors mit Feldwerten hin: *At least one selector AND field path is needed. Please press the 'Add selector / field' button.*

Add-field-Button fügt dabei zusätzliche Feldwerte ein. Das Löschen dieser Werte ist mit Hilfe des *Minus-Buttons* (-) möglich, wobei mindestens ein Feldwert standardkonform vorhanden sein muss.[36]

Der *Punkte-Button* (...) öffnet den Konfigurationsdialog der Abbildung 7.16. In

Abbildung 7.16.: Konfigurationsdialog einer Constraint - XPath-Spezifikation

diesem Dialog können statt der textuellen Formulierung von XPath-Ausdrücken alle Deklarationen von EMX ausgewählt werden.[37] Dies ist eine Möglichkeit zur Lösung bzw. Vereinfachung der in Kapitel 6.4.4 thematisierten Problematik des Vergleichs von XPath-Ausdrücken. Statt eines Ausdrucks würden in CodeX somit EIDs verwendet werden, sodass die Analyse von Auswirkungen von Schemaänderungen auf einen einfacheren Mengenvergleich reduziert werden würde.

Dies würde allerdings voraussetzen, dass ein konzeptuelles Modell vollständig nur innerhalb von CodeX erzeugt und verarbeitet wird. Ein Import mit anschließender Transformation von beliebigen XPath-Ausdrücken wäre allerdings wiederum problematisch. Dennoch könnte für eine reduzierte Menge von XPath-Ausdrücken eine automatisierte Evolution mit den vorhandenen Mechanismen durchgeführt werden. Diese Problematik bietet Potential für weitere Arbeiten, sodass eine vollständige Integration innerhalb von CodeX abschließend nicht erfolgt ist.

Module

Module können mittels *Modul-Button* der *Toolbar* ins *Canvas* hinzugefügt werden. Der entsprechende Konfigurationsdialog ist in Abbildung A.78 dargestellt.

In Abhängigkeit des Modus (*mode*) sind die Attribute *namespace* und *prefix* spezifizierbar. Dies ist standardkonform nur für einen Import möglich, sodass bei den anderen Modi die entsprechenden Felder nicht zur Verfügung stehen.

Die detaillierte Ansicht (*detailedView*) öffnet bei deren Auswahl eine zusätzliche Übersicht. Diese enthält sowohl das Anlagedatum des Moduls, als auch alle externen Entitäten, welche im konzeptuellen Model durch das Modul eingebunden

[36] Hinweis: Wurde ein Feldwert unbeabsichtigt gelöscht, sollte der Cancel-Button betätigt werden.
[37] Hinweis: Beim Selektor ist durch das Halten von STRG eine Mengenvereinigung von EIDs (|) möglich.

wurden. In Abbildung A.79 ist dies exemplarisch für das durch den *Add-Null-Button* der *Toolbar* eingefügte Modul dargestellt.

Es werden nur globale, externe Entitäten eingebunden, die zum Anlagedatum verfügbar sind. Eine erneute Prüfung der Schemaquelle ist nicht vorgesehen, da externe Quellen deren Semantik jederzeit unbemerkt ändern können. Dies kann durch CodeX weder geprüft noch verhindert werden.

Nachdem die Realisierung des konzeptuellen Modells in CodeX beschrieben wurde, wird nachfolgend an einem Beispiel die XML-Schemaevolution erläutert.

7.2.4. Anwendung des EMX-Editors

Ein konzeptuelles Modell kann mit den im vorherigen Abschnitt beschriebenen Möglichkeiten vollständig neu spezifiziert werden. Eine Alternative stellt der Import eines XML-Schemas dar, der nachfolgend angewendet wird.

Es wird hierfür das XML-Schema des XML-Beispiels 1.2 verwendet. Anschließend werden das automatisch erzeugte EMX verändert und die XML-Schemaevolution mit unterschiedlichen Nutzerkonfigurationen beschrieben. Der Export des angepassten XML-Schemas wird zum Abschluss erläutert.

Importieren eines XML-Schemas

Nachdem ein Projekt angelegt wurde (File → New → Project), kann ein XML-Schema importiert werden (File → Import). Es öffnet sich der Dialog der Abbildung A.80, in welchem das Projekt *bsp* ausgewählt wurde.[38] Zum Erstellen eines konzeptuellen Modells muss die Option *Also create EMX* ausgewählt werden.[39] Nach dem Betätigen des OK-Buttons wird das Modell der Abbildung 7.17 erstellt.

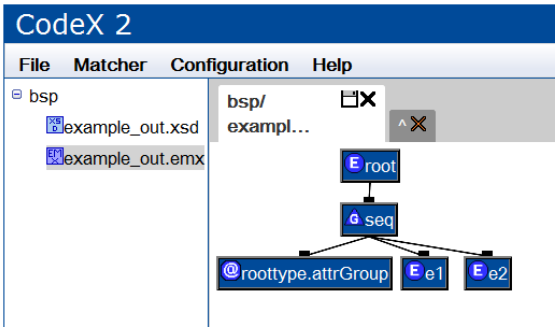


Abbildung 7.17.: EMX des importierten XML-Schemas des XML-Beispiels 1.2

[38] Hinweis: Ist im Projektfenster ein Projekt selektiert, wird dieses automatisch im Dialog ausgewählt.
[39] Hinweis: *Allow type redundancy* ist relevant bzgl. der Transformation von Modellierungsstilen [Kap13].

Im Projektfenster werden sowohl das konzeptuelle Modell (*example_out.emx*), als auch dessen importiertes XML-Schema (*example_out.xsd*) angelegt. Im Vergleich zum Ausgangsschema des XML-Beispiels 1.2 wurde Modell-bedingt allerdings die Attributgruppe *roottype.attrGroup* erzeugt, welche als Container der Attributreferenzen der Deklarationen *a1* und *a2* dient.[40]

Nachdem die Konfiguration gemäß Abbildung A.81 geändert wurde (Configuration → generateValue)[41], werden Anpassungen am Modell vorgenommen. Es werden sowohl das Inhaltsmodell der Gruppe auf *choice* umgestellt, als auch dessen minimale Häufigkeit auf zwei erhöht (jeweils Abbildung A.82) und die Auftrittshäufigkeiten der Attributreferenzen geändert (Abbildung A.83). Die Attributreferenz *a1* ist somit optional (*use = 'optional'*), während *a2* verboten ist (*use = 'prohibited'*).

Anschließend wird das Modell gespeichert, was in Abbildung 7.18 dargestellt ist. Es wird sowohl der Informationsdialog zum erfolgreichen Speichern angezeigt, als

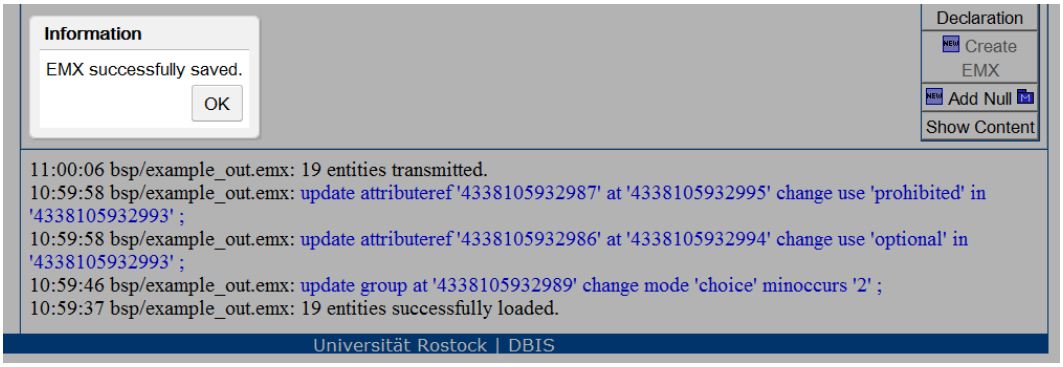


Abbildung 7.18.: Speicherung des EMX mit Darstellung der Konsole

auch die Konsole mit den getätigten Änderungen. Die ELaX-Statements wurden durch obige Anpassungen automatisch erzeugt und entsprechend gespeichert.

Die Transformation des EMX

Die Transformation eines konzeptuellen Modells wird durch den Rechtsklick auf den entsprechenden Eintrag im Projektfenster mit anschließender Auswahl von *Optimize* begonnen. Dadurch wird der gegebenenfalls noch geöffnete EMX-Editor geschlossen[42] und ein Prozessdialog wird sichtbar.[43]

Es wird der Dialog der Abbildung A.84 geöffnet. Dies entspricht der *Optimierung von EMX Anpassungen* der Abbildung 6.25. Es kann konfiguriert werden, welche

[40] siehe auch: Kapitel 4.1.1 (Entitätstypen von EMX)
[41] Hinweis: In CodeX ist die Standardkonfiguration der Wertgenerierung der Eintrag *use NULL*.
[42] Hinweis: Das serverseitig gespeicherte EMX wird genutzt, ungesicherte Änderungen werden verworfen.
[43] siehe auch: Kapitel 7.2.2 (EMX-Editor)

Änderungen ausgewertet werden sollen.[44] Dies sind standardmäßig alle Anpassungen, sodass *lastChanges* entsprechend selektiert ist.

Durch den *Continue-Button* wird der Dialog um eine Liste von ELaX-Operationen erweitert. Der angepasste Prozessdialog ist in Abbildung 7.19 dargestellt. Die

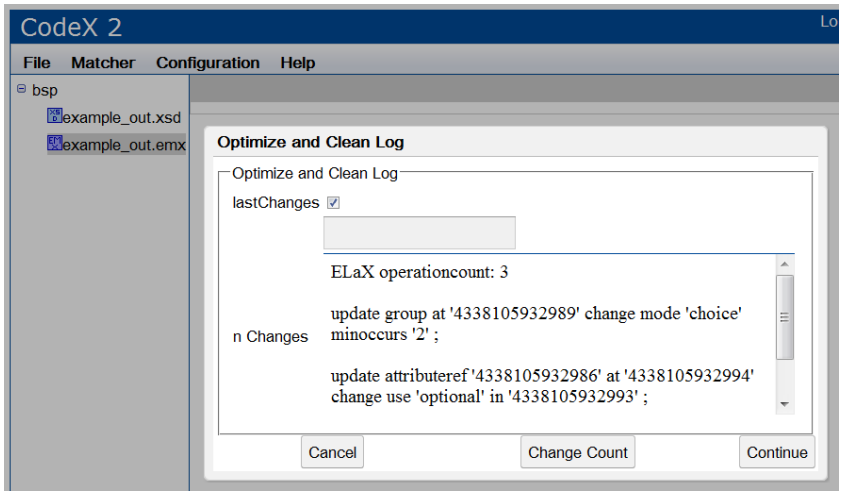


Abbildung 7.19.: Prozessdialog nach der Anwendung von ROfEL

Liste ist die mit Hilfe von ROfEL ermittelte, minimierte Sequenz von Schemaänderungen. Die nachträgliche Änderung ist durch den *Change-Count-Button* möglich, wodurch ROfEL mit der veränderten Konfiguration erneut aufgerufen wird.

Durch die Betätigung des Continue-Buttons wird der Prozessdialog der Abbildung A.85 geöffnet. Dieser realisiert die Analyse der Auswirkungen auf die Instanzen (d.h. *Instanzanpassungen notwendig*) der Abbildung 6.25. Es sind sowohl das Ausgangsschema (*EMX*), als auch die kompletten Logeinträge der analysierten Änderungen (*ELaX*) und das Zielschema (*EMX'*) enthalten.

Wird der *Check-Log-Button* betätigt, dann wird auf Grundlage des konzeptuellen Ausgangsmodells analysiert, ob die ELaX-Statements Auswirkungen auf die Instanzen haben. Es werden nur solche Schemaoperationen für nachfolgende Analysen übernommen, die Instanz- und/oder Folgekosten gemäß Abbildung 6.2 hervorrufen. Als Resultat wird der Prozessdialog ein weiteres Mal angepasst, dies ist in Abbildung A.86 dargestellt. Es wurde hier die Anpassung der Attributreferenz *a1* entfernt, da die Änderung der Auftrittshäufigkeit von *required* auf *optional* keine Instanzanpassungen hervorruft. Folgerichtig ist der ELaX-Operationszähler der Abbildung A.86 von drei auf zwei reduziert worden.

Ist die Anzahl der verbleibenden ELaX-Statements Null (*Checked ELaX operation count: 0*), dann wird der Prozessdialog durch den Continue-Button geschlos-

[44] Hinweis: Änderungen werden blockweise gespeichert, wobei die Nachricht *X entities transmitted* als Trennelement gilt. Blöcke ohne normale ELaX-Statements werden bei lastChanges nicht mitgezählt.

sen.[45] Ansonsten wird der Dialog der Abbildung 7.20 geöffnet, welcher wiederum die Liste der verbleibenden Statements enthält. Der Dialog realisiert die *Erzeugung*

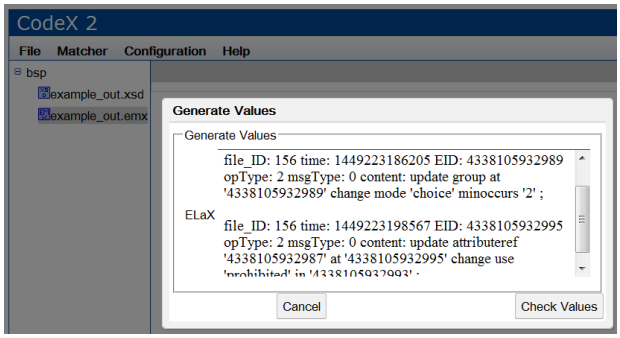


Abbildung 7.20.: Prozessdialog vor der Generierung von Werten

von Transformationsschritten der Abbildung 6.25.

Wird anschließend der *Check-Values-Button* betätigt, werden alle Entitäten aufgezählt, für die innerhalb des konzeptuellen Modells keine Werte vorgegeben sind. Dies wird in der Abbildung A.87 dargestellt und ist die Konsequenz des in Abbildung A.81 spezifizierten *generateValue* Wertes (d.h. *ask USER*).[46]

Durch die Angabe eines Wertes für *e1* verschwindet das rote Ausrufezeichen, und der Prozessdialog kann fortgesetzt werden. Alternativ zur Eingabe kann mit Hilfe des *Punkte-Buttons* (...) und dem Dialog der Abbildung A.88 eine Beispieldatei zum Auffinden eines Wertes verwendet werden. Dies sollte ein wohlgeformtes XML-Dokument sein, in welchem nach dem Knotennamen *e1* gesucht wird.

Die Fortsetzung des Prozessdialogs durch den Continue-Button öffnet den abschließenden Dialog der Abbildung 7.21. In diesem kann ein XML-Dokument an-

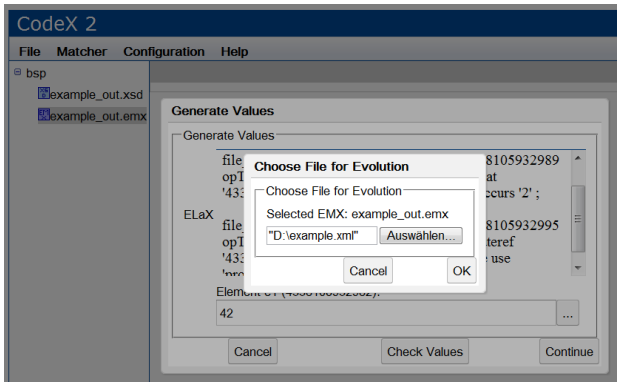


Abbildung 7.21.: Prozessdialog zur Auswahl eines XML-Dokuments

[45] Hinweis: Ein Informationsdialog beinhaltet: *No ELaX-Statement remained for evolution.*

[46] Die Transformation unter Anwendung der Nullwertfähigkeit (*use NULL*) wird ebenso noch erläutert.

gegeben werden, welches evolutioniert werden soll. Aktuell wird nur das anzupassende XML-Dokument geladen und die internen Strukturen in einem Kommentar hinzugefügt. Dazu zählen sowohl die Nutzervariablen, als auch die optimierten Logeinträge, die Wildcardflags, die EID-Ketten und deren Lokalisierungspfade in XPath, sowie die generierten Werte.

Der durch den OK-Button geöffnete Informationsdialog ist in Abbildung A.89 dargestellt. Da das Inhaltsfeld von diesem Dialog keine Kommentare visualisiert, wird zusätzlich ein Popup[47] erzeugt, welches mit einem externen Texteditor geöffnet werden kann. Dies wird durch die Abbildung A.90 dargestellt. Die Öffnung des XML-Dokuments mit Kommentaren innerhalb des externen Texteditors *Notepad++* wird in Abbildung A.91 visualisiert.

Es werden obige Strukturen aufgelistet, wobei die EIDs der Entitäten der ELaX-Statements zur Unterscheidung der Einträge dienen. Die Erhöhung der minimalen Häufigkeit der Gruppe (*EID = 4338105932989*) benötigt zum Beispiel generierte Werte für die zwingende Elementreferenz *e1* (*GenValues: <e1>42</e1>*). Für die Änderung der Häufigkeit der Attributreferenz (*EID = 4338105932995*) auf *prohibited* ist dies nicht notwendig, sodass *GenValues* entsprechend leer ist.[48]

Konfiguration und Anwendung der Nullwertfähigkeit

Um die XML-Schemaevolution weitergehend zu automatisieren, sollte die Nutzerinteraktion möglichst vermieden werden. Die Einführung der Nullwertfähigkeit ist diesbezüglich eine Möglichkeit, welche durch die Änderung der Nutzerkonfiguration (Configuration → generateValue) ermöglicht wird. In Abbildung A.92 wird dargestellt, wie die standardmäßig vorkonfigurierte Einstellung *use NULL* im Drop-Down-Menü *generateValue* ausgewählt wird.

Anschließend kann mit Hilfe der obigen Beschreibung die Transformation des EMX bis zur Abbildung 7.20 ein weiteres Mal vollzogen werden. Wird nun allerdings der *Check-Values-Button* betätigt, dann erscheint der Informationsdialog der Abbildung A.93. Dieser weist darauf hin, dass im EMX das für die Nullwertfähigkeit notwendig, externe Modul `http://www.ls-dbis.de/codex` noch nicht eingebunden ist.[49] Des Weiteren wird beschrieben, mit welchem Vorgehen diese Voraussetzung erfüllt werden kann.

Das erneute Öffnen des konzeptuellen Modells mit anschließendem Einbinden des Moduls wird empfohlen. Dies ist durch den *Add-Null-Button*[50] der *Toolbar* möglich, durch welchen das notwendige Modul automatisch im EMX eingebunden wird.[51] Anschließend muss das Modell gespeichert werden, was in Abbildung 7.22

[47] Hinweis: Das Popup wird gegebenenfalls durch den Browser blockiert und muss zugelassen werden.
[48] Hinweis: [] ist in Abbildung A.91 gleichbedeutend mit einem leeren Feld.
[49] Hinweis: Ohne die Einbindung des Moduls ist die Nutzung der Nullwertfähigkeit nicht möglich.
[50] Hinweis: Alternativ könnte ein Modul mittels des Modul-Buttons der Toolbar mit der schemaLocation `http://www.ls-dbis.de/codex` importiert werden. Diese Variante ist allerdings aufwendiger.
[51] Hinweis: Eine Verbindung ins Internet ist notwendig, da sonst ein leeres Modul eingebunden wird.

dargestellt ist. Somit ist die Voraussetzung für die Nullwertfähigkeit gegeben, die

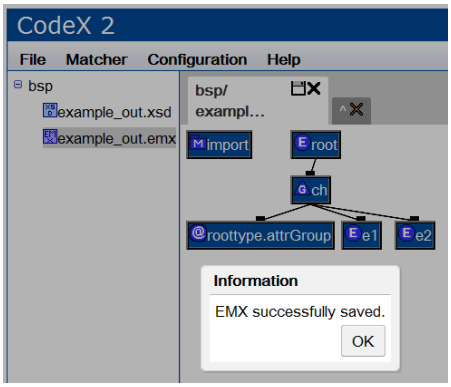


Abbildung 7.22.: Erweiterung des EMX der Abbildung 7.17 um das Modul

Transformation kann ein weiteres Mal begonnen werden.

Als Konsequenz der Einbindung des Moduls wird beim Betätigen des *Check-Values-Buttons* der Abbildung 7.20 nicht der Informationsdialog über die fehlende Voraussetzung geöffnet, sondern der Prozessdialog der Abbildung A.94. Im Vergleich zum Vorgehen mit der Konfiguration *ask USER* fehlt das Feld zur Eingabe eines Wertes für *e1* (vgl. Abbildung A.87). Nach der Betätigung des Continue-Buttons wird wiederum der Dialog zur Auswahl eines XML-Dokuments geöffnet (siehe Abbildung 7.21). Anschließend kann das in Abbildung A.95 dargestellte, veränderte Popup in einem externen Texteditor geöffnet werden. In diesem wird der zwingenden Elementreferenz *e1* aufgrund der Nullwertfähigkeit der Nullwert *null* zugeordnet (*GenValues: <e1>null</e1>*).

Durch die letztmalige Betätigung des Continue-Buttons wurde der Typ der Elementdeklaration *e1* verändert. Es wurde automatisch ein Vereinigungstyp erzeugt, deren Teilnehmer der ursprüngliche Typ (*decimal*) und der Nullwerttyp des Moduls sind (*null*). Der Konfigurationsdialog des einfachen Vereinigungstyps ist in der Abbildung 7.23 dargestellt, wobei der Nulltyp wiederum als Entität eines Moduls mit *(m)* gekennzeichnet ist. Die angepasste Elementdeklaration ist im Konfigurationsdialog der Abbildung A.96 dargestellt. Als Folge der obigen Einbindung des Moduls ist im Drop-Down-Menü *type* der neue Vereinigungstyp spezifiziert.

Wird nun abermals die Transformation des EMX vollzogen, enthält der Prozessdialog zur Analyse der Auswirkungen auf die Instanzen weitere Einträge (siehe Abbildung A.97). Nach der Überprüfung des Logs mittels *Check-Log-Button* und der anschließenden Betätigung des Continue-Buttons, wird der Prozessdialog der Abbildung A.98 geöffnet. Dieser Dialog enthält nun zusätzlich die Operation der Typänderung der Elementdeklaration *e1*, wodurch abschließend das in Abbildung A.99 dargestellte Popup in einem externen Texteditor geöffnet werden kann.

Sowohl die Einführung des Vereinigungstyps zur Umsetzung der Nullwertfähig-

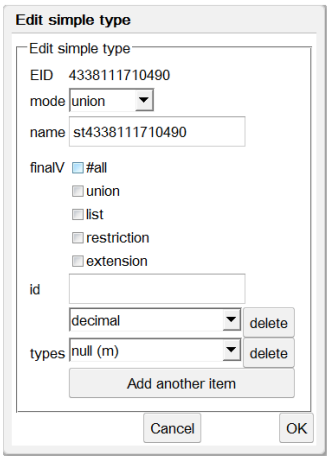


Abbildung 7.23.: Konfigurationsdialog des einfachen Typs mit Nullwertfähigkeit

keit, als auch die Änderung des Typs der Elementdeklaration wurden automatisch durchgeführt und geloggt. Da dies allerdings im Hintergrund geschieht und nicht direkt in der Konsole von CodeX ersichtlich ist, wurde für das obige Beispiel ein Auszug aus der Datenbank extrahiert. Dieser ist in Abbildung A.100 dargestellt.

Es werden die letzten beiden, durch das Einbinden des externen Moduls hinzugefügten, einfachen Typen (*dayTimeDuration* und *dateTimeStamp*) und die anschließende Speicherung (*68 entities transmitted*) gezeigt. Des Weiteren sind sowohl die normale Aktionsmeldung über die Anwendung der Nullwertfähigkeit (*Automatic [..] adaption*), als auch die Einführung des Vereinigungstyps (*add simpletype [..]*) und die Änderung der Elementdeklaration (*update element [..]*) enthalten.

Exportieren eines veränderten XML-Schemas

Nachdem ein XML-Schema importiert und anschließend dessen konzeptuelles Modell angepasst wurde, kann das veränderte Schema exportiert werden (File → Export).[52] Dies ist ausgehend vom Dialog der Abbildung A.101 möglich, in welchem das konzeptuelle Modell *example_out.emx* ausgewählt wurde.[53]

Der Exportdialog konstruiert aus dem gespeicherten EMX ein XML-Schema, wobei Informationen über nicht exportierte Entitäten gegeben werden. In Abbildung A.102 ist dieser Informationsdialog dargestellt. Es werden zum Beispiel standardkonform alle Elementreferenzen nicht exportiert, welche kein Elternelement mit Ausnahme des Schemas besitzen. Dies ist im EMX des Beispiels die Referenz *root*.

Nachdem der Informationsdialog mit dem OK-Button beendet wurde, erscheint das Popup der Abbildung A.103. Die Datei *example_out.xsd*[54] enthält das Ergebnis

[52]Hinweis: Die Anpassung eines konzeptuellen Modells ist keine Voraussetzung für den Export.
[53]Hinweis: Die Option *XSD 1.1 Schema* fügt im exportierten XSD eine Versionsinformation hinzu.
[54]Hinweis: Der Name der exportierten Datei wird analog zum konzeptuellen Modell gewählt.

