Management von Typhierarchien in der XML-Schemaevolution

Masterarbeit

Universität Rostock
Fakultät für Informatik und Elektrotechnik
Institut für Informatik
Lehrstuhl Datenbank- und Informationssysteme

eingereicht von:	Chris Kaping
Matrikelnummer:	209204822
geboren am:	03.11.1988 in Wismar
Erstgutachter:	Dr.-Ing. habil. Meike Klettke
Zweitgutachter:	Prof. Dr.-Ing. habil. Peter Forbrig
Betreuer:	Dipl.-Inf. Thomas Nösinger
Abgabedatum:	15.10.2014

Selbstständigkeitserklärung

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

Rostock, den 15. Oktober 2014

Zusammenfassung

XML (eXtensible Markup Language) ist eine Auszeichnungssprache, die unter anderem als Format für den plattformunabhängigen Austausch von Daten eingesetzt wird. Mit der Schemasprache XML-Schema (XML Schema Definition Language - XSD) existiert eine Möglichkeit, Strukturen für XML-Dokumente zu spezifizieren. Dokumente werden als gültig bezeichnet, wenn sie solchen Strukturen entsprechen. Diese Strukturen setzen sich hauptsächlich aus Attribut- und Elementdeklarationen zusammen, die durch ihre Verwendung in Typdefinitionen angeordnet werden. Zwischen den Komponenten entstehen dadurch Abhängigkeiten. Eine Form dieser Abhängigkeiten sind Typhierarchien, die sich aufgrund des Mechanismus zur Typableitung in XML-Schema bilden können. In einer Ableitung lassen sich die Bestandteile eines Typen auf verschiedene Weisen verändern, z. B. durch eine Einschränkung oder eine Erweiterung.

Die Anpassung eines XML-Schemas an neue Anforderungen wird als XML-Schemaevolution bezeichnet. Je nachdem, welche Komponenten in einem Schema manipuliert werden, kann dies die Veränderung der bislang gültigen Dokumente erfordern, damit diese der neuen Struktur entsprechen (Co-Evolution). Im Speziellen können Änderungen der Typhierarchien weitreichende Auswirkungen nach sich ziehen.

Um die Tragweite der Auswirkungen zu erkennen und einen besseren Überblick über ein XML-Schema zu erhalten, ist es vorteilhaft, die Typhierarchien zu visualisieren. Für diese Aufgabe existieren mehrere Lösungen. In der Arbeit wird zunächst untersucht, welche Vorgehensweisen verschiedene Werkzeuge dafür nutzen, die in einer Gegenüberstellung bewertet werden. Die Erkenntnisse werden genutzt, um ein eigenes Konzept für die Visualisierung zu entwerfen.

Anschließend erfolgt eine umfassende Analyse darüber, unter welchen Bedingungen Änderungen an Typhierarchien möglich sind. Die Bedingungen ergeben sich dabei aus den schemaspezifischen Komponenten. Für Situationen, in denen Änderungen die Bedingungen verletzen, werden zudem Lösungsvorschläge in der Form von weiteren Schemaanpassungen genannt. Das Ziel dieser Vorschläge ist es, die Konsistenz des Schemas zu sichern und gleichzeitig die Informationen der XML-Dokumente zu erhalten. Wird beispielsweise ein Typ aus einer Hierarchie gelöscht, muss ein neuer Typ für abhängige Definitionen und Deklarationen als Kompensation gefunden werden. Die Arbeit zeigt, wie sich die Informationen von Typdefinitionen ausnutzen lassen, um einen geeigneten Typen zu ermitteln, der wenig Informationsverlust für die Dokumente verspricht.

Abschließend erfolgte eine Implementierung in den Prototypen CodeX (Conceptual Design and Evolution for XML-Schema). Gegenstand dieser ist sowohl das entwickelte Konzept für die Visualisierung von Typhierarchien als auch die Kompensation nach der Entfernung eines Typen. Als Ergebnis erweitert der erste Aspekt CodeX um weitere Sichten, welche die Typhierarchien in den Vordergrund stellen und somit den Prozess der XML-Schemamodellierung visuell unterstützen. Der zweite Aspekt erzeugt ELaX-Ausdrücke (Evolution Language for XML-Schema) für das Löschen und Kompensieren von Typen, was die XML-Schemaevolution mit CodeX bereichert.

Inhaltsverzeichnis

1. Einleitung

Für die Entstehung von Wissen sind Informationen bzw. Daten unabdingbar. Das Aufkommen des Internets eröffnete zweifelsohne eine neue Möglichkeit, abgespeicherte Informationen aus verschiedensten Domänen für den Zugriff und den Austausch leichter zugänglich zu machen. Durch den rasanten Anstieg an verfügbaren Informationen stellte sich allerdings schnell die Notwendigkeit heraus, eine Repräsentation zu finden, die der inhärenten Struktur der Informationen gerecht wird und dabei trotzdem eine einheitliche, effiziente und einfach zu verstehende Form besitzt. Eine solche Repräsentation musste also möglichst anwendungsunabhängig, von Maschinen leicht zu verarbeiten und vom Menschen lesbar, also „sprechend“ sein.

Angetrieben von diesen Zielen[1], entwickelte das World Wide Web Consortium (W3C) mit der eXtensible Markup Language (XML) einen Vorschlag für eine Auszeichnungssprache zur Speicherung und für den Austausch von Daten, vor allem im Web ([BPSM+08]). Der Inhalt dieser Dokuemnte besteht dabei zum größten Teil aus den Informationen selbst, die mit Hilfe von gewissen Markup-Zeichen auf einfache Weise in baumartige Hierarchien angeordnet oder mit weiteren (Meta-)Informationen angereichert werden können.

Die Komplexität solcher XML-Dokumente entspricht dabei der Komplexität