des Exports. In Abbildung 7.24 ist die Datei im externen Texteditor dargestellt.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:cx="file://codex-null.xsd"
    xmlns:vc="http://www.w3.org/2007/XMLSchema-versioning" vc:minVersion="1.1">
  <xs:import namespace="file://codex-null.xsd" schemaLocation="http://www.ls-dbis.de/codex" />
  <xs:element name="root" type="roottype">
    <!--Possible root.-->
  </xs:element>
  <xs:element name="e1" type="st4338111710490" />
  <xs:element name="e2" type="xs:string" />
  <xs:attribute name="a1" type="xs:decimal" />
  <xs:attribute name="a2" type="xs:string" />
  <xs:complexType name="roottype">
    <xs:choice maxOccurs="2" minOccurs="2">
      <xs:element ref="e1" maxOccurs="2" minOccurs="1" />
      <xs:element ref="e2" maxOccurs="2" minOccurs="0" />
    </xs:choice>
    <xs:attributeGroup ref="roottype.attrGroup" />
  </xs:complexType>
  <xs:attributeGroup name="roottype.attrGroup">
    <xs:attribute ref="a1" use="optional" />
    <xs:attribute ref="a2" use="prohibited" />
  </xs:attributeGroup>
  <xs:simpleType name="st4338111710490">
    <xs:union memberTypes="xs:decimal cx:null" />
  </xs:simpleType>
</xs:schema>
```

Abbildung 7.24.: Ergebnis des Exports des veränderten XML-Schemas

Das Ergebnis des Exports entspricht grundlegend dem XML-Schema des XML-Beispiels 1.3. Es wurden allerdings sowohl Versionsinformationen, als auch das externe Modul zur Anwendbarkeit der Nullwertfähigkeit, der Vereinigungstyp zur Umsetzung der Nullwertfähigkeit und eine Modell-bedingte Attributgruppe hinzugefügt. Des Weiteren wurde durch einen Kommentar vermerkt, welche globale Elementdeklaration auf der Grundlage des konzeptuellen Modells als Wurzelelement für XML-Dokumente dienen kann. Dies ist im Beispiel das Element *root*.

Nachdem mit einem ausführlichen Beispiel die XML-Schemaevolution mit Hilfe von CodeX erläutert wurde, werden im nächsten und letzten Abschnitt des Kapitels bisherige Erweiterungen des Prototyps beschrieben. Diese wurden primär in studentischen Arbeiten konzipiert und gegebenenfalls in CodeX ergänzt.

7.2.5. Weitere Features von CodeX 2.0

Der Forschungsprototyp CodeX wurde über einen längeren Zeitraum hinweg konzipiert und umgesetzt. Zur Ergänzung der Funktionalität wurden Themen am Lehrstuhl für Datenbank- und Informationssysteme der Universität Rostock ausgeschrieben, die in studentischen Arbeiten realisiert wurden. Nachfolgend werden diese Arbeiten in chronologischer Reihenfolge überblicksartig präsentiert.

XML-Editor

Im Projektfenster werden zusätzlich zu den konzeptuellen Modellen (*EMX*) sowohl XML-Schemas (*XSD*) als auch XML-Dokumente (*XML*) angezeigt. Durch den doppelten Linksklick auf eine XML-Datei, wird im Editorfenster ein *XML-Editor* geöffnet. In der Abbildung A.104 ist der XML-Editor mit dessen Quellansicht (*Source*), in Abbildung A.105 mit der Modellansicht (*Model*) dargestellt.

Die Quellansicht unterstützt eine sprachspezifische Syntaxhervorhebung, während die Modellansicht eine kastenbasierte Visualisierung der Komponentenstruktur einer XML-Datei ist. Beide Darstellungsformen wurden ursprünglich Mitte 2012 von Herrn Hannes Grunert als studentische Hilfskraft in CodeX integriert, und bis Ende 2013 erweitert, modifiziert und gegebenenfalls korrigiert.

Der XML-Editor könnte als Alternative zur Nutzung des obigen, externen Texteditors dienen. Dies erfordert allerdings eine entsprechende Weiterentwicklung.

XSD-Matcher

In [Def13] wird eine alternative Variante zur Ermittlung von Schemaänderungen vorgestellt. Statt wie bisher die Nutzerinteraktion zu loggen, werden ein Ausgangs- und Zielschema miteinander verglichen. Die ermittelten Korrespondenzen zwischen den Schemakomponenten werden anschließend in ELaX-Statements umgewandelt. Diese könnten ebenso als Grundlage der XML-Schemaevolution dienen.

Der *XSD-Matcher* wurde in CodeX integriert und ist in der Menüleiste auswählbar (*Matcher*). Es öffnet sich die GUI der Abbildung 7.25. In dieser können ein

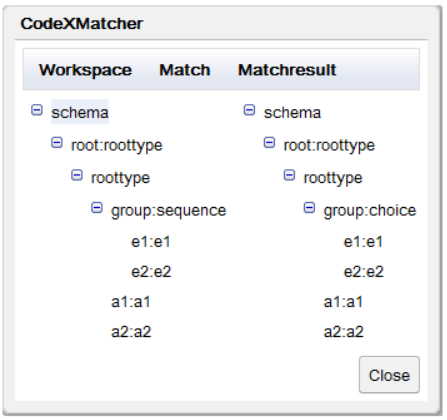


Abbildung 7.25.: Grafisches Frontend des XSD-Matchers von CodeX

Ausgangs- und Zielschema ausgewählt werden (Workspace → Schemas), welche anschließend verglichen werden können (Match → Execute Workflow). Das Ergebnis ist unter anderem eine Liste von ELaX-Statements (Matchresult → ELaX), mit welcher das Ausgangsschema in das Zielschema überführt werden kann.

In der aktuellen Realisierung des XSD-Matchers ist eine veraltete Sprachspezifikation von ELaX verwendet worden. Dies führt zu Inkompatibilitäten mit den vorhandenen Analysealgorithmen von CodeX, sodass eine automatisierte Auswertung verhindern wird. Des Weiteren ist eine endgültige Integration in CodeX noch nicht vollzogen, da unter anderem Schnittstellen zur Speicherung innerhalb der relationalen Strukturen fehlen. Ein weiterer Nachteil des XSD-Matchers ist, dass Expertenwissen beim Vergleich der Schemakomponenten erforderlich ist. Dies gilt besonders bei Umbenennungen und Umsortierungen innerhalb umfangreicherer XML-Schemata. Hier sind die automatisch ermittelten Korrespondenzen nicht hilfreich, sodass vergleichsweise viele Nutzerinteraktionen und Korrekturen notwendig sind. Dies stellt einen Widerspruch zur angestrebten XML-Schemaevolution dar.

Die Erweiterung wird somit in CodeX nicht angewendet, dennoch ist dies ein Ansatz mit hohem Potential. Es könnte nach entsprechenden Anpassungen zum Beispiel ein Vergleich der durch das Logging ermittelten und durch ROfEL optimierten ELaX-Statements mit dem Ergebnis des Matchers erfolgen. Dies wäre eine Evaluierung der Minimalität und Notwendigkeit von ELaX-Statements. Des Weiteren könnte ein Anwender im bevorzugten Tool ein XML-Schema ändern, und anschließend die Adaption der XML-Dokumente mit Hilfe von CodeX durchführen.

Management der Typhierarchie

In [Kap14] wird ein Konzept zum Management der Typhierarchie in CodeX erläutert. Es werden unter anderem Möglichkeiten zur Visualisierung der Hierarchien vorgestellt, welche in der *Toolbar* des EMX-Editors für komplexe Typen über den *ComplexType-Button* und für einfache Typen über den *SimpleType-Button* aufgerufen werden können. Es öffnet sich bei der Betätigung des *Show-current-[..]-type-hierarchy-Buttons* entweder ein *TypeViewer-Dialog* für komplexe (Abbildung A.106) oder einfache Typen (Abbildung A.107).

Die Kompensation der Löschoperation eines Typs wird ebenso in [Kap14] vorgestellt. Ausgehend von der Typhierarchie werden beim Löschen eines Typs Kompensationstypen vorgeschlagen, welche eine Instanzanpassung überflüssig machen und/oder die Folgekosten minimieren. Der entsprechende Dialog ist in Abbildung 7.26 dargestellt. Es wird durch die Löschung des Typs *resS* (vgl. Abbildung A.107) angemerkt, dass die Elementdeklaration *elemResS* und der einfache Typ *unionS* betroffen wären. Somit kann vor der Schemaänderung abgeschätzt werden, ob Folgekosten entstehen oder nicht. Des Weiteren wird als Kompensation die Verwendung des Typs *string* vorgeschlagen, wodurch trotz der Schemaänderung keine Instanzanpassung notwendig ist. Diesbezüglich werden automatisch die betroffenen Entitäten angepasst und die ELaX-Statements erzeugt.

Diese Erweiterung ist vollständig in CodeX integriert worden und wird bei der XML-Schemaevolution angewendet. Vor allem bei Schemaänderungen auf dem konzeptuellen Modell werden die Funktionalitäten eingesetzt, sodass Anwender

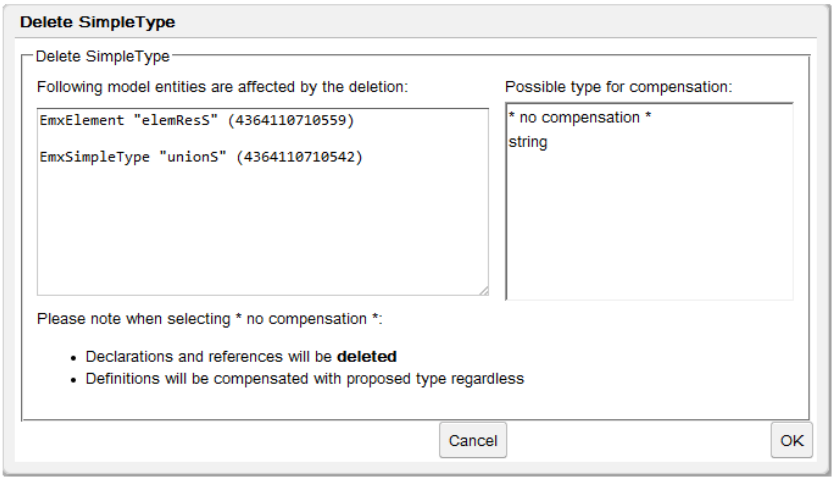


Abbildung 7.26.: Dialog zur Kompensation der Löschung eines einfachen Typs

bei der Modellierung frühzeitig über mögliche Auswirkungen informiert werden.

Abschließende Betrachtung

In diesem Kapitel wurde die letzte Zielsetzung der Arbeit behandelt. Dies ist die Unterstützung von Nicht-Experten bei der hochkomplexen, fehleranfälligen Evolution durch ein geeignetes Tool und sinnvolle Abstraktionen.

Der Prototyp *CodeX* erfüllt diese Anforderung.[55] Die Architektur wurde vorgestellt, sowie eine Einordnung der Ansätze der vorherigen Kapitel getätigt. Des Weiteren wurden sowohl die grafische Benutzeroberfläche, als auch der *EMX-Editor*, die Realisierung des konzeptuellen Modells und Erweiterungen präsentiert. An einem Beispiel ist darüber hinaus die XML-Schemaevolution erläutert worden.

Die vorliegende Arbeit wird im nächsten Kapitel mit der Schlussbetrachtung beendet. Dabei wird die Lösung der Problemstellung abschließend bewertet, bevor im Ausblick auf zukünftige, sinnvolle Erweiterungen eingegangen wird.

[55] siehe auch: These 15

8. Schlussbetrachtung

In diesem Kapitel werden abschließende Betrachtungen zur vorliegenden Arbeit getätigt. Dabei wird in **Abschnitt 8.1** eine Zusammenfassung gegebenen, welche rückblickend auf die Realisierung sowohl der Fragestellungen der Problemstellung, als auch die Zielsetzungen und Schwerpunkte eingeht. In **Abschnitt 8.2** wird anschließend auf zukünftige, sinnvolle Erweiterungen der Arbeit eingegangen.

8.1. Zusammenfassung

In Kapitel 1.1 wurde die zentrale *Problemstellung* der XML-Schemaevolution definiert. Diese beinhaltet primär die Fragestellung, ob und wie es möglich ist, das *Gültigkeitsproblem* zwischen einem sich ändernden XML-Schema und vormals gültigen XML-Dokumenten zu lösen.

Es wurde die These aufgestellt, dass durch die Erfassung, Charakterisierung und Analyse der Schemaänderungen die zur Adaption notwendigen Transformationsschritte automatisch hergeleitet werden können. Diese These beeinflusste die nachfolgend wiederholten *Zielsetzungen* und *Schwerpunkte der Arbeit*.
Zielsetzungen der Arbeit aus Kapitel 1.1.1:

- Spezifikation und Erfassung von Änderungen, die ein Nutzer an einem XML-Schema vornimmt.
- Analyse, Optimierung und Bereinigung der erfassten Änderungen, sowie die weitestgehend automatische Erstellung von daraus resultierenden Transformationsschritten zur Adaption der XML-Dokumente.
- Unterstützung von Nicht-Experten bei der hochkomplexen, fehleranfälligen Evolution durch ein geeignetes Tool und sinnvolle Abstraktionen.

Schwerpunkte der Arbeit aus Kapitel 1.1.2:

- *Änderungen* (Kapitel 4 - Lösungsansatz)
  - Konzeptuelle Modellierung von XML-Schema
  - Verwaltung und Speicherung von Modellen
- *Bestimmung* (Kapitel 5 - Transformationssprache)
  - Spezifikation/Umsetzung von Änderungsoperationen

  - Definition einer Updatesprache und deren Optimierung
  - Logging der Nutzeraktion und deren Auswertung

- *Adaptionen* (Kapitel 6 - Adaption der Instanzen)
  - Automatisierte Erzeugung von Transformationsschritten zur Wahrung und/oder Wiederherstellung der Gültigkeit einer Datenbasis

In Abbildung 8.1 wird der Zusammenhang zwischen den Zielsetzungen und den Schwerpunkten dargestellt. Ein Kreuz (x) markiert jeweils eine Abhängigkeit bzw.

Zielsetzung / Schwerpunkt	1	2	3 (Kap7)
Änderungen (Kap4)	x		x
Bestimmung (Kap5)	x	x	x
Adaptionen (Kap6)		x	x

Abbildung 8.1.: Zusammenhang zwischen Zielsetzungen und Schwerpunkten

einen Zusammenhang. Die erste Zielsetzung wird zum Beispiel durch die Schwerpunkte *Änderungen* und *Bestimmung* realisiert. Somit ist diese primär in den Kapiteln 4 und 5 gelöst worden. Die zweite Zielsetzung wird wiederum durch die *Bestimmung* aber ebenso durch die *Adaptionen* in den Kapiteln 5 und 6 thematisiert. Die dritte Zielsetzung (d.h. die *Unterstützung*) wird in Kapitel 7 behandelt.

Änderungen

Dieser Schwerpunkt wurden primär im Kapitel 4 realisiert und beinhaltet sowohl die konzeptuelle Modellierung von XML-Schema, als auch die Verwaltung und Speicherung von Modellen.

Diesbezüglich wurde das konzeptuelle Modell **EMX** (Entity Model for XML-Schema) als Abstraktion von XML-Schema vorgestellt. Somit entsteht eine **Drei-Ebenen-Architektur** zur Lösung der Problemstellung. Es existiert dabei eine eindeutige *Korrespondenz* zwischen der zusätzlich eingeführten Modellebene und der Schemaebene, sodass Schemaänderungen stellvertretend am EMX durchgeführt werden. Die Vorteile einer konzeptuellen Modellierung, wie diese in Kapitel 4.1 vorgestellt wurden, werden somit bei der XML-Schemaevolution angewendet.

Ausgehend vom EMX wurde eine *logische Struktur* eingeführt, welche die zur Speicherung der Entitäten notwendigen Relationen visualisiert. Diese Struktur wurde für den **Garden-of-Eden-Modellierungsstil** konzipiert, sodass es neben den primären Schemakomponenten des XML-Schemas (d.h. einfache und komplexe Typdefinitionen, sowie Attribut- und Elementdeklarationen) zum Beispiel eine Unterscheidung zwischen einer Elementdeklaration und einer Elementreferenz gibt.

Die Identifikation einer jeden Entität wird durch eine eineindeutige **EID** (EMX ID) realisiert, sodass diese IDs innerhalb eines konzeptuellen Modells zur Beschrei-

bung der vorliegenden Beziehungen verwendet werden. Eine Elementreferenz beinhaltet zum Beispiel die EID der referenzierten Deklaration.

Mit den relationalen Strukturen ist die Verwaltung und Speicherung der Modelle umgesetzt, sodass in Kombination mit der konzeptuellen Modellierung der erste Schwerpunkt durch die vorliegende Arbeit vollständig realisiert wird.

Bestimmung

Der zweite Schwerpunkt wurde primär im Kapitel 5 realisiert und beinhaltet sowohl die Spezifikation und Umsetzung von Änderungsoperationen, als auch die Definition einer Updatesprache und deren Optimierung, sowie das Logging der Nutzeraktionen und deren Auswertung.

Es wurde ausgehend vom EMX die domainspezifische Transformationssprache **ELaX** (Evolution Language for XML-Schema) spezifiziert, welche die in Kapitel 5.1 formulierten Kriterien umsetzt. Mit ELaX werden Änderungsoperationen beschrieben, wobei eine Unterscheidung zwischen dem Hinzufügen (*add*), Löschen (*delete*) und Ändern (*update*) von Entitäten getätigt wird. Des Weiteren enthält ein ELaX-Statement in Abhängigkeit des konzeptuellen Modells eine Liste von Attribut-Wert-Paaren, mit denen eine Änderung entsprechend ausgedrückt wird.

Die ELaX-Statements werden analog zum EMX in relationalen Strukturen gespeichert. Dabei wird unter anderem der Zeitpunkt vermerkt, sodass eine *Historie von Änderungen* aufgebaut wird. Es entsteht ein **Log**, welches mit Hilfe des regelbasierten Algorithmus **ROfEL** (Rule-based Optimizer for ELaX) analysiert wird. ROfEL besteht aus unterschiedlichen Ersetzungsregeln, welche die *Minimierung der Operationsanzahl* durch das Erkennen und Beseitigen von unnötigen, redundanten und ungültigen ELaX-Operationen umsetzen.

Der zweite Schwerpunkt ist somit in Kapitel 5, mit Ausnahme der Auswertung der Nutzeraktionen, ebenso realisiert. Dieser Aspekt ist thematisch eher im dritten Schwerpunkt umgesetzt worden, obwohl mit ROfEL eine Auswertung bereits implizit vollzogen wird. Dies ist wiederum ein Indiz für die Abhängigkeiten der Schwerpunkte untereinander. Der zweite Schwerpunkt wird unter Beachtung der gesamten Arbeit vollständig realisiert.

Adaptionen

Der dritte und letzte Schwerpunkt wird primär im Kapitel 6 realisiert und beinhaltet die automatisierte Erzeugung von Transformationsschritten zur Wahrung und/oder Wiederherstellung der Gültigkeit einer Datenbasis.

Die ELaX-Statements wurden bezüglich deren *Kapazität* und *Informationsgehalt* klassifiziert, sodass eine Auswertung der durch ROfEL optimierten Sequenz von Schemaänderungen umgesetzt wird. Dies entspricht der Realisierung des letzten, noch offenen Aspekts des zweiten Schwerpunkts. Die **Klassifikation** mit deren *Instanz-* und *Folgekosten* ist die Grundlage der Analyse der Auswirkungen von

ELaX-Statements auf die Gültigkeit von Instanzen. Der Ablauf der Analyse wird durch diverse Programmablaufpläne (*PAP*) ausführlich beschrieben.

Sind Instanz- oder Folgekosten nicht auszuschließen, dann wird ausgehend vom EMX die *Lokalisierung von Komponenten* auf Instanzebene vollzogen. Diese unterste Ebene der Drei-Ebenen-Architektur besitzt aufgrund der Optionalität und/oder des Verbots von Entitäten eine mehrdeutige Korrespondenz zur Schemaebene. Daher wird eine Menge von *EID-Ketten* konstruiert, welche anschließend in **Lokalisierungspfade** umgewandelt werden. Diese Pfade sind absolute XPath-Ausdrücke, welche alle gültigen Möglichkeiten des Vorhandenseins einer veränderten Entität innerhalb eines XML-Dokuments beinhalten.

Zusätzlich zu den Lokalisierungspfaden sind für die Transformationsschritte die **Generierung von Informationen** notwendig, insofern zwingende Entitäten ohne explizite Defaultwerte des Schemas auf Instanzebene eingefügt werden müssen. Es wird diesbezüglich unterschieden zwischen einfachen und komplexen Inhalt, sowie dem von Wildcards. Sowohl komplexe Inhalte, als auch die Wildcards werden auf eine minimale Realisierung der notwendigen Informationen reduziert und benötigen letztendlich ebenso einen einfachen Inhalt. Daher wurde eine standardkonforme Erweiterung des XML-Schemas konzipiert, mit welcher unabhängig vom Datentyp ein gültiger, einfacher Inhalt erzeugt werden kann. Die aus der Erweiterung resultierende **Nullwertfähigkeit** bedingt selber keine Instanzanpassung.

Abschließend wurde beschrieben, inwieweit die bis dahin automatisiert erzeugten Transformationsschritte zur Anpassung des **DOM** (Document Object Model) eines XML-Dokuments der Datenbasis verwendet werden. Primär besteht ein Transformationsschritt sowohl aus einer *Existenzbedingung*, als auch dem eventuell notwendigen, generierten Inhalt und kontextabhängigen *Matchbedingungen*. Durch wiederholte Auswertung der Bedingungen wird die Gültigkeit aller XML-Dokumente der Datenbasis geprüft und anschließend gegebenenfalls wieder hergestellt.

Der dritte und letzte Schwerpunkt ist somit durch die vorliegende Arbeit vollständig realisiert, wodurch folgerichtig ebenso die ersten beiden Zielsetzungen erfolgreich umgesetzt sind. Die letzte Zielsetzung, welche von allen bisherigen Schwerpunkten ebenso abhängig ist bzw. diese behandelt, wird nachfolgend thematisiert.

Unterstützung

Die Unterstützung von Nicht-Experten bei der hochkomplexen, fehleranfälligen Evolution durch ein geeignetes Tool und sinnvolle Abstraktionen wurde ausgehend von der Problemstellung als letzte Zielsetzung definiert.

In Kapitel 7 wurde diesbezüglich der Forschungsprototyp **CodeX** (Conceptual design and evolution of XML schemas) vorgestellt. Dieses webbasierte Tool unterstützt die XML-Schemaevolution, indem die Konzepte der Schwerpunkte umgesetzt wurden. CodeX dient dabei als **Demonstrator**, mit welchem die Ansätze und vorgestellten Mechanismen entsprechend evaluiert werden können.

Der **EMX-Editor** des Prototyps ist die wichtigste Komponente zur konzeptuellen Modellierung in CodeX. In einem *Canvas* wird dabei die *Dokument-zentrierte Darstellungsweise* des EMX aufgebaut, verändert oder ergänzt. Der Editor wird *dialogbasiert* bedient, wobei ein Anwender durch diverse Informationsdialoge, Konsolenausgaben und Markierungen (d.h. rote Ausrufezeichen) über die Vollständigkeit und Korrektheit der Modellierung informiert wird.

Die *Nutzerinteraktion* mit CodeX wird geloggt, wobei durch Schemaänderungen im EMX-Editor automatisch ELaX-Statements erzeugt und gespeichert werden. Mit Hilfe eines Prozessdialogs kann ausgehend vom Log die Adaption von Instanzen gesteuert werden. Die entsprechende Transformation beinhaltet sowohl die Anwendung von ROfEL, als auch die Analyse der Auswirkungen, die Lokalisierung von Komponenten und die Generierung von Inhalten. Die standardmäßig aktivierte Nullwertfähigkeit kann durch eine *Nutzerkonfiguration* deaktiviert werden.

Die letzte Zielsetzung ist insofern realisiert, als dass mit CodeX ein Tool existiert, welches Nicht-Experten bei der Evolution entsprechend unterstützt. Allerdings ist hier eine Einschränkung vorzunehmen, wie in Kapitel 7.1.2 dargestellt ist. In diesem werden die vorgestellten Ansätze der Schwerpunkte bezüglich deren Realisierung analysiert. Die finale Anpassung des DOM eines XML-Dokuments ist nicht implementiert, sodass lediglich die Transformationsschritte erzeugt und innerhalb eines Kommentars im XML-Dokument ausgegeben werden. Somit ist die letzte Zielsetzung weitestgehend, aber nicht vollständig realisiert.

Die obige These kann zum Abschluss der Arbeit dennoch positiv beantwortet werden. Durch die Erfassung, Charakterisierung und Analyse der Schemaänderungen können die zur Adaption notwendigen Transformationsschritte automatisch hergeleitet werden. Wie in der vorliegenden Promotionsschrift gezeigt, wird mit den vorgestellten Ansätzen die Problemstellung der XML-Schemaevolution gelöst. Dies entspricht der Lösung des Gültigkeitsproblems im dargestellten Szenario.

8.2. Ausblick

Während der Bearbeitung der vorliegenden Arbeit sind über die Zielstellung hinausgehende Fragestellungen aufgeworfen worden, die in weiterführenden Arbeiten betrachtet werden sollten. Dazu zählen ebenso Teilaspekte vorheriger Kapitel, die im Rahmen dieser Arbeit allerdings nicht umgesetzt wurden.

Allen voran sollte die *Anpassung des DOM* eines XML-Dokuments abschließend implementiert werden. In diesem Zusammenhang könnte ebenso geklärt werden, ob durch *Nutzung weiterführender XML-Technologien* dieser Schritt eventuell erweitert werden kann. Es wäre denkbar, dass aus den Transformationsschritten zum Beispiel XSLT-Skripte (Extensible Stylesheet Language Transformation) erzeugt werden. Damit könnten XML-Dokumente außerhalb von CodeX angepasst werden. Solche Skripte sind allerdings sehr umfangreich und schwer zu warten. Des Weiteren wäre zu untersuchen, ob eine automatisierte Erzeugung möglich ist.

In Kapitel 6.4.4 wurde beschrieben, dass das Ändern von *Constraints* in der XML-Schemaevolution nicht automatisierbar ist. Ein möglicher Ansatz auf einer reduzierten Menge von XPath-Ausdrücken wurde in Kapitel 7.2.3 skizziert. Dieser Ansatz bzw. ähnliche Mechanismen könnten in weiterführenden Arbeiten ebenso betrachtet werden. Es wäre zum Beispiel denkbar, speziell für die Auswertung von XPath-Ausdrücken Algorithmen zu entwickeln bzw. diese aus dritten Quellen zu beziehen und analog zum XSD-Matcher [Def13] in CodeX zu integrieren. Zusätzlich könnte in diesem Zusammenhang die in Kapitel 6.4.1 beschriebene, *Constraint-basierte Wissensgenerierung* nochmals thematisiert werden.

In Kapitel 2.1.1 wurden Substitutionsgruppen vorgestellt, mit welchen Stellvertreter von Elementen definiert werden können. Eine *Integration von Substitutionsgruppen* wäre, obwohl dies wie dargelegt nicht empfohlen wird, ein weiteres Alleinstellungsmerkmal von CodeX. Substitutionsgruppen würden eine zusätzliche Elementhierarchie aufbauen, die unter anderem bei der Lokalisierung von Komponenten entsprechend mit berücksichtigt werden müsste. Dass die nachträgliche Integration einer solchen Hierarchie in CodeX möglich ist, wurde durch das Konzept zum Management der Typhierarchie in [Kap14] erfolgreich gezeigt.

Werden Elementreferenzen oder Elementdeklarationen insofern geändert, dass ein vorhandener, komplexer Typ angepasst wird, dann wird in CodeX der Elementinhalt durch den entsprechend neu generierten ersetzt. Dies ist ein pragmatisches Vorgehen, das durch die *Verwendung eines Matchingverfahrens* verbessert werden könnte. Ein solches Verfahren wurde konzipiert und ist in den Programmablaufplänen der Abbildungen A.46 bis A.53 dargestellt. Ob die Integration dieses Konzepts sinnvoll ist und inwieweit der Lösungsansatz umsetzbar ist, sind ebenso interessante Fragestellungen für weiterführende Arbeiten.

Eine letzte Erweiterung beinhaltet die *Entwicklung eines Kostenmodells*, mit welchem Schemaänderungen und die daraus resultierenden Instanzanpassungen feingranularer ermittelt werden könnten. Diesbezüglich wurden in [Gru11] erste Ansätze entwickelt, allerdings noch nicht unter Berücksichtigung von ELaX. In [Nös15a] wurden darüber hinaus Schemakosten für ELaX-Operationen innerhalb von CodeX zusammengetragen, welche deren Aufwand anhand der Anzahl von internen Vergleichen zur Absicherung der Konsistenz des EMX abschätzen. Dies sind erste Ansätze, welche erweitert und integriert werden könnten. Ein entsprechendes Kostenmodell hätte den Vorteil, dass schon während der Schemaänderungen der Anwender über die Kosten informiert wird. Somit könnte unter anderem auch eine Obergrenze festgelegt werden, bis zu welcher Änderungen zugelassen werden. In diesem Zusammenhang wäre es ebenso denkbar, dass Evolutionsvoreinstellungen getätigt werden, wodurch ausschließlich bspw. kapazitätserhaltende oder -erweiternde, aber immer instanzerhaltende Operationen möglich sind.

Nachdem abschließend auf eine Auswahl zukünftiger, sinnvoller Erweiterungen der Arbeit eingegangen wurde, ist das Kapitel der Schlussbetrachtungen abgeschlossen. Es folgen nun die unterschiedlichen Verzeichnisse und Anhänge.

Literaturverzeichnis

[ABD+89] Atkinson, Malcolm, François Bancilhon, David DeWitt, Klaus Dittrich, David Maier und Stanley Zdonik: *The Object-Oriented Database System Manifesto*, 1989.

[Ada14] Adams, Drew: *Oracle XML DB Developer's Guide 12c Release 1 (12.1) E41152-10*. http://docs.oracle.com/database/121/ADXDB/toc.htm, 2014. Online Dokumentation, Accessed: 2015-03-09.

[Alt15a] Altova GmbH: *Altova DiffDog 2015 Benutzer- und Referenzhandbuch*. http://www.altova.com/de/documents/DiffDogEnt.pdf, 2015.

[Alt15b] Altova GmbH: *Altova XML-Schema-Vergleich*. http://www.altova.com/de/diffdog/xml-schema-diff-tool.html, 2015. Accessed: 2015-05-15.

[Alt15c] Altova GmbH: *Altova XML-Tools*. http://www.altova.com/de/xml_tools.html, 2015. Accessed: 2015-05-15.

[Alt15d] Altova GmbH: *Altova XMLSpy 2015 Enterprise Edition Benutzer- und Referenzhandbuch*. http://www.altova.com/de/documents/XMLSpyEnt.pdf, 2015.

[And08] Andrey Simanovsky: *Data Schema Evolution Support in XML-Relational Database Systems*. Programming and Computer Software, 34(1):16–26, 2008.

[Apa11] Apache Software Foundation: *Apache Xindice*. http://xml.apache.org/xindice/, 2011. Accessed: 2015-05-21.

[Bas15] BaseX Team: *BaseX Documentation Version 8.1*. http://docs.basex.org/wiki/Documentation, March 2015. Accessed: 2015-05-19.

[BBC+10] Berglund, Anders, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie und Jérôme Siméon: *XML Path Language (XPath) 2.0 (Second Edition)*. http://www.w3.org/TR/2010/REC-xpath20-20101214/, 2010. Accessed: 2015-01-25.

[BBGO12] Brahmia, Zouhaier, Rafik Bouaziz, Fabio Grandi und Barbara Oliboni: *A Study of Conventional Schema Versioning in the τXSchema Framework*. Technischer Bericht, TIMECENTER, June 2012.

[BFM+10] Berglund, Anders, Mary Fernández, Ashok Malhotra, Jonathan Marsh, Marton Nagy und Norman Walsh: *XQuery 1.0 and XPath 2.0 Data Model (XDM) (Second Edition)*. `http://www.w3.org/TR/2010/REC-xpath-datamodel-20101214/`, 2010. Accessed: 2015-01-25.

[BGOB14] Brahmia, Zouhaier, Fabio Grandi, Barbara Oliboni und Rafik Bouaziz: *High-level Operations for Changing Temporal Schema, Conventional Schema and Annotations, in the τXSchema Framework*. Technischer Bericht, TIMECENTER, January 2014.

[BL01] Bonifati, Angela und Dongwon Lee: *Technical Survey of XML Schema and Query Languages*, 2001.

[BM04] Biron, Paul V. und Ashok Malhotra: *XML Schema Part 2: Datatypes Second Edition*. `http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/`, October 2004. Accessed: 2015-01-15.

[BNdB04] Bex, Geert Jan, Frank Neven und Jan Van den Bussche: *DTDs Versus XML Schema: A Practical Study*. In: *Proceedings of the 7th International Workshop on the Web and Databases: Colocated with ACM SIGMOD/PODS 2004*, WebDB '04, Seiten 79–84, New York, NY, USA, 2004. ACM.

[Bor11] Born, Matthias: *Überblick: W3C XML Schema 1.1*. `http://www.oio.de/public/xml/w3c-xml-schema-1-1-ueberblick.htm`, 2011. Accessed: 2015-01-15.

[BPSM+08] Bray, Tim, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler und François Yergeau: *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. `http://www.w3.org/TR/2008/REC-xml-20081126/`, November 2008. Accessed: 2015-01-13.

[Bra02] Bray, Tim: *Comments on April XQuery drafts*. `http://lists.w3.org/Archives/Public/public-qt-comments/2002Jul/0007.html`, 2002. Accessed: 2015-06-20.

[Cas09] Castro, Pablo: *Microsoft Developer Network - Evolving schema*. `https://social.msdn.microsoft.com/Forums/en-US/6de984bb-a7b0-4eec-9a9b-95658b76fb02/evolving-schema?forum=dataservices`, March 2009. Forum, Accessed: 2015-04-28.

[Cav09] Cavalieri, Federico: *Querying and Evolution of XML Schemas and Related Documents.* Master Thesis, University of Genova, 2009.

[Cav10] Cavalieri, Federico: *EXup: An Engine for the Evolution of XML Schemas and Associated Documents.* In: *Proceedings of the 2010 EDBT/ICDT Workshops*, EDBT '10, Seiten 21:1–21:10, New York, NY, USA, 2010. ACM.

[CD99] Clark, James und Steve DeRose: *XML Path Language (XPath) Version 1.0.* `http://www.w3.org/TR/1999/REC-xpath-19991116/`, 1999. Accessed: 2015-01-25.

[CGM08a] Cavalieri, Federico, Giovanna Guerrini und Marco Mesiti: *Navigational Path Expressions on XML Schemas.* In: *DEXA*, Seiten 718–726, 2008.

[CGM08b] Cavalieri, Federico, Giovanna Guerrini und Marco Mesiti: *Navigational Path Expressions on XML Schemas.* Technischer Bericht, Dipartimento di Informatica e Scienze dell'Informazione, Universita‘ di Genova, 2008.

[CGM11a] Cavalieri, Federico, Giovanna Guerrini und Marco Mesiti: *Dynamic Reasoning on XML Updates.* In: *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT/ICDT '11, Seiten 165–176, New York, NY, USA, 2011. ACM.

[CGM11b] Cavalieri, Federico, Giovanna Guerrini und Marco Mesiti: *Updates on XML Documents and Schemas.* In: *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on*, Seiten 308–311, April 2011.

[CGM11c] Cavalieri, Federico, Giovanna Guerrini und Marco Mesiti: *Updating XML Schemas and Associated Documents through EXup.* In: *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, Seiten 1320–1323, April 2011.

[CGMO11] Cavalieri, Federico, Giovanna Guerrini, Marco Mesiti und Barbara Oliboni: *On the Reduction of Sequences of XML Document and Schema Update Operations.* In: *ICDE Workshops*, Seiten 77–86, 2011.

[Cla02] Clark, J.: *Relax NG and W3C XML Schema.* `http://www.imc.org/ietf-xml-use/mail-archive/msg00217.html`, 2002. Try to accessed: 2015-06-20; not longer online available.

[CM01] Clark, James und MURATA Makoto: *RELAX NG Specification*. `http://relaxng.org/spec-20011203.html`, December 2001. Accessed: 2015-01-13.

[Cod70] Codd, E. F.: *A Relational Model of Data for Large Shared Data Banks*. Commun. ACM, 13(6):377–387, Juni 1970.

[Cos02] Costello, Roger L.: *XML Schema*. `http://www.xfront.com/xml-schema.html`, 2002. Accessed: 2015-01-20.

[Cos09] Costello, Roger L.: *XML Schema 1.1*. `http://www.xfront.com/xml-schema-1-1/xml-schema-1-1.ppt?v=10`, 2009. Accessed: 2015-01-20.

[CRP09] Cicchetti, Antonio, Davide Di Ruscio und Alfonso Pierantonio: *Managing Dependent Changes in Coupled Evolution*. In: *ICMT*, Seiten 35–51, 2009.

[CSK01] Chang, Ben, Mark Scardina und Stefan Kiritzov: *Oracle 9i XML Handbook*. Mcgraw-Hill Professional, 2001.

[Dat12] Datenbanken Online Lexikon: *Datenbanken: XML-Datenbank*. `http://wikis.gm.fh-koeln.de/wiki_db/Datenbanken/XML-Datenbank`, 2012. Accessed: 2015-05-20.

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

[Def13] Deffke, Jan: *Entwicklung eines Matching- und Mappingverfahres zur Verbesserung der XML-Schemaevolution*. Master Thesis, Universität Rostock, 2013.

[DGGN08] Delima, Neil, Sandy Gao, Michael Glavassevich und Khaled Noaman: *XML Schema 1.1, Part 1: An introduction to XML Schema 1.1*. `http://www.ibm.com/developerworks/xml/library/x-xml11pt1/`, 2008. Accessed: 2015-01-18.

[DGGN09] Delima, Neil, Sandy Gao, Michael Glavassevich und Khaled Noaman: *XML Schema 1.1, Part 2: An introduction to XML Schema 1.1*. `http://www.ibm.com/developerworks/library/x-xml11pt2/`, 2009. Accessed: 2015-01-15.

[DLP+07] Domínguez, Eladio, Jorge Lloret, Beatriz Pérez, Áurea Rodríguez, Angel Luis Rubio und María Antonia Zapata: *A Survey of UML Models to XML Schemas Transformations*. In: *WISE*, Seiten 184–195, 2007.

[DLP+11] Domínguez, Eladio, Jorge Lloret, Beatriz Pérez, Áurea Rodríguez, Angel Luis Rubio und María Antonia Zapata: *Evolution of XML schemas and documents from stereotyped UML class models: A traceable approach.* Information & Software Technology, 53(1):34–50, 2011.

[DLRZ04] Domínguez, Eladio, Jorge Lloret, Angel Luis Rubio und María Antonia Zapata: *Elementary Translations: The Seesaws for Achieving Traceability Between Database Schemata.* In: *Conceptual Modeling for Advanced Application Domains, ER 2004 Workshops CoMoGIS, COMWIM, ECDM, CoMoA, DGOV, and ECOMO, Shanghai, China, November 8-12, 2004, Proceedings*, Seiten 377–389, 2004.

[DLRZ05] Domínguez, Eladio, Jorge Lloret, Angel Luis Rubio und María Antonia Zapata: *Evolving XML Schemas and Documents Using UML Class Diagrams.* In: *DEXA*, Seiten 343–352, 2005.

[DLRZ06] Domínguez, Eladio, Jorge Lloret, Angel Luis Rubio und María Antonia Zapata: *Validation of XML Documents: From UML Models to XML Schemas and XSLT Stylesheets.* In: *Advances in Information Systems, 4th International Conference, ADVIS 2006, Izmir, Turkey, October 18-20, 2006, Proceedings*, Seiten 48–59, 2006.

[DLRZ08] Domínguez, Eladio, Jorge Lloret, Angel Luis Rubio und María Antonia Zapata: *MeDEA: A database evolution architecture with traceability.* Data Knowl. Eng., 65(3):419–441, 2008.

[DT01] Deutsch, Alin und Val Tannen: *Containment and Integrity Constraints for XPath.* In: *Proceedings of the 8th International Workshop on Knowledge Representation meets Databases (KRDB 2001), Rome, Italy, September 15, 2001*, 2001.

[eXi14a] eXist-db Team: *Documentation.* `http://exist-db.org/exist/apps/doc/documentation.xml`, 2014. Accessed: 2015-05-19.

[eXi14b] eXist-db Team: *XML Validation.* `http://exist-db.org/exist/apps/doc/validation.xml`, 2014. Accessed: 2015-05-19.

[eXi14c] eXist-db Team: *XQuery Function Documentation.* `http://exist-db.org/exist/apps/fundocs/view.html`, 2014. Accessed: 2015-05-21.

[eXi14d] eXist-db Team: *XQuery Update Extension.* `http://exist-db.org/exist/apps/doc/update_ext.xml`, 2014. Accessed: 2015-05-19.

[Faj10] Fajt, Stanislav: *Mining XML Integrity Constraints*. Master Thesis, Charles University in Prague, 2010.

[Fak11] Fakhroutdinov, Kirill: *UML Profile Diagrams*. `http://www.uml-diagrams.org/profile-diagrams.html`, 2011. Accessed: 2015-06-01.

[FMN11] Fajt, Stanislav, Irena Mlýnková und Martin Necaský: *On Mining XML Integrity Constraints*. In: *Digital Information Management (ICDIM), 2011 Sixth International Conference on*, Seiten 23–29, Sept 2011.

[FMZ+95] Ferrandina, Fabrizio, Thorsten Meyer, Roberto Zicari, Guy Ferran und Joëlle Madec: *Schema and Database Evolution in the O2 Object Database System*. In: *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland.*, Seiten 170–181, 1995.

[FW04] Fallside, David C. und Priscilla Walmsley: *XML Schema Part 0: Primer Second Edition*. `http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/`, October 2004. Accessed: 2015-01-13.

[Gep02] Geppert, Andreas: *Objektrelationale und objektorientierte Datenbankkonzepte und -systeme*. dpunkt, 2002.

[GLQ08] Genevès, Pierre, Nabil Layaïda und Vincent Quint: *Ensuring Query Compatibility with Evolving XML Schemas*. CoRR, abs/0811.4324, 2008.

[GM08] Guerrini, Giovanna und Marco Mesiti: *X-Evolution: A Comprehensive Approach for XML Schema Evolution*. In: *DEXA Workshops*, Seiten 251–255, 2008.

[GMR05] Guerrini, Giovanna, Marco Mesiti und Daniele Rossi: *Impact of XML Schema Evolution on Valid Documents*. In: *WIDM*, Seiten 39–44, 2005.

[GMR06] Guerrini, Giovanna, Marco Mesiti und Daniele Rossi: *XML Schema Evolution*. Technischer Bericht, University of Genova, 2006.

[GMS07] Guerrini, Giovanna, Marco Mesiti und Matteo Alberto Sorrenti: *XML Schema Evolution: Incremental Validation and Efficient Document Adaptation*. In: Barbosa, Denilson, Angela Bonifati, Zohra Bellahséne, Ela Hunt und Rainer Unland (Herausgeber): *Database and XMLTechnologies*, Band 4704 der Reihe *Lecture Notes in Computer Science*, Seiten 92–106. Springer Berlin Heidelberg, 2007.

[Grü10] Grün, Christian: *Storing and Querying Large XML Instances*. Doktorarbeit, Universität Konstanz, Konstanz, 2010.

[Gru11] Grunert, Hannes: *XML-Schema Evolution: Kategorisierung und Bewertung*. Bachelor Thesis, Universität Rostock, 2011.

[Gru13] Grunert, Hannes: *Integration von Integritätsbedingungen bei der XML-Schemaevolution*. Master Thesis, Universität Rostock, 2013.

[GSMT12] Gao, Shudi (Sandy), C. M. Sperberg-McQueen und Henry S. Thompson: *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. `http://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/`, April 2012. Accessed: 2014-10-29.

[Gue15] Guerrini, Giovanna: *Website - Technical Reports*. `http://www.disi.unige.it/person/GuerriniG/tr.html`, 2015. Accessed: 2015-05-27.

[Har07] Hartung, Michael: *Automatisierte Umsetzung von komplexen XML-Schemaänderungen*. In: *BTW Workshops*, Seiten 64–78, 2007.

[Har11] Hartung, Michael: *Evolution von Ontologien in den Lebenswissenschaften*. Doktorarbeit, Universität Leipzig, `http://dbs.uni-leipzig.de/file/DissHartung_Final.pdf`, 2011.

[Hei10] Heidemann, Julia: *Online Social Networks - Ein sozialer und technischer Überblick*. Informatik-Spektrum, 33(3):262–271, 2010.

[Heu97] Heuer, Andreas: *Objektorientierte Datenbanken: Konzepte, Modelle, Systeme, 2. Auflage*. Addison-Wesley, 1997.

[HH06] Hick, Jean-Marc und Jean-Luc Hainaut: *Database application evolution: A transformational approach*. Data Knowl. Eng., 59(3):534–558, Dezember 2006.

[HHW+04] Hors, Arnaud Le, Philippe Le Hégaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion und Steve Byrne: *Document Object Model (DOM) Level 3 Core Specification*. `http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407`, April 2004. Accessed: 2015-07-13.

[HP09] Hidders, Jan und Jan Paredaens: *XPath/XQuery*. In: Liu, Ling und M. Tamer Özsu (Herausgeber): *Encyclopedia of Database Systems*, Seiten 3659–3665. Springer US, 2009.

[HT07] Hanson, Robert und Adam Tacy: *GWT im Einsatz - AJAX-Anwendungen entwickeln mit dem Google Web Toolkit*. Carl Hanser Verlag GmbH & Co. KG, München, 2007. 567 Seiten.

[HTR11] Hartung, Michael, James Terwilliger und Erhard Rahm: *Schema Matching and Mapping*, Kapitel 6. Recent Advances in Schema and Ontology Evolution. Data-Centric Systems and Applications. Springer-Verlag, 1st Edition Auflage, 1 2011.

[Hul86] Hull, Richard: *Relative Information Capacity of Simple Relational Database Schemata*. SIAM J. Comput., 15(3):856–886, 1986.

[IBM13] IBM: *IBM DB2 10.5 for Linux, UNIX and Windows - pureXML - Handbuch*. `http://www-01.ibm.com/support/knowledgecenter/SSEPGG_10.5.0/com.ibm.db2.luw.xml.doc/doc/c0022308.html?lang=de`, 2013. Online Dokumentation, Accessed: 2015-03-09.

[ISO11a] ISO: *Information technology - Database languages - SQL - Part 01: Framework (SQL/Framework)*. Technischer Bericht ISO/IEC FCD 9075-1, International Organization for Standardization, 2011.

[ISO11b] ISO: *Information technology - Database languages - SQL Part 02: Foundation (SQL/Foundation)*. Technischer Bericht ISO/IEC FCD 9075-2, International Organization for Standardization, 2011. Final Committee Draft of Date 2010-02-05.

[ISO11c] ISO: *Information technology - Database languages - SQL Part 14: XML-Related Specifications (SQL/XML)*. Technischer Bericht ISO/IEC FCD 9075-14, International Organization for Standardization, 2011. Final Committee Draft of Date 2010-02-05.

[ISO11d] ISO: *ISO/IEC 9075-1:2011 - Website*. `http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=53681`, 2011. Accessed: 2015-02-14.

[JTC06] JTC1/SC34: *Information technology - Document Schema Definition Languages (DSDL) - Part 3: Rule-based validation - Schematron*. `http://www.schematron.com/`, Juni 2006. Accessed: 2015-01-13.

[JTC08] JTC1/SC34: *Information technology - Document Schema Definition Language (DSDL) - Part 2: Regular-grammar-based validation - RELAX NG*. `http://standards.iso.org/ittf/PubliclyAvailableStandards/index.html`, December 2008. Accessed: 2015-01-13.

[JW10] Jordan, Casey D und Dale Waldt: *Schema scope: Primer and best practices*. `http://www.ibm.com/developerworks/library/x-schemascope/`, 2010. Accessed: 2015-01-15.

[Kap13] Kaping, Chris: *Transformation von Modellierungsstilen.* Bachelor Thesis, Universität Rostock, 2013.

[Kap14] Kaping, Chris: *Management von Typhierarchien in der XML-Schemaevolution.* Master Thesis, Universität Rostock, 2014.

[KAW01] KAWAGUCHI, Kohsuke: *W3C XML Schema: DOs and DON'Ts.* `http://www.kohsuke.org/xmlschema/XMLSchemaDOsAndDONTs.html`, 2001. Accessed: 2015-01-15.

[Kay01] Kay, Michael H.: *XSLT Programmer's Reference 2nd Edition.* Peer Information, April 2001.

[Kim88] Kim, Hyoung: *Issues in Object-oriented Database Schemas.* Technischer Bericht, University of Texas at Austin, Austin, TX, USA, 1988. Dissertation.

[KK03] Krumbein, Tobias und Thomas Kudrass: *Rule-Based Generation of XML Schemas from UML Class Diagrams.* In: *In Berliner XML Tage 2003*, Seiten 213–227, 2003.

[KKLM09] Klímek, Jakub, Lukás Kopenec, Pavel Loupal und Jakub Malý: *XCase - A Tool for Conceptual XML Data Modeling.* In: *ADBIS (Workshops)*, Seiten 96–103, 2009.

[Kle07a] Klettke, Meike: *Conceptual XML Schema Evolution - the CoDEX approach for Design and Redesign.* In: *BTW Workshops*, Seiten 53–63, 2007.

[Kle07b] Klettke, Meike: *Modellierung, Bewertung und Evolution von XML-Dokumentkollektionen.* Habilitation, Fakultät für Informatik und Elektrotechnik, Universität Rostock, 2007.

[KM03] Klettke, Meike und Holger Meyer: *XML & Datenbanken : Konzepte, Sprachen und Systeme.* xml.bibliothek. dpunkt-Verlag, Heidelberg, 1. Auflage, 2003.

[KMH05] Klettke, Meike, Holger Meyer und Birger Hänsel: *Evolution — The Other Side of the XML Update Coin.* In: *2nd International Workshop on XML Schema and Data Management (XSDM), Tokyo*, April 2005.

[KMN12] Klímek, Jakub, Jakub Malý und Irena Mlýnková Martin Necaský: *eXolutio: Tool for XML Schema and Data Management.* In: *Proceedings of the Dateso 2012 Annual International Workshop on DAtabases, TExts, Specifications and Objects, Zernov, Rovensko pod Troskami, Czech Republic, April 18, 2012*, Seiten 69–80, 2012.

[KN10a] Klímek, Jakub und Martin Necaský: *Integration and Evolution of XML Data via Common Data Model*. In: *EDBT/ICDT Workshops*, 2010.

[KN10b] Klímek, Jakub und Martin Necaský: *Reverse-engineering of XML Schemas: A Survey*. In: *International Workshop on DAtabases, TExts, Specifications and Objects (DATESO 2010)*, Seiten 96–107, 2010.

[KN12] Klímek, Jakub und Martin Necaský: *On Inheritance in Conceptual Modeling for XML*. In: *Proceedings of the 3rd International Conference on Ambient Systems, Networks and Technologies (ANT 2012), the 9th International Conference on Mobile Web Information Systems (MobiWIS-2012), Niagara Falls, Ontario, Canada, August 27-29, 2012*, Seiten 54–61, 2012.

[Kol99] Kolmschlag, Silvia: *Schemaevolution in Föderierten Datenbanksystemen*. Shaker Verlag, August 1999. Dissertation.

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

[KTGH13] Klein, Dominik, Phuoc Tran-Gia und Matthias Hartmann: *Big Data*. Informatik-Spektrum, 36(3):319–323, 2013.

[Lau05] Lausen, Georg: *Datenbanken : Grundlagen und XML-Technologien*. Spektrum Akademischer Verlag, 1. Auflage, 2005.

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

[LHBM05] Leonardi, Erwin, Tran T. Hoai, Sourav S. Bhowmick und Sanjay Madria: *DTD-Diff: A Change Detection Algorithm for DTDs*. Technischer Bericht, Nanyang Technological University, Singapore, 2005.

[LHBM06] Leonardi, Erwin, Tran T. Hoai, Sourav S. Bhowmick und Sanjay Madria: *DTD-Diff: A Change Detection Algorithm for DTDs*. In: Lee, Mong, Kian-Lee Tan und Vilas Wuwongse (Herausgeber): *Database Systems for Advanced Applications*, Band 3882 der Reihe *Lecture Notes in Computer Science*, Seiten 817–827. Springer Berlin Heidelberg, 2006.

[Li99] Li, Xue: *A Survey of Schema Evolution in Object-Oriented Databases*. In: *Proceedings of the 31st International Conference on Technology of Object-Oriented Language and Systems*, TOOLS '99, Seiten 362–, Washington, DC, USA, 1999. IEEE Computer Society.

[Liq14] Liquid: *XML Schema Tutorial, Part 2 : Best Practices, Conventions & Recommendations*. `http://www.liquid-technologies.com/Tutorials/XmlSchemas/XsdTutorial_02.aspx`, 2014. Accessed: 2015-01-15.

[LS04] Lehner, Wolfgang und Harald Schöning: *XQuery : Grundlagen und fortgeschrittene Methoden*. xml.bibliothek. dpunkt-Verlag, Heidelberg, 1. Auflage, 2004.

[LV96] Lausen, Georg und Gottfried Vossen: *Objekt-orientierte Datenbanken - Modelle und Sprachen*. Oldenbourg, 1996.

[Mal02] Maler, Eve: *Schema Design Rules for UBL...and Maybe for You*, 2002.

[Mal10] Malý, Jakub: *XML Schema Evolution*. Master Thesis, Charles University in Prague, 2010.

[Mar03] Marchal, Benoit: *Tip: When to use local and global declarations*. `http://www.ibm.com/developerworks/xml/library/x-tiplocdec/index.html`, 2003. Accessed: 2015-01-24.

[MCSG06] Mesiti, Marco, Roberto Celle, Matteo Alberto Sorrenti und Giovanna Guerrini: *X-Evolution: A System for XML Schema Evolution and Document Adaptation*. In: Ioannidis, Yannis, Marc H. Scholl, Joachim W. Schmidt, Florian Matthes, Mike Hatzopoulos, Klemens Boehm, Alfons Kemper, Torsten Grust und Christian Boehm (Herausgeber): *Advances in Database Technology - EDBT 2006*, Band 3896 der Reihe *Lecture Notes in Computer Science*, Seiten 1143–1146. Springer Berlin Heidelberg, 2006.

[MF10] Mattern, Friedemann und Christian Flörkemeier: *Vom Internet der Computer zum Internet der Dinge*. Informatik-Spektrum, 33(2):107–121, 2010.

[Mic14] Microsoft: *Microsoft SQL Server 2014 XML Data*. `https://msdn.microsoft.com/de-de/library/bb522446.aspx`, 2014. Online Dokumentation, Accessed: 2015-03-09.

[MKMN11] Malý, Jakub, Jakub Klímek, Irena Mlýnková und Martin Necaský: *XML Document Versioning and Revalidation.* In: *Proceedings of the Dateso 2011: Annual International Workshop on DAtabases, TExts, Specifications and Objects, Pisek, Czech Republic, April 20, 2011*, Seiten 49–60, 2011.

[ML05] McAffer, Jeff und Jean-Michel Lemieux: *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java(TM) Applications.* Addison-Wesley Professional, 2005.

[MLMK05] Murata, Makoto, Dongwon Lee, Murali Mani und Kohsuke Kawaguchi: *Taxonomy of XML Schema Languages Using Formal Language Theory.* ACM Trans. Internet Technol., 5(4):660–704, November 2005.

[Mlý15] Mlýnková, Irena: *Website - Publikationen.* `http://www.ksi.mff.cuni.cz/~holubova/publikace.html`, 2015. Accessed: 2015-06-10.

[MML07] Moro, Mirella Moura, Susan Malaika und Lipyeow Lim: *Preserving XML Queries during Schema Evolution.* In: *WWW*, Seiten 1341–1342, 2007.

[MMN11] Malý, Jakub, Irena Mlýnková und Martin Necaský: *XML Data Transformations as Schema Evolves.* In: *ADBIS*, Seiten 375–388, 2011.

[MN09] Mlýnková, Irena und Martin Necaský: *Five-Level Multi-Application Schema Evolution.* In: *DATESO'09: Databases, Texts, Specifications, and Objects*, Seiten 90 – 104, April 2009.

[MN12] Malý, Jakub und Martin Necaský: *XML Document Versioning, Revalidation and Constraints.* In: Harth, Andreas und Nora Koch (Herausgeber): *Current Trends in Web Engineering*, Band 7059 der Reihe *Lecture Notes in Computer Science*, Seiten 317–321. Springer Berlin Heidelberg, 2012.

[MN13] Malý, Jakub und Martin Necaský: *When Grammars do not Suffice: Data and Content Integrity Constraints Verification in XML through a Conceptual Model.* In: *Ninth Asia-Pacific Conference on Conceptual Modelling, APCCM 2013, Adelaide, Australia, January 29-Feburary 1, 2013.*, Seiten 21–30, 2013.

[MNM12] Malý, Jakub, Martin Necaský und Irena Mlýnková: *Efficient adaptation of XML data using a conceptual model.* Information Systems Frontiers, Seiten 1–34, 2012.

[mon11] MONETDB TEAM: *MonetDB/XQuery*. `https://www.monetdb.org/XQuery`, March 2011. Accessed: 2015-05-21.

[Nec09] NECASKÝ, MARTIN: *Conceptual Modeling for XML*. Dissertation, Charles University in Prague, January 2009.

[Nec15a] NECASKÝ, MARTIN: *Website - Publikationen*. `http://www.ksi.mff.cuni.cz/en/publikace.php`, 2015. Accessed: 2015-06-10.

[Nec15b] NECASKÝ, MARTIN: *Website - Technical Reports*. `http://www.ksi.mff.cuni.cz/en/publikace.php?typ=TECHREP`, 2015. Accessed: 2015-06-10.

[NKH12] NÖSINGER, THOMAS, MEIKE KLETTKE und ANDREAS HEUER: *Evolution von XML-Schemata auf konzeptioneller Ebene - Übersicht: Der CodeX-Ansatz zur Lösung des Gültigkeitsproblems*. In: *Proceedings of the 24th GI-Workshop "Grundlagen von Datenbanken 2012", Lübbenau, Germany, May 29 - June 01, 2012*, Seiten 29–34, 2012.

[NKH13a] NÖSINGER, THOMAS, MEIKE KLETTKE und ANDREAS HEUER: *Automatisierte Modelladaptionen durch Evolution - (R)ELaX in the Garden of Eden*. Technischer Bericht CS-01-13, Institut für Informatik, Universität Rostock, Rostock, Germany, January 2013. Published as technical report CS-01-13 under ISSN 0944-5900.

[NKH13b] NÖSINGER, THOMAS, MEIKE KLETTKE und ANDREAS HEUER: *A Conceptual Model for the XML Schema Evolution*. In: *Proceedings of the 25th GI-Workshop "Grundlagen von Datenbanken 2013", Ilmenau, Germany, May 28 - 31, 2013*, Seiten 28–33, 2013.

[NKH13c] NÖSINGER, THOMAS, MEIKE KLETTKE und ANDREAS HEUER: *XML Schema Transformations - The ELaX Approach*. In: *Database and Expert Systems Applications - 24th International Conference, DEXA 2013, Prague, Czech Republic, August 26-29, 2013. Proceedings, Part I*, Seiten 293–302, 2013.

[NKH13d] NÖSINGER, THOMAS, MEIKE KLETTKE und ANDREAS HEUER: *XML Schema Transformations - The ELaX Approach*. Technischer Bericht CS-02-13, Institut für Informatik, Universität Rostock, Rostock, Germany, 2013. Published as technical report CS-02-13 under ISSN 0944-5900.

[NKH14] NÖSINGER, THOMAS, MEIKE KLETTKE und ANDREAS HEUER: *Optimization of Sequences of XML Schema Modifications - The ROfEL Approach*. In: *Proceedings of the 26th GI-Workshop Grundlagen von*

Datenbanken, Bozen-Bolzano, Italy, October 21st to 24th, 2014., Seiten 11–16, 2014.

[NKMM11] Necaský, Martin, Jakub Klímek, Jakub Malý und Irena Mlýnková: *Evolution and change management of XML-based systems.* Journal of Systems and Software, 85(3):683 – 707, 2011. Novel Approaches in the Design and Implementation of Systems/Software Architecture.

[NMKM12] Necaský, Martin, Irena Mlýnková, Jakub Klímek und Jakub Malý: *When conceptual model meets grammar: A dual approach to XML data modeling.* Data Knowl. Eng., 72:1–30, 2012.

[Nös15a] Nösinger, Thomas: *ELaX-Operationen.* siehe: `http://www.noesinger.net/`, Juli 2015. Übersicht der Klassifikation der ELaX-Operationen.

[Nös15b] Nösinger, Thomas: *ELaX (Evolution Language for XML-Schema).* `www.ls-dbis.de/elax`, 2015. Accessed: 2015-07-03.

[Nös15c] Nösinger, Thomas: *Programm Ablauf Pläne (PAP) der Evolution.* siehe: `http://www.noesinger.net/`, Juli 2015. Übersicht von PAPs der Evolution.

[Nös15d] Nösinger, Thomas: *Transformationsschritte der Programm Ablauf Pläne (PAP) der Update-Operationen.* siehe: `http://www.noesinger.net/`, Oktober 2015. Übersicht der Transformationsschritte der Update-Operationen.

[NS03] Neven, Frank und Thomas Schwentick: *XPath Containment in the Presence of Disjunction, DTDs, and Variables.* In: Calvanese, Diego, Maurizio Lenzerini und Rajeev Motwani (Herausgeber): *Database Theory - ICDT 2003*, Band 2572 der Reihe *Lecture Notes in Computer Science*, Seiten 315–329. Springer Berlin Heidelberg, 2003.

[OAS05] OASIS Entity Resolution Technical Committee: *XML Catalogs - OASIS Standard V1.1.* `https://www.oasis-open.org/committees/download.php/14809/xml-catalogs.html`, October 2005. Accessed: 2015-05-21.

[Oba03a] Obasanjo, Dare: *W3C XML Schema Design Patterns: Avoiding Complexity.* `http://msdn.microsoft.com/en-us/library/aa468564.aspx`, 2003. Accessed: 2015-01-15.

[Oba03b] Obasanjo, Dare: *W3C XML Schema Design Patterns: Dealing With Change*. http://msdn.microsoft.com/en-us/library/aa468563.aspx, 2003. Accessed: 2015-01-15.

[Oba03c] Obasanjo, Dare: *XML Schema Design Patterns: Is Complex Type Derivation Unnecessary?* http://msdn.microsoft.com/en-us/library/aa468548.aspx, 2003. Accessed: 2015-01-15.

[OMFB02] Olteanu, Dan, Holger Meuss, Tim Furche und Franqois Bry: *XPath: Looking Forward*. In: Chaudhri, Akmal B., Rainer Unland, Chabane Djeraba und Wolfgang Lindner (Herausgeber): *XML-Based Data Management and Multimedia Engineering - EDBT 2002 Workshops*, Band 2490 der Reihe *Lecture Notes in Computer Science*, Seiten 109–127. Springer Berlin Heidelberg, 2002.

[OMG11] OMG - Object Management Group: *OMG Unified Modeling Language (OMG UML), Superstructure - Version 2.4.1*. http://www.omg.org/spec/UML/2.4.1/Superstructure/, August 2011.

[Ora09] Oracle: *DB2 9.5 pureXML Support - A In Depth Look at DB2 9.5 pureXML*, February 2009.

[Orc07] Orchard, David: *Guide to Versioning XML Languages using new XML Schema 1.1 features*. http://www.w3.org/TR/xmlschema-guide2versioning/, July 2007. Accessed: 2015-01-15.

[PGM+12] Peterson, David, Shudi (Sandy) Gao, Ashok Malhotra, C. M. Sperberg-McQueen und Henry S. Thompson: *W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes*. http://www.w3.org/TR/2012/REC-xmlschema11-2-20120405/, April 2012. Accessed: 2014-10-29.

[RAJB+00] Roddick, John F., Lina Al-Jadir, Leopoldo Bertossi, Marlon Dumas, Florida Estrella, Heidi Gregersen, Kathleen Hornsby, Jens Lufter, Federica Mandreoli, Tomi Männistö, Enric Mayol und Lex Wedemeijer: *Evolution and Change in Data Management - Issues and Directions*. SIGMOD Rec., 29(1):21–25, März 2000.

[RB06] Rahm, Erhard und Philip A. Bernstein: *An Online Bibliography on Schema Evolution*. SIGMOD Record, 35(4):30–31, 2006.

[RCD+11] Robie, Jonathan, Don Chamberlin, Michael Dyck, Daniela Florescu, Jim Melton und Jérôme Siméon:

XQuery Update Facility 1.0. `http://www.w3.org/TR/2011/REC-xquery-update-10-20110317/`, 2011. Accessed: 2015-05-26.

[Rod09a] Roddick, John F.: *Schema Evolution*. In: Liu, Ling und M. Tamer Özsu (Herausgeber): *Encyclopedia of Database Systems*, Seiten 2479–2481. Springer US, 2009.

[Rod09b] Roddick, John F.: *Schema Versioning*. In: Liu, Ling und M. Tamer Özsu (Herausgeber): *Encyclopedia of Database Systems*, Seiten 2499–2502. Springer US, 2009.

[Ron10] Ronald Bourret: *XML Database Products*. `http://www.rpbourret.com/xml/XMLDatabaseProds.htm`, 2010. Accessed: 2015-05-20.

[Rys09] Rys, Michael: *XML Schema*. In: Liu, Ling und M. Tamer Özsu (Herausgeber): *Encyclopedia of Database Systems*, Seiten 3621–3623. Springer US, 2009.

[Sch03] Schöning, Harald: *XML und Datenbanken : Konzepte und Systeme*. Carl Hanser Verlag München Wien, 2003.

[Sch04] Schwentick, Thomas: *XPath Query Containment*. SIGMOD Rec., 33(1):101–109, März 2004.

[Sch05] Schmidhauser, Arno: *Tamino XML-Datenbank*. Technischer Bericht, Software AG, April 2005.

[SDG12] Solimando, Alessandro, Giorgio Delzanno und Giovanna Guerrini: *Static Analysis of XML Document Adaptations*. In: *ER Workshops*, Seiten 57–66, 2012.

[Sed11] Sedna Team: *Sedna Programmer's Guide*. http://sedna.org/onepage/ProgGuide.html, November 2011. Accessed: 2015-05-19.

[SKC+01] Su, Hong, Diane Kramer, Li Chen, Kajal T. Claypool und Elke A. Rundensteiner: *XEM: Managing the Evolution of XML Documents*. In: *RIDE-DM*, Seiten 103–110, 2001.

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

[SS12] Schmidt-Schauss, Manfred: *Reduktionssysteme und Termersetzung*. `http://www.ki.informatik.uni-frankfurt.de/lehre/SS2014/AD/skript/Termers-1.pdf`, 2012. Accessed: 2014-10-29.

[SSH11] Saake, Gunter, Kai-Uwe Sattler und Andreas Heuer: *Datenbanken - Implementierungstechniken (3. Aufl.)*. MITP, 2011.

[SSH13] Saake, Gunter, Kai-Uwe Sattler und Andreas Heuer: *Datenbanken - Konzepte und Sprachen (5. Aufl)*. MITP, 2013.

[SST97] Saake, Gunter, Ingo Schmitt und Can Türker: *Objektdatenbanken - Konzepte, Sprachen, Architekturen*. Informatik Lehrbuch-Reihe. International Thomson, 1997.

[ST01] Salminen, Airi und Frank Wm. Tompa: *Requirements for XML Document Database Systems*. In: *Proceedings of the 2001 ACM Symposium on Document engineering*, DocEng '01, Seiten 85–94, New York, NY, USA, 2001. ACM.

[Ste06] Stephan, Robert: *Entwicklung und Implementierung einer Methode zum konzeptuellen Entwurf von XML-Schemas*. Diplomarbeit, Universität Rostock, 2006.

[TBMM04] Thompson, Henry S., David Beech, Murray Maloney und Noah Mendelsohn: *XML Schema Part 1: Structures Second Edition*. `http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/`, October 2004. Accessed: 2015-01-15.

[TG04] Tan, Marvin B. L. und Angela Goh: *Keeping Pace with Evolving XML-Based Specifications*. In: *EDBT Workshops*, Seiten 280–288, 2004.

[Tre95] Tresch, Markus: *Evolution in Objekt-Datenbanken*, Band 10. Teubner-Texte zur Informatik, 1995. `http://www.amazon.de/Evolution-Objekt-Datenbanken-Teubner-Texte-Informatik/dp/3815420598`.

[vdV02] Vlist, Eric van der: *XML Schema*. O'Reilly & Associates, Sebastopol, CA, 1. Auflage, 2002. `http://docstore.mik.ua/orelly/xml/schema/index.htm`.

[W3C15a] W3C: *World Wide Web Consortium (W3C)*. `http://www.w3.org/`, 2015. Accessed: 2015-01-13.

[W3C15b] W3C: *XML Technology*. `http://www.w3.org/standards/xml/`, 2015. Accessed: 2015-01-13.

[W3C15c] W3C: *XML Technology - Schema*. `http://www.w3.org/standards/xml/schema`, 2015. Accessed: 2015-01-13.

[Whi04] Whitmer, Ray: *Document Object Model (DOM) Level 3 XPath Specification*. http://www.w3.org/TR/2004/NOTE-DOM-Level-3-XPath-20040226, February 2004. Accessed: 2015-10-15.

[Wik14] Wikipedia: *Sedna (database) — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Sedna_(database)&oldid=603817204, 2014. Accessed: 2015-05-21.

[Wik15] Wikipedia: *XML-Datenbank — Wikipedia, Die freie Enzyklopädie*. http://de.wikipedia.org/w/index.php?title=XML-Datenbank&oldid=140917029, 2015. Accessed: 2015-05-20.

[XLWB09] Xu, Liang, Tok Wang Ling, Huayu Wu und Zhifeng Bao: *DDE: From Dewey to a Fully Dynamic XML Labeling Scheme*. In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, Seiten 719–730, New York, NY, USA, 2009. ACM.

[xml01] xml-dev list group: *XML Schemas: Best Practices*. http://www.xfront.com/BestPracticesHomepage.html, 2001. Accessed: 2015-01-15.

[XML15] XML Research Group - XRG: *Website - Publikationen*. http://www.ksi.mff.cuni.cz/xrg/publications/, 2015. Accessed: 2015-06-10.

Abbildungsverzeichnis

Quellcode und Dateien

A. Anhang

<table>
<tr><th>#</th><th>Name</th><th>length</th><th>minLength</th><th>maxLength</th><th>pattern</th><th>enumeration</th><th>whiteSpace</th><th>maxInclusive</th><th>maxExclusive</th><th>minInclusive</th><th>minExclusive</th><th>totalDigits</th><th>fractionDigits</th><th>Assertions</th><th>explicitTimezone</th><th>#</th><th>pattern</th><th></th></tr>
<tr><td>1</td><td>string</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>R p</td><td></td><td></td><td></td><td></td><td></td><td></td><td>x</td><td></td><td>1</td><td>#x9 | #xA | #xD | [#x20-#xD7FF] | [#xE000-#xFFFD] | [#x10000-#x10FFFF]</td><td></td></tr>
<tr><td>2</td><td>decimal</td><td></td><td></td><td></td><td>x</td><td>x</td><td>f c</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td></td><td>2</td><td>(\\+|-)?([0-9]+(\\.[0-9]+)?)</td><td></td></tr>
<tr><td>3</td><td>integer</td><td></td><td></td><td></td><td>R</td><td>x</td><td>f c</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>f 0</td><td>x</td><td></td><td>3</td><td>[\-+]?[0-9]+</td><td></td></tr>
<tr><td>4</td><td>float</td><td></td><td></td><td></td><td>x</td><td>x</td><td>f c</td><td>x</td><td>x</td><td>x</td><td>x</td><td></td><td></td><td>x</td><td></td><td>4</td><td>(\+|-)?([0-9]+(\.[0-9]*)?|\.[0-9]+)([Ee](\+|-)?[0-9]+)?|(\+|-)?INF|NaN</td><td></td></tr>
<tr><td>5</td><td>boolean</td><td></td><td></td><td></td><td>x</td><td></td><td>f c</td><td></td><td></td><td></td><td></td><td></td><td></td><td>x</td><td></td><td>5</td><td>'true' | 'false' | '1' | '0'</td><td></td></tr>
<tr><td>6</td><td>anyURI</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>f c</td><td></td><td></td><td></td><td></td><td></td><td></td><td>x</td><td></td><td>6</td><td>\\b(https?|ftp|file)://[-a-zA-Z0-9+&@#/%?=~_|!:,.;]*[-a-zA-Z0-9+&@#/%=~_|]</td><td></td></tr>
<tr><td>7</td><td>QName</td><td>~~x~~</td><td>x</td><td>x</td><td>x</td><td>x</td><td>f c</td><td></td><td></td><td></td><td></td><td></td><td></td><td>x</td><td></td><td>7</td><td>(NCName ':' NCName) | NCName</td><td></td></tr>
<tr><td>8</td><td>NCName</td><td>x</td><td>x</td><td>x</td><td>R</td><td>x</td><td>R c</td><td></td><td></td><td></td><td></td><td></td><td></td><td>x</td><td></td><td>8</td><td>\i\c* ∩ [\i-[:]][\c-[:]]*</td><td>[a-zA-Z_][a-zA-Z0-9._-]*</td></tr>
<tr><td>9</td><td>gMonth</td><td></td><td></td><td></td><td>x</td><td>x</td><td>f c</td><td>x</td><td>x</td><td>x</td><td>x</td><td></td><td></td><td>x</td><td>R op</td><td>9</td><td>--(0[1-9]|1[0-2])(Z|(\+|-)((0[0-9]|1[0-3]):[0-5][0-9]|14:00))?</td><td></td></tr>
<tr><td>10</td><td>gDay</td><td></td><td></td><td></td><td>x</td><td>x</td><td>f c</td><td>x</td><td>x</td><td>x</td><td>x</td><td></td><td></td><td>x</td><td>R op</td><td>10</td><td>---(0[1-9]|[12][0-9]|3[01])(Z|(\+|-)((0[0-9]|1[0-3]):[0-5][0-9]|14:00))?</td><td></td></tr>
<tr><td>11</td><td>gMonthDay</td><td></td><td></td><td></td><td>x</td><td>x</td><td>f c</td><td>x</td><td>x</td><td>x</td><td>x</td><td></td><td></td><td>x</td><td>R op</td><td>11</td><td>--(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))?</td><td></td></tr>
<tr><td>12</td><td>gYear</td><td></td><td></td><td></td><td>x</td><td>x</td><td>f c</td><td>x</td><td>x</td><td>x</td><td>x</td><td></td><td></td><td>x</td><td>R op</td><td>12</td><td>-?([1-9][0-9]{3,}|0[0-9]{3})(Z|(\+|-)((0[0-9]|1[0-3]):[0-5][0-9]|14:00))?</td><td></td></tr>
<tr><td>13</td><td>gYearMonth</td><td></td><td></td><td></td><td>x</td><td>x</td><td>f c</td><td>x</td><td>x</td><td>x</td><td>x</td><td></td><td></td><td>x</td><td>R op</td><td>13</td><td>-?([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))?</td><td></td></tr>
<tr><td>14</td><td>date</td><td></td><td></td><td></td><td>x</td><td>x</td><td>f c</td><td>x</td><td>x</td><td>x</td><td>x</td><td></td><td></td><td>x</td><td>R op</td><td>14</td><td>-?([1-9][0-9]{3,}|0[0-9]{3})-(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))?</td><td></td></tr>
<tr><td>15</td><td>time</td><td></td><td></td><td></td><td>x</td><td>x</td><td>f c</td><td>x</td><td>x</td><td>x</td><td>x</td><td></td><td></td><td>x</td><td>R op</td><td>15</td><td>(([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))?</td><td></td></tr>
<tr><td>16</td><td>dateTime</td><td></td><td></td><td></td><td>x</td><td>x</td><td>f c</td><td>x</td><td>x</td><td>x</td><td>x</td><td></td><td></td><td>x</td><td>R op</td><td>16</td><td>-?([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]+)? **^1**</td><td></td></tr>
<tr><td>17</td><td>duration</td><td></td><td></td><td></td><td>x</td><td>x</td><td>f c</td><td>x</td><td>x</td><td>x</td><td>x</td><td></td><td></td><td>x</td><td></td><td>17</td><td>-?P[0-9]+Y?([0-9]+M)?([0-9]+D)?(T([0-9]+H)?([0-9]+M)?([0-9]+(\.[0-9]+)?S)?)?</td><td></td></tr>
<tr><td>18</td><td>base64Binary</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>f c</td><td></td><td></td><td></td><td></td><td></td><td></td><td>x</td><td></td><td>18</td><td>((([A-Za-z0-9+/] ?){4})*(([A-Za-z0-9+/] ?){3}[A-Za-z0-9+/]|([A-Za-z0-9+/] ?){2}[AEIMQUYcgkosw048] ?=| **^2**</td><td></td></tr>
<tr><td>19</td><td>hexBinary</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>f c</td><td></td><td></td><td></td><td></td><td></td><td></td><td>x</td><td></td><td>19</td><td>([0-9a-fA-F]{2})*</td><td></td></tr>
<tr><td>20</td><td>double</td><td></td><td></td><td></td><td>x</td><td>x</td><td>f c</td><td>x</td><td>x</td><td>x</td><td>x</td><td></td><td></td><td>x</td><td></td><td>20</td><td>(\+|-)?([0-9]+(\.[0-9]*)?|\.[0-9]+)([Ee](\+|-)?[0-9]+)? |(\+|-)?INF|NaN</td><td></td></tr>
<tr><td>21</td><td>NOTATION</td><td>~~x~~</td><td>x</td><td>x</td><td>x</td><td>x</td><td>f c</td><td></td><td></td><td></td><td></td><td></td><td></td><td>x</td><td></td><td>21</td><td>Menge der QNamen</td><td></td></tr>
<tr><td>22</td><td>normalizedString</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>R r</td><td></td><td></td><td></td><td></td><td></td><td></td><td>x</td><td></td><td>22</td><td>[#x20-#xD7FF] | [#xE000-#xFFFD] | [#x10000-#x10FFFF]</td><td></td></tr>
<tr><td>23</td><td>token</td><td>x</td><td>x</td><td>x</td><td>x</td><td>x</td><td>R c</td><td></td><td></td><td></td><td></td><td></td><td></td><td>x</td><td></td><td>23</td><td>[a-zA-Z0-9.:_-]+</td><td></td></tr>
<tr><td>24</td><td>language</td><td>x</td><td>x</td><td>x</td><td>R</td><td>x</td><td>R c</td><td></td><td></td><td></td><td></td><td></td><td></td><td>x</td><td></td><td>24</td><td>[a-zA-Z]{1,8}(-[a-zA-Z0-9]{1,8})*</td><td></td></tr>
<tr><td>25</td><td>Name</td><td>x</td><td>x</td><td>x</td><td>R</td><td>x</td><td>R c</td><td></td><td></td><td></td><td></td><td></td><td></td><td>x</td><td></td><td>25</td><td>\i\c*</td><td>[a-zA-Z_:][a-zA-Z0-9.:_-]*</td></tr>
<tr><td>26</td><td>ID</td><td>x</td><td>x</td><td>x</td><td>R</td><td>x</td><td>R c</td><td></td><td></td><td></td><td></td><td></td><td></td><td>x</td><td></td><td>26</td><td>\i\c* ∩ [\i-[:]][\c-[:]]*</td><td>[a-zA-Z_][a-zA-Z0-9._-]*</td></tr>
<tr><td>27</td><td>IDREF</td><td>x</td><td>x</td><td>x</td><td>R</td><td>x</td><td>R c</td><td></td><td></td><td></td><td></td><td></td><td></td><td>x</td><td></td><td>27</td><td>\i\c* ∩ [\i-[:]][\c-[:]]*</td><td>[a-zA-Z_][a-zA-Z0-9._-]*</td></tr>
<tr><td>28</td><td>IDREFS</td><td>x</td><td>R 1</td><td>x</td><td>x</td><td>x</td><td>R c</td><td></td><td></td><td></td><td></td><td></td><td></td><td>x</td><td></td><td>28</td><td>Liste von IDREF</td><td></td></tr>
<tr><td>29</td><td>ENTITY</td><td>x</td><td>x</td><td>x</td><td>R</td><td>x</td><td>R c</td><td></td><td></td><td></td><td></td><td></td><td></td><td>x</td><td></td><td>29</td><td>\i\c* ∩ [\i-[:]][\c-[:]]*</td><td>[a-zA-Z_][a-zA-Z0-9._-]*</td></tr>
<tr><td>30</td><td>ENTITIES</td><td>x</td><td>R 1</td><td>x</td><td>x</td><td>x</td><td>R c</td><td></td><td></td><td></td><td></td><td></td><td></td><td>x</td><td></td><td>30</td><td>Liste von ENTITY</td><td></td></tr>
<tr><td>31</td><td>NMTOKEN</td><td>x</td><td>x</td><td>x</td><td>R</td><td>x</td><td>R c</td><td></td><td></td><td></td><td></td><td></td><td></td><td>x</td><td></td><td>31</td><td>\c+</td><td>[a-zA-Z0-9.:_-]+</td></tr>
<tr><td>32</td><td>NMTOKENS</td><td>x</td><td>R 1</td><td>x</td><td>x</td><td>x</td><td>R c</td><td></td><td></td><td></td><td></td><td></td><td></td><td>x</td><td></td><td>32</td><td>Liste von NMTOKEN</td><td></td></tr>
<tr><td>33</td><td>nonPositiveInteger</td><td></td><td></td><td></td><td>R</td><td>x</td><td>f c</td><td>R 0</td><td>x</td><td>x</td><td>x</td><td>x</td><td>f 0</td><td>x</td><td></td><td>33</td><td>[\-+]?[0-9]+</td><td>**variety = list -> "whiteSpace = collapse + fixed = true"**</td></tr>
<tr><td>34</td><td>negativeInteger</td><td></td><td></td><td></td><td>R</td><td>x</td><td>f c</td><td>R -1</td><td>x</td><td>x</td><td>x</td><td>x</td><td>f 0</td><td>x</td><td></td><td>34</td><td>[\-+]?[0-9]+</td><td></td></tr>
<tr><td>35</td><td>long</td><td></td><td></td><td></td><td>R</td><td>x</td><td>f c</td><td>R</td><td>x</td><td>R</td><td>x</td><td>x</td><td>f 0</td><td>x</td><td></td><td>35</td><td>[\-+]?[0-9]+</td><td>maxInclusive = 9223372036854775807 minInclusive = -9223372036854775808</td></tr>
<tr><td>36</td><td>int</td><td></td><td></td><td></td><td>R</td><td>x</td><td>f c</td><td>R</td><td>x</td><td>R</td><td>x</td><td>x</td><td>f 0</td><td>x</td><td></td><td>36</td><td>[\-+]?[0-9]+</td><td>maxInclusive = 2147483647 minInclusive = -2147483648</td></tr>
<tr><td>37</td><td>short</td><td></td><td></td><td></td><td>R</td><td>x</td><td>f c</td><td>R</td><td>x</td><td>R</td><td>x</td><td>x</td><td>f 0</td><td>x</td><td></td><td>37</td><td>[\-+]?[0-9]+</td><td>maxInclusive = 32767 minInclusive = -32768</td></tr>
<tr><td>38</td><td>byte</td><td></td><td></td><td></td><td>R</td><td>x</td><td>f c</td><td>R</td><td>x</td><td>R</td><td>x</td><td>x</td><td>f 0</td><td>x</td><td></td><td>38</td><td>[\-+]?[0-9]+</td><td>maxInclusive = 127 minInclusive = -128</td></tr>
<tr><td>39</td><td>nonNegativeInteger</td><td></td><td></td><td></td><td>R</td><td>x</td><td>f c</td><td>x</td><td>x</td><td>R 0</td><td>x</td><td>x</td><td>f 0</td><td>x</td><td></td><td>39</td><td>[\-+]?[0-9]+</td><td></td></tr>
<tr><td>40</td><td>unsignedLong</td><td></td><td></td><td></td><td>R</td><td>x</td><td>f c</td><td>R</td><td>x</td><td>R 0</td><td>x</td><td>x</td><td>f 0</td><td>x</td><td></td><td>40</td><td>[\-+]?[0-9]+</td><td>maxInclusive = 18446744073709551615</td></tr>
<tr><td>41</td><td>unsignedInt</td><td></td><td></td><td></td><td>R</td><td>x</td><td>f c</td><td>R</td><td>x</td><td>R 0</td><td>x</td><td>x</td><td>f 0</td><td>x</td><td></td><td>41</td><td>[\-+]?[0-9]+</td><td>maxInclusive = 4294967295</td></tr>
<tr><td>42</td><td>unsignedShort</td><td></td><td></td><td></td><td>R</td><td>x</td><td>f c</td><td>R</td><td>x</td><td>R 0</td><td>x</td><td>x</td><td>f 0</td><td>x</td><td></td><td>42</td><td>[\-+]?[0-9]+</td><td>maxInclusive = 65535</td></tr>
<tr><td>43</td><td>unsignedByte</td><td></td><td></td><td></td><td>R</td><td>x</td><td>f c</td><td>R</td><td>x</td><td>R 0</td><td>x</td><td>x</td><td>f 0</td><td>x</td><td></td><td>43</td><td>[\-+]?[0-9]+</td><td>maxInclusive = 255</td></tr>
<tr><td>44</td><td>positiveInteger</td><td></td><td></td><td></td><td>R</td><td>x</td><td>f c</td><td>x</td><td>x</td><td>R 1</td><td>x</td><td>x</td><td>f 0</td><td>x</td><td></td><td>44</td><td>[\-+]?[0-9]+</td><td></td></tr>
<tr><td>45</td><td>yearMonthDuration</td><td></td><td></td><td></td><td>R</td><td>x</td><td>f c</td><td>x</td><td>x</td><td>x</td><td>x</td><td></td><td></td><td>x</td><td></td><td>45</td><td>[^DT]*</td><td>^1 |(24:00:00(\.0+)?))(Z|(\+|-)((0[0-9]|1[0-3]):[0-5][0-9]|14:00))?</td></tr>
<tr><td>46</td><td>dayTimeDuration</td><td></td><td></td><td></td><td>R</td><td>x</td><td>f c</td><td>x</td><td>x</td><td>x</td><td>x</td><td></td><td></td><td>x</td><td></td><td>46</td><td>[^YM]*[DT].*</td><td>^2 [A-Za-z0-9+/] ?[AQgw] ?= ?=))?</td></tr>
<tr><td>47</td><td>dateTimeStamp</td><td></td><td></td><td></td><td>x</td><td>x</td><td>f c</td><td>x</td><td>x</td><td>x</td><td>x</td><td></td><td></td><td>x</td><td>f req</td><td>47</td><td>.*(Z|(\+|-)[0-9][0-9]:[0-9][0-9])</td><td></td></tr>
</table>

req required R restricted
pr prohibited
op optional
c collapse f fixed
r replace
p preserve
c > r > p req > pr > op
~~deprecated~~

Abbildung A.1.: Built-in-Typen mit Facetten gemäß [PGM+12]

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="root" type="roottype"/>
    <xs:element name="e1" type="xs:string"></xs:element>
    <xs:element name="e2" type="xs:string"/>
    <xs:attribute name="a1" type="xs:string"></xs:attribute>
    <xs:attribute name="a2" type="xs:string"/>
    <xs:attribute name="a3" type="xs:string"/>
    <xs:complexType name="roottype">
        <xs:sequence>
            <xs:element ref="e1" minOccurs="1" maxOccurs="2"/>
            <xs:element ref="e2" minOccurs="0" maxOccurs="2"/>
        </xs:sequence>
        <xs:attribute ref="a1" use="required"/>
        <xs:attribute ref="a2" use="optional"/>
        <xs:attribute ref="a3" use="prohibited"></xs:attribute>
    </xs:complexType>
</xs:schema>
```

XML-Beispiel A.1: Ausgangsschema für das Altova DiffDog-Mapping

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="root" type="roottype"/>
    <xs:element name="e1" type="xs:decimal"/>
    <xs:element name="e2" type="xs:string"/>
    <xs:attribute name="a1" type="xs:decimal"/>
    <xs:attribute name="a2" type="xs:string"/>
    <xs:attribute name="a3" type="xs:string"/>
    <xs:complexType name="roottype">
        <xs:choice>
            <xs:element ref="e1" minOccurs="1" maxOccurs="2"/>
            <xs:element ref="e2" minOccurs="0" maxOccurs="2"/>
        </xs:choice>
        <xs:attribute ref="a1" use="required"/>
        <xs:attribute ref="a2" use="optional"/>
        <xs:attribute ref="a3" use="required"/>
    </xs:complexType>
</xs:schema>
```

XML-Beispiel A.2: Zielschema für das Altova DiffDog-Mapping (Unterschiede zu dem XML-Schema aus XML-Beispiel A.1 sind rot gekennzeichnet)

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
    <xsl:output method="xml" encoding="UTF-8" indent="yes"/>
    <xsl:template match="/root">
        <root>
            <xsl:attribute name="a1">
                <xsl:value-of select="@a1"/>
            </xsl:attribute>
            <xsl:attribute name="a2">
                <xsl:value-of select="@a2"/>
            </xsl:attribute>
            <xsl:for-each select="e1">
                <e1>
                    <xsl:value-of select="."/>
                </e1>
            </xsl:for-each>
            <xsl:for-each select="e2">
                <e2>
                    <xsl:value-of select="."/>
                </e2>
            </xsl:for-each>
        </root>
    </xsl:template>
    <xsl:template match="e1">
        <e1/>
    </xsl:template>
    <xsl:template match="e2">
        <e2/>
    </xsl:template>
</xsl:stylesheet>
```

XML-Beispiel A.3: XSLT-Dokument durch Mapping von Altova DiffDog des Ausgangs- (XML-Beispiel A.1) und Zielschemas (XML-Beispiel A.2)

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="root" type="roottype"/>
    <xs:element name="e1" type="xs:decimal"/>
    <xs:element name="e2" type="xs:string"/>
    <xs:complexType name="roottype">
        <xs:sequence minOccurs="1" maxOccurs="2">
            <xs:element ref="e1" minOccurs="0" maxOccurs="42"/>
        </xs:sequence>
    </xs:complexType>
    <xs:element name="e3" type="xs:string"/>
</xs:schema>
```

XML-Beispiel A.4: XML-Schema des XML-Beispiels 5.1 nach Anwendung von ELaX

```
XSUpdExpr        ::= "UPDATE SCHEMA" ObjectSpec UpdateSpec AdaptSpec?
ObjectSpec       ::= "('" SchemaURI "')" SinglePath
UpdateSpec       ::= STypeUpdSpec | InsertSpec | ChangeSpec |
                          RemoveSpec | MoveSpec | MigrateSpec

STypeUpdSpec     ::= "CHANGE BASE TYPE TO" QName |
                     "ADD RESTRICTIONS" RestrictionSpecs |
                     "REPLACE RESTRICTIONS" RestrictionNames
                          "WITH" RestrictionSpecs |
                     "REMOVE RESTRICTIONS" RestrictionNames |
                     "ADD MEMBER" TypeName MemberPosSpec |
                     "CHANGE MEMBER AT" PositiveInteger "TO" TypeName |
                     "REPLACE MEMBERS WITH" MemberNames |
                     "REMOVE MEMBER AT" PositiveInteger

InsertSpec       ::= "INSERT TYPE {" XMLTypeDef "}" InsPosSpec?  |
                     "INSERT ELEMENT" ("REFERENCING" QName CardSpec?)  |
                           ((QName "OF TYPE" QName CardSpec?)  |
                          XMLElemDef) InsPosSpec?  |
                     "INSERT ATTRIBUTE" (( "{" XMLAttrDef "}")|
                          (QName "OF TYPE" QName "REQUIRED"?))
                           InsPosSpec?  |
                     "INSERT OPERATOR" ((OperatorType CardSpec?)  |
                          "{" XmlOpDef "}") InsPosSpec?  |
                     "INSERT SUBSTRUCTURE" "{" XMLDef "}" InsPosSpec?

ChangeSpec       ::= "CHANGE TYPE TO" (QName | "{" XMLTypeDef"}" |
                          ("REFERENCING" QName)) |
                     "CHANGE CARDINALITY TO" (CardSpec | "DEFAULT") |
                     "CHANGE OPERATOR TO" OperatorType CardSpec?  |
                     "RENAME TO" QName |

RemoveSpec       ::= "REMOVE" ("CASCADE" | "WITHOUT SUBSTRUCTURE")?

MoveSpec         ::= "MOVE TO" XSchemaPathExpr InsPosSpec?
MigrateSpec      ::= "MIGRATE" ( "GLOBAL TO LOCAL" |
                          ("LOCAL TO GLOBAL ("TYPE" | "ELEMENT")
                           QName)) |

OperatorType     ::= "SEQUENCE" | "CHOICE" | "ALL"
RestrictionNames ::= RestrictionName ("," RestrictionNames)?
RestrictionSpecs ::= RestrictionName "=" Value ("," RestrictionSpecs)?
MemberNames      ::= QName ("," MemberNames)?
MemberPosSpec    ::= "AS" ("FIRST" | "LAST") | "AT" PositiveInteger
InsPosSpec       ::= "AS" ("FIRST" | "LAST") "CHILD"| ("BEFORE" |
                           "AFTER") SimpleStep | "AT" PositiveInteger
CardSpec         ::= "(" MinCard "," MaxCard ")"
MinCard          ::= "0" | PositiveInteger
MaxCard          ::= PositiveInteger | "UNBOUNDED"
```

Abbildung A.2.: XSchemaUpdate-Spezifikation nach [Cav09]

UML Element	XML Schema
class	element, complex type, with ID attribute, and key
abstract class	abstract element and complex type, with ID attribute
attribute	subelement of the corresponding class complex type
stereotype	attribute of the corresponding element
package	element without attributes
association aggregation	reference element, with IDREF attribute referencing the associated class and keyref for type safety (key/keyref references)
association class	association class element and an additional IDREF references to the association class element and a keyref in the corresponding reference elements in the associated classes
qualified association	extension of the reference element, keyref and key of the target class with the qualified attributes
composition	reference element, with subordinated class elem. (hierarch. rel.)
generalization	complex type of the subclass is defined as an extension of the complex type of the superclass
association constraint	currently not mapped
n-ary association	association element with IDREF references to all associated classes (resolution of the n-ary association)

Abbildung A.3.: Mapping von UML-Elementen zu XML-Schema aus [KK03]

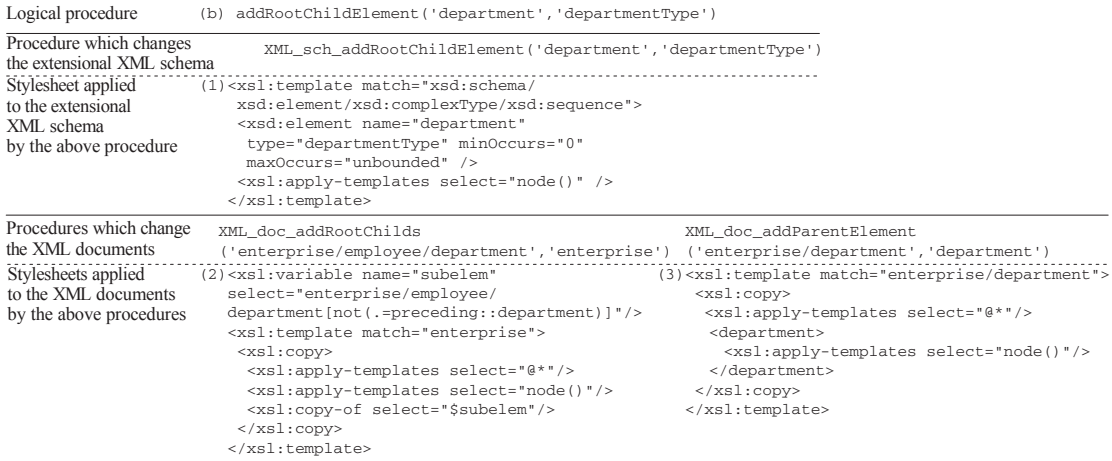


Abbildung A.4.: XSLT-Stylesheets zur Anpassung der textuellen Struktur aus [DLRZ05]

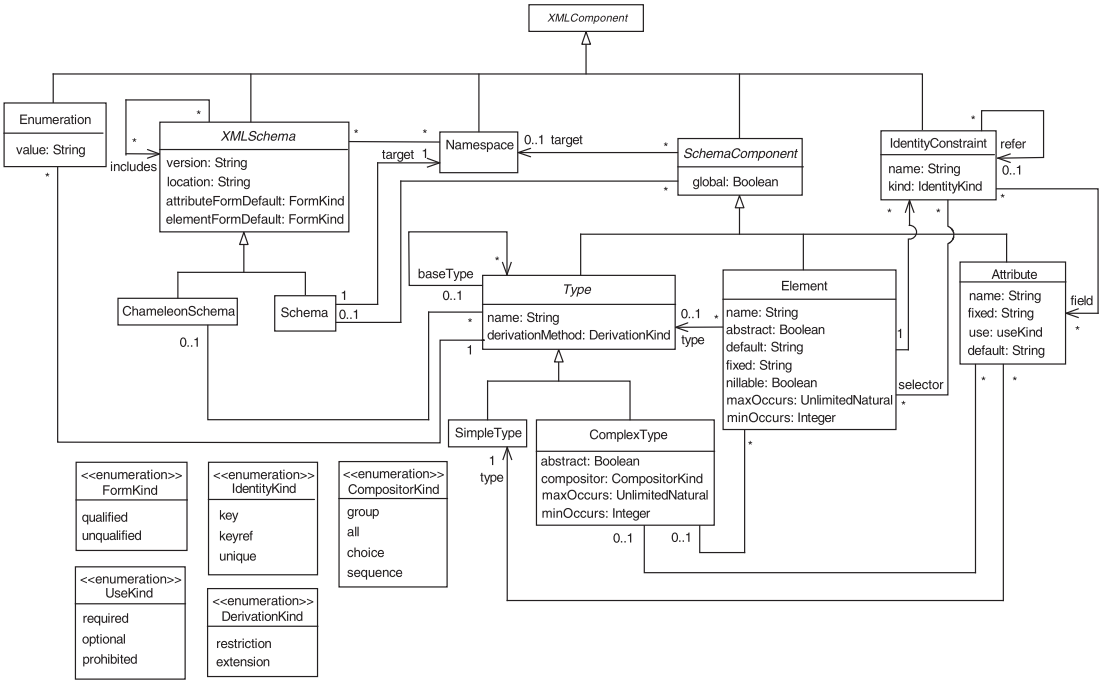


Abbildung A.5.: Grafische Repräsentation der XML-Komponenten eines XML-Schemas mit Bezug zum UML-XML-Ansatz aus [DLP+11]

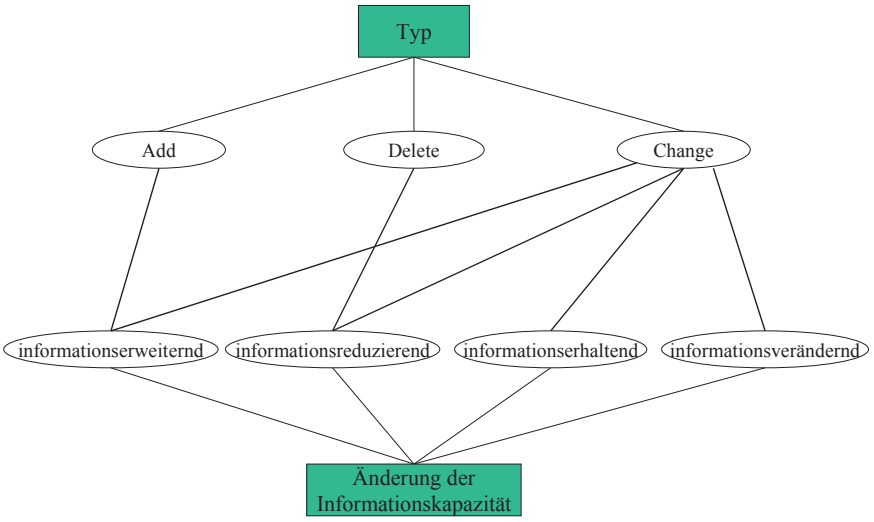


Abbildung A.6.: Zusammenhang zwischen dem Typ eines Operators und der Änderung der Informationskapazität aus [Har07]

Change predicate	Category	Description
$classAdded(\widetilde{C}', \widetilde{R}')$	Addition	A new class $\widetilde{C}'$ is added as a child of association $\widetilde{R}'$ (if $\widetilde{R}' = \perp$, $\widetilde{C}'$ is added as a new root class).
$classRemoved(C')$	Removal	Class C' is removed.
$classRenamed(\widetilde{C}', \tilde{n}')$	Sedentary	The name of class $\widetilde{C}'$ is changed to $\tilde{n}' \in \mathcal{L}$. The name is mandatory for PSM classes, but can be changed.
$classMoved(\widetilde{C}', \widetilde{R}'_n)$	Migratory	Class $\widetilde{C}'$ is moved and becomes a child of association $\widetilde{R}'_n$ in version $\tilde{v}$ (or becomes a new root class, in that case $\widetilde{R}'_n = \perp$). This change encompasses changes of the *child* participant of associations (in contrast to *associationMoved*—see below).
$srIntroduced(\widetilde{C}', \widetilde{C}'_r)$	Sedentary	Class $\widetilde{C}'$ becomes a structural representative of another class $\widetilde{C}'_r$ in the schema. In the previous version, it was not a structural representative.
$srRemoved(\widetilde{C}')$	Sedentary	Class $\widetilde{C}'$ is converted to a regular class. In the previous version, it was a structural representative.
$srChanged(\widetilde{C}', \widetilde{C}'_r)$	Sedentary	Class $\widetilde{C}'$ becomes a structural representative of another class $\widetilde{C}'_r$ in the schema. In the previous version, it was a structural representative of a different class.
$attributeAdded(\widetilde{A}', \widetilde{C}', \tilde{i}')$	Addition	A new attribute $\widetilde{A}'$ is added to class $\widetilde{C}'$ at position $\tilde{i}' \in \mathbb{N}_0$.
$attributeRemoved(A')$	Removal	Attribute A' is removed.
$attributeRenamed(\widetilde{A}', \tilde{n}')$	Sedentary	The name of attribute $\widetilde{A}'$ is changed to $\tilde{n}' \in \mathcal{L}$.
$attributeMoved(\widetilde{A}', \widetilde{C}'_n, \tilde{i}')$	Migratory	The value of $class(\widetilde{A}')$ is changed, i.e. attribute $\widetilde{A}'$ is moved from class C'_o to class $\widetilde{C}'_n$ at position $\tilde{i}' \in \mathbb{N}_0$. Moves within the same class are detected by *attributeIndexChanged*.
$attributeXFormChanged(\widetilde{A}', \tilde{f}')$	Sedentary	The value of $xform$ is changed from a to e or vice versa for attribute $\widetilde{A}'(\tilde{f}' \in \{a, e\})$.
$attributeTypeChanged(\widetilde{A}', \widetilde{D}')$	Sedentary	The type of attribute $\widetilde{A}'$ is changed to $\widetilde{D}' \in \mathcal{D}$.
$attributeIndexChanged(\widetilde{A}', \tilde{i}')$	Migratory	Attribute $\widetilde{A}'$ is moved to position $\tilde{i}' \in \mathbb{N}_0$ within the same class as in version v. Moves between classes are detected by *attributeMoved*.
$attributeCardinalityChanged(\widetilde{A}', \tilde{c}')$	Sedentary	The cardinality of attribute $\widetilde{A}'$ is changed to $\tilde{c}' \in \mathcal{C}$.
$associationAdded(\widetilde{R}', \widetilde{C}', \tilde{i}')$	Addition	A new association $\widetilde{R}'$ is added to the content of class $\widetilde{C}'$ at position $\tilde{i}' \in \mathbb{N}_0$.
$associationRemoved(R')$	Removal	Association R' is removed.
$associationRenamed(\widetilde{R}', \tilde{n}')$	Sedentary	The name of association $\widetilde{R}'$ is changed to $\tilde{n}' \in \mathcal{L}$.
$associationMoved(\widetilde{R}', \widetilde{P}'_n, \tilde{i}')$	Migratory	Association $\widetilde{R}'$ is moved from the content of node P'_o to the content of node $\widetilde{P}'_n$ at position $\tilde{i}' \in \mathbb{N}_0$. This change encompasses changes of the *parent* participant of associations (in contrast to *classMoved* and *contentModelMoved*—see below).
$associationCardinalityChanged(\widetilde{R}', \tilde{c}')$	Sedentary	The cardinality of association $\widetilde{R}'$ is changed to $\tilde{c}' \in \mathcal{C}$.
$associationIndexChanged(\widetilde{R}', \tilde{i}')$	Migratory	Association $\widetilde{R}'$ is moved to position $\tilde{i}' \in \mathbb{N}_0$ (within the same class as in version v).
$contentModelAdded(\widetilde{M}', \widetilde{R}')$	Addition	A new content model $\widetilde{M}'$ is added as a child of association $\widetilde{R}'$.
$contentModelRemoved(M')$	Removal	Content model M' is removed.
$contentModelMoved(\widetilde{M}', \widetilde{R}'_n)$	Migratory	Content model $\widetilde{M}'$ is moved and becomes a child of association $\widetilde{R}'_n$ in version $\tilde{v}$. Content models cannot be roots in a normalized PSM schema (see Definition 5). Thus, unlike *classMoved*, $\widetilde{R}'_n$ is never null for *contentModelMoved*.
$contentModelTypeChanged(\widetilde{M}', \tilde{t}')$	Sedentary	The type of content model (`sequence`, `set`, `choice`) $\widetilde{M}'$ is changed to $\tilde{t}' \in$ {`sequence`, `set`, `choice`} .

There are no predicates dedicated to the changes in the set $\mathcal{S}'_e$ and function *participant*, because each change in $\mathcal{S}'_e$ and *participant* is an inherent part of another change (*classAdded*, *classRemoved*, *classMoved*, *contentModelAdded*, *contentModelRemoved*, *contentModelMoved*, *associationAdded*, *associationRemoved*). Thus, changes in $\mathcal{S}'_e$ and *participant* are detected and documents adapted within the scope of the changes listed above

Abbildung A.7.: Klassifikation von Änderungen aus [MNM12]

Das formale Modell für EMX basiert auf einem gemischten Graph (*engl.* mixed graph).

Es handelt sich also um ein 3-Tupel: $G = (V, E, A)$, mit

V, einer Menge von Knoten

E, einer Menge ungerichteter Kanten. Eine ungerichtete Kante $e \in E$ ist ein ungeordnetes Knotenpaar: $e = (v_1, v_2)$ mit $v_1, v_2 \in V$

A, einer Menge gerichteter Kanten. Eine gerichtete Kante $a \in A$ ist ein geordnetes Knotenpaar: $a = (v_1, v_2)$ mit $v_1, v_2 \in V$

Für die unterschiedlichen Entity-Typen werden *disjunkte* Teilmengen von V eingeführt. Es gilt:

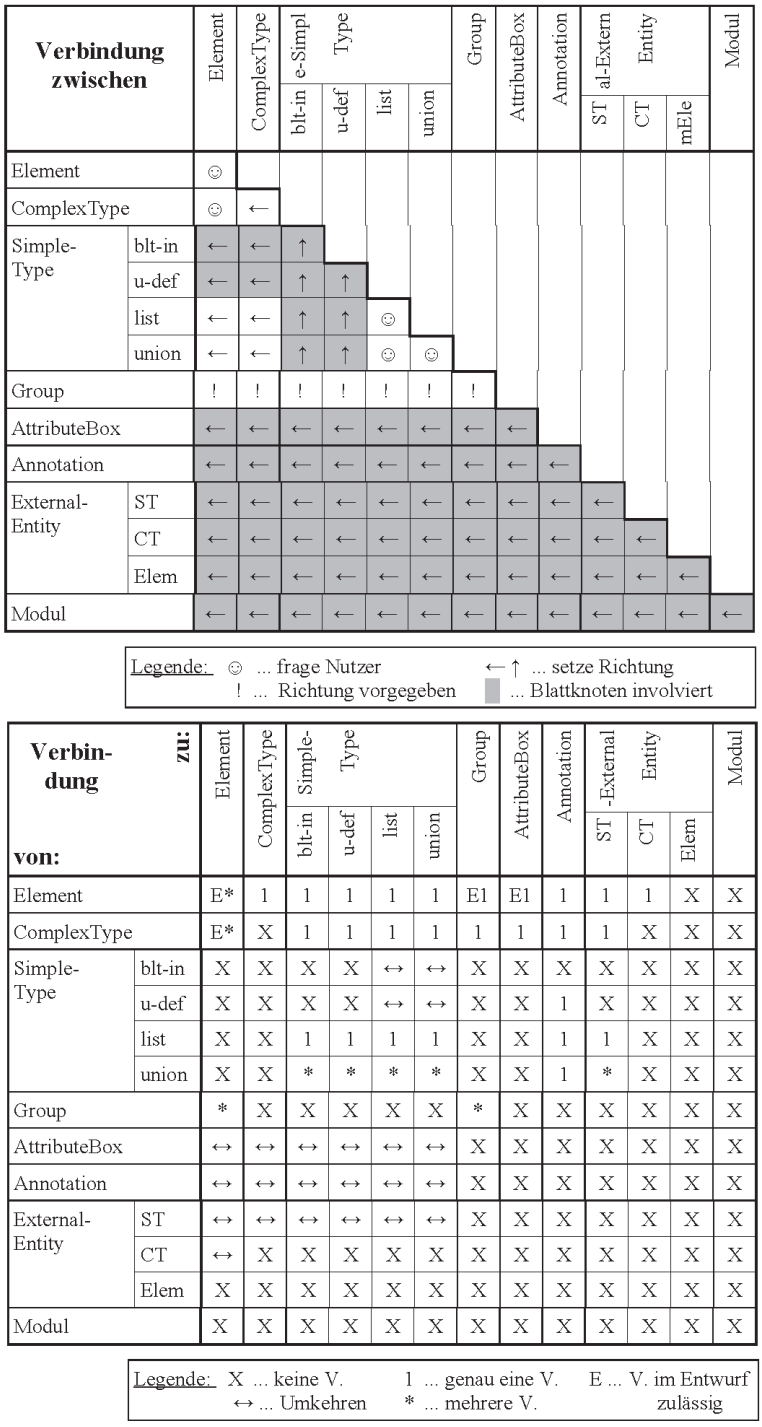
$$V = El \cup CT \cup ST_{blt-in} \cup ST_{udef} \cup ST_{list} \cup ST_{union} \cup Grp \cup AttrBox \cup Ann \cup EE_{ST} \cup EE_{CT} \cup EE_{El} \cup M$$

mit
- El = Menge der Element-Entities,
- CT = Menge der ComplexType-Entities,
- ST_{blt-in} = Menge der SimpleType-Entities, (für Datentypen der XML-Schema-Spezifikation [XS101]),
- ST_{udef} = Menge der benutzerdefinierten SimpleType-Entities,
- ST_{list} = Menge der List-SimpleType-Entities,
- ST_{union} = Menge der Union-SimpleType-Entities,
- Grp = Menge der Group-Entities,
- $AttrBox$ = Menge der AttributeBox-Enties,
- Ann = Menge der Annotation-Entities,
- EE_{ST} = Menge der SimpleType-ExternalEntities,
- EE_{CT} = Menge der ComplexType-ExternalEntities,
- EE_{El} = Menge der Element-ExternalEntities,
- M = Menge der Module.

Um das formale Modell zu vervollständigen, werden noch zwei Menge von Regeln R_a und R_e eingeführt. Die Regeln haben die Form:

$$r_a : \quad (X \subset V) \times (Y \subset V) \rightarrow \{true, false\} \in R_a$$
$$r_e : \quad (X \subset V) \times (Y \subset V) \rightarrow \{true, false\} \in R_e$$

Abbildung A.8.: Formales Modell von EMX (Entity Model for XML-Schema) aus [Ste06]



Verbindung zwischen		Element	ComplexType	Simple-Type: blt-in	Simple-Type: u-def	Simple-Type: list	Simple-Type: union	Group	AttributeBox	Annotation	External-Entity: ST	External-Entity: CT	External-Entity: mEle	Modul
Element		☺												
ComplexType		☺	←											
Simple-Type	blt-in	←	←	↑										
	u-def	←	←	↑	↑									
	list	←	←	↑	↑	☺								
	union	←	←	↑	↑	☺	☺							
Group		!	!	!	!	!	!	!						
AttributeBox		←	←	←	←	←	←	←	←					
Annotation		←	←	←	←	←	←	←	←	←				
External-Entity	ST	←	←	←	←	←	←	←	←	←	←			
	CT	←	←	←	←	←	←	←	←	←	←	←		
	Elem	←	←	←	←	←	←	←	←	←	←	←	←	
Modul		←	←	←	←	←	←	←	←	←	←	←	←	←

Legende: ☺ ... frage Nutzer ← ↑ ... setze Richtung
! ... Richtung vorgegeben ... Blattknoten involviert

Verbindung zu: / **von:**		Element	ComplexType	Simple-Type: blt-in	Simple-Type: u-def	Simple-Type: list	Simple-Type: union	Group	AttributeBox	Annotation	External-Entity: ST	External-Entity: CT	External-Entity: Elem	Modul
Element		E*	1	1	1	1	1	E1	E1	1	1	1	X	X
ComplexType		E*	X	1	1	1	1	1	1	1	1	X	X	X
Simple-Type	blt-in	X	X	X	X	↔	↔	X	X	X	X	X	X	X
	u-def	X	X	X	X	↔	↔	X	X	1	X	X	X	X
	list	X	X	1	1	1	1	X	X	1	1	X	X	X
	union	X	X	*	*	*	*	X	X	1	*	X	X	X
Group		*	X	X	X	X	X	*	X	X	X	X	X	X
AttributeBox		↔	↔	↔	↔	↔	↔	X	X	X	X	X	X	X
Annotation		↔	↔	↔	↔	↔	↔	X	X	X	X	X	X	X
External-Entity	ST	↔	↔	↔	↔	↔	↔	X	X	X	X	X	X	X
	CT	↔	X	X	X	X	X	X	X	X	X	X	X	X
	Elem	X	X	X	X	X	X	X	X	X	X	X	X	X
Modul		X	X	X	X	X	X	X	X	X	X	X	X	X

Legende: X ... keine V. 1 ... genau eine V. E ... V. im Entwurf zulässig
↔ ... Umkehren * ... mehrere V.

Abbildung A.9.: Ungerichtete (oben) und gerichtete (unten) Kantenkombinationen des formalen Modells von EMX in Abbildung A.8 aus [Ste06]

Kante(X,Y) von X / zu Y	element	attribute-group	group	complex-type	simple-type	annotation	constraint	module	schema
element			x						
attribute-group			x						
group	x								
complex-type									
simple-type									
annotation	x	x	x				x	x	
constraint	x								
module									

Abbildung A.10.: Visualisierung gerichteter Kanten zwischen Entitätstypen im EMX

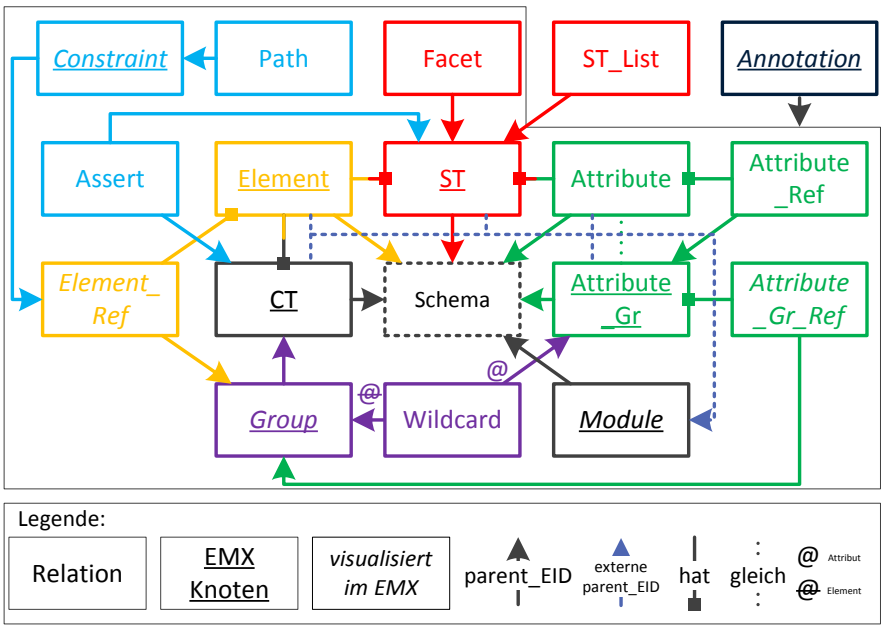


Abbildung A.11.: Logisches Modell aus Perspektive der EMX-Knoten

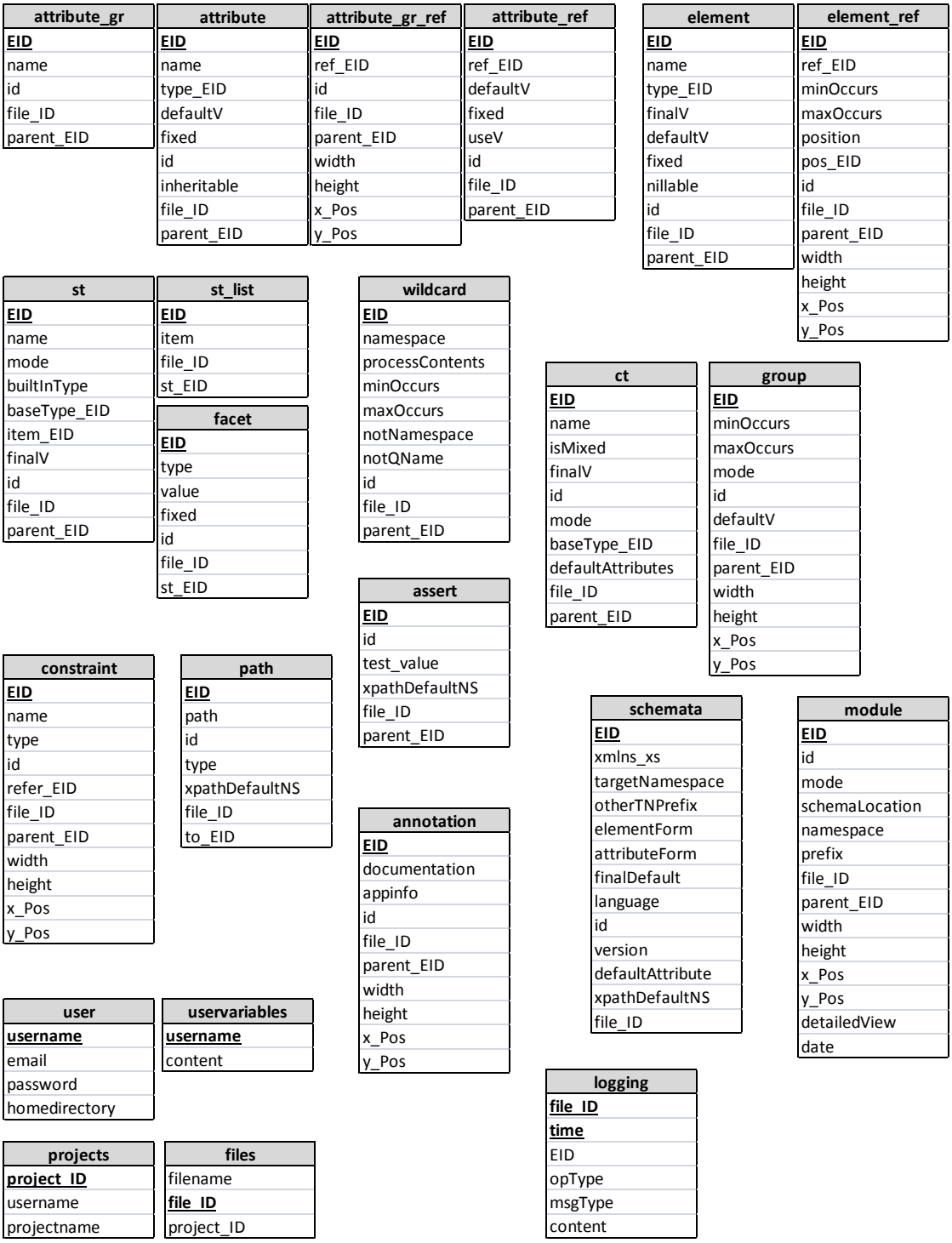


Abbildung A.12.: Relationsschemata zur Speicherung und Verwaltung von EMX

time	EID	opType	content
1	1	0	add element name 'e1' type 'xs:decimal' id 'EID1' ;
3	2	0	add element name 'e2' type 'xs:string' id 'EID2' ;
5	3	0	add complextype name 'roottype' id 'EID3' ;
6	4	0	add group mode sequence minoccurs '1' maxoccurs '2' id 'EID4' in '3' ;
8	5	0	add elementref 'e1' minoccurs '1' maxoccurs '2' id 'EID5' in '4' ;
9	6	0	add elementref 'e2' minoccurs '0' maxoccurs '2' id 'EID6' in '4' ;
11	7	0	add element name 'root' type '3' id 'EID7' ;

Abbildung A.13.: Log der Abbildung 5.2 nach Anwendung der ROfEL-Regeln

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="root" type="roottype"/>
    <xs:element name="e1" type="xs:decimal"/>
    <xs:element name="e2" type="xs:string"/>
    <!--Durch das Einfügen wird die Kapazität erweitert.-->
    <xs:element name="e3" type="xs:decimal">
    <xs:attribute name="a1" type="xs:decimal"/>
    <xs:attribute name="a2" type="xs:string"/>
    <xs:complexType name="roottype">
        <xs:sequence minOccurs="1" maxOccurs="2">
            <!--e3 kann als Referenz neu verwendet werden.-->
            <xs:element ref="e1" minOccurs="1" maxOccurs="2"/>
            <xs:element ref="e2" minOccurs="0" maxOccurs="2"/>
        </xs:sequence>
        <xs:attribute ref="a1" use="required"/>
        <xs:attribute ref="a2" use="optional"/>
    </xs:complexType>
</xs:schema>
```

XML-Beispiel A.5: Kapazitätserweiternde Operation auf einem XML-Schema

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="root" type="roottype"/>
    <!--Durch das Löschen wird die Kapazität reduziert.-->
    <xs:element name="e1" type="xs:decimal"/>
    <xs:element name="e2" type="xs:string"/>
    <xs:attribute name="a1" type="xs:decimal"/>
    <xs:attribute name="a2" type="xs:string"/>
    <xs:complexType name="roottype">
        <xs:sequence minOccurs="1" maxOccurs="2">
            <!--Diese Referenz kann nicht mehr verwendet werden.-->
            <xs:element ref="e1" minOccurs="1" maxOccurs="2"/>
            <xs:element ref="e2" minOccurs="0" maxOccurs="2"/>
        </xs:sequence>
        <xs:attribute ref="a1" use="required"/>
        <xs:attribute ref="a2" use="optional"/>
    </xs:complexType>
</xs:schema>
```

XML-Beispiel A.6: Kapazitätsreduzierende Operation auf einem XML-Schema

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="root" type="roottype"/>
    <xs:element name="e1" type="xs:decimal"/>
    <xs:element name="e2" type="xs:string"/>
    <!--Durch das Einfügen wird die Kapazität erhalten.-->
    <xs:annotation></xs:annotation>
    <xs:attribute name="a1" type="xs:decimal"/>
    <xs:attribute name="a2" type="xs:string"/>
    <xs:complexType name="roottype">
        <xs:sequence minOccurs="1" maxOccurs="2">
            <xs:element ref="e1" minOccurs="1" maxOccurs="2"/>
            <xs:element ref="e2" minOccurs="0" maxOccurs="2"/>
        </xs:sequence>
        <xs:attribute ref="a1" use="required"/>
        <xs:attribute ref="a2" use="optional"/>
    </xs:complexType>
</xs:schema>
```

XML-Beispiel A.7: Kapazitätserhaltende Operation auf einem XML-Schema

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="e1" type="xs:decimal"/>
    <xs:element name="e2" type="xs:string"/>
    <!--Durch das Ändern (final hinzugefügt) wird die Kapazität reduziert.-->
    <xs:simpleType name="st1" final="#all">
        <xs:list itemType="xs:string"/>
    </xs:simpleType>
    <xs:simpleType name="st2">
        <!--st1 kann nicht mehr verwendet werden.-->
        <xs:restriction base="st1"/>
    </xs:simpleType>
    <!--Durch das Ändern (final gelöscht) wird die Kapazität erweitert.-->
    <xs:simpleType name="st3" final="#all">
        <xs:list itemType="xs:string"/>
    </xs:simpleType>
    <xs:simpleType name="st4">
        <!--st3 kann neu verwendet werden.-->
        <xs:restriction base="xs:string"/>
    </xs:simpleType>
    <!--Durch das Ändern (Umbenennung) wird die Kapazität erhalten.-->
    <xs:simpleType name="neuerName">
        <xs:restriction base="xs:string"/>
    </xs:simpleType>
</xs:schema>
```

XML-Beispiel A.8: Kapazitätsverändernde Operation auf einem XML-Schema

```
<!--Hinzufügen einer zwingenden Elementreferenz e2 im XML-Schema.-->
<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="example.xsd"
      a1="0">
    <e1>0</e1>
    <e1>0</e1>
    <!--Der Informationsgehalt wird durch e2 erweitert.-->
    <e2>42</e2>
</root>
```

XML-Beispiel A.9: Instanzerweiternde Operation auf einem XML-Dokument

```
<!--Löschen einer zwingenden Elementreferenz e2 im XML-Schema.-->
<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="example.xsd"
      a1="0">
    <e1>0</e1>
    <e1>0</e1>
    <!--Der Informationsgehalt wird ohne e2 reduziert.-->
    <e2>0</e2>
</root>
```

XML-Beispiel A.10: Instanzreduzierende Operation auf einem XML-Dokument

```
<!--Hinzufügen einer optionalen Elementreferenz e2 im XML-Schema.-->
<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="example.xsd"
      a1="0">
    <!--Der Informationsgehalt wird erhalten.-->
    <e1>0</e1>
    <e1>0</e1>
</root>
```

XML-Beispiel A.11: Instanzerhaltende Operation auf einem XML-Dokument

```
<!--Ändern von maxOccurs von Elementreferenzen e1 und e2 im XML-Schema.-->
<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="example.xsd"
      a1="0">
    <!--Der Informationsgehalt wird durch Änderung von 2 auf 3 erhalten.-->
    <e1>0</e1>
    <e1>0</e1>
    <!--Der Informationsgehalt wird durch Änderung von 2 auf 1 reduziert.-->
    <e2>0</e2>
    <e2>0</e2>
</root>
```

XML-Beispiel A.12: Instanzverändernde Operation auf einem XML-Dokument

Operation	Kapazität S	Kapazität S'	Informations-gehalt I	Informations-gehalt I'	Instanz-kosten CodeX	Folge-kosten CodeX	Dokumentanpassung Kriterien (Analyse Anpassung)	Dokument Auswirkung (Analyse Auswirkung)	Parent-Check	Child-Check	Überprüfung und Lokalisierung
addannotation	=	<= >	=	=	0						
addattributegroupdef	>		=	=	0						
addattribute	>		=	=	0						
addattributeref	=		= >	= >	1		use = "required"	add value ?	X		[use = "required"]; (in AG, mit AGRef)?, ((in Group)?, in CT, in E, mit ERef)+
addattributegroupref	=		= >	= >	1		Anzahl ARefs [use = "required"] > 0	add value ?	2..X	1..X	1. ARef [use = "required"] in AG, 2. ((in Group)?, in CT, in E, mit ERef)+
addattributewildcard	=		=	=	0						
addgroup	=		=	=	0						
addst	>		=	=	0						
addct	>		=	=	0						
addelementdef	>		=	=	0						
addelementref	=		= >	= >	1		minoccurs > 0	add value ?	X		[minoccurs > 0]; in Group [mode = "sequence" | mode = "all"], ((in Group)?, in CT, in E, mit ERef)+
addelementwildcard	=		= >	= >	1		minoccurs > 0	add value ?	X		[minoccurs > 0]; in Group [mode = "sequence" | mode = "all"], ((in Group)?, in CT, in E, mit ERef)+
addmodule	= >		=	=	0						
addconstraint	= >		<=	=	1		Komponente K in Dokument	mod value ?	X	X	1. Durch Selector und Field Kombination addressierte Komponente K ermitteln 2. Für K ermitteln, wie oft in Instanz + Sammeln dortiger Werte + Eindeutigkeit
delannotation	=		=	=	0						
delattributegroupdef	<		<=	<=	0	X					
delattribute	<		<=	<=	0	X					
delattributeref	=		<=	<=	1		use != "prohibited"	del value ?	X		(in AG, mit AGRef)?, (((in Group)?, in CT, in E, in ERef)+
delattributegroupref	=		<=	<	1		Anzahl ARefs [use != "prohibited"] > 0	del value ?	2..X	1..X	1. ARef [use != "prohibited"] in AG, 2. ((in Group)?, in CT, in E, mit ERef)+
delattributewildcard	=		<=	<=	1			del value ?	X		(in AG, mit AGRef)?, (((in Group)?, in CT, in E, in ERef)+
delgroup	=		<=	<=	0	X					
delst	<		<=	<=	0	X					
delct	<		<=	<=	0	X					
delelementdef	<		<=	<=	0	X					
delelementref	=		<=	<	1		maxoccurs > 0	del value ?	X		[maxoccurs > 0]; (in Group, in CT, in E, mit ERef)+
delelementwildcard	=		<=	<=	1		maxoccurs > 0	del value ?	X		[maxoccurs > 0]; (in Group, in CT, in E, mit ERef)+
delmodule	<=		<=	<=	0	X					
delconstraint	<=		=	=	0						
updschema	< = >		< = >	< = >	0 1 0		targetnamespace defaultattribute sonst	del/mod/add value ?		X	CT [mit defaultattributeapply != "false" UND AGRef nicht bereits vorhanden], in E, mit ERef; (in Group, in CT, in E, mit ERef)*
updannotation	=		=	=	0						
updattributegroupdef	=		=	=	0						
updattribute	< = >		< = >	< = >	1		name	mod Markup ?	X		ARef [use != "prohibited"], (in AG, mit AGRef)?, (((in Group)?, in CT, in E, mit ERef)+ UND Wildcard [processContent != "lax"], (in AG , mit AGRef)?, (((in Group)?, in CT, in E, mit ERef)+
					1		sonst	siehe extra Liste	X		[Restriktiverer Typ]: ARef [use != "prohibited"], (in AG, mit AGRef)?, (((in Group)?, in CT, in E, mit ERef)+ UND Wildcard [processContent != "lax"], (in AG , mit AGRef)?, (((in Group)?, in CT, in E, mit ERef)+
updattributeref	=		< = >	< = >	1		ref	mod value / Markup ?	X		[Restriktiverer Typ]: (in AG, mit AGRef)?, (((in Group)?, in CT, in E, mit ERef)+
							sonst	siehe extra Liste	X		(in AG, mit AGRef)?, (((in Group)?, in CT, in E, mit ERef)+
updattributegroupref	=		< = >	< = >	1		ref	mod value / Markup ?	2..X	1..X	0. keine Überdeckung (Matcher?) -> NichtSchnitt löschen 1. ARef [use != "prohibited" UND nicht im Schnitt] in AG, 2. (((in Group)?, in CT, in E, mit ERef)+
updattributewildcard	=		<=	1				siehe extra Liste	X		(in AG, mit AGRef)?, (((in Group)?, in CT, in E, mit ERef)+
updgroup	=		< = >	< = >	1			siehe extra Listen	2..X	1..X	1.ERef [minoccurs != 0] oder Any [minoccurs != 0], 2. (in Group, in CT, in E, in ERef)+
updst	< = >		< = >	< = >	1		name	mod value ?	X		in E, mit ERef, (in Group, in CT , in E, mit ERef)*
							sonst	siehe extra Liste	X		[Restriktiverer Typ]: (in E, mit ERef UND in A, mit ARef [in AG, mit AGRef]?), (((in Group)?, in CT, in E, mit ERef)*
updct	< = >		< = >	< = >	1		name	mod value ?	X		(((in Group)?, in CT, in E, mit ERef)+
							sonst	siehe extra Liste	X		(((in Group)?, in CT, in E, mit ERef)+
updelementref	< = >		< = >	< = >	1		name	mod Markup ?	X		ERef [maxoccurs > 0], (in Group, in CT in E, mit ERef+ UND Wildcard [processcontent="strict"], (in Group, in CT, in E, mit ERef)+
							sonst	siehe extra Liste	X		ERef [maxoccurs > 0], (in Group, in CT in E, mit ERef+ UND Wildcard [processcontent="strict"], (in Group, in CT, in E, mit ERef)+
updelementref	=		< = >	< = >	1		ref	mod value / Markup ?	X		[maxoccurs > 0]; (in Group, in CT, in E, mit ERef)+
							sonst	siehe extra Liste	X		[maxoccurs > 0]; (in Group, in CT, in E, mit ERef)+
							xPos, yPos	reorder Markup ?	X		[maxoccurs > 0]; (in Group [mode != "choice"], in CT, in E, mit ERef)+
updelementwildcard	=		< = >	< = >	1			siehe extra Liste	X		[maxoccurs > 0]; (in Group, in CT, in E, mit ERef)+
updmodule	< = >		< = >	< = >	0	X					
updconstraint	< = >		< =	< =	1		Komponente K in Dokument	mod value ?	X	X	1. Durch Selector und Field Kombination addressierte Komponente K ermitteln 2. Für K ermitteln, wie oft in Instanz + Sammeln dortiger Werte + Eindeutigkeit

Abbildung A.14.: Übersicht der Klassifikation der ELaX-Operationen aus [Nös15a]

neu ->

aktuell	Werte	default		fixed	
		Wert bleibt	change	Wert bleibt	change
	default	nix	nix	immer WP: add/mod value	immer WP: add/mod value
	fixed	nix	nix	nix	immer WP: mod value
	not(default) & not(fixed)	nix	nix	nix	immer WP: add/mod value

neu ->

aktuell	type	ST		
		UT	OT	change
	ST	immer WP: mod value?	nix	immer WP: mod value?

neu ->

aktuell	inheritable	WAHR	FALSCH
	WAHR	nix	nix
	FALSCH	nix	nix

Abbildung A.15.: Extraliste der ELaX-Operation updattribute aus [Nös15a]

neu ->

aktuell	Werte	default		fixed	
		Wert bleibt	change	Wert bleibt	change
	default	nix	nix	immer WP: add/mod value	immer WP: add/mod value
	fixed	nix	nix	nix	immer WP: mod value
	not(default) & not(fixed)	nix	nix	nix	immer WP: add/mod value

neu ->

aktuell	use	prohibited	optional	required
	probibited	nix	nix	immer WP: add value
	optional	immer WP: del value	nix	immer WP: add value
	required	immer WP: del value	nix	nix

Abbildung A.16.: Extraliste der ELaX-Operation updattributeref aus [Nös15a]

aktuell \ neu -> not	QNAME insert	QNAME remove	##defined
QNAME	immer WP: del value?	nix	immer WP: del value?
##defined	keine Module eingebunden: nix, sonst: immer WP: del value?	nicht möglich	nix

aktuell \ neu -> namespace	##any	##other	##local	ANYURI insert	ANYURI remove	##targetnamespace
##any	nix	immer WP: del value?	immer WP: del value?	immer WP: del value?	nicht möglich	immer WP: del value?
##other	nix	nix	immer WP: del value?	immer WP: del value?	nicht möglich	immer WP: del value?
##local	nix	nix	nix	nix	nicht möglich	nix
ANYURI	nix	immer WP: del value?	nix	nix	immer WP: del value?	nix
##targetnamespace	nix	immer WP: del value?	nix	nix	nicht möglich	nix

aktuell \ neu -> notNamespace	ANYURI insert	ANYURI remove	##targetnamespace	##local
ANYURI	immer WP: del value?	nix	immer WP: del value?	nix
##targetnamespace	immer WP: del value?	nicht möglich	nix	nix
##local	immer WP: del value?	nicht möglich	immer WP: del value?	nix

aktuell \ neu -> processcontent	lax	skip	strict
lax	nix	nix	immer WP: del value?
skip	nix	nix	immer WP: del value?
strict	nix	nix	nix

Abbildung A.17.: Extraliste der ELaX-Operation updattributewildcard aus [Nös15a]

mode	mMin	mMax	e1Min	e1Max	e2Min	e2Max	aMin	aMax	sequence	choice			mMin--	mMax--			mMin++			mMax++
										e1	e2	a		e1	e2	a	e1	e2	a	
all	0	0	0	0	0	0	0	0			Kombi							nix		
all	0	0	0	1	0	0	0	0			nix									
all	0	0	0	e1Max	0	0	0	0							invalid					Kombi
all	0	0	1	1	0	0	0	0		add							add			
all	0	0	1	e1Max	0	0	0	0		add							add			
all	0	0	e1Min	e1Max	0	0	0	0		add*							add*			
																		nix		
all	0	0	0	0	0	1	0	0												
all	0	0	0	1	0	1	0	0												
all	0	0	0	e1Max	0	1	0	0			nix				invalid					Kombi
all	0	0	1	1	0	1	0	0									add			
all	0	0	1	e1Max	0	1	0	0									add			
all	0	0	e1Min	e1Max	0	1	0	0									add*			
																		nix		
all	0	0	0	0	0	e2Max	0	0												
all	0	0	0	1	0	e2Max	0	0												
all	0	0	0	e1Max	0	e2Max	0	0			nix				invalid					Kombi
all	0	0	1	1	0	e2Max	0	0									add			
all	0	0	1	e1Max	0	e2Max	0	0									add			
all	0	0	e1Min	e1Max	0	e2Max	0	0									add*			
											Kombi									
all	0	0	0	0	1	1	0	0			nix							add		
all	0	0	0	1	1	1	0	0										add		
all	0	0	0	e1Max	1	1	0	0							invalid			add		Kombi
all	0	0	1	1	1	1	0	0		add							add	add		
all	0	0	1	e1Max	1	1	0	0		add							add	add		
all	0	0	e1Min	e1Max	1	1	0	0		add							add*	add		
											Kombi									
all	0	0	0	0	1	e2Max	0	0			nix							add		
all	0	0	0	1	1	e2Max	0	0										add		
all	0	0	0	e1Max	1	e2Max	0	0							invalid			add		Kombi
all	0	0	1	1	1	e2Max	0	0		add							add	add		
all	0	0	1	e1Max	1	e2Max	0	0		add							add	add		
all	0	0	e1Min	e1Max	1	e2Max	0	0		add							add*	add		

Abbildung A.18.: Auszug der Extraliste der ELaX-Operation updgroup aus [Nös15a]

aktuell \ neu -> mode	built-in	list	union insert	union remove	restriction insert	restriction remove	modify
built-in	OT: nix, UT: WP (mod value?)	LT = BT ODER OT: nix, UT: WP (mod value?)	Mit BT drin: nix, sonst: WP (mod value?)	nix	immer WP: (mod value?)	nix	nix
list	nix	Neuer LT ist OT: nix, UT: WP (mod value?)	LT enthalten: nix, sonst: WP (mod value?)	nix	immer WP: (mod value?)	nix	nix
union	nix	Ein MT, der dann LT UND WS = preserve: nix, sonst: WP (mod value?)	nix	Ein OT oder MT deckt Typ ab: nix, sonst: WP (mod value?)	immer WP: (mod value?)	nix	nix
restriction	nix	LT = RT UND WS = preserve: nix, sonst: WP (mod value?)	Mit RT: nix, sonst: WP (mod value?)	nix	immer WP UND alle UT prüfen (mod value?)	nix	Facette allgemeiner: nix, sonst: WP UND alle UT prüfen (mod value?)

UT - Untertyp / OT - Obertyp

WP - Wertprüfung / WS - WhiteSpace

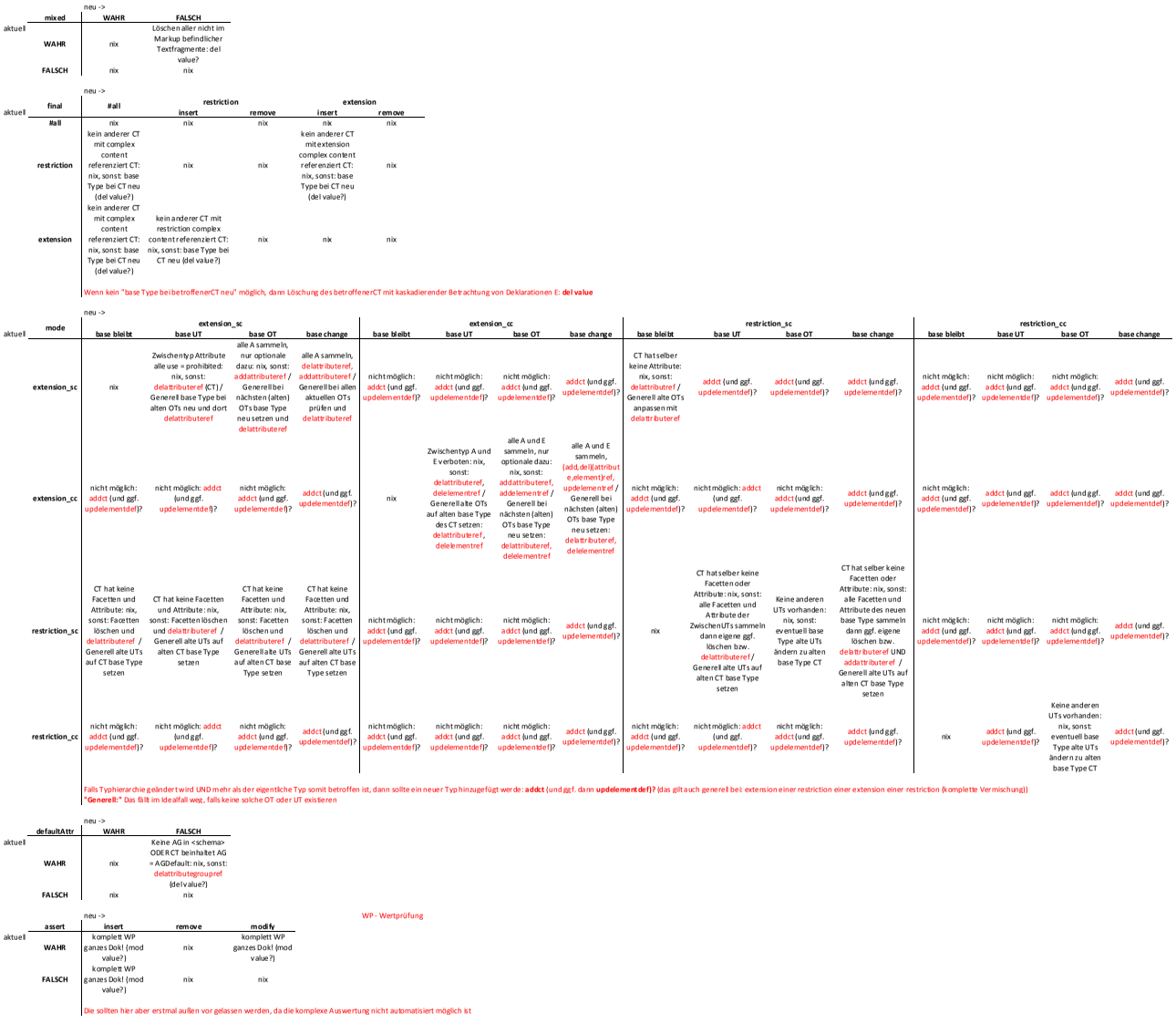
BT - Built-in Type / LT ListType (list)

MT - MemberType (union) / RT - RestrictionType (restriction)

aktuell \ neu -> final	#all	union insert	union remove	list insert	list remove	restriction insert	restriction remove	extension insert	extension remove
#all	nix	nix	nix	nix	nix	nix	nix	nix	nix
union	alle UT mit mode = res ODER list prüfen, nirgend als OT: nix, sonst: RT oder LT neu (del value?)	nix	nix	alle UT mit mode = list prüfen, nirgends als OT: nix, sonst LT neu (del value?)	nix	alle UT mit mode = res prüfen, nirgends als OT: nix, sonst RT neu (del value?)	nix	kein CT mit extension simple content: nix, sonst: base Type bei CT neu (del value?)	nix
list	alle UT mit mode = union ODER res prüfen, nirgends als OT: nix, sonst: MT oder RT neu (del value?)	alle UT mit mode = union prüfen, nirgends als OT: nix, sonst MT neu (del value?)	nix	nix	nix	alle UT mit mode = res prüfen, nirgends als OT: nix, sonst RT neu (del value?)	nix	kein CT mit extension simple content: nix, sonst: base Type bei CT neu (del value?)	nix
restriction	alle UT mit mode = union ODER list prüfen, nirgends als OT: nix, sonst: MT oder LT neu (del value?)	alle UT mit mode = union prüfen, nirgends als OT: nix, sonst MT neu (del value?)	nix	alle UT mit mode = list prüfen, nirgends als OT: nix, sonst LT neu (del value?)	nix	nix	nix	kein CT mit extension simple content: nix, sonst: base Type bei CT neu (del value?)	nix
extension	kein CT mit extension simple content: nix, sonst: base Type bei CT neu (del value?)	alle UT mit mode = union prüfen, nirgends als OT: nix, sonst MT neu (del value?)	nix	alle UT mit mode = list prüfen, nirgends als OT: nix, sonst LT neu (del value?)	nix	alle UT mit mode = res prüfen, nirgends als OT: nix, sonst RT neu (del value?)	nix	nix	nix

Wenn kein "MT, LT, RT oder base Type bei CT neu" möglich, dann Löschung des UT mit kaskadierender Betrachtung von Deklarationen E und A: **del value**

Abbildung A.19.: Extraliste der ELaX-Operation updst aus [Nös15a]



aktuell \ neu →	mixed	WAHR	FALSCH
	WAHR	nix	Löschen aller nicht im Markup befindlicher Textfragmente: del value?
	FALSCH	nix	nix

aktuell \ neu →	final	#all	restriction: insert	restriction: remove	extension: insert	extension: remove
	#all	nix	nix	nix	nix	nix
	restriction	kein anderer CT mit complex content referenziert CT: nix, sonst: base Type bei CT neu (del value?)	nix	nix	kein anderer CT mit extension complex content referenziert CT: nix, sonst: base Type bei CT neu (del value?)	nix
	extension	kein anderer CT mit complex content referenziert CT: nix, sonst: base Type bei CT neu (del value?)	kein anderer CT mit restriction complex content referenziert CT: nix, sonst: base Type bei CT neu (del value?)	nix	nix	nix

Wenn kein "base Type bei betroffenerCT neu" möglich, dann Löschung des betroffenerCT mit kaskadierender Betrachtung von Deklarationen E: **del value**

aktuell \ neu →	mode	extension_sc: base bleibt	extension_sc: base UT	extension_sc: base OT	extension_sc: base change	extension_cc: base bleibt	extension_cc: base UT	extension_cc: base OT	extension_cc: base change	restriction_sc: base bleibt	restriction_sc: base UT	restriction_sc: base OT	restriction_sc: base change	restriction_cc: base bleibt	restriction_cc: base UT	restriction_cc: base OT	restriction_cc: base change
	extension_sc	nix	Zwischentyp Attribute alle use = prohibited: nix, sonst: delattributeref (CT) / Generell base Type bei alten OTs neu und dort delattributeref	alle A sammeln, nur optionale dazu: nix, sonst: addattributeref / Generell bei nächsten (alten) OTs base Type neu setzen und delattributeref	alle A sammeln, delattributeref, addattributeref / Generell bei allen aktuellen OTs prüfen und delattributeref	nicht möglich: addct (und ggf. updelementdef)?	nicht möglich: addct (und ggf. updelementdef)?	nicht möglich: addct (und ggf. updelementdef)?	addct (und ggf. updelementdef)?	CT hat selber keine Attribute: nix, sonst: delattributeref / Generell alte OTs anpassen mit delattributeref	addct (und ggf. updelementdef)?	addct (und ggf. updelementdef)?	addct (und ggf. updelementdef)?	nicht möglich: addct (und ggf. updelementdef)?	nicht möglich: addct (und ggf. updelementdef)?	nicht möglich: addct (und ggf. updelementdef)?	addct (und ggf. updelementdef)?
	extension_cc	nicht möglich: addct (und ggf. updelementdef)?	nicht möglich: addct (und ggf. updelementdef)?	nicht möglich: addct (und ggf. updelementdef)?	addct (und ggf. updelementdef)?	nix	Zwischentyp A und E verboten: nix, sonst: delattributeref, delelementref / Generell alte OTs auf alten base Type des CT setzen: delattributeref, delelementref	alle A und E sammeln, nur optionale dazu: nix, sonst: addattributeref, addelementref / Generell bei nächsten (alten) OTs base Type neu setzen: delattributeref, delelementref	alle A und E sammeln, (add, del)(attribute,element)ref, updelementref / Generell bei nächsten (alten) OTs base Type neu setzen: delattributeref, delelementref	nicht möglich: addct (und ggf. updelementdef)?	nicht möglich: addct (und ggf. updelementdef)?	nicht möglich: addct (und ggf. updelementdef)?	addct (und ggf. updelementdef)?	nicht möglich: addct (und ggf. updelementdef)?	addct (und ggf. updelementdef)?	addct (und ggf. updelementdef)?	addct (und ggf. updelementdef)?
	restriction_sc	CT hat keine Facetten und Attribute: nix, sonst: Facetten löschen und delattributeref / Generell alte UTs auf CT base Type setzen	CT hat keine Facetten und Attribute: nix, sonst: Facetten löschen und delattributeref / Generell alte UTs auf alten CT base Type setzen	CT hat keine Facetten und Attribute: nix, sonst: Facetten löschen und delattributeref / Generell alte UTs auf alten CT base Type setzen	CT hat keine Facetten und Attribute: nix, sonst: Facetten löschen und delattributeref / Generell alte UTs auf alten CT base Type setzen	nicht möglich: addct (und ggf. updelementdef)?	nicht möglich: addct (und ggf. updelementdef)?	nicht möglich: addct (und ggf. updelementdef)?	addct (und ggf. updelementdef)?	nix	CT hat selber keine Facetten oder Attribute: nix, sonst: alle Facetten und Attribute der ZwischenUTs sammeln dann eigene ggf. löschen bzw. delattributeref / Generell alte UTs auf alten CT base Type setzen	Keine anderen UTs vorhanden: nix, sonst: eventuell base Type alte UTs ändern zu alten base Type CT	CT hat selber keine Facetten oder Attribute: nix, sonst: alle Facetten und Attribute des neuen base Type sammeln dann ggf. eigene löschen bzw. delattributeref UND addattributeref / Generell alte UTs auf alten CT base Type setzen	nicht möglich: addct (und ggf. updelementdef)?	nicht möglich: addct (und ggf. updelementdef)?	nicht möglich: addct (und ggf. updelementdef)?	addct (und ggf. updelementdef)?
	restriction_cc	nicht möglich: addct (und ggf. updelementdef)?	nicht möglich: addct (und ggf. updelementdef)?	nicht möglich: addct (und ggf. updelementdef)?	addct (und ggf. updelementdef)?	nicht möglich: addct (und ggf. updelementdef)?	nicht möglich: addct (und ggf. updelementdef)?	nicht möglich: addct (und ggf. updelementdef)?	addct (und ggf. updelementdef)?	nicht möglich: addct (und ggf. updelementdef)?	nicht möglich: addct (und ggf. updelementdef)?	nicht möglich: addct (und ggf. updelementdef)?	addct (und ggf. updelementdef)?	nix	addct (und ggf. updelementdef)?	Keine anderen UTs vorhanden: nix, sonst: eventuell base Type alte UTs ändern zu alten base Type CT	addct (und ggf. updelementdef)?

Falls Typhierarchie geändert wird UND mehr als der eigentliche Typ somit betroffen ist, dann sollte ein neuer Typ hinzugefügt werde: **addct** (und ggf. dann **updelementdef)?** (das gilt auch generell bei extension einer restriction einer extension einer restriction (komplette Vermischung))
"**Generell:**" Das fällt im Idealfall weg, falls keine solche OT oder UT existieren

aktuell \ neu →	defaultAttr	WAHR	FALSCH
	WAHR	nix	Keine AG in <schema> ODER CT beinhaltet AG = AGdefault: nix, sonst: delattributegroupref (del value?)
	FALSCH	nix	nix

aktuell \ neu →	assert	insert	remove	modify
	WAHR	komplett WP ganzes Dok! (mod value?)	nix	komplett WP ganzes Dok! (mod value?)
	FALSCH	komplett WP ganzes Dok! (mod value?)	nix	nix

WP - Wertprüfung

Die sollten hier aber erstmal außen vor gelassen werden, da die komplexe Auswertung nicht automatisiert möglich ist

Abbildung A.20.: Extraliste der ELaX-Operation updct aus [Nös15a]

aktuell \ neu ->	Werte	default: Wert bleibt	default: change	fixed: Wert bleibt	fixed: change
	default	nix	nix	immer WP: add/mod value	immer WP: add/mod value
	fixed	nix	nix	nix	immer WP: mod value
	not(default) & not(fixed)	nix	nix	nix	immer WP: add/mod value

aktuell \ neu ->	final	#all	restriction: insert	restriction: remove	extension: insert	extension: remove
	#all	Hängen mit SubstitutionGroups zusammen, die noch nicht im CodeX Umfang enthalten sind. Die Behandlung erfolgt ab analog / ähnlich zu Vererbungen der komplexen Typen.				
	restriction					
	extension					

aktuell \ neu ->	nillable	WAHR: xsi:nil='true'	WAHR: xsi:nil='false'	WAHR: not(xsi:nil='')	FALSCH: xsi:nil='true'	FALSCH: xsi:nil='false'	FALSCH: not(xsi:nil='')
	WAHR	nix	nix	nix	immer WP: mod Markup und add value	mod Markup	nix
	FALSCH	nicht möglich	nicht möglich	nix	nicht möglich	nicht möglich	nix

aktuell \ neu ->	type	ST: UT	ST: OT	ST: change	CT: UT	CT: OT	CT: change
	ST	immer WP: mod value?	nix	immer WP: mod value?	Löschen des einfachen Wertes UND das gesamte Contentmodell des CT muss bei jedem <Element> im Dokument hinzugefügt werden (als Kind)		
	CT	Das gesamte Contentmodell des CT muss bei jedem <Element> im Dokument gelöscht werden (child) UND Ersetzung durch einfachen Wert (add value) ... falls einfacher Wert schon vorhanden (CT mixed Content mit Mgl. des leeren Contentmodell): immer WP: mod value?			Vergleich Contentmodell Matcher (alter CT, neuer CT): NichtSchnitt erlaubter E und A löschen del (attribute, agroup, element) ref, updgroup	Vergleich Contentmodell Matcher (alter CT, neuer CT): NichtSchnitt zwingender E und A hinzufügen add (attribute, agroup, element) ref, updgroup	Vergleich Contentmodell Matcher (alter CT, neuer CT): NichtSchnitt zwingender hinzufügen (wie CT OT) und erlaubter löschen (wie CT UT), updgroup

Abbildung A.21.: Extraliste der ELaX-Operation updelementdef aus [Nös15a]

neu ->

aktuell	not	QNAME		##defined	##definedsibling
		insert	remove		
	QNAME	maxoccurs = 0: nix, sonst: immer WP (del value?)	nix	maxoccurs = 0: nix, sonst: immer WP (del value?)	maxoccurs = 0: nix, sonst: immer WP (del value?)
	##defined	maxoccurs = 0 ODER keine Module eingebunden: nix, sonst: immer WP (del value?)	nicht möglich	nix	nix
	##definedsibling	maxoccurs = 0: nix, sonst: immer WP (del value?)	nicht möglich	maxoccurs = 0: nix, sonst: immer WP (del value?)	nix

neu ->

aktuell	namespace	##any	##other	##local	ANYURI		##targetnamespace
					insert	remove	
	##any	nix	maxoccurs = 0: nix, sonst: immer WP (del value?)	maxoccurs = 0: nix, sonst: immer WP (del value?)	maxoccurs = 0: nix, sonst: immer WP (del value?)	nicht möglich	maxoccurs = 0: nix, sonst: immer WP (del value?)
	##other	nix	nix	maxoccurs = 0: nix, sonst: immer WP (del value?)	maxoccurs = 0: nix, sonst: immer WP (del value?)	nicht möglich	maxoccurs = 0: nix, sonst: immer WP (del value?)
	##local	nix	nix	nix	nix	nicht möglich	nix
	ANYURI	nix	maxoccurs = 0: nix, sonst: immer WP (del value?)	nix	nix	maxoccurs = 0: nix, sonst: immer WP (del value?)	nix
	##targetnamespace	nix	maxoccurs = 0: nix, sonst: immer WP (del value?)	nix	nix	nicht möglich	nix

neu ->

aktuell	notNamespace	ANYURI		##targetnamespace	##local
		insert	remove		
	ANYURI	maxoccurs = 0: nix, sonst: immer WP (del value?)	nix	maxoccurs = 0: nix, sonst: immer WP (del value?)	nix
	##targetnamespace	maxoccurs = 0: nix, sonst: immer WP (del value?)	nicht möglich	nix	nix
	##local	maxoccurs = 0: nix, sonst: immer WP (del value?)	nicht möglich	maxoccurs = 0: nix, sonst: immer WP (del value?)	nix

neu ->

aktuell	processcontent	lax	skip	strict
	lax	nix	nix	maxoccurs = 0: nix, sonst: immer WP (del value?)
	skip	nix	nix	maxoccurs = 0: nix, sonst: immer WP (del value?)
	strict	nix	nix	nix

neu ->

aktuell	minoccurs	min--	min++
	0	nicht möglich	add value
	1	nix	immer WP: add value?
	n	nix	immer WP: add value?

neu ->

aktuell	maxoccurs	max--	max++
	0	nicht möglich	nix
	1	immer WP: del value?	nix
	n	immer WP: del value?	nix

Abbildung A.22.: Extraliste der ELaX-Operation updelementwildcard aus [Nös15a]

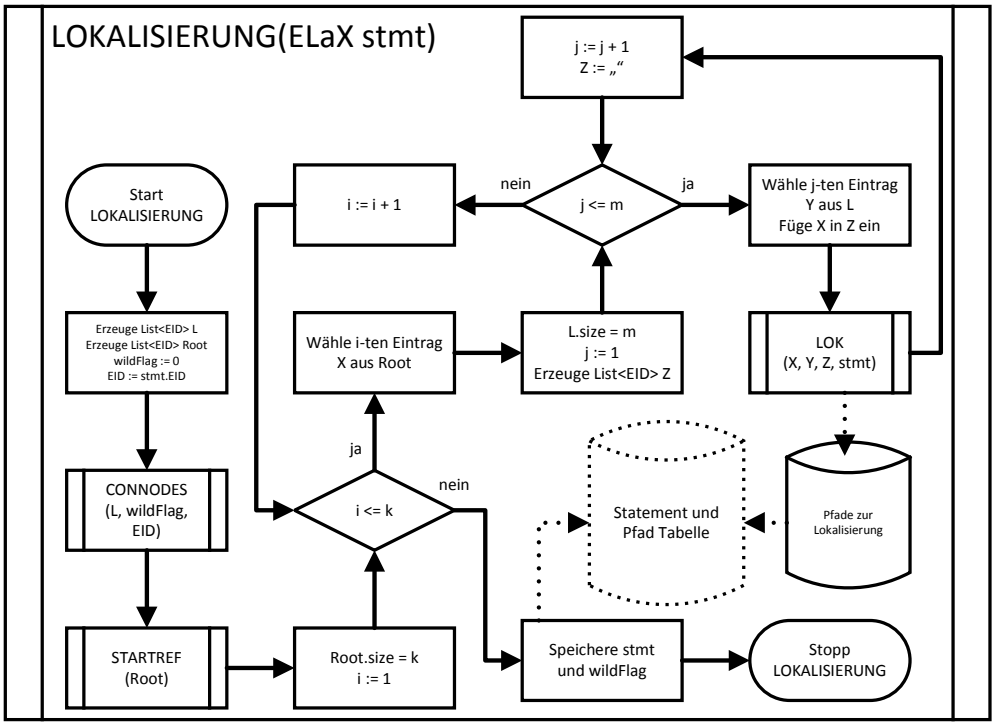


Abbildung A.23.: PAP - Lokalisierung aus [Nös15c]

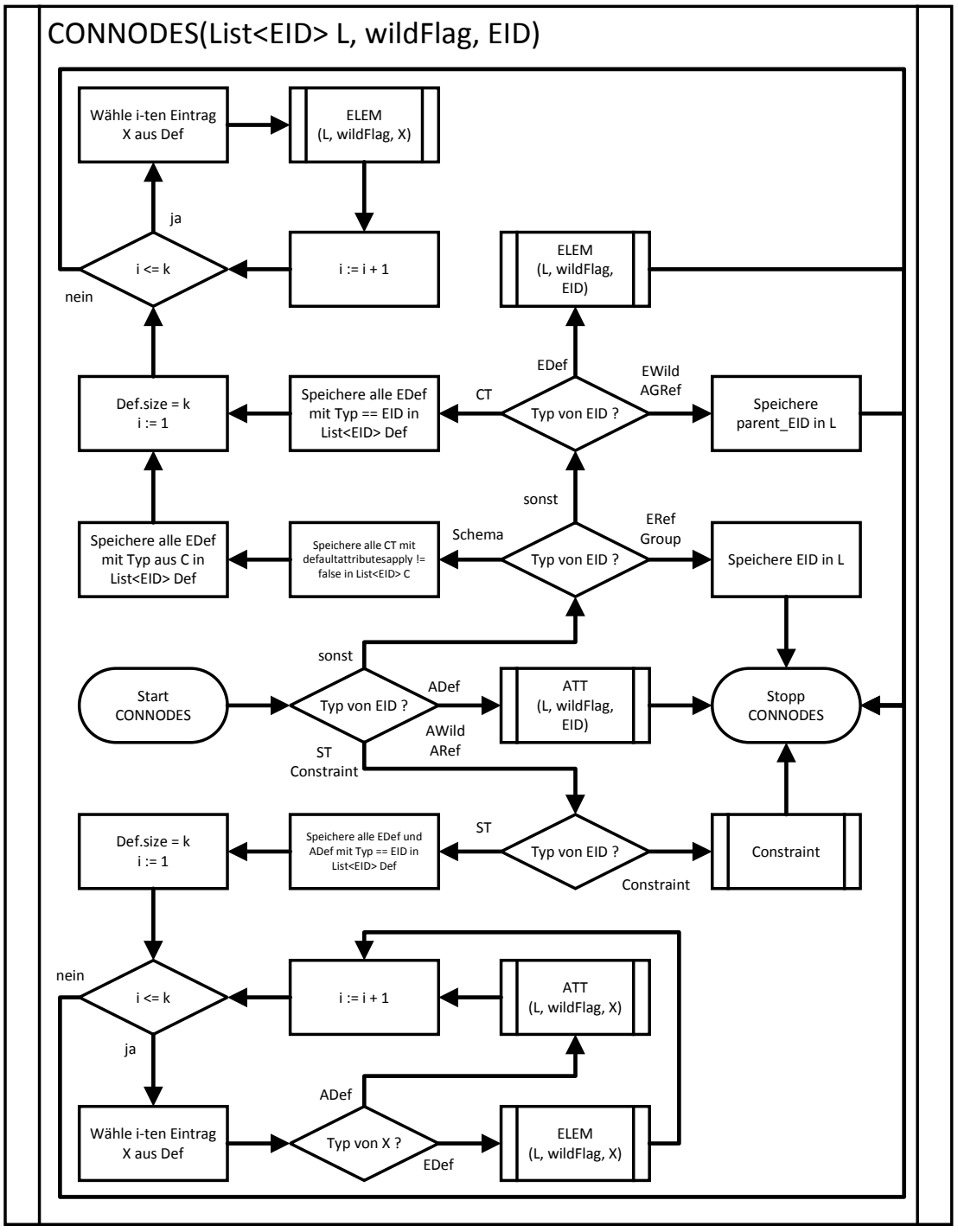


Abbildung A.24.: PAP - CONNODES aus [Nös15c]

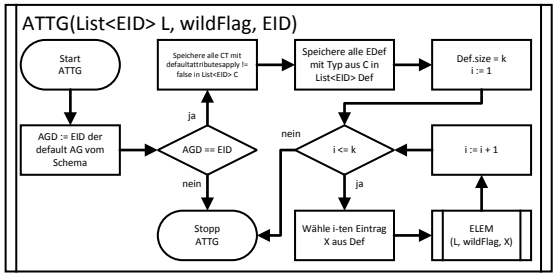


Abbildung A.25.: PAP - ATTG aus [Nös15c]

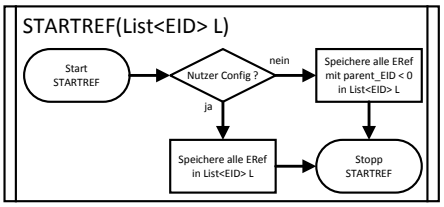


Abbildung A.26.: PAP - STARTREF aus [Nös15c]

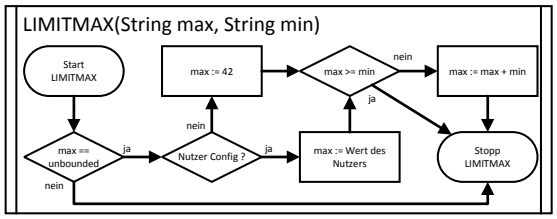


Abbildung A.27.: PAP - LIMITMAX aus [Nös15c]

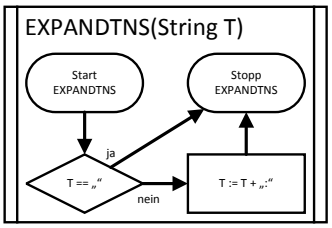


Abbildung A.28.: PAP - EXPANDTNS aus [Nös15c]

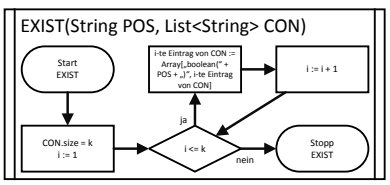


Abbildung A.29.: PAP - EXIST aus [Nös15c]

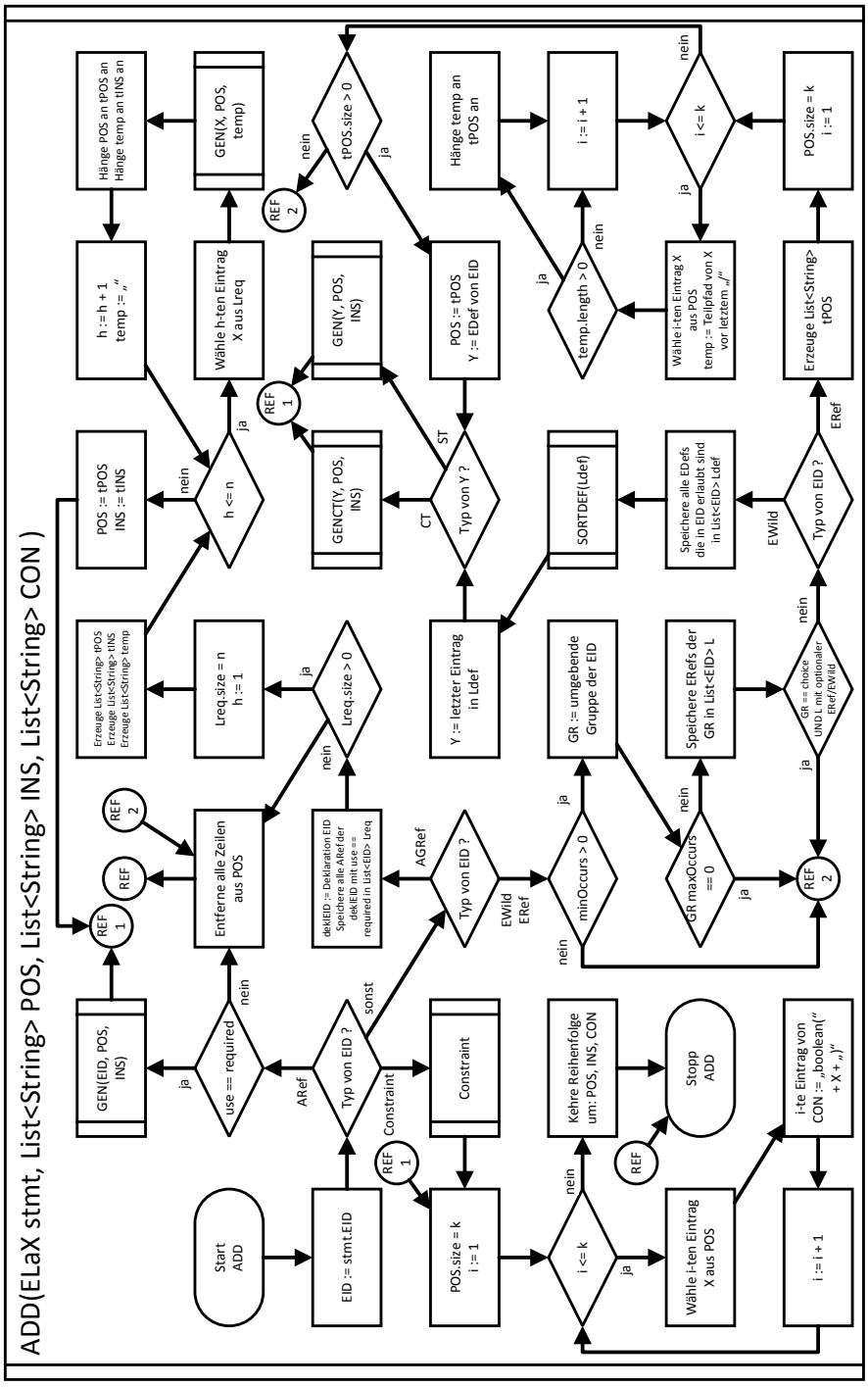


Abbildung A.30.: PAP - ADD aus [Nös15c]

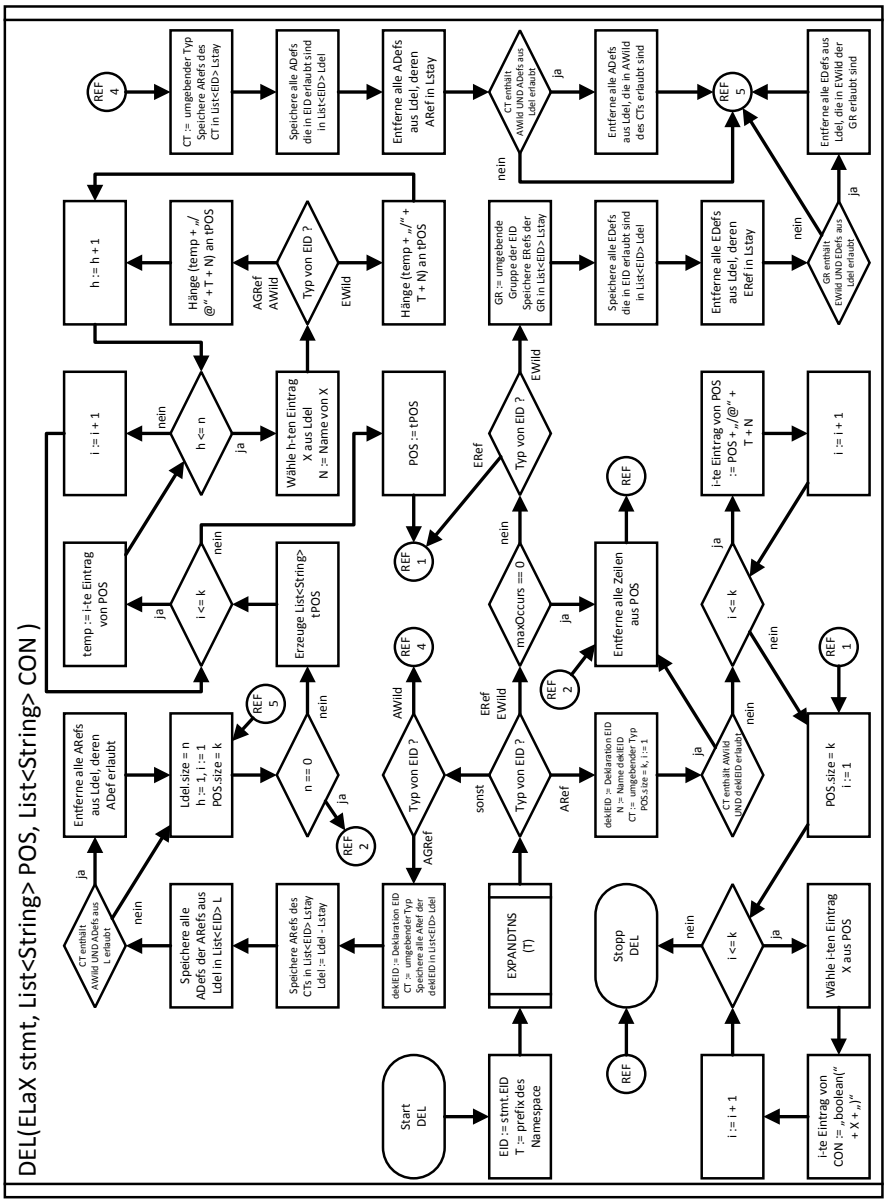


Abbildung A.31.: PAP - DEL aus [Nös15c]

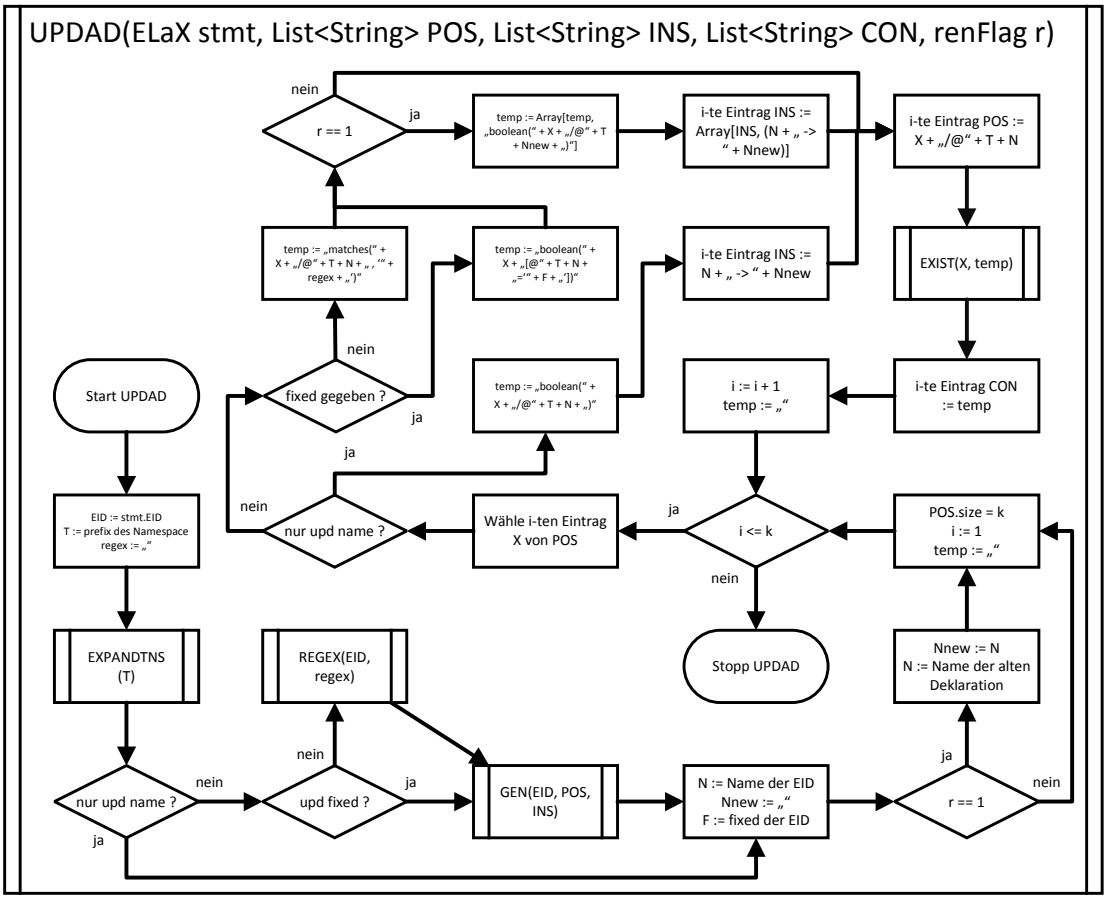


Abbildung A.32.: PAP - UPDAD aus [Nös15c]

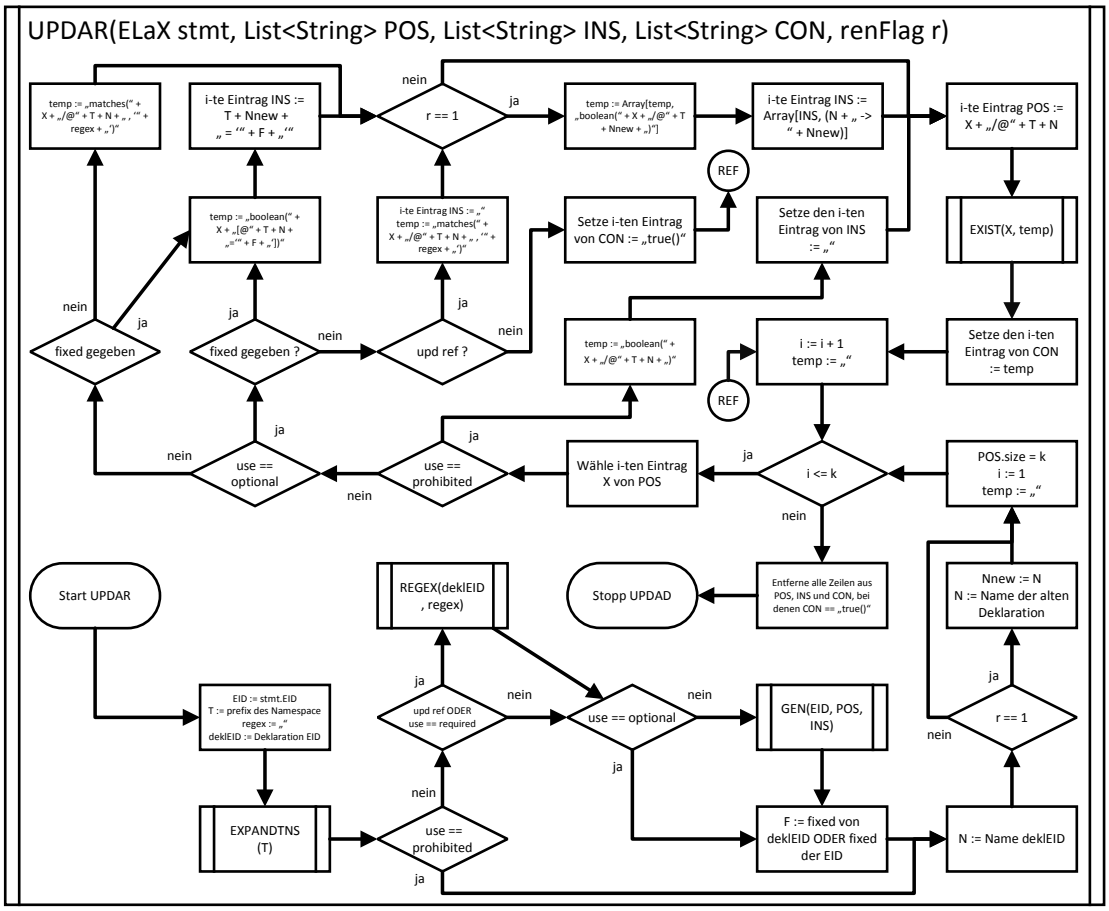


Abbildung A.33.: PAP - UPDAR aus [Nös15c]

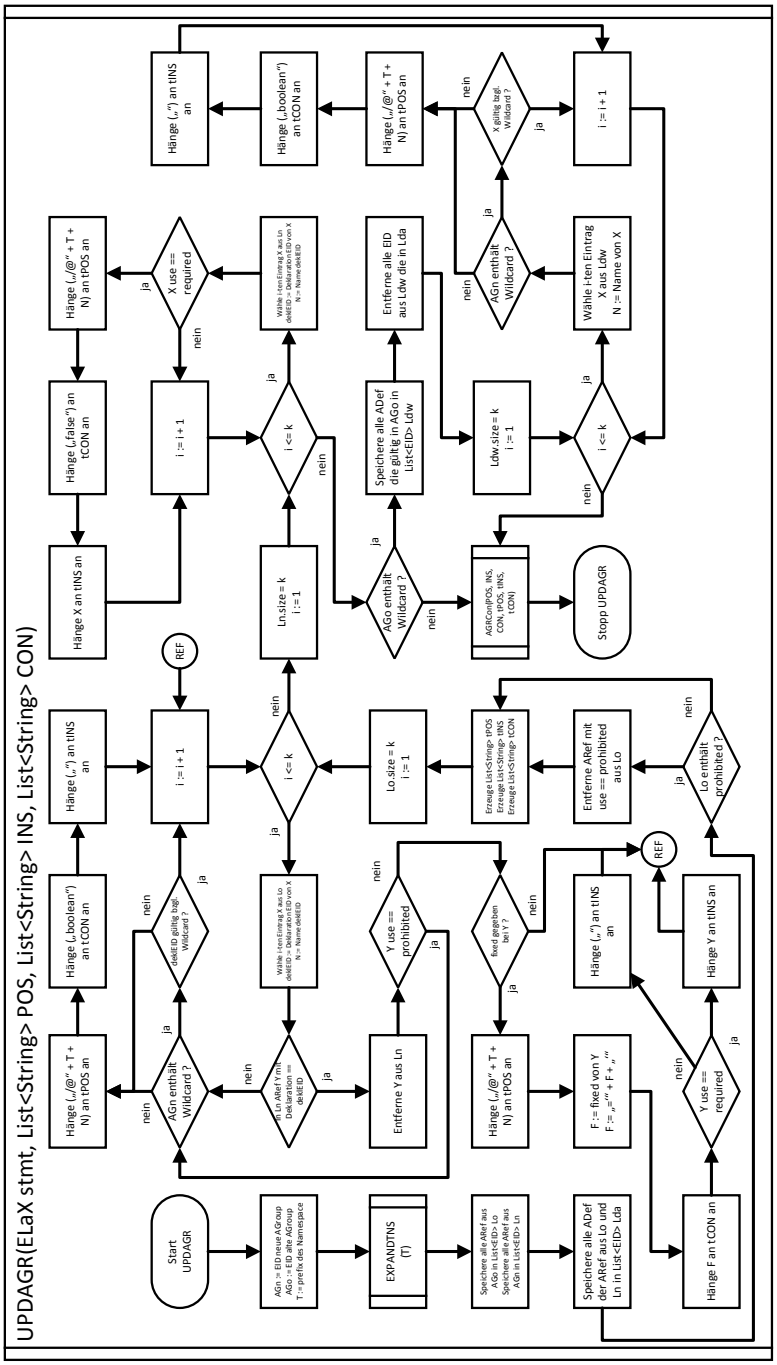


Abbildung A.34.: PAP - UPDAGR aus [Nös15c]

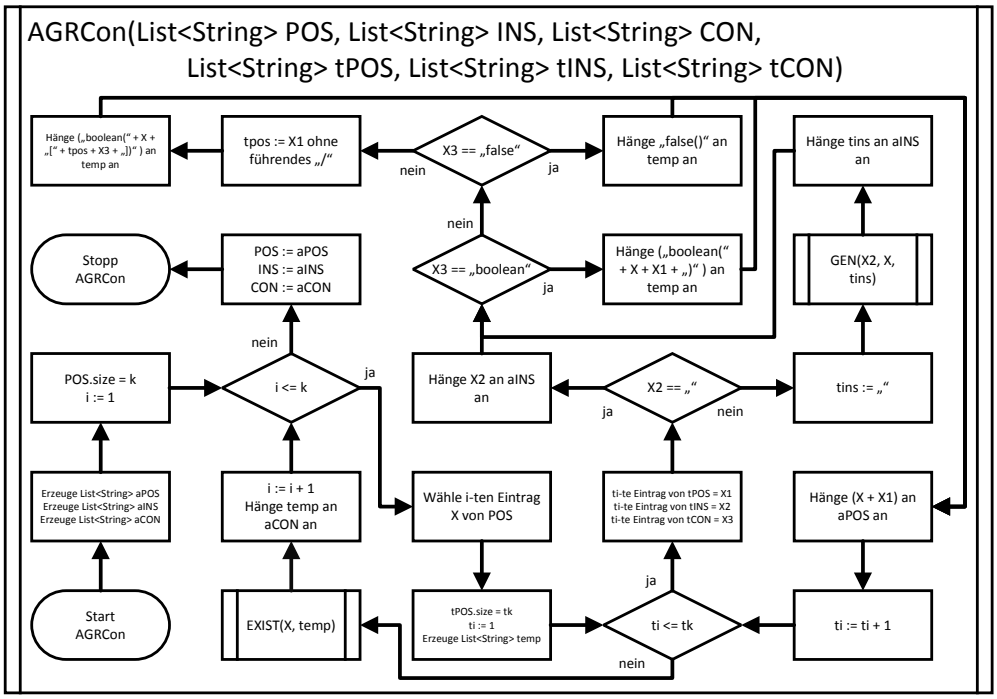


Abbildung A.35.: PAP - AGRCon aus [Nös15c]

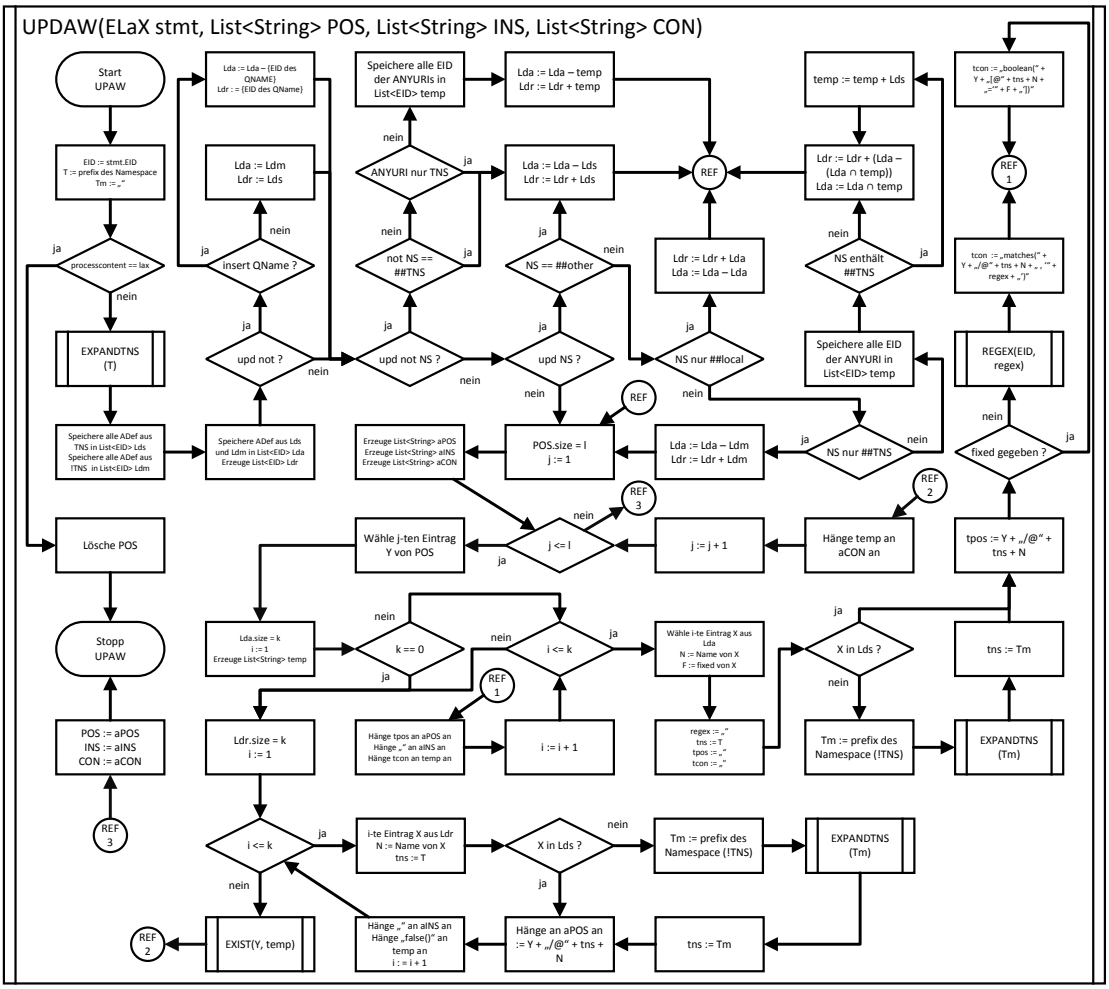
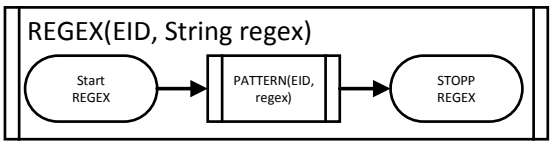


Abbildung A.36.: PAP - UPDAW aus [Nös15c]



PATTERN(EID, regex):
Die *checkDomainIntegrity()* Funktion aus studentischer Arbeit *2013_Integration von Integritaetsbedingungen bei der XML-Schemaevolution(MA Hannes Grunert).pdf* müsste angepasst werden, um statt *true* und *false* ein Pattern des obersten built-in Grundtypen auszugeben (Dissertation: *Abbildung A.1.: Built-in Typen mit Facetten gemäß [PGM+12]*). Siehe auch: *2014_Management von Typhierarchien in der XML-Schemaevolution(MA Chris Kaping).pdf*

Abbildung A.37.: PAP - REGEX aus [Nös15c]

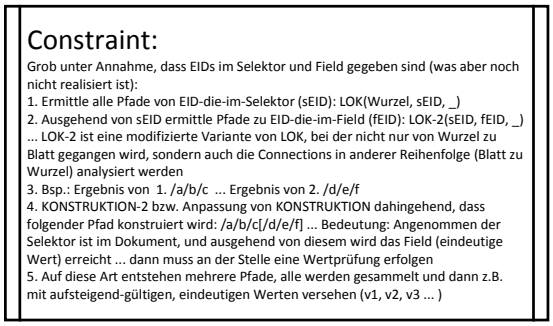


Abbildung A.38.: PAP - Constraint aus [Nös15c]

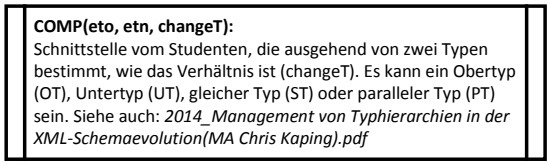


Abbildung A.39.: PAP - COMP aus [Nös15c]

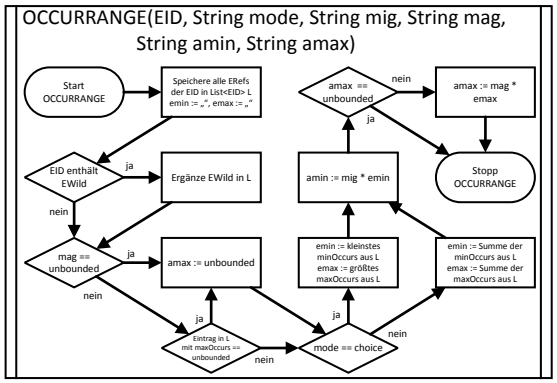


Abbildung A.40.: PAP - OCCURRANGE aus [Nös15c]

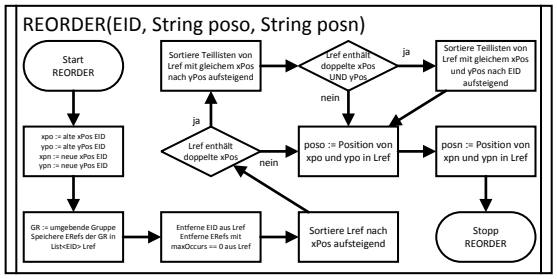


Abbildung A.41.: PAP - REORDER aus [Nös15c]

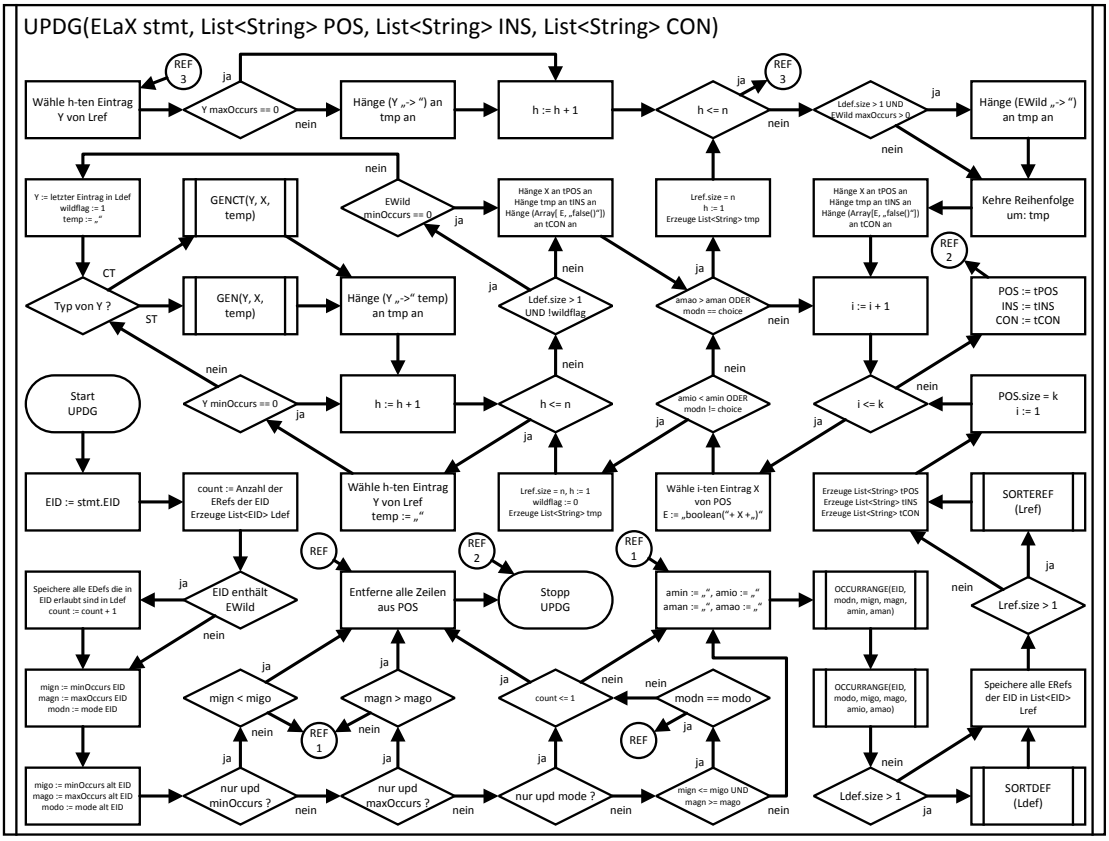


Abbildung A.42.: PAP - UPDG aus [Nös15c]

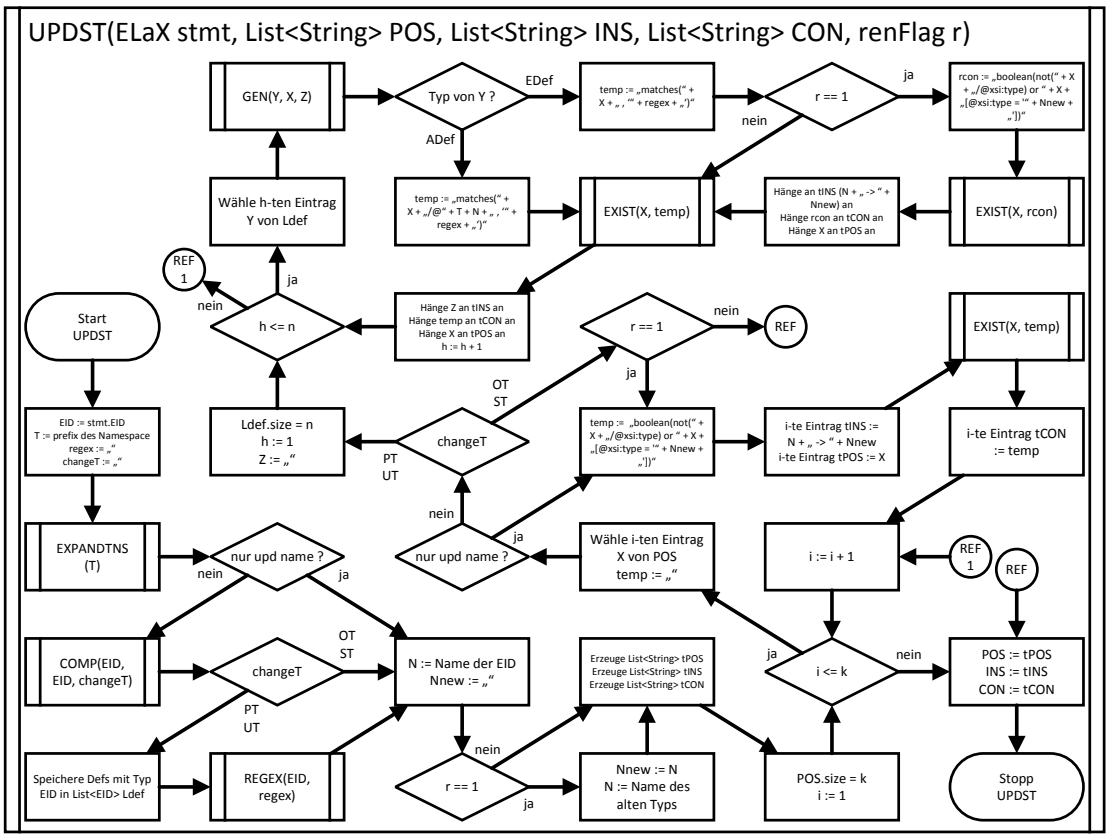


Abbildung A.43.: PAP - UPDST aus [Nös15c]

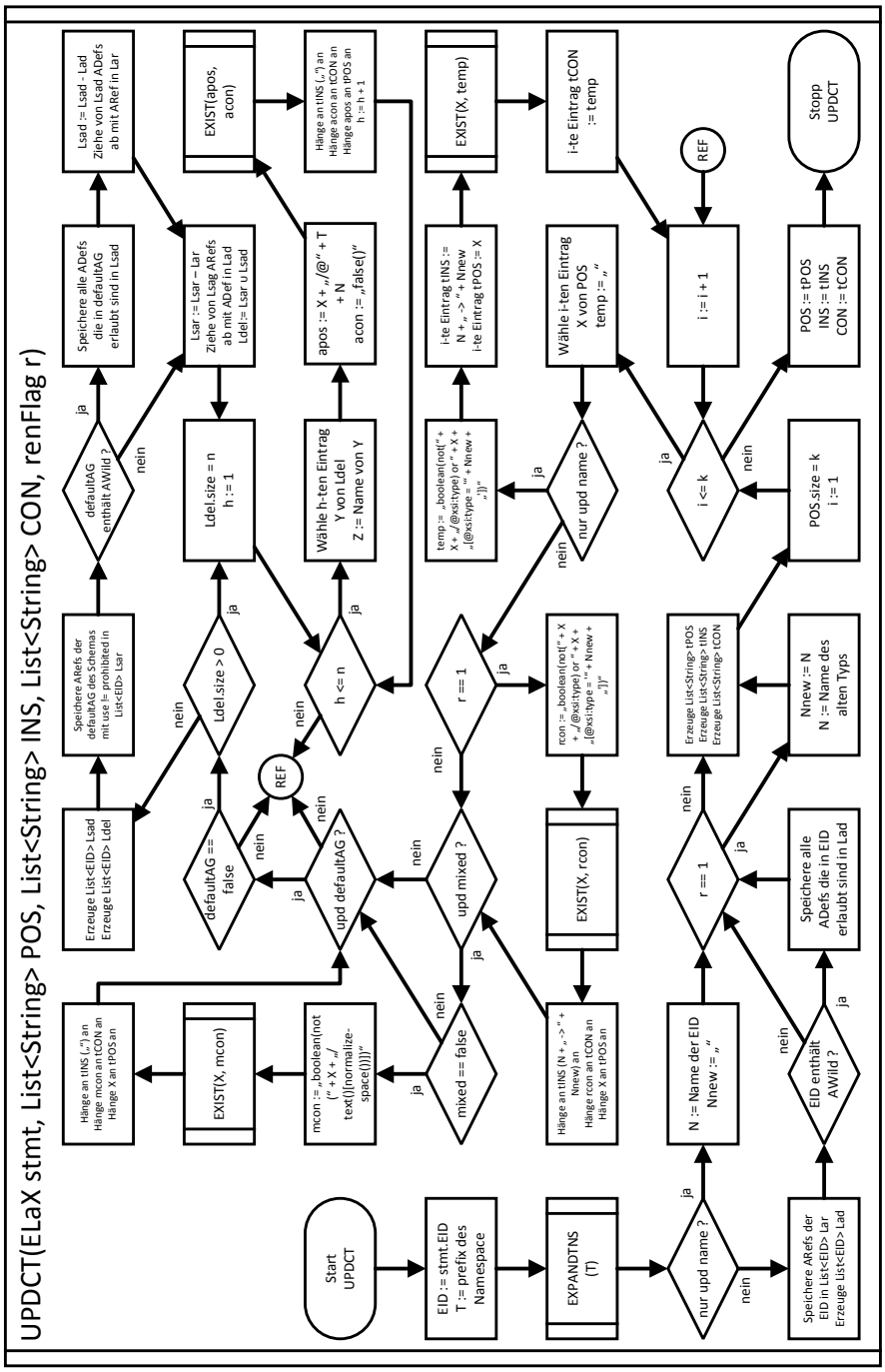


Abbildung A.44.: PAP - UPDCT aus [Nös15c]

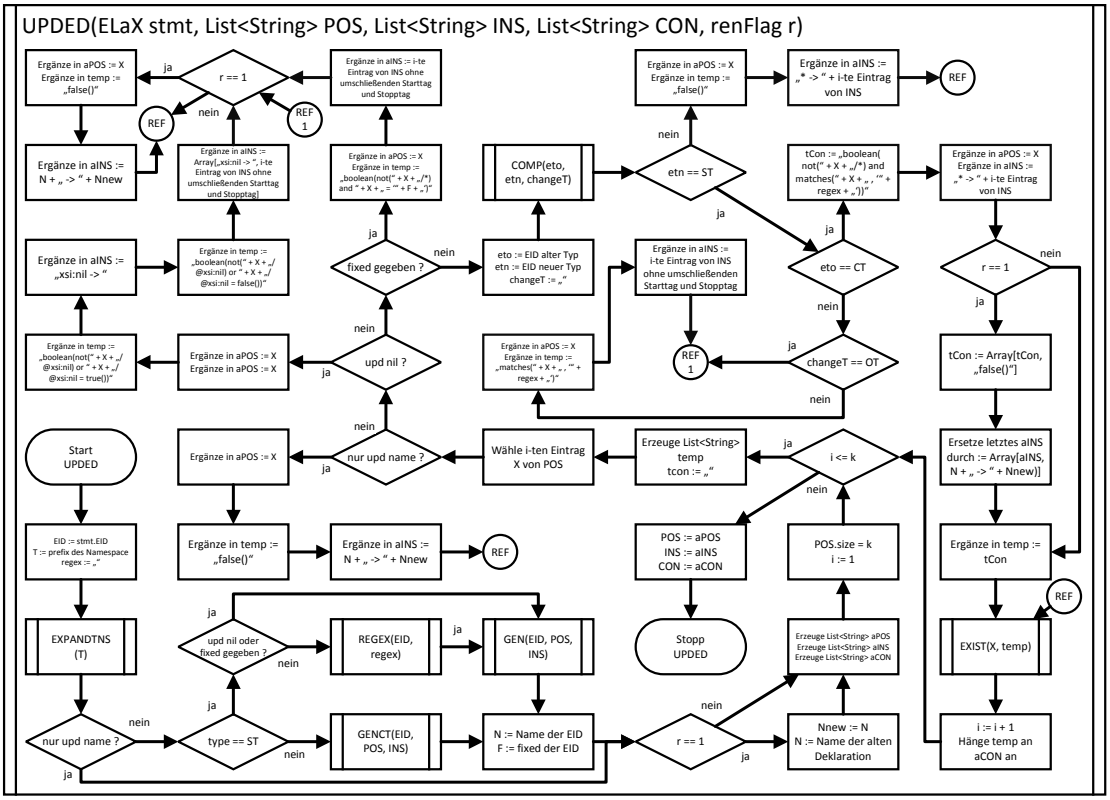


Abbildung A.45.: PAP - UPDED aus [Nös15c]

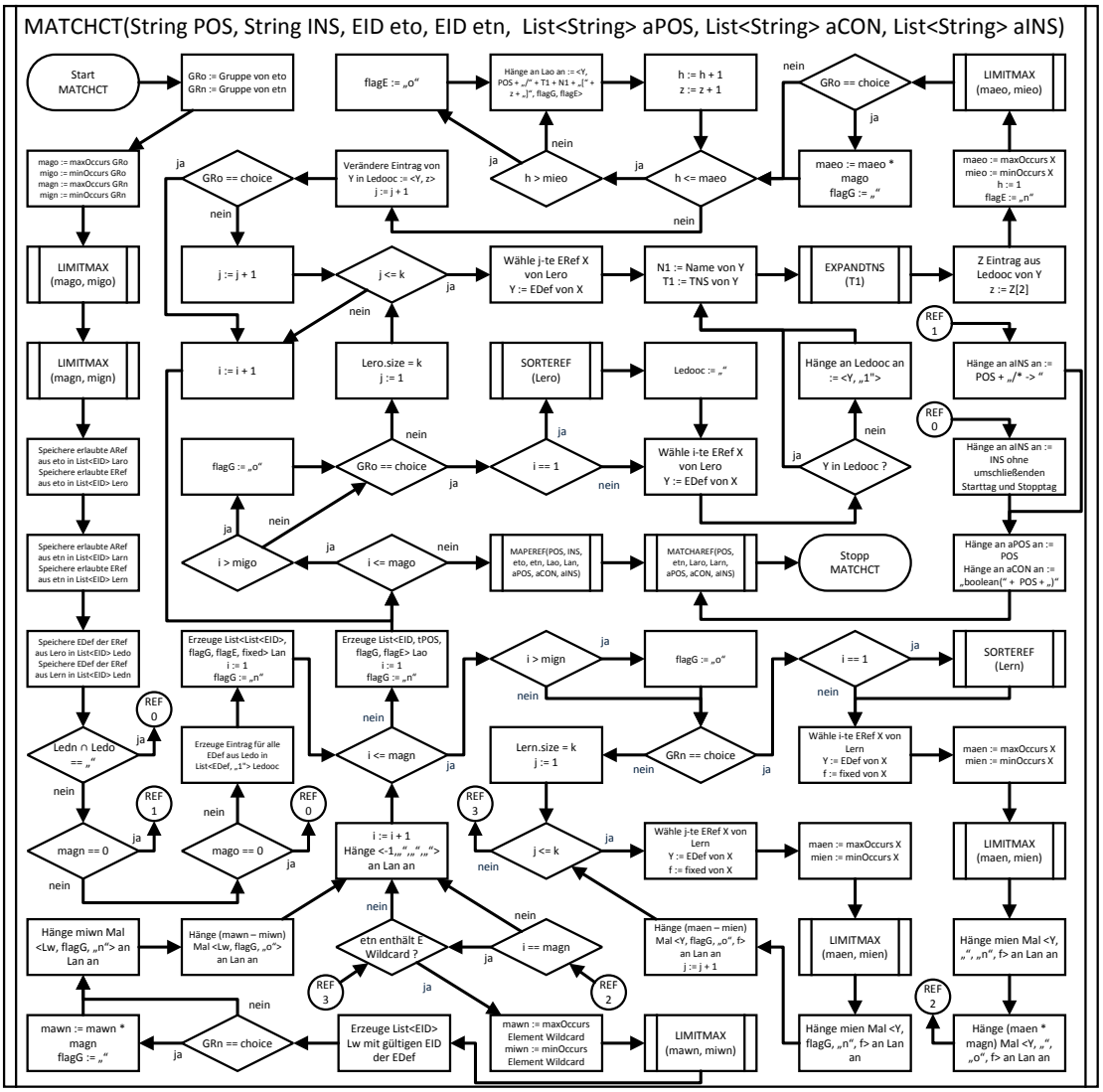


Abbildung A.46.: PAP - MATCHCT aus [Nös15c]

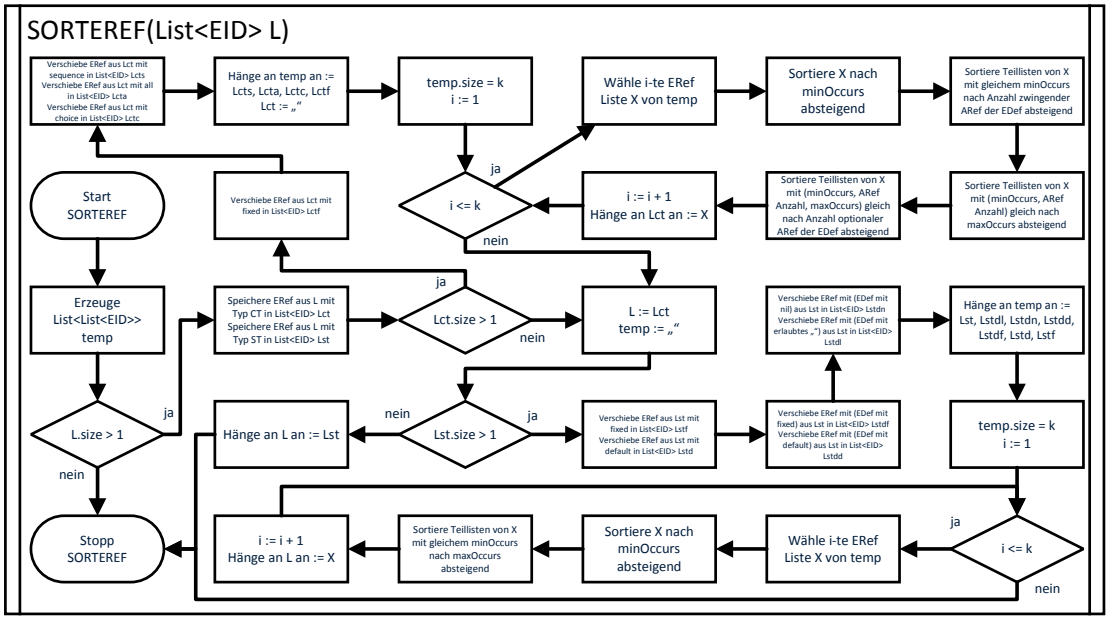


Abbildung A.47.: PAP - SORTEREF aus [Nös15c]

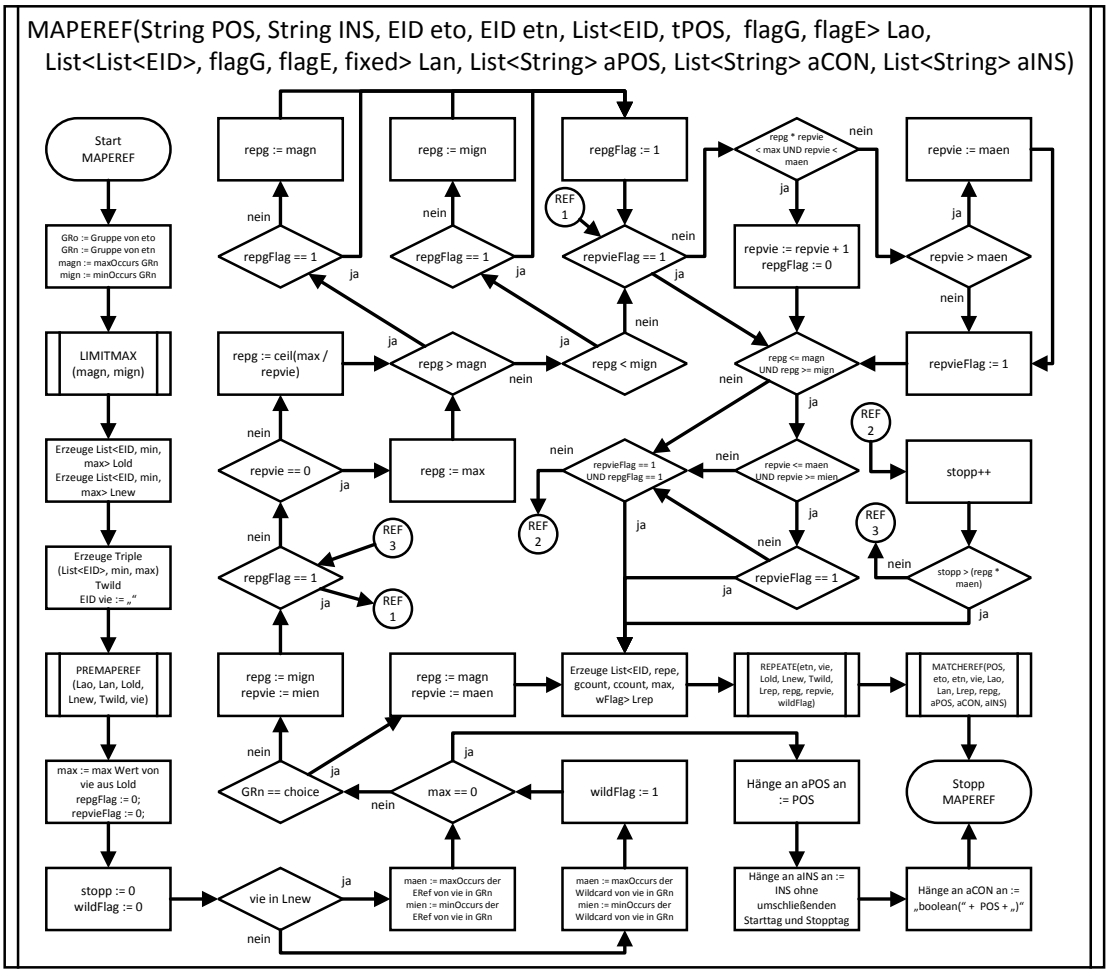


Abbildung A.48.: PAP - MAPEREF aus [Nös15c]

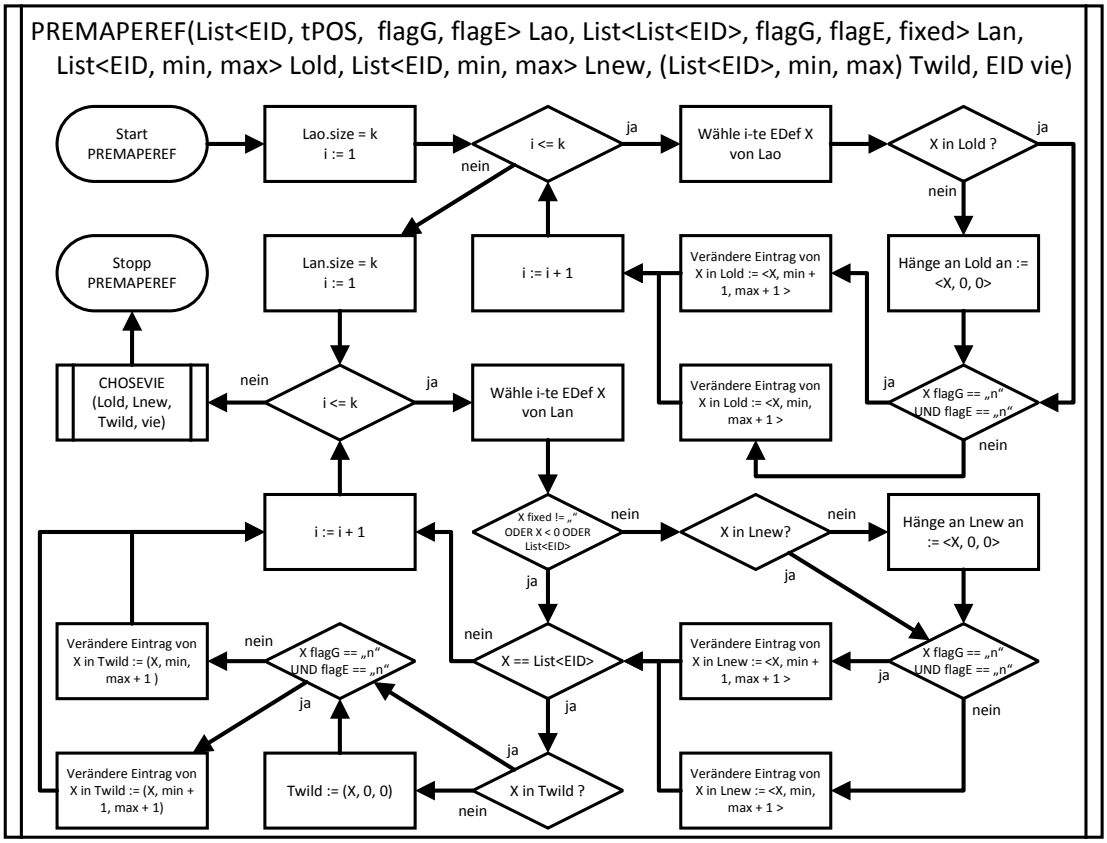


Abbildung A.49.: PAP - PREMAPEREF aus [Nös15c]

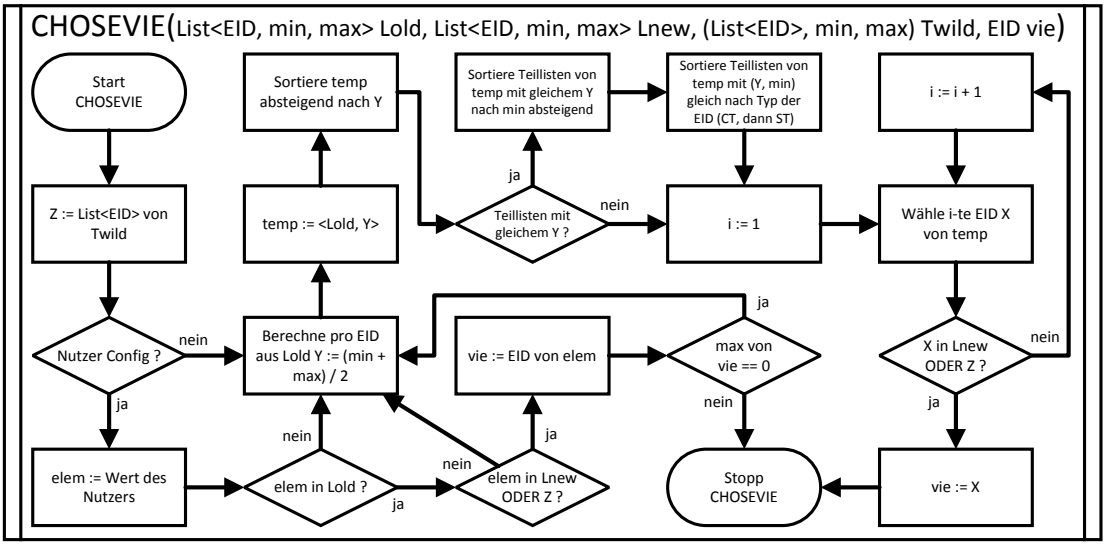


Abbildung A.50.: PAP - CHOSEVIE aus [Nös15c]

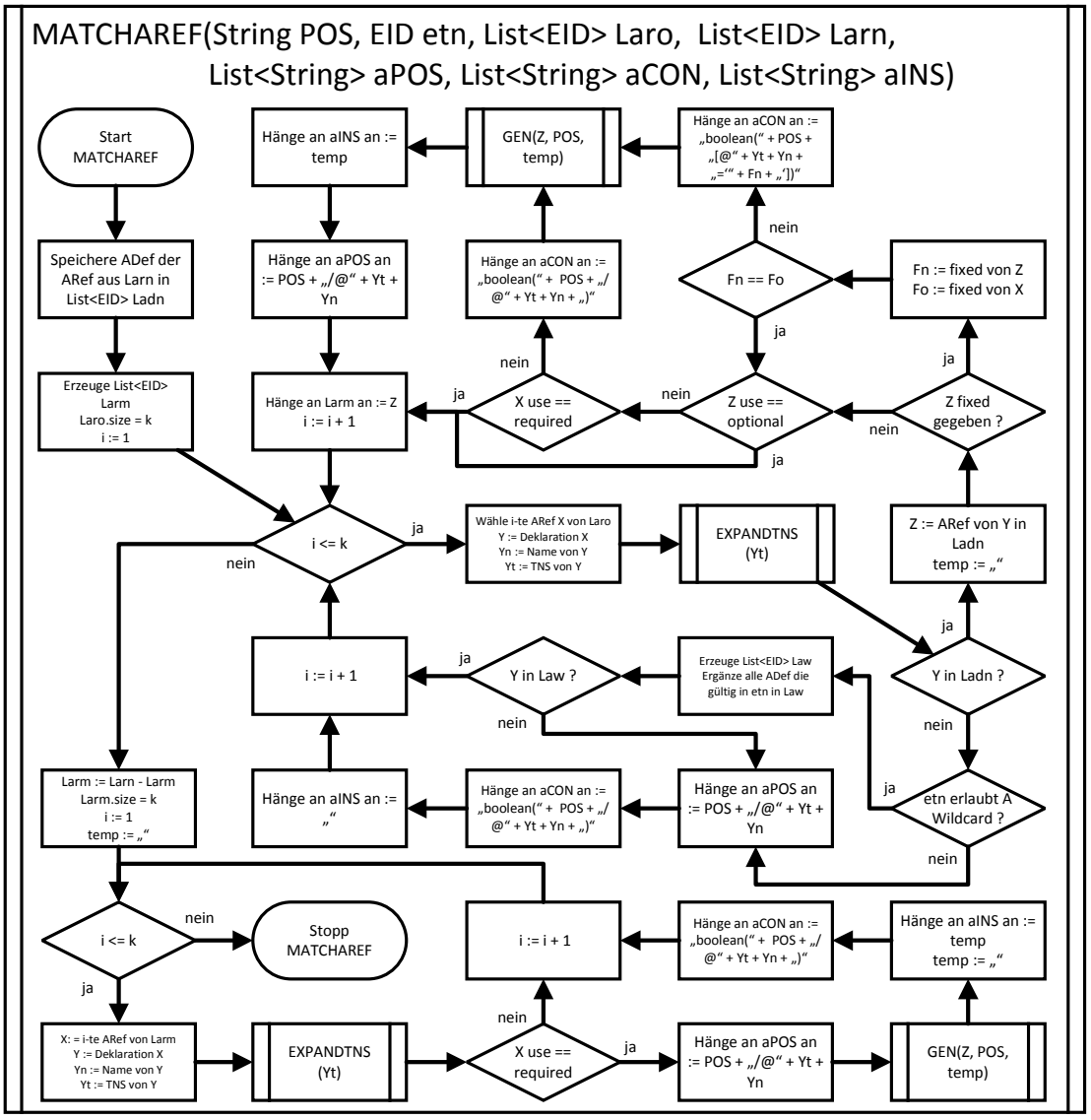


Abbildung A.51.: PAP - MATCHAREF aus [Nös15c]

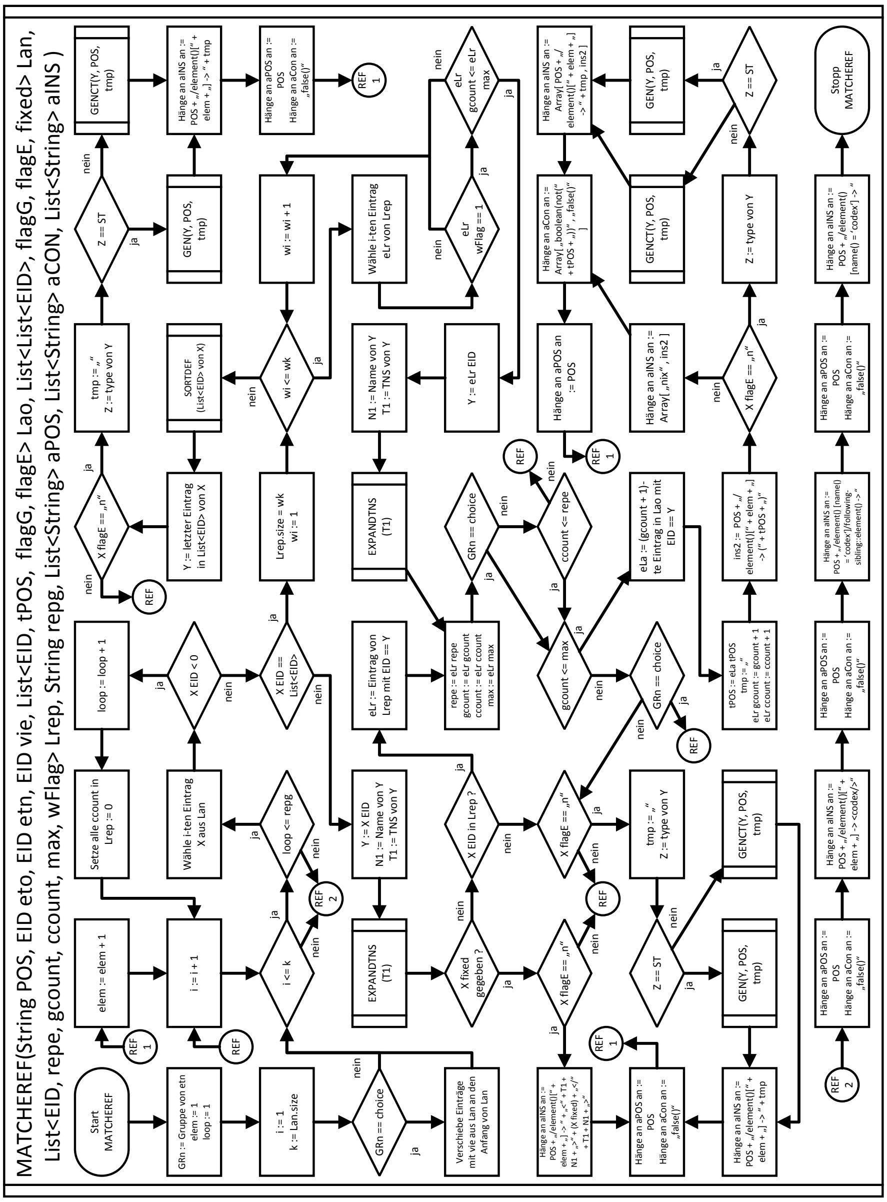


Abbildung A.52.: PAP - MATCHEREF aus [Nös15c]

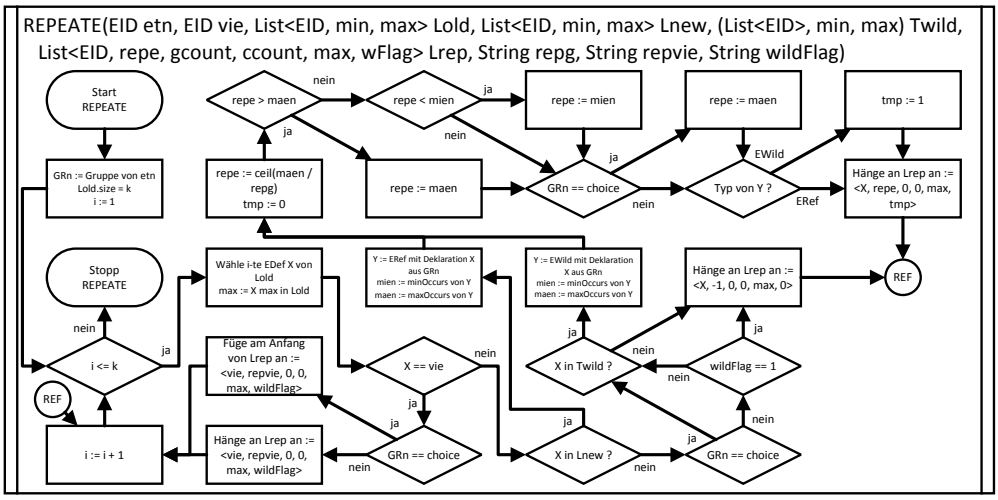


Abbildung A.53.: PAP - REPEATE aus [Nös15c]

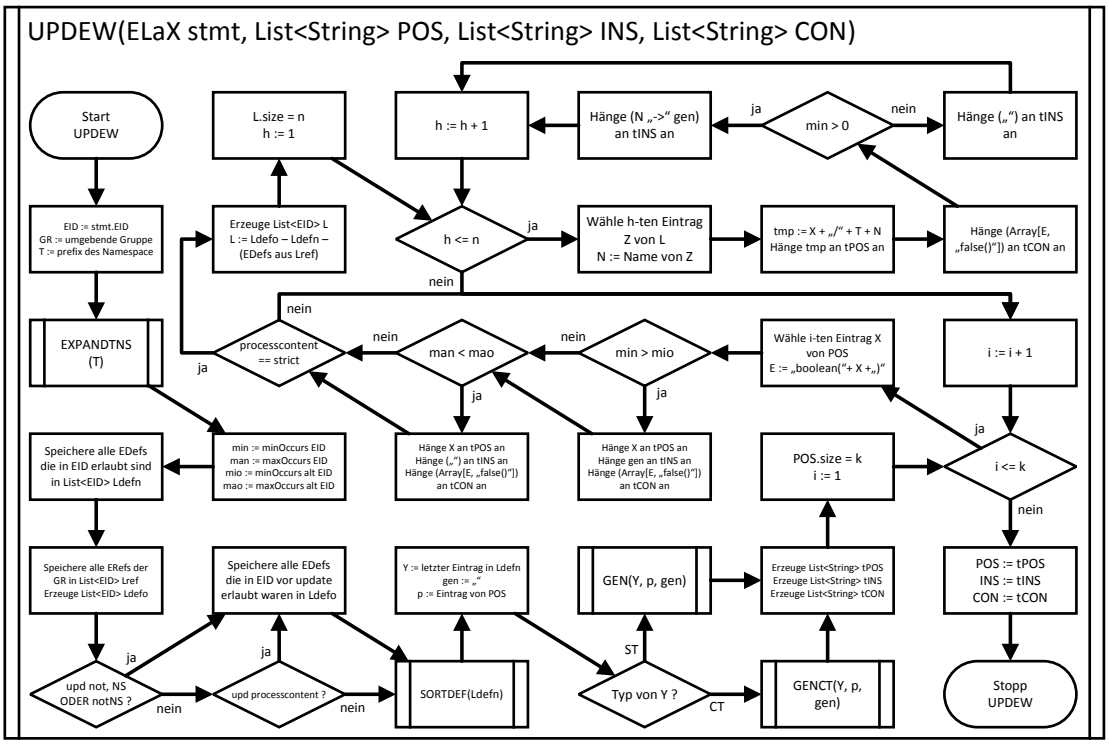


Abbildung A.54.: PAP - UPDEW aus [Nös15c]

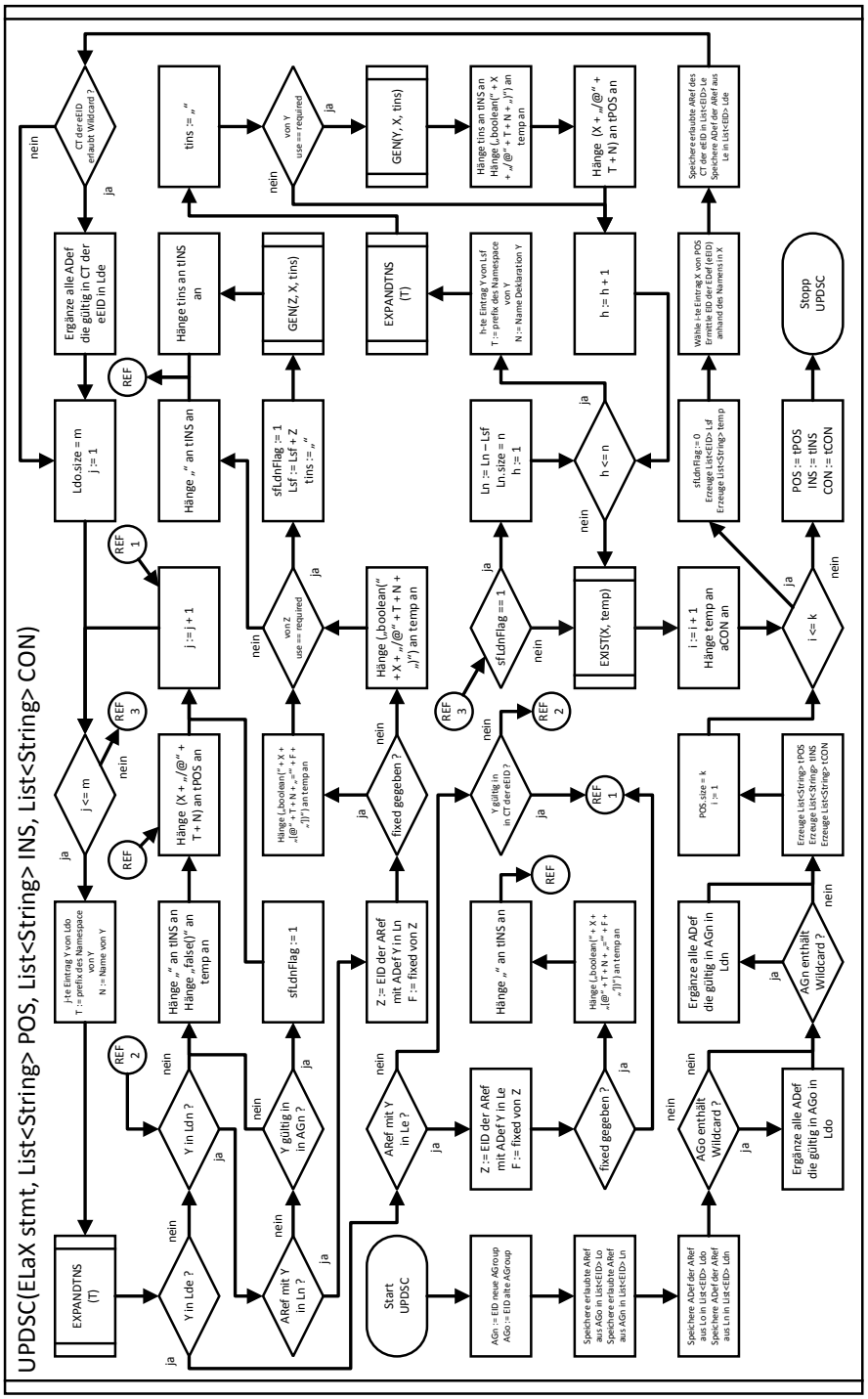


Abbildung A.55.: PAP - UPDSC aus [Nös15c]

```
<?xml version="1.0" encoding="UTF-8"?>
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns:vc="http://www.w3.org/2007/XMLSchema-versioning"
      vc:minVersion="1.1" xmlns:cx="file://codex-null.xsd"
      targetNamespace="file://codex-null.xsd">
        <xs:simpleType name="null">
                <xs:restriction base="xs:string">
                    <xs:enumeration value="null"/>
                    <xs:enumeration value="exist"/>
                    <xs:enumeration value="never"/>
                    <xs:enumeration value=""/>
                </xs:restriction>
        </xs:simpleType>
        <xs:simpleType name="string">
                <xs:union memberTypes="xs:string␣cx:null"/>
        </xs:simpleType>
        <xs:simpleType name="decimal">
                <xs:union memberTypes="xs:decimal␣cx:null"/>
        </xs:simpleType>
        <xs:simpleType name="dateTime">
                <xs:union memberTypes="xs:dateTime␣cx:null"/>
        </xs:simpleType>
        <xs:simpleType name="integer">
                <xs:union memberTypes="xs:integer␣cx:null"/>
        </xs:simpleType>
        <xs:simpleType name="long">
                <xs:union memberTypes="xs:long␣cx:null"/>
        </xs:simpleType>
        <xs:simpleType name="int">
                <xs:union memberTypes="xs:int␣cx:null"/>
        </xs:simpleType>
        <xs:simpleType name="short">
                <xs:union memberTypes="xs:short␣cx:null"/>
        </xs:simpleType>
        <xs:simpleType name="byte">
                <xs:union memberTypes="xs:byte␣cx:null"/>
        </xs:simpleType>
        <xs:simpleType name="nonNegativeInteger">
                <xs:union memberTypes="xs:nonNegativeInteger␣cx:null"/>
        </xs:simpleType>
        <xs:simpleType name="positiveInteger">
                <xs:union memberTypes="xs:positiveInteger␣cx:null"/>
        </xs:simpleType>
        <xs:simpleType name="unsignedLong">
                <xs:union memberTypes="xs:unsignedLong␣cx:null"/>
        </xs:simpleType>
        <xs:simpleType name="unsignedInt">
                <xs:union memberTypes="xs:unsignedInt␣cx:null"/>
        </xs:simpleType>
        <xs:simpleType name="unsignedShort">
                <xs:union memberTypes="xs:unsignedShort␣cx:null"/>
        </xs:simpleType>
        <xs:simpleType name="unsignedByte">
```

```
        <xs:union memberTypes="xs:unsignedByte cx:null"/>
</xs:simpleType>
<xs:simpleType name="nonPositiveInteger">
        <xs:union memberTypes="xs:nonPositiveInteger cx:null"/>
</xs:simpleType>
<xs:simpleType name="negativeInteger">
        <xs:union memberTypes="xs:negativeInteger cx:null"/>
</xs:simpleType>
<xs:simpleType name="base64Binary">
        <xs:union memberTypes="xs:base64Binary cx:null"/>
</xs:simpleType>
<xs:simpleType name="boolean">
        <xs:union memberTypes="xs:boolean cx:null"/>
</xs:simpleType>
<xs:simpleType name="date">
        <xs:union memberTypes="xs:date cx:null"/>
</xs:simpleType>
<xs:simpleType name="double">
        <xs:union memberTypes="xs:double cx:null"/>
</xs:simpleType>
<xs:simpleType name="duration">
        <xs:union memberTypes="xs:duration cx:null"/>
</xs:simpleType>
<xs:simpleType name="float">
        <xs:union memberTypes="xs:float cx:null"/>
</xs:simpleType>
<xs:simpleType name="gDay">
        <xs:union memberTypes="xs:gDay cx:null"/>
</xs:simpleType>
<xs:simpleType name="gMonth">
        <xs:union memberTypes="xs:gMonth cx:null"/>
</xs:simpleType>
<xs:simpleType name="gMonthDay">
        <xs:union memberTypes="xs:gMonthDay cx:null"/>
</xs:simpleType>
<xs:simpleType name="gYear">
        <xs:union memberTypes="xs:gYear cx:null"/>
</xs:simpleType>
<xs:simpleType name="gYearMonth">
        <xs:union memberTypes="xs:gYearMonth cx:null"/>
</xs:simpleType>
<xs:simpleType name="hexBinary">
        <xs:union memberTypes="xs:hexBinary cx:null"/>
</xs:simpleType>
<xs:simpleType name="QName">
        <xs:union memberTypes="xs:QName cx:null"/>
</xs:simpleType>
<xs:simpleType name="time">
        <xs:union memberTypes="xs:time cx:null"/>
</xs:simpleType>
<xs:simpleType name="normalizedString">
        <xs:union memberTypes="xs:normalizedString cx:null"/>
</xs:simpleType>
```

```
    <xs:simpleType name="token">
            <xs:union memberTypes="xs:token␣cx:null"/>
    </xs:simpleType>
    <xs:simpleType name="language">
            <xs:union memberTypes="xs:language␣cx:null"/>
    </xs:simpleType>
    <xs:simpleType name="Name">
            <xs:union memberTypes="xs:Name␣cx:null"/>
    </xs:simpleType>
    <xs:simpleType name="NCName">
            <xs:union memberTypes="xs:NCName␣cx:null"/>
    </xs:simpleType>
    <xs:simpleType name="ENTITY">
            <xs:union memberTypes="xs:ENTITY␣cx:null"/>
    </xs:simpleType>
    <xs:simpleType name="ID">
            <xs:union memberTypes="xs:ID␣cx:null"/>
    </xs:simpleType>
    <xs:simpleType name="IDREF">
            <xs:union memberTypes="xs:IDREF␣cx:null"/>
    </xs:simpleType>
    <xs:simpleType name="NMTOKEN">
            <xs:union memberTypes="xs:NMTOKEN␣cx:null"/>
    </xs:simpleType>
    <xs:simpleType name="anyURI">
            <xs:union memberTypes="xs:anyURI␣cx:null"/>
    </xs:simpleType>
    <xs:simpleType name="NMTOKENS">
            <xs:union memberTypes="xs:NMTOKENS␣cx:null"/>
    </xs:simpleType>
    <xs:simpleType name="IDREFS">
            <xs:union memberTypes="xs:IDREFS␣cx:null"/>
    </xs:simpleType>
    <xs:simpleType name="ENTITIES">
            <xs:union memberTypes="xs:ENTITIES␣cx:null"/>
    </xs:simpleType>
    <xs:simpleType name="NOTATION">
            <xs:union memberTypes="xs:NOTATION␣cx:null"/>
    </xs:simpleType>
    <xs:simpleType name="yearMonthDuration">
            <xs:union memberTypes="xs:yearMonthDuration␣cx:null"/>
    </xs:simpleType>
    <xs:simpleType name="dayTimeDuration">
            <xs:union memberTypes="xs:dayTimeDuration␣cx:null"/>
    </xs:simpleType>
    <xs:simpleType name="dateTimeStamp">
            <xs:union memberTypes="xs:dateTimeStamp␣cx:null"/>
    </xs:simpleType>
</xs:schema>
```

XML-Beispiel A.13: XML-Schema Nullwerte: `http://www.ls-dbis.de/codex`

Statement	POS	INS	CON	w	r
update attribute name 'a1' change fixed 'fixed' ;	/root[1]/@a1	a1='fixed'	[boolean(/root[1]/@a1), boolean(/root[1][@a1='fixed'])]	1	0
update attribute name 'a1' change fixed 'fixed' ;	/root[1]/e1[1]/@a1	a1='fixed'	[boolean(/root[1]/e1[1]/@a1), boolean(/root[1]/e1[1][@a1='fixed'])]	1	0
update attribute name 'a1' change fixed 'fixed' ;	/root[1]/e1[2]/@a1	a1='fixed'	[boolean(/root[1]/e1[2]/@a1), boolean(/root[1]/e1[2][@a1='fixed'])]	1	0
update attribute name 'a2' change type 'xs:integer' ;	/root[1]/@a2	a2=''	[boolean(/root[1]/@a2), matches(/root[1]/@a2, '^(\+|-)?([0-9])+')]	1	0
update attribute name 'a2' change type 'xs:integer' ;	/root[1]/e1[1]/@a2	a2=''	[boolean(/root[1]/e1[1]/@a2), matches(/root[1]/e1[1]/@a2, '^(\+|-)?([0-9])+')]	1	0
update attribute name 'a2' change type 'xs:integer' ;	/root[1]/e1[2]/@a2	a2=''	[boolean(/root[1]/e1[2]/@a2), matches(/root[1]/e1[2]/@a2, '^(\+|-)?([0-9])+')]	1	0
delete attributeref at '12' ;	/root[1]/@a3		boolean(/root[1]/@a3)	0	0
delete attributeref at '12' ;	/root[1]/e1[1]/@a3		boolean(/root[1]/e1[1]/@a3)	0	0
delete attributeref at '12' ;	/root[1]/e1[2]/@a3		boolean(/root[1]/e1[2]/@a3)	0	0
update elementref 'e1' at '16' change xpos '2' ypos '2' ;	/root[1]/*	1 -> 2	[boolean(/root[1]/*), false()]	0	0
update elementref 'e2' at '17' change minoccurs '2' ;	/root[1]/*	<e2></e2>	[boolean(/root[1]/*), false()]	0	0
add elementref 'e4' minoccurs '1' id 'EID23' in '15' ;	/root[1]	<e4></e4>	boolean(/root[1])	0	0
update elementref 'e3' at '21' change maxoccurs '1' ;	/root[1]/e1[1]/*	""	[boolean(/root[1]/e1[1]/*), false()]	0	0
update elementref 'e3' at '21' change maxoccurs '1' ;	/root[1]/e1[2]/*	""	[boolean(/root[1]/e1[2]/*), false()]	0	0
update complextype name ctype change name 'ctypex' ;	/root[1]/e1[1]	ctype -> ctypex	[boolean(/root[1]/e1[1]), boolean(not(/root[1]/e1[1]/@xsi:type) or /root[1]/e1[1][@xsi:type ='ctypex'])]	0	1
update complextype name ctype change name 'ctypex' ;	/root[1]/e1[2]	ctype -> ctypex	[boolean(/root[1]/e1[2]), boolean(not(/root[1]/e1[2]/@xsi:type) or /root[1]/e1[2][@xsi:type ='ctypex'])]	0	1
update element name 'e3' change name 'ex' ;	/root[1]/e1[2]/e3[4]	e3 -> ex	[boolean(/root[1]/e1[2]/e3[4]), false()]	0	1
update element name 'e3' change name 'ex' ;	/root[1]/e1[2]/e3[3]	e3 -> ex	[boolean(/root[1]/e1[2]/e3[3]), false()]	0	1
update element name 'e3' change name 'ex' ;	/root[1]/e1[2]/e3[2]	e3 -> ex	[boolean(/root[1]/e1[2]/e3[2]), false()]	0	1
update element name 'e3' change name 'ex' ;	/root[1]/e1[2]/e3[1]	e3 -> ex	[boolean(/root[1]/e1[2]/e3[1]), false()]	0	1
update element name 'e3' change name 'ex' ;	/root[1]/e1[1]/e3[4]	e3 -> ex	[boolean(/root[1]/e1[1]/e3[4]), false()]	0	1
update element name 'e3' change name 'ex' ;	/root[1]/e1[1]/e3[3]	e3 -> ex	[boolean(/root[1]/e1[1]/e3[3]), false()]	0	1
update element name 'e3' change name 'ex' ;	/root[1]/e1[1]/e3[2]	e3 -> ex	[boolean(/root[1]/e1[1]/e3[2]), false()]	0	1
update element name 'e3' change name 'ex' ;	/root[1]/e1[1]/e3[1]	e3 -> ex	[boolean(/root[1]/e1[1]/e3[1]), false()]	0	1

Abbildung A.56.: Transformationsschritte nach Anwendung PAP Generierung von Werten der Abbildung 6.22 (komplett im Vergleich zu Abbildung 6.29)

Time	Statement	POS	INS	CON	wf	r	Rückgabewert CON Existenz- / Matchbedingung	min XML	max XML	avg XML	Return Type
2	update attribute name 'a1' change fixed 'fixed' ;	/root[1]/@a1	a1='fixed'	[boolean(/root[1]/@a1), boolean(/root[1]/@a1='fixed')]	1	0	true / false	true / false	true / false	true / false	Node
2	update attribute name 'a1' change fixed 'fixed' ;	/root[1]/e1[1]/@a1	a1='fixed'	[boolean(/root[1]/e1[1]/@a1), boolean(/root[1]/e1[1]/@a1='fixed')]	1	0	true / false	true / false	true / false	true / false	Node
2	update attribute name 'a1' change fixed 'fixed' ;	/root[1]/e1[2]/@a1	a1='fixed'	[boolean(/root[1]/e1[2]/@a1), boolean(/root[1]/e1[2]/@a1='fixed')]	1	0	false	true / false	false	false	Node
3	update attribute name 'a2' change type 'xs:integer' ;	/root[1]/@a2	a2=''	[boolean(/root[1]/@a2), matches(/root[1]/@a2, '^(\+\-)?([0-9])+')]	1	0	false	true / false	false	false	Node
3	update attribute name 'a2' change type 'xs:integer' ;	/root[1]/e1[1]/@a2	a2=''	[boolean(/root[1]/e1[1]/@a2), matches(/root[1]/e1[1]/@a2, '^(\+\-)?([0-9])+')]	1	0	false	true / false	true / false	true / false	Node
3	update attribute name 'a2' change type 'xs:integer' ;	/root[1]/e1[2]/@a2	a2=''	[boolean(/root[1]/e1[2]/@a2), matches(/root[1]/e1[2]/@a2, '^(\+\-)?([0-9])+')]	1	0	false	true / false	false	false	Node
5	delete attributeref at '12' ;	/root[1]/@a3		boolean(/root[1]/@a3)	0	0	false	true	false	false	Node
5	delete attributeref at '12' ;	/root[1]/e1[1]/@a3		boolean(/root[1]/e1[1]/@a3)	0	0	false	true	true	false	Node
5	delete attributeref at '12' ;	/root[1]/e1[2]/@a3		boolean(/root[1]/e1[2]/@a3)	0	0	false	true	false	false	Node
6	update elementref 'e1' at '16' change xpos '2' ypos '2' ;	/root[1]/*	1 -> 2	[boolean(/root[1]/*), false()]	0	0	true / false	true / false	true / false	true / false	NodeList
7	update elementref 'e2' at '17' change minoccurs '2' ;	/root[1]/*	<e2></e2>	[boolean(/root[1]/*), false()]	0	0	true / false	true / false	true / false	true / false	NodeList
8	add elementref 'e4' minoccurs '1' id 'EID23' in '15' ;	/root[1]	<e4></e4>	boolean(/root[1])	0	0	true	true	true	true	NodeList
9	update elementref 'e3' at '21' change maxoccurs '1' ;	/root[1]/e1[1]/*	''	[boolean(/root[1]/e1[1]/*), false()]	0	0	true / false	true / false	true / false	true / false	NodeList
9	update elementref 'e3' at '21' change maxoccurs '1' ;	/root[1]/e1[2]/*	''	[boolean(/root[1]/e1[2]/*), false()]	0	0	false	true / false	false	false	NodeList
10	update complextype name ctype change name 'ctypex' ;	/root[1]/e1[1]	ctype -> ctypex	[boolean(/root[1]/e1[1]), boolean(not(/root[1]/e1[1]/@xsi:type) or /root[1]/e1[1][@xsi:type ='ctypex'])]	0	1	true / true	true / false	true / false	true / false	Node
10	update complextype name ctype change name 'ctypex' ;	/root[1]/e1[2]	ctype -> ctypex	[boolean(/root[1]/e1[2]), boolean(not(/root[1]/e1[2]/@xsi:type) or /root[1]/e1[2][@xsi:type ='ctypex'])]	0	1	false	true / false	false	false	Node
11	update element name 'e3' change name 'ex' ;	/root[1]/e1[2]/e3[4]	e3 -> ex	[boolean(/root[1]/e1[2]/e3[4]), false()]	0	1	false	true / false	false	false	Node
11	update element name 'e3' change name 'ex' ;	/root[1]/e1[2]/e3[3]	e3 -> ex	[boolean(/root[1]/e1[2]/e3[3]), false()]	0	1	false	true / false	false	false	Node
11	update element name 'e3' change name 'ex' ;	/root[1]/e1[2]/e3[2]	e3 -> ex	[boolean(/root[1]/e1[2]/e3[2]), false()]	0	1	false	true / false	false	false	Node
11	update element name 'e3' change name 'ex' ;	/root[1]/e1[2]/e3[1]	e3 -> ex	[boolean(/root[1]/e1[2]/e3[1]), false()]	0	1	false	true / false	false	false	Node
11	update element name 'e3' change name 'ex' ;	/root[1]/e1[1]/e3[4]	e3 -> ex	[boolean(/root[1]/e1[1]/e3[4]), false()]	0	1	false	true / false	false	false	Node
11	update element name 'e3' change name 'ex' ;	/root[1]/e1[1]/e3[3]	e3 -> ex	[boolean(/root[1]/e1[1]/e3[3]), false()]	0	1	false	true / false	true / false	true / false	Node
11	update element name 'e3' change name 'ex' ;	/root[1]/e1[1]/e3[2]	e3 -> ex	[boolean(/root[1]/e1[1]/e3[2]), false()]	0	1	true / false	true / false	true / false	true / false	Node
11	update element name 'e3' change name 'ex' ;	/root[1]/e1[1]/e3[1]	e3 -> ex	[boolean(/root[1]/e1[1]/e3[1]), false()]	0	1	true / false	true / false	true / false	true / false	Node

Abbildung A.57.: Boolesche Rückgabewerte der Existenz- und Matchbedingungen aus CON der Abbildung A.56 (komplett im Vergleich zu Abbildung 6.32)

Operation	Abarbeitungsreihenfolge							CON (Condition)	CON bei rename	INS (Insert)	INS bei rename	Notizen	
									Array[CON, Wert hier unter]		Array[INS, Wert hier unter]		
	Exist: boolean(X)											X: Position bis Element	
	Exist: true, dann PAR ... false, dann nix											N: Name	X -> Y: von links nach rechts
PAR:	if (!CON[1]) then INS[1] elseif (CON[2] exists AND !CON[2]) then INS[2] else stop;											T: Prefix vom TNS	
	if (CON[1]) then CON[2]											F: Fixed Wert	c: Konstante/Wert aus GEN
1 updattribute	nur name							„boolean(“ + X + „/@“ + T + N + „)“		N + „ -> “ + Nnew			
	fixed gegeben	ja						„boolean(“ + X + „[@“ + T + N + „='“ + F + „'])“	„boolean(“ + X + „/@“ + T + Nnew + „)“	GEN: T + Nnew + „ = '“ + c + „'“	N + „ -> “ + Nnew	fixed > type	
		nein						„matches(“ + X + „/@“ + T + N + „ , '“ + regex + „')“	„boolean(“ + X + „/@“ + T + Nnew + „)“	GEN: T + Nnew + „ = '“ + c + „'“	N + „ -> “ + Nnew	Type Prüfung	
2 updattributeref	use prohibited							„boolean(“ + X + „/@“ + T + N + „)“		„“			
	use optional	fixed gegeben	ja					„boolean(“ + X + „[@“ + T + N + „='“ + F + „'])“	„boolean(“ + X + „/@“ + T + Nnew + „)“	T + Nnew + „ = '“ + c + „'“	N + „ -> “ + Nnew	fixed > ref	
			nein	upd ref	ja			„matches(“ + X + „/@“ + T + N + „ , '“ + regex + „')“	„boolean(“ + X + „/@“ + T + Nnew + „)“	„“	N + „ -> “ + Nnew	Type Prüfung	
					nein							hier soll nix passieren	
	use required	fixed gegeben	ja					„boolean(“ + X + „[@“ + T + N + „='“ + F + „'])“	„boolean(“ + X + „/@“ + T + Nnew + „)“	T + Nnew + „ = '“ + c + „'“	N + „ -> “ + Nnew	fixed > ref	
			nein	upd ref	ja			„matches(“ + X + „/@“ + T + N + „ , '“ + regex + „')“	„boolean(“ + X + „/@“ + T + Nnew + „)“	GEN: T + Nnew + „ = '“ + c + „'“	N + „ -> “ + Nnew	Type Prüfung	
					nein			„matches(“ + X + „/@“ + T + N + „ , '“ + regex + „')“		GEN: T + Nnew + „ = '“ + c + „'“		Type Prüfung	
3 updattributegroupref	**alten ARef Deklarationen (ohne prohibited)**												
	gegeben in neuer AG	use prohibited in neuer Aref	gültige Wildcard in neuer AG									kein Eintrag	
			keine / ungültige Wildcard in neuer AG					„boolean(“ + X + „/@“ + T + N + „)“		„“		Löschen (tINS = "" ; tCON = "boolean")	
		use != prohibited in neuer Aref	fixed in neuer Aref	use optional				„boolean(“ + X + „[@“ + T + N + „='“ + F + „'])“		„“		tINS = "" ; tCON = "='fixed-Wert'"	
				use required				„boolean(“ + X + „[@“ + T + N + „='“ + F + „'])“		GEN: T + N + „ = '“ + c + „'“		tINS = "EID-der-ARef" ; tCON = "='fixed-Wert'"	
			fixed nicht in neuer Aref									kein Eintrag	
	nicht gegeben in neuer AG	gültige Wildcard in neuer AG										kein Eintrag	
		keine / ungültige Wildcard in neuer AG						„boolean(“ + X + „/@“ + T + N + „)“		„“		Löschen (tINS = "" ; tCON = "boolean")	
	neue ARef Deklarationen (noch nicht behandelt)												
	use required	ja						„false()“		GEN: T + N + „ = '“ + c + „'“		tINS = "EID-der-ARef" ; tCON = "false"	
		nein										kein Eintrag	
	alte AG enthält Wildcard (alle noch nicht behandelten, gültigen ADef)												
	gültige Wildcard in neuer AG											kein Eintrag	
	keine / ungültige Wildcard in neuer AG							„boolean(“ + X + „/@“ + T + N + „)“		„“		Löschen (tINS = "" ; tCON = "boolean")	
4 updattributewildcard	**Erlaubte EIDs der ADef in Lda, nicht erlaubte in Ldr**											a all, m modul, s schema, r remove	
	Erlaubt gemäß not, NS und not NS	ja	fixed gegeben	ja				„boolean(“ + X + „[@“ + T + N + „='“ + F + „'])“		„“			
				nein				„matches(“ + X + „/@“ + T + N + „ , '“ + regex + „')“		„“		Type Prüfung	
		nein						„false()“		„“		nicht erlaubte löschen	
5 updschema	**alten ARef Deklarationen (ohne prohibited) in Ldo**											ARef Lx, ADef Ldx, sf "something from", n new, o old, e element	
	in Lde	ja	in Le (Ref oder Wild)	ja	fixed	ja		„boolean(“ + X + „[@“ + T + N + „='“ + F + „'])“		„“			
						nein						einfache Übernahme	
				nein	gültig in CT	ja						einfache Übernahme	
						nein	s.u. in Ldn ?					weiter bei in Ldn	
	in Ldn (nicht in Lde)	ja	in Ln (Ref oder Wild)	ja	fixed gegeben	ja	use required	„boolean(“ + X + „[@“ + T + N + „='“ + F + „'])“		GEN: T + N + „ = '“ + c + „'“		sfLdnFlag := 1	
							use != required	„boolean(“ + X + „[@“ + T + N + „='“ + F + „'])“		„“			
						nein	use required	„boolean(“ + X + „/@“ + T + N + „)“		GEN: T + N + „ = '“ + c + „'“		sfLdnFlag := 1	
							use != required	„boolean(“ + X + „/@“ + T + N + „)“		„“			
				nein	gültig in AGn	ja						einfache Übernahme, aber sfLdnFlag := 1	
						nein		„false()“		„“		nicht erlaubte löschen	
		nein						„false()“		„“		nicht erlaubte löschen	
	zum Informationserhalt Verwendung der neuen AG (sfLdnFlag == 1), somit notwendige, fehlende (oben noch nicht behandelte) Attribute ergänzen												
	use required	ja						„boolean(“ + X + „/@“ + T + N + „)“		GEN: T + N + „ = '“ + c + „'“			
		nein										kein Eintrag	

Abbildung A.58.: Transformationsschritte 1T der Update-Operationen gemäß [Nös15d]

Operation	Abarbeitungsreihenfolge				CON (Condition)	CON bei rename	INS (Insert)	INS bei rename	Notizen
6 updelementdef	nur name				„false]“		N + „> “ + Nnew		
	upd nil	zusätzlich bei rename	nein		„boolean(not(“ + X + „/@xsi:nil) or “ + X + „/@xsi:nil = true))“ „boolean(not(“ + X + „/@xsi:nil) or “ + X + „/@xsi:nil = false))“		„xsi:nil -> “ Array(„xsi:nil -> “, GEN: ohne umschließenden Start- und Stopptag)		xsi:nil = “false“ im Dokument xsi:nil = “true“ im Dokument	Verdoppelung da Tests abhängig von Instanz, bei Rename sogar weiterer Test, weil GEN ohne Tags
			ja		„false]“		N + „> “ + Nnew			
	fixed gegeben	zusätzlich bei rename	nein		„boolean(not(“ + X + „/“ + A + „=‘“ + F + „‘)“		GEN: ohne umschließenden Start- und Stopptag (c)			bei Rename sogar weiterer Test, weil GEN ohne Tags
			ja		„false]“		N + „> “ + Nnew			
	ST neu	CT alt			„boolean(not(“ + X + „/“) and matches(“ + X + „,‘“ + regex + „‘))“	„false]“	„ “ -> “ + GEN	N + „ -> “ + Nnew	komplett ersetzen, außer dortiger Wert ok	
		ST alt	neu OT ohne rename		„false]“		N + „> “ + Nnew		hier soll nix passieren	
			neu OT mit rename		„false]“		N + „> “ + Nnew		bei Rename ein Test	
			neu OT zusätzlich bei kein OT rename	nein	„matches(“ + X + „, ‘“ + regex + „‘)“		GEN: ohne umschließenden Start- und Stopptag			
				ja	„false]“		N + „> “ + Nnew			bei Rename sogar weiterer Test, weil GEN ohne Tags
	CT neu (nicht ST neu)				„false]“		“ -> “ + GEN		immer komplett ersetzen	
7 updelementref	**Lösche alle Duplikate aus (POS, tNS, tCON) und verschiebe Einzeleinträge längensortiert ans Ende (betrifft Einträge mit POS: Umgebene Gruppe)**									
	upd min (++)	zusätzlich bei rename	nein		„false]“		GEN: T + N + „ =‘“ + c + „‘“		POS: Umgebene Gruppe	bei Rename weiterer Test (alte ändern)
			ja		„false]“		N + „> “ + Nnew		POS: Element selber	
	upd max (--)	zusätzlich bei rename	nein		„false]“		“		POS: Umgebene Gruppe	bei Rename weiterer Test (alte ändern)
			ja		„false]“		N + „> “ + Nnew		POS: Element selber	
	upd Pos	zusätzlich bei rename	nein		„false]“		posto „>“ posn		POS: Umgebene Gruppe	bei Rename weiterer Test (alte ändern)
			ja		„false]“		N + „> “ + Nnew		POS: Element selber	
	upd ref	changeT mit OT oder ST	rename	nein	„false]“		N + „> “ + Nnew		kein Eintrag	
			rename	ja	„false]“		GEN: T + N + „ =‘“ + c + „‘“			Match nicht erfolgreich, Rename obsolet
		changeT mit UT oder PT	neu ST zusätzlich bei rename	nein	„matches(“ + Y + „, ‘“ + regex + „‘)“		N + „> “ + Nnew			
				ja	„false]“		“ -> “ + GEN		immer komplett ersetzen	
		neu CT			„false]“					
8 updew	upd min (++)				„false]“		GEN: T + N + „ =‘“ + c + „‘“		GEN mit Element aus SORTDEF (kleinste Mgl.)	
	upd max (--)				„false]“		“			
	Processcontent = strict (L bilden als (alte, erlaubte EDef) - (neue , erlaubte EDef + ERef der Gruppen) - Überbleibsel behandeln: zwingend -> GEN, optional -> “ “)								Ersetzen durch ermittelte Wildcard	
	in L Element mit min > 0	ja			„false]“		N + „> “ + GEN			
		nein			„false]“		“			
9 updg	**Neuer Gruppenaufbau mit zwingenden ERefs (Uref) und zwingender EWild (Udef)**								Count Durchläufe + ggf. min/max-Vergleich	
	upd min (++) ODER mode != choice				„false]“		ERef „ -> “ GEN, ERef „ -> “ GEN, …, ERef „ -> “ GEN, ERef/EDef -> “ GEN		Liste von Komponenten in Gruppenreihenfolge zum Einfügen (letztes ggf. EWild)	
	Gruppenaufbau mit nicht verbotenen ERefs (Eref) und nicht verbotenen EWild (Edef)									
	upd max (--) ODER mode == choice				„false]“		EWild „ -> “, ERef „ -> “, …, ERef „ -> “		Umgekehrte Reihenfolge beim Löschen	
10 updst	nur name				„boolean(not(“ + X + „/@xsi:type) or “ + X + „/@xsi:type = “ + Nnew + „‘])“		N + „> “ + Nnew			
	changeT mit OT oder ST	rename	nein						kein Eintrag	
			ja		„boolean(not(“ + X + „/@xsi:type) or “ + X + „/@xsi:type = “ + Nnew + „‘])“		N + „> “ + Nnew			
	Ermittle alle Deklarationen (Ldef) in denen der Typ gegeben									
	changeT mit PT oder UT	EDef	zusätzlich bei rename	nein	„matches(“ + X + „, ‘“ + regex + „‘)“		GEN: T + N + „ =‘“ + c + „‘“			bei Rename weiterer Test (xsi:type ändern)
				ja	„boolean(not(“ + X + „/@xsi:type) or “ + X + „/@xsi:type = “ + Nnew + „‘])“		N + „> “ + Nnew			
		ADef	zusätzlich bei rename	nein	„matches(“ + X + „/@“ + T + N + „, ‘“ + regex + „‘)“		GEN: T + N + „ =‘“ + c + „‘“			bei Rename weiterer Test (xsi:type ändern)
				ja	„boolean(not(“ + X + „/@xsi:type) or “ + X + „/@xsi:type = “ + Nnew + „‘])“		N + „> “ + Nnew			
	upd mixed	mixed ist false			„boolean(not(“ + X + „/text()[normalize-space()])“		“			bei mixed weiterer Test
11 updct	**default AG war gegeben, alle nicht durch eigene AG definierten Attribute (oder durch AWild begründbar) müssen gelöscht werden (gesammelt in Ldel)**									
	nur name				„boolean(not(“ + X + „/@xsi:type) or “ + X + „/@xsi:type = “ + Nnew + „‘])“		N + „> “ + Nnew			
	zusätzlich bei rename	nein							kein Eintrag	bei Rename weiterer Test (xsi:type ändern)
		ja			„boolean(not(“ + X + „/@xsi:type) or “ + X + „/@xsi:type = “ + Nnew + „‘])“		N + „> “ + Nnew			
	upd mixed	mixed ist false			„boolean(not(“ + X + „/text()[normalize-space()])“		“			bei mixed weiterer Test
	upd defaultAG	defaultAG ist false	Ldelsize > 0		„false]“		“		für jeden Eintrag in Ldel (X + / @ + T + N) bei defaultAG weiterer Test pro zu löschendem Attribut	

Abbildung A.59.: Transformationsschritte 2T der Update-Operationen gemäß [Nös15d]

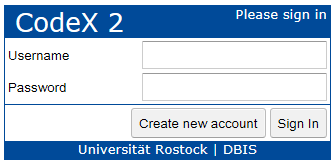


Abbildung A.60.: Login-Bildschirm des Prototypen CodeX

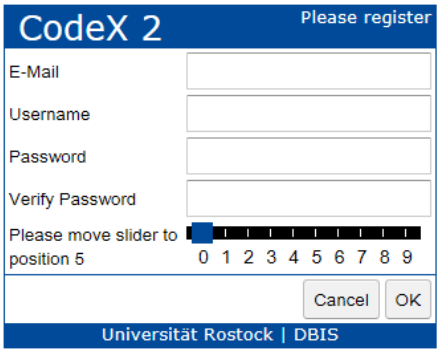


Abbildung A.61.: Registrierungsbildschirm des Prototypen CodeX

Abbildung A.62.: Informationsdialog des Prototypen CodeX

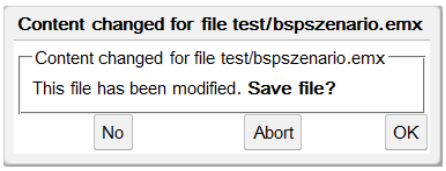


Abbildung A.63.: Bestätigungsdialog des Prototypen CodeX

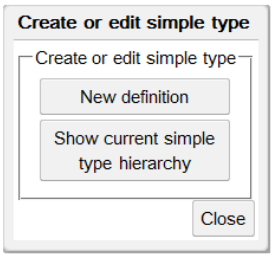


Abbildung A.64.: Übersichtsdialog des Prototypen CodeX

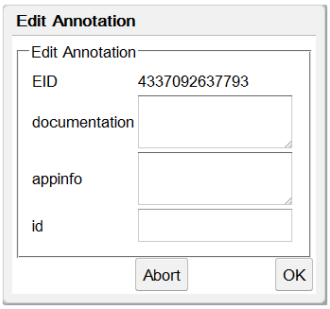


Abbildung A.65.: Konfigurationsdialog einer Annotation des Prototypen CodeX

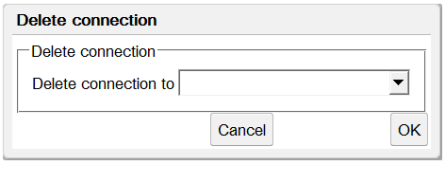


Abbildung A.66.: Dialog zum Löschen einer Kante des Prototypen CodeX

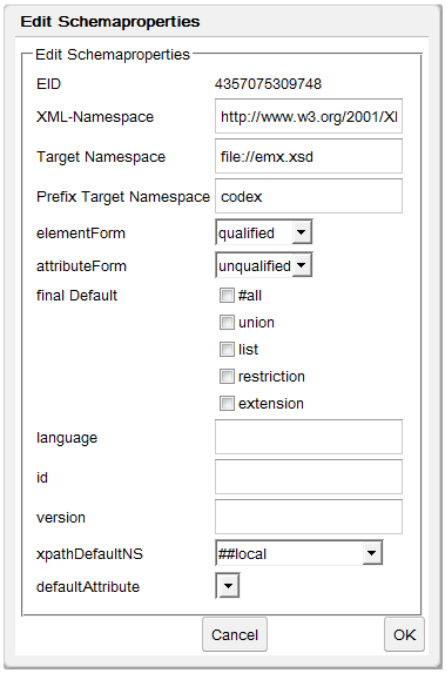


Abbildung A.67.: Konfigurationsdialog eines Schemas

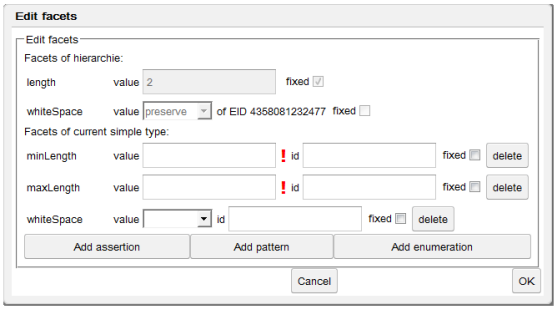


Abbildung A.68.: Angepasster Konfigurationsdialog von Facetten eines Restriktionstyps

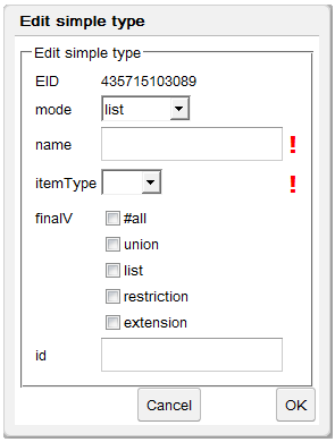


Abbildung A.69.: Konfigurationsdialog eines Listentyps

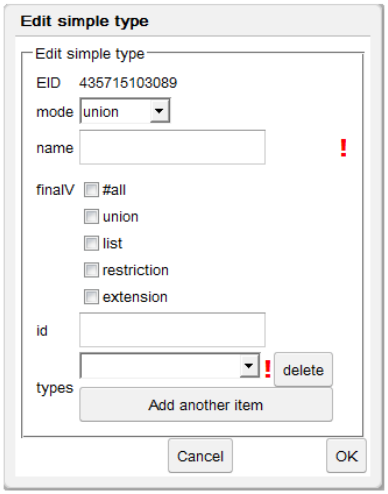


Abbildung A.70.: Konfigurationsdialog eines Vereinigungstyps

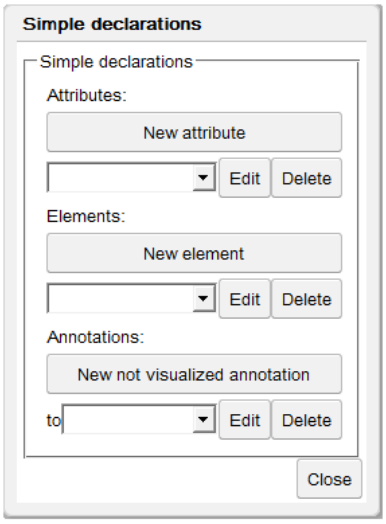


Abbildung A.71.: Übersichtsdialog von Attribut- und Elementdeklarationen mit einfachen Typen, sowie nicht visualisierter Annotationen

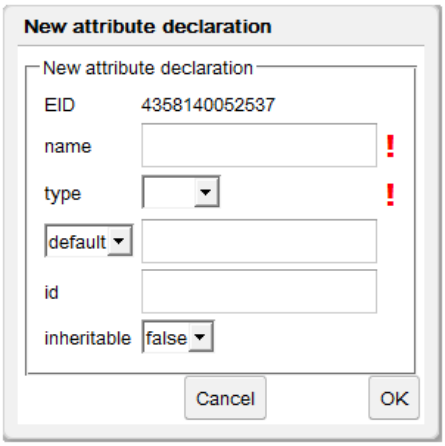


Abbildung A.72.: Konfigurationsdialog einer Attributdeklaration

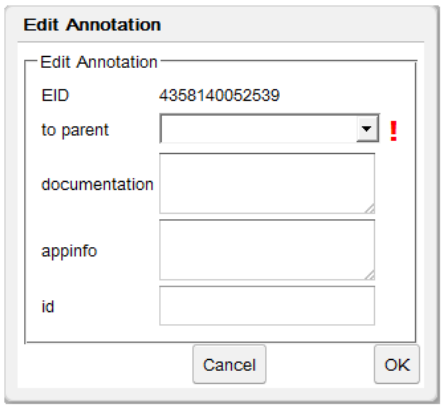


Abbildung A.73.: Konfigurationsdialog einer nicht visualisierten Annotation

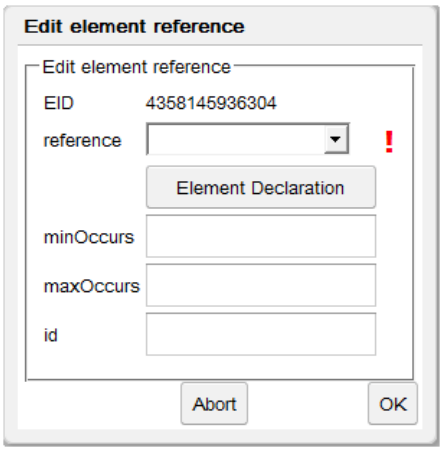


Abbildung A.74.: Konfigurationsdialog einer Elementreferenz

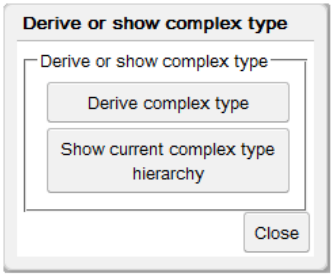


Abbildung A.75.: Übersichtsdialog von komplexen Typen

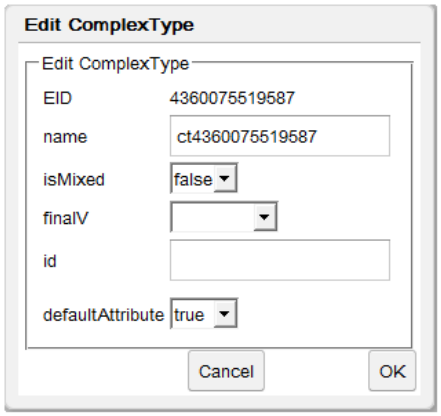


Abbildung A.76.: Konfigurationsdialog eines komplexen Typen

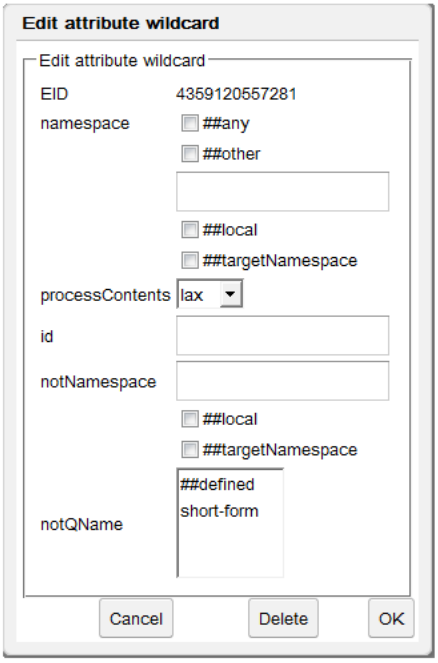


Abbildung A.77.: Konfigurationsdialog einer Attributwildcard

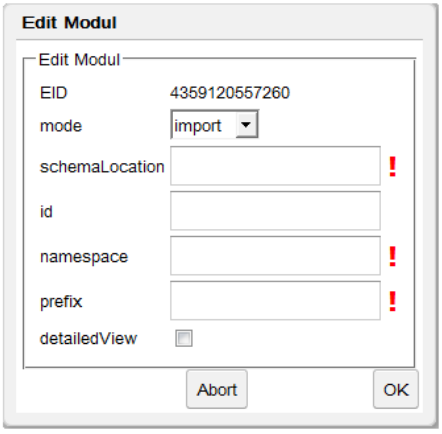


Abbildung A.78.: Konfigurationsdialog eines Moduls

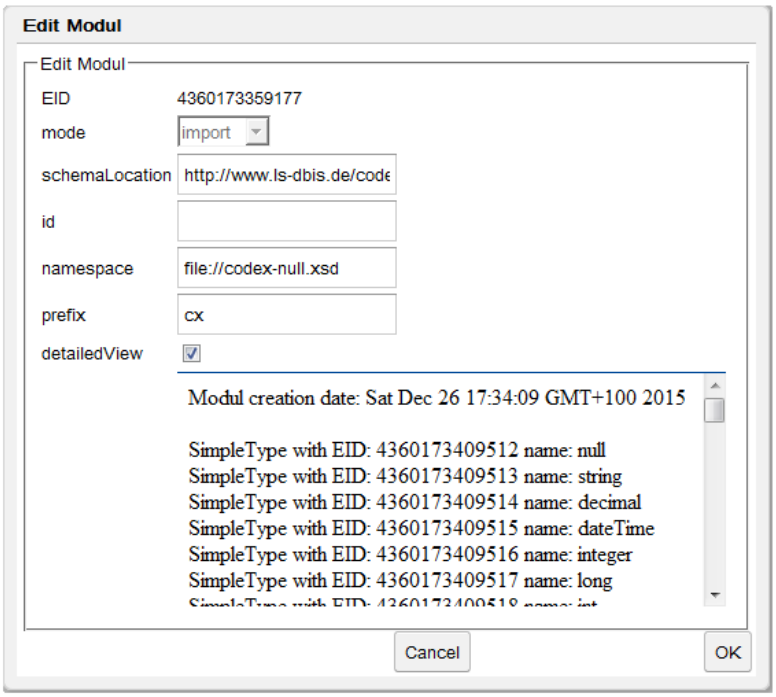


Abbildung A.79.: Konfigurationsdialog eines Moduls mit detaillierter Ansicht

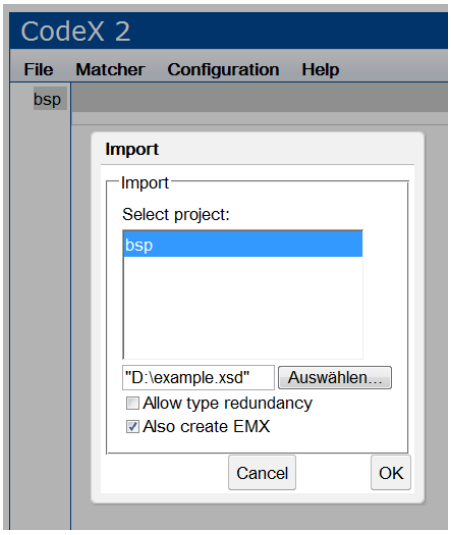


Abbildung A.80.: Dialog des Imports eines XML-Schemas

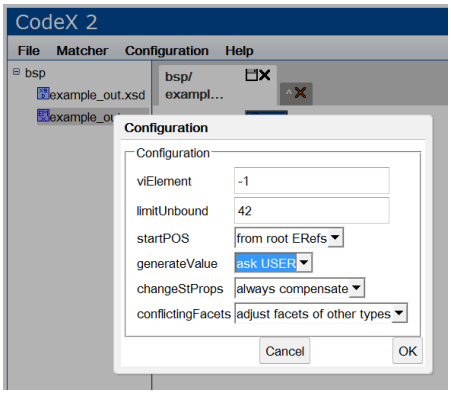


Abbildung A.81.: Konfigurationsdialog mit Wertgenerierung ask USER

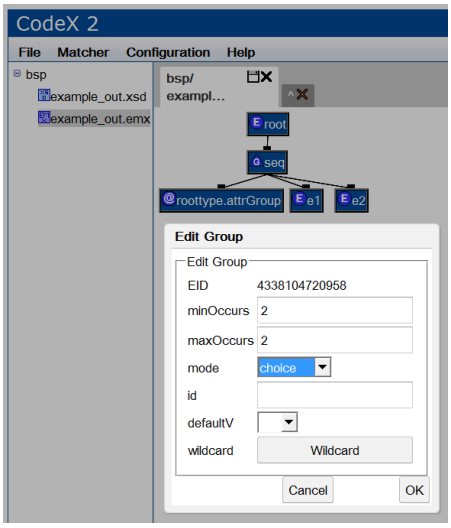


Abbildung A.82.: Änderungen des Inhaltsmodells und der Häufigkeit der Gruppe

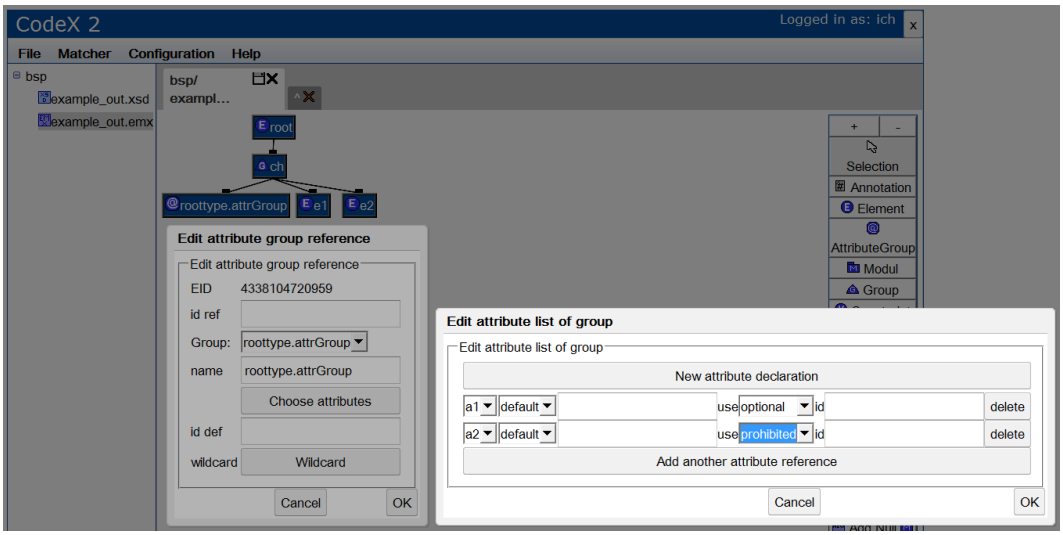


Abbildung A.83.: Änderung der Auftrittshäufigkeiten der Attributreferenzen

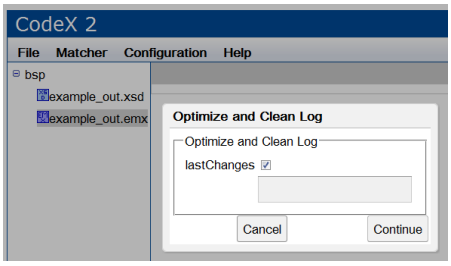


Abbildung A.84.: Prozessdialog vor der Anwendung von ROfEL

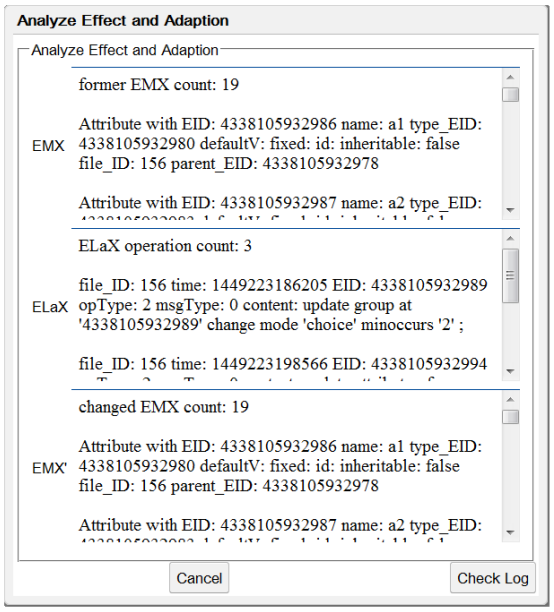


Abbildung A.85.: Prozessdialog vor der ELaX-Analyse

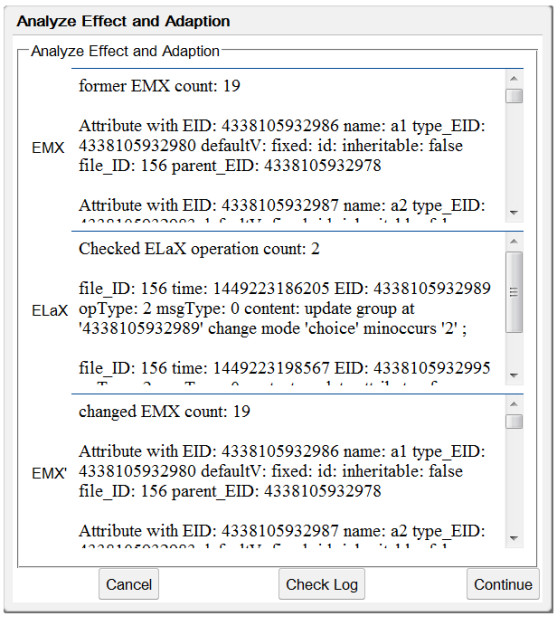


Abbildung A.86.: Prozessdialog nach der ELaX-Analyse

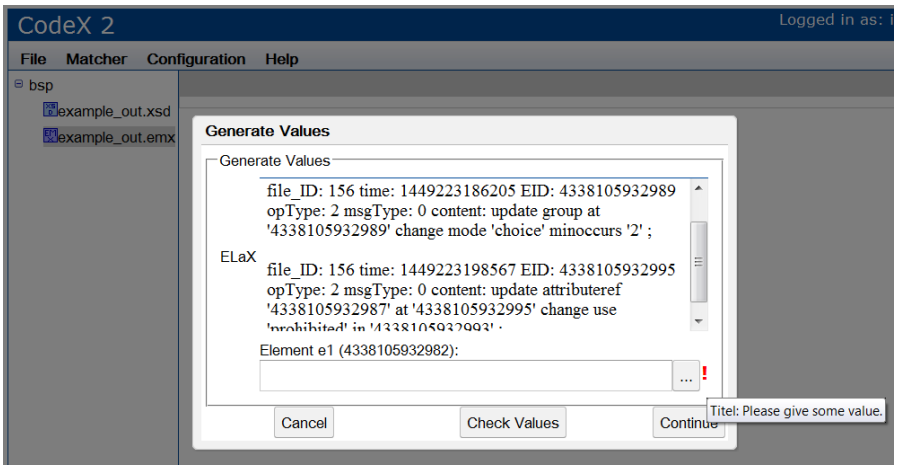


Abbildung A.87.: Prozessdialog während der Generierung von Werten

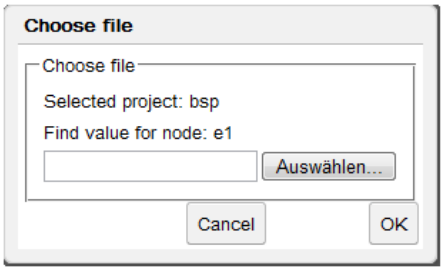


Abbildung A.88.: Dialog zur Auswahl einer Beispieldatei zur Generierung von Werten

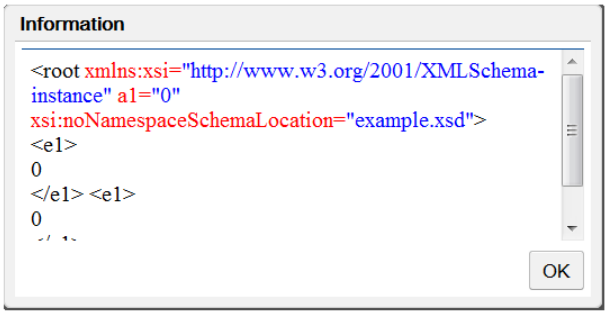


Abbildung A.89.: Informationsdialog mit Inhalt des XML-Dokuments

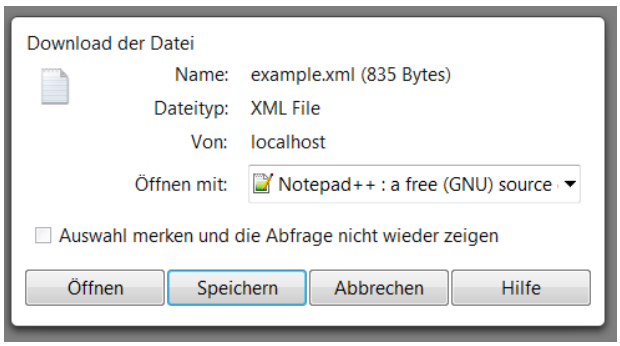


Abbildung A.90.: Popup des Informationsdialogs mit Inhalt des XML-Dokuments

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?><root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" a1="0" xsi:noNamespaceSchemaLocation="example.xsd">
    <e1>0</e1>
    <e1>0</e1>
</root><!--
user: ich project: bsp file:example_out.emx (file_ID: 156) time: 1449223172978 count: -1

Log: 2
4338105932989: update group at '4338105932989' change mode 'choice' minoccurs '2' ;
4338105932995: update attributeref '4338105932987' at '4338105932995' change use 'prohibited'  in '4338105932993' ;

WildFlag: 2
4338105932989: false
4338105932995: false

EidChain: 2
4338105932989: [[4338105932996], [4338105932989, 4338105932996]]
4338105932995: [[4338105932989, 4338105932996]]

XPathes: 2
4338105932989: [/root[1], /root[2]]
4338105932995: [/root[1], /root[2]]

GenValues: 2
4338105932989: [<e1>42</e1>, ]
4338105932995: []
 -->
```

Abbildung A.91.: Inhalt des Informationsdialogs des XML-Dokuments

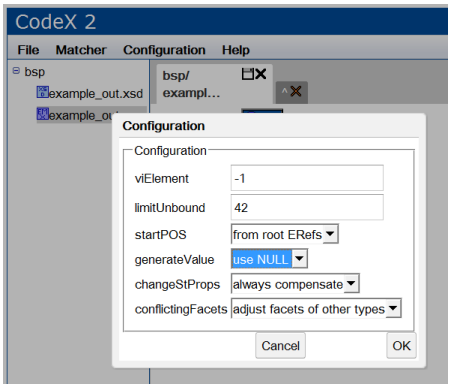


Abbildung A.92.: Konfigurationsdialog mit Wertgenerierung use NULL

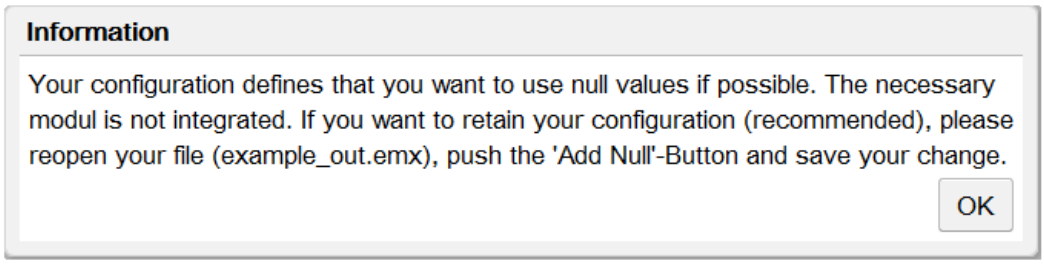


Abbildung A.93.: Informationsdialog fehlender Voraussetzungen zur Nullwertfähigkeit

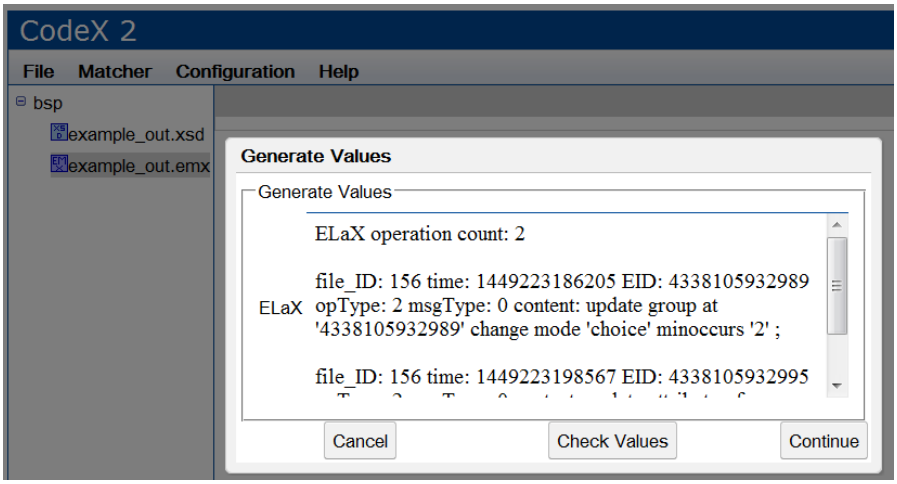


Abbildung A.94.: Prozessdialog der Generierung von Werten mit Nullwertfähigkeit

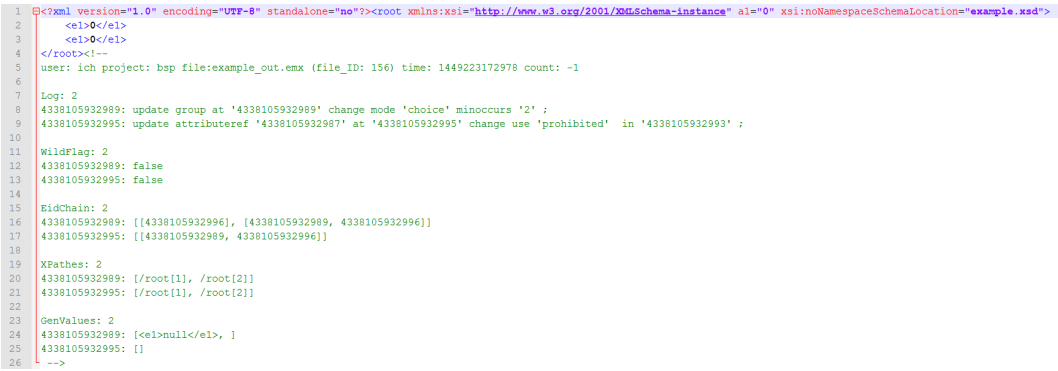


Abbildung A.95.: Inhalt des Informationsdialogs mit Nullwertfähigkeit

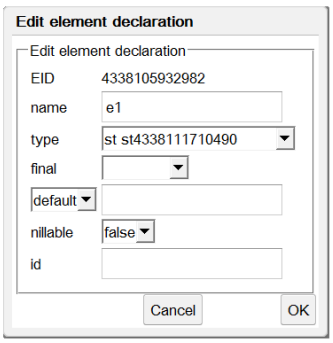


Abbildung A.96.: Konfigurationsdialog der Deklaration (nach der Nullwertfähigkeit)

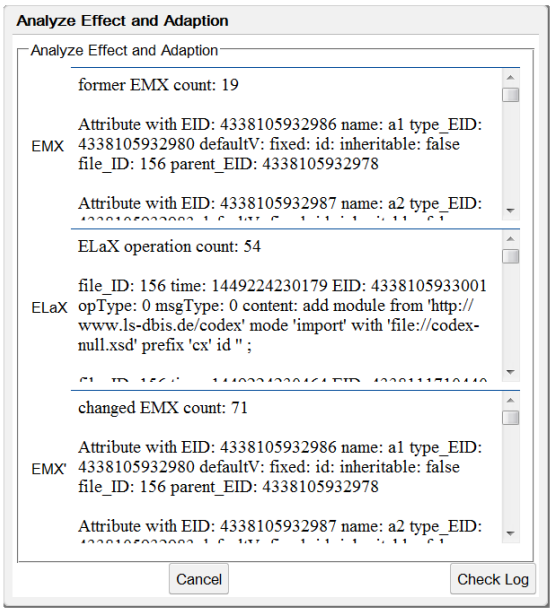


Abbildung A.97.: Prozessdialog vor der ELaX-Analyse (nach der Nullwertfähigkeit)

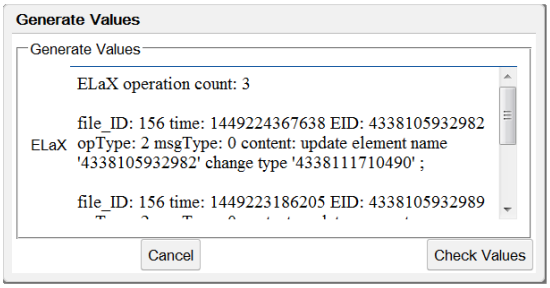
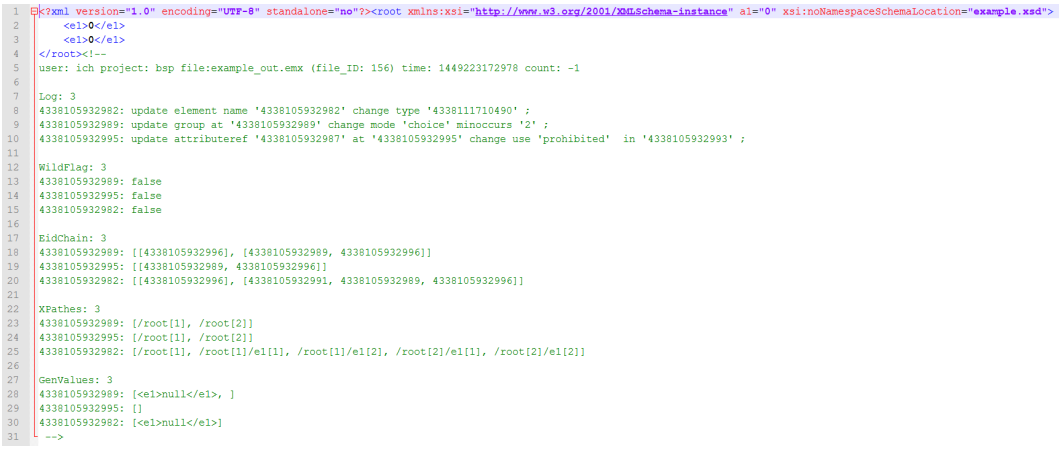
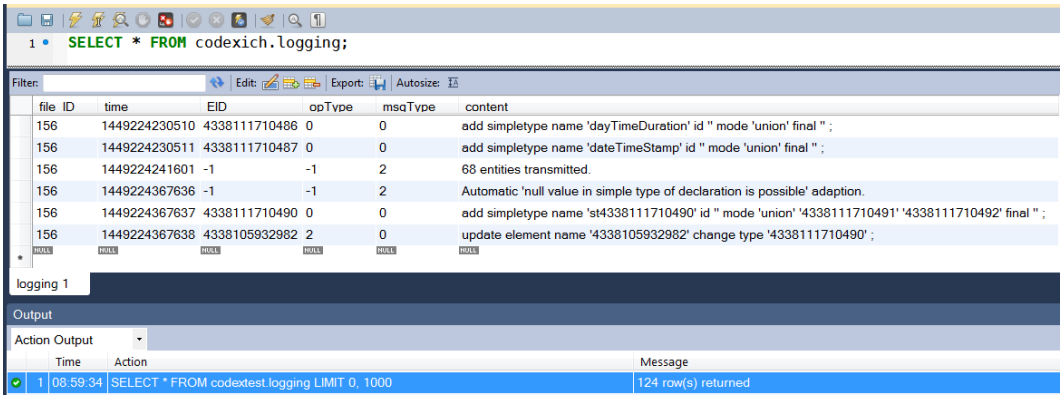


Abbildung A.98.: Prozessdialog der Wertgenerierung (nach der Nullwertfähigkeit)



```
<?xml version="1.0" encoding="UTF-8" standalone="no"?><root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" a1="0" xsi:noNamespaceSchemaLocation="example.xsd">
    <e1>0</e1>
    <e1>0</e1>
</root><!--
user: ich project: bsp file:example_out.emx (file_ID: 156) time: 1449223172978 count: -1

Log: 3
4338105932982: update element name '4338105932982' change type '4338111710490' ;
4338105932989: update group at '4338105932989' change mode 'choice' minoccurs '2' ;
4338105932995: update attributeref '4338105932987' at '4338105932995' change use 'prohibited'  in '4338105932993' ;

WildFlag: 3
4338105932989: false
4338105932995: false
4338105932982: false

EidChain: 3
4338105932989: [[4338105932996], [4338105932989, 4338105932996]]
4338105932995: [[4338105932989, 4338105932996]]
4338105932982: [[4338105932996], [4338105932991, 4338105932989, 4338105932996]]

XPathes: 3
4338105932989: [/root[1], /root[2]]
4338105932995: [/root[1], /root[2]]
4338105932982: [/root[1], /root[1]/e1[1], /root[1]/e1[2], /root[2]/e1[1], /root[2]/e1[2]]

GenValues: 3
4338105932989: [<e1>null</e1>, ]
4338105932995: []
4338105932982: [<e1>null</e1>]
-->
```

Abbildung A.99.: Inhalt des Informationsdialogs (nach der Nullwertfähigkeit)



```
SELECT * FROM codexich.logging;
```

Filter: | Edit: | Export: | Autosize:

file_ID	time	EID	opType	msgType	content
156	1449224230510	4338111710486	0	0	add simpletype name 'dayTimeDuration' id '' mode 'union' final '' ;
156	1449224230511	4338111710487	0	0	add simpletype name 'dateTimeStamp' id '' mode 'union' final '' ;
156	1449224241601	-1	-1	2	68 entities transmitted.
156	1449224367636	-1	-1	2	Automatic 'null value in simple type of declaration is possible' adaption.
156	1449224367637	4338111710490	0	0	add simpletype name 'st4338111710490' id '' mode 'union' '4338111710491' '4338111710492' final '' ;
156	1449224367638	4338105932982	2	0	update element name '4338105932982' change type '4338111710490' ;
NULL	NULL	NULL	NULL	NULL	NULL

logging 1

Output

Action Output

	Time	Action	Message
1	08:59:34	SELECT * FROM codextest.logging LIMIT 0, 1000	124 row(s) returned

Abbildung A.100.: Auszug des gespeicherten Logs (nach der Nullwertfähigkeit)

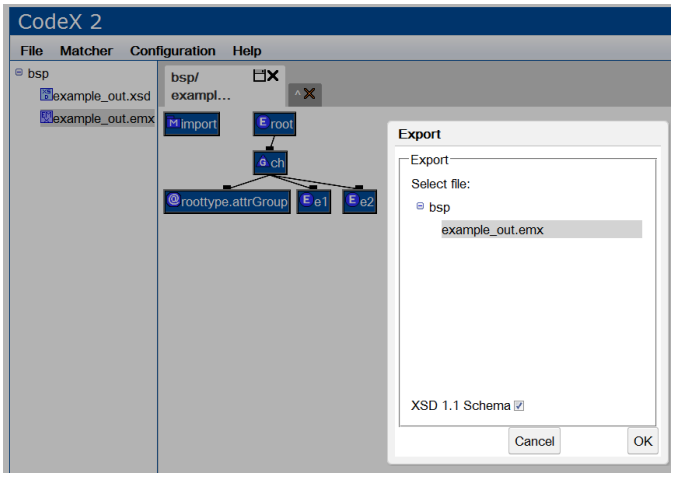


Abbildung A.101.: Dialog des Exports eines XML-Schemas

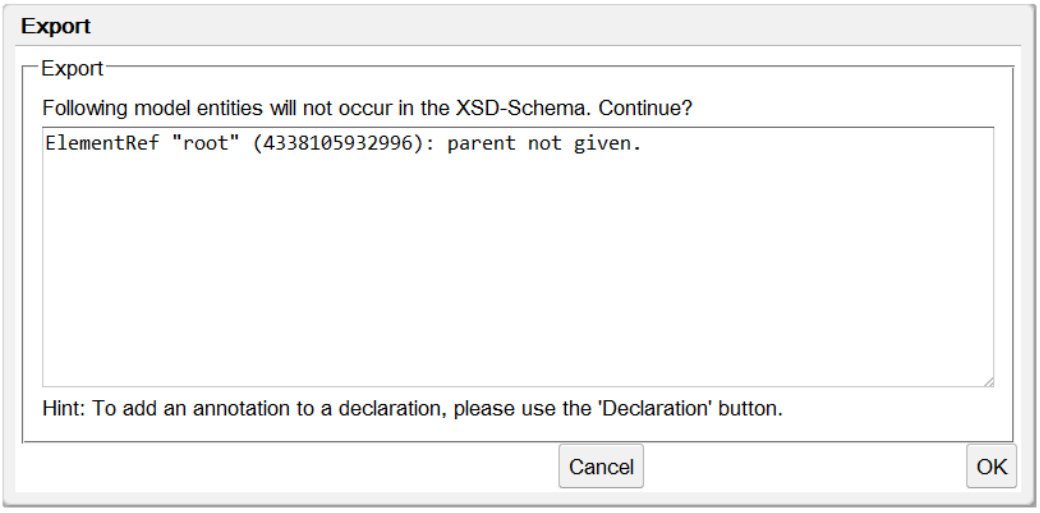


Abbildung A.102.: Informationsdialog des Exports eines XML-Schemas

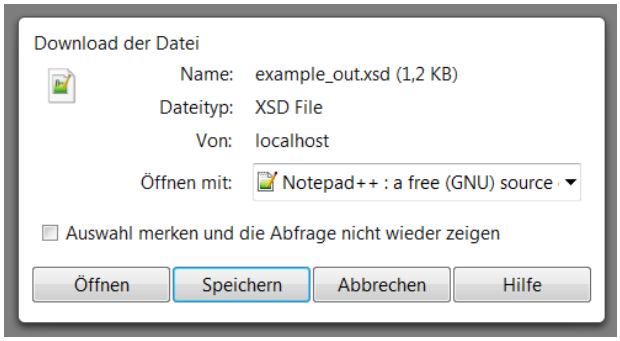


Abbildung A.103.: Popup des Exports eines XML-Dokuments

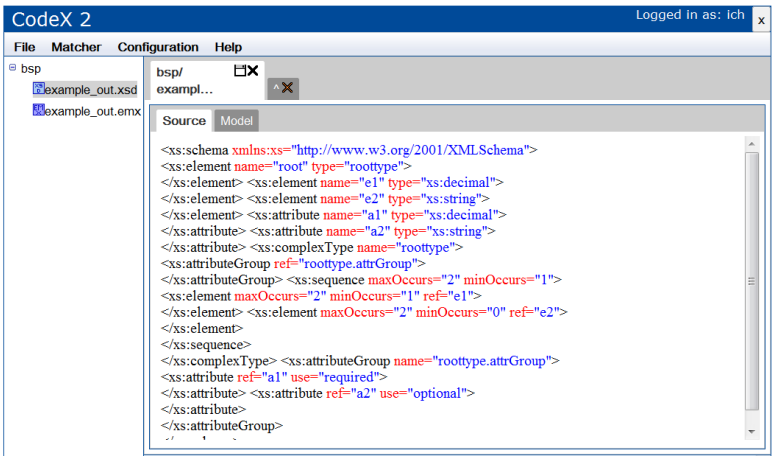


Abbildung A.104.: XML-Editor von CodeX - Quellansicht

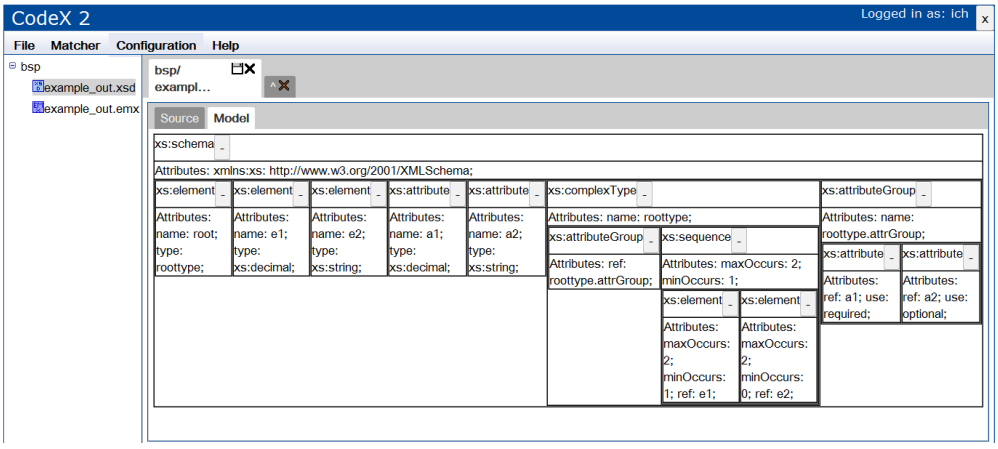


Abbildung A.105.: XML-Editor von CodeX - Modellansicht

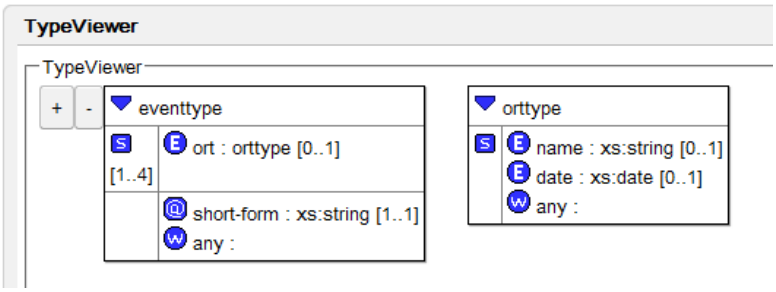


Abbildung A.106.: Überblick der Typhierarchie von komplexen Typen in CodeX

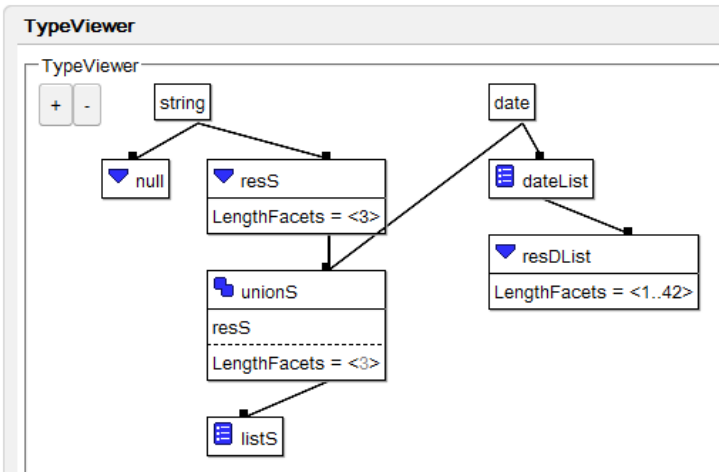


Abbildung A.107.: Überblick der Typhierarchie von einfachen Typen in CodeX

B. Überblick der Sprachspezifikation

ELaX - Evolution Language for XML-Schema

- elax ::= ((<add> | <delete> | <update>) ";")+ ;

Operationen zur Identifikation und Lokalisierung von Komponenten

- position ::= (”after” | ”before” | (”as” (”first” | ”last”) ”into”) | ”in”) <locator> ;
- refposition ::= ((”first” | ”last” | ”all” | (”position” INT)) ”in” <xpathexpr>) | emxid ;
- locator ::= <xpathexpr> | emxid ;
- xpathexpr ::= (”/” (”.” | (”node()” | (”node()[@name=”’ NCNAME ”’]”)) (”[”INT”]”)?))+ ;
- eid ::= QNAME | emxid ;

Weitere Hilfskomponenten der Operationen

- facet ::= ((((”minexclusive” | ”mininclusive” | ”maxexclusive” | ”maxinclusive” | ”totaldigits” | ”fractiondigits” | ”length” | ”minlength” | ”maxlength” | ”whitespace” | ”explicittimezone”) STRING (”fixed” (”true” | ”false”))?) | ((”enumeration”| ”pattern”) STRING)) (”id” ID)?) | <assert> ;
- assert :: = ”assert” STRING (<xpathdefaultnamespace>)? (”id” ID)? ;
- xpathdefaultnamespace ::= ”xpathdefaultnamespace” (ANYURI | (”##defaultnamespace” | ”##targetnamespace” | ”##local”)) ;
- selectorpath ::= (”.//”)? <constraintstep> (”/” <constraintstep>)* ;
- fieldpath ::= (”.//”)? (<constraintstep> ”/”)* (<constraintstep> | (”@” (QNAME | ”*” | (NCNAME ”:” ”*”)))) ;
- constraintstep ::= ”.” | (QNAME | ”*” | (NCNAME ”:” ”*”)) ;

Überblick der add-Operation

- add ::= ”add” (<addannotation> | <addattributegroup> | <addgroup> | <addst> | <addct> | <addelement> | <addmodule> | <addconstraint>) ;
- addannotation ::= ”annotation” (”appinfo” STRING)? (”documentation” STRING)? (”id” ID)? ”in” <locator> ;
- addattributegroup ::= <addattributegroupdef> | <addattribute> | <addattributeref> | <addattributegroupref> | <addattributewildcard> ;
- addattributegroupdef ::= ”attributegroup” ”name” NCNAME (”id” ID)? ”with” (<addattributeref>)+ (<addattributewildcard>)? ;
- addattribute ::= ”attribute” ”name” NCNAME ”type” <eid> ((”default” | ”fixed”) STRING)? (”id” ID)? (”inheritable” (”true” | ”false”))? ;
- addattributeref ::= ”attributeref” <eid> ((”default” | ”fixed”) STRING)? (”use” (”prohibited” | ”optional” | ”required”))? (”id” ID)? ”in” <locator> ;
- addattributegroupref ::= ”attributegroupref” <eid> (”id” ID)? ”in” <locator> ;
- addattributewildcard ::= ”anyattribute” (”not” (<eid> | ”##defined”)+)? (”namespace” ((”##any” | ”##other” | (”##local” | ANYURI | ”##targetnamespace”)+)?) | (((”not” (ANYURI | (”##targetnamespace” | ”##local”))+)?))? (”processcontent” (”lax” | ”skip” | ”strict”))? (”id” ID)? ”in” <locator> ;
- addgroup ::= ”group” ”mode” (”sequence” | (”choice” (”with” <groupdefault>)?) | ”all”) (”minoccurs” INT)? (”maxoccurs” STRING)? (”id” ID)? ”in” <locator> ;
- groupdefault ::= ”first” | ”last” | INT ;
- addst ::= ”simpletype” ”name” NCNAME (”id” ID)? ”mode” ((”built-in” | ”list”) <eid> | ”union” <eid>+ | ”restriction” ”of” <eid> ”with” <facet>+) (”final” (”#all” | (”union” | ”list” | ”restriction” | ”extension”)+))? ;
- addct ::= ”complextype” ”name” NCNAME (”mixed” (”true” | ”false”))? (”final” (”#all” | ”restriction” | ”extension”))? (”mode” (”extension_cc” | ”extension_sc” | ”restriction_cc” | (”restriction_sc” <facet>*)) ”with” ”base” <eid>)? (”id” ID)? (”defaultattributesapply” (”true” | ”false”))? (<assert>*)? ;
- addelement ::= <addelementdef> | <addelementref> | <addelementwildcard> ;

- addelementdef ::= ”element” ”name” NCNAME ”type” <eid>
  ((”default” | ”fixed”) STRING)?
  (”final” (”#all” | ”restriction” | ”extension”))?
  (”nillable” (”true” | ”false”))? (”id” ID)? ;

- addelementref ::= ”elementref” <eid> (”minoccurs” INT)? (”maxoccurs” STRING)? (”id” ID)? <position> (”xPos” INT ”yPos” INT)? ;

- addelementwildcard ::= ”any”
  (”not” (<eid> | (”##defined” | ”##definedsibling”))+)?
  (”namespace” ((”##any” | ”##other” |
    (”##local” | ANYURI | ”##targetnamespace”)+)?) |
    ((”not” (ANYURI | (”##targetnamespace” | ”##local”))+)?))?
  (”processcontent” (”lax” | ”skip” | ”strict”))?
  (”minoccurs” INT)? (”maxoccurs” STRING)? (”id” ID)? ”in” <locator> ;

- addmodule ::= ”module” ”from” ANYURI
  ”mode” ((”import” ”with” ”namespace” ANYURI ”prefix” NCNAME) |
    (”redefine” (<addst> | <addct> | <addattributegroupdef>)*) |
    ”include” |
    (”override” (<addst> | <addct> | <addattributegroupdef> |
      <addelementdef> | <addattribute>)*))
  (”id” ID)? ;

- addconstraint ::= ”constraint” ”name” NCNAME (”id” ID)?
  ”type” (”key” | ”unique” | (”keyref” ”refer” <eid>))
  ”with” <addconstraintpath> ”in” <locator> ;

- addconstraintpath ::= (”selector” <selectorpath> (”|” <selectorpath>)*
    (”id” ID)? (<xpathdefaultnamespace>)?)
  (”field” <fieldpath> (”id” ID)? (<xpathdefaultnamespace>)?)+ ;

Überblick der delete-Operation

- delete ::= ”delete” (<delannotation> | <delattributegroup> | <delgroup> | <delst> | <delct> | <delelement> | <delmodule> | <delconstraint>) ;

- delannotation ::= ”annotation” ”at” <locator> ;

- delattributegroup ::= <delattributegroupdef> | <delattribute> | <delattributeref> | <delattributegroupref> | <delattributewildcard> ;

- delattributegroupdef ::= ”attributegroup” ”name” <eid> ;

- delattribute ::= ”attribute” ”name” <eid> ;

- delattributeref ::= ”attributeref” ”at” <locator> ;
- delattributegroupref ::= ”attributegroupref” ”at” <locator> ;
- delattributewildcard ::= ”anyattribute” ”at” <locator> ;
- delgroup ::= ”group” ”at” <locator> ;
- delst ::= ”simpletype” ”name” <eid> ;
- delct ::= ”complextype” ”name” <eid> ;
- delelement ::= <delelementdef> | <delelementref> | <delelementwildcard>;
- delelementdef ::= ”element” ”name” <eid> ;
- delelementref ::= ”elementref” ”at” (<locator> | <refposition>) ;
- delelementwildcard ::= ”any” ”at” <locator> ;
- delmodule ::= ”module” ”at” <locator> ;
- delconstraint ::= ”constraint” ”name” <eid> ;

Überblick der update-Operation

- update ::= ”update” (<updannotation> | <updattributegroup> | <updgroup> | <updst> | <updct> | <updelement> | <updmodule> | <updconstraint> | <updschema>);
- updschema ::= ”schema” ”**change**” (”targetnamespace” ANYURI)? (”targetnamespaceprefix” NCNAME)? (”language” NCNAME)? (”version” STRING)? (”elementform” (”qualified” | ”unqualified”))? (”attributeform” (”qualified” | ”unqualified”))? (”finaldefault” (”#all” | (”extension” | ”list” | ”restriction” | ”union”)+))? (”id” ID)? (”defaultattribute” <eid>)? (<xpathdefaultnamespace>)? ;
- updannotation ::= ”annotation” ”at” <locator> ”**change**” (”appinfo” STRING)? (”documentation” STRING)? (”id” ID)? (”move” ”into” <locator>)? ;
- updattributegroup ::= <updattributegroupdef> | <updattribute> | <updattributeref> | <updattributegroupref> | <updattributewildcard> ;
- updattributegroupdef ::= ”attributegroup” ”name” <eid> ”**change**” (”name” NCNAME)? (”id” ID)? (”add” <addattributeref>)* (”delete” <delattributeref>)* ((”add” <addattributewildcard>) | (”delete” <delattributewildcard>))? ;

- updattribute ::= ”attribute” ”name” <eid> ”**change**” (”name” NCNAME)?
  (”type” <eid>)? ((”default” | ”fixed”) STRING)? (”id” ID)?
  (”inheritable” (”true” | ”false”))?;

- updattributeref ::= ”attributeref” <eid> ”at” <locator> ”**change**”
  (”ref” <eid>)? ((”default” | ”fixed”) STRING)?
  (”use” (”prohibited” | ”optional” | ”required”))? (”id” ID)?
  (”move” ”into” <locator>)? ;

- updattributegroupref ::= ”attributegroupref” <eid> ”at” <locator>
  ”**change**” (”ref” <eid>)? (”id” ID)? (”move” ”into” <locator>)? ;

- updattributewildcard ::= ”anyattribute” ”at” <locator> ”**change**”
  (”not” (<eid> | ”##defined”)+)?
  (”namespace” ((”##any” | ”##other” | (”##local” | ANYURI |
  ”##targetnamespace”)+)?) |
  ((”not” (ANYURI | (”##targetnamespace” | ”##local”))+)?))?
  (”processcontent” (”lax” | ”skip” | ”strict”))? (”id” ID)? ;

- updgroup ::= ”group” ”at” <locator> ”**change**”
  (”mode” (”sequence” | (”choice” (”with” <groupdefault>)?) | ”all”))?
  (”minoccurs” INT)? (”maxoccurs” STRING)? (”id” ID)? ;

- updst ::= ”simpletype” ”name” <eid> ”**change**”
  (”name” NCNAME)? (”id” ID)?
  (”mode” ((”built-in” | ”list”) <eid> |
  (”union” (”insert” <eid>+)* (”remove” <eid>+)*) |
  (”restriction” ”of” <eid> ((”insert” | ”remove” | ”modify”) <facet>
  ”at” <locator>)*)))?
  (”final” (”#all” | (”union” | ”list” | ”restriction” | ”extension”)+))? ;

- updct ::= ”complextype” ”name” <eid> ”**change**” (”name” NCNAME)?
  (”mixed” (”true” | ”false”))? (”final” (”#all” | ”restriction” | ”extension”))?
  (”mode” (”extension_cc” | ”extension_sc” | ”restriction_cc” |
  (”restriction_sc” ((”insert” | ”remove” | ”modify”) <facet>
  ”at” <locator>)*))
  ”with” ”base” <eid>)?
  (”id” ID)? (”defaultattributesapply” (”true” | ”false”))?
  (((”insert” | ”remove” | ”modify”) <assert> ”at” <locator>)*)?;

- updelement ::= <updelementdef> | <updelementref> |
  <updelementwildcard> ;

- updelementdef ::= ”element” ”name” <eid> ”**change**” (”name” NCNAME)?
  (”type” <eid>)? ((”default” | ”fixed”) STRING)?
  (”final” (”#all” | ”restriction” | ”extension”))? (”nillable” (”true” | ”false”))?
  (”id” ID)? ;

- updelementref ::= ”elementref” <eid> ”at” (<locator> | <refposition>)
  ”**change**” (”ref” <eid>)? (”minoccurs” INT)? (”maxoccurs” STRING)?
  (”id” ID)? (”move” ”to” <position>)? (”xPos” INT ”yPos” INT)? ;

- updelementwildcard ::= ”any” ”at” <locator> ”**change**”
  (”not” (<eid> | (”##defined” | ”##definedsibling”))+)?
  (”namespace” ((”##any” | ”##other” | (”##local” | ANYURI |
  ”##targetnamespace”)+)?) |
  ((”not” (ANYURI | (”##targetnamespace” | ”##local”))+)?))?
  (”processcontent” (”lax” | ”skip” | ”strict”))?
  (”minoccurs” INT)? (”maxoccurs” STRING)? (”id” ID)? ;

- updmodule ::= ”module” ”at” <locator> ”**change**” (”from” ANYURI)?
  (”mode” ((”import” ”with” ”namespace” ANYURI ”prefix” NCNAME) |
  ”redefine” | ”include” | ”override”))?
  (”id” ID)? ;

- updconstraint ::= ”constraint” ”name” <eid> ”at” <locator>
  ”**change**” (”name” NCNAME)? (”id” ID)?
  (”type” (”key” | ”unique” | (”keyref” ”refer” <eid>)))?
  ((”insert” | ”remove” | ”modify”) <updconstraintpath>)*
  (”move” ”into” <locator>)? ;

- updconstraintpath ::= (”selector” <selectorpath> (”id” ID)?
  (<xpathdefaultnamespace>)?)?
  (”field” <fieldpath> (”id” ID)? (<xpathdefaultnamespace>)?
  ”at” <locator>)* ;

C. Hinweise zum Prototypen

CodeX (Conceptual design and evolution of XML schemas), bzw. auch *CodeX 2* als GWT-Webapplikation (Google Web Toolkit) genannt, ist ein Forschungsprototyp, der über einen längeren Zeitraum an der Universität Rostock, Institut für Informatik, Lehrstuhl für Datenbank- und Informationssysteme entwickelt wurde. Dabei sind sowohl studentische Arbeiten, als auch externe Komponenten anderer Entwickler (z.B. der Slider in Abbildung A.61) integriert worden. Da allerdings keine professionelle Softwareentwicklung vollzogen wurde, sind gewisse Kinderkrankheiten nicht auszuschließen.

Verwendete Technologien

- MySQL 5.5.25a
- Apache Tomcat 7.0.29 x64
- GWT (Google Web Toolkit) 2.5.1
- Java JDK 7u21 x64
- Eclipse Java EE IDE for Web Developers (Version: Kepler)
- Opera Browser 12.17

Vorbereitende Schritte zur Nutzung des Prototypen

1. Download des WAR-Files von *CodeX 2* unter: `http://www.noesinger.net/`
2. MySQL starten (insofern dies nicht automatisch passiert).
3. Die Datenbank **codex** initial in MySQL anlegen:

   CREATE DATABASE IF NOT EXISTS codex DEFAULT CHARACTER SET = latin1 DEFAULT COLLATE = latin1_german1_ci;

4. Dem User **codexuser** die notwendigen Rechte zum Logging geben:
   - grant all privileges on codex%.* to codexuser@localhost identified by 'codexpass';
   - grant create, delete, select, insert, update on codex.* to codexuser@localhost identified by 'codexpass';
5. Die Datei **CodeX_2.war** (siehe Schritt 1) in das *webapps* Verzeichnis von Tomcat kopieren und den Tomcat starten. Tomcat erstellt daraufhin standardmäßig aus der obigen Datei ein lokales Webprojekt, welches unter der URL `http://localhost:8080/CodeX_2/` aufgerufen werden kann.
6. Die obige URL mit dem Internet Explorer (oder alternativ Opera Version 12.17) aufrufen, da neuere Browserengines (u.a. Chromium) bei der Platzierung von Modellelementen im EMX-Editor nicht kompatibel sind (diese implementieren die notwendigen Schnittstellen nicht länger korrekt).
7. Beim erstmaligen Login den *codexuser* mit einem beliebigen Passwort auswählen und den Button *Sign In* anklicken. Es werden daraufhin sowohl die zur Verwaltung des konzeptuellen Modells, als auch der Anwender notwendigen, relationalen Strukturen in der obigen MySQL-Datenbank *codex* angelegt.
8. Einen neuen Nutzer registrieren und anschließend den Prototypen verwenden.

Eidesstattliche Versicherung

Ich erkläre hiermit, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.
Die Arbeit wurde weder im Inland noch im Ausland in gleicher oder ähnlicher Form einer Prüfungsbehörde zur Erlangung eines akademischen Grades vorgelegt.

Rostock, 1. Februar 2016 Thomas Nösinger

Thesen

These 1 Die Modellierung und Evolution von XML-Schema wird durch ein konzeptuelles Modell signifikant erleichtert.

These 2 Sind sowohl die Operation des Nutzers auf dem EMX (Entity Model for XML-Schema) als auch die Korrespondenzen zwischen dem konzeptuellen Modell und dem XML-Schema bekannt, so kann die Operation zum Anpassen des XML-Schemas hergeleitet werden.

These 3 Sind sowohl die Operation des Nutzers auf dem EMX (Entity Model for XML-Schema) als auch die Korrespondenzen zwischen dem konzeptuellen Modell, dem XML-Schema und den XML-Instanzen bekannt, so kann die Operation zum Anpassen der XML-Instanzen hergeleitet werden.

These 4 Eine Transformationssprache wie ELaX (Evolution Language for XML-Schema) ist absolut notwendig, um Änderungen auf einem XML-Schema durchführen und formal ausdrücken zu können.

These 5 Durch die Aufzeichnung (Logging) von ELaX-Operationen (Evolution Language for XML-Schema) wird eine feingranulare Historie aufgebaut, aus der evolutionsrelevante Informationen automatisch ermittelt werden können.

These 6 Mit dem regelbasierten Algorithmus ROfEL (Rule-based Optimizer for ELaX) werden unnötige, redundante und ungültige ELaX-Operationen (Evolution Language for XML-Schema) in einem Log erkannt und beseitigt.

These 7 Die Minimierung der Anzahl von notwendigen Änderungen an einem XML-Schema ist eine Voraussetzung für die effiziente XML-Schemaevolution.

These 8 Der regelbasierte Algorithmus ROfEL (Rule-based Optimizer for ELaX) ist korrekt, da er ein konfluentes, terminierendes Regelsystem ist.

These 9 Die Konzentration auf einen Modellierungsstil von XML-Schema (d.h. Garden of Eden) vereinfacht die Analyse von Anpassungen auf Schemaebene und trägt somit zur effizienten XML-Schemaevolution bei.

These 10 Die Liste von absoluten Lokalisierungspfaden beinhaltet alle Positionen, an denen eine durch die Anwendung einer ELaX-Operation (Evolution Language for XML-Schema) auf Schema- und Modellebene betroffene Komponente in der Instanzebene vorhanden sein kann.

These 11 Referentielle Integritäten auf Instanzebene sind auf Schema- und Modellebene nicht analysierbar und können daher in der XML-Schemaevolution nicht automatisiert behandelt werden.

These 12 Die Generierung von sowohl einfachen und komplexen Inhalten, als auch von Wildcardinhalten bei vorhandenen, gültigen Deklarationen, ist unter Verwendung der vorgestellten Mechanismen auf Schema- und Modellebene in der XML-Schemaevolution automatisierbar.

These 13 Die durch die Optionalität von Strukturen auf Schemaebene bedingte Heterogenität der Instanzebene ist ein wesentlicher Grund für die hohe Komplexität und feingranulare Betrachtungsweise der XML-Schemaevolution.

These 14 Die Gültigkeit der transformierten Instanzen bezüglich eines veränderten XML-Schemas kann durch die Anwendung des vorgestellten Ansatzes der XML-Schemaevolution weitestgehend automatisiert gewährleistet werden.

These 15 Der Prototyp CodeX (Conceptual design and evolution of XML schemas) ist ein Tool mit sinnvollen Abstraktionen, durch das Nicht-Experten bei der hochkomplexen, fehleranfälligen Evolution unterstützt werden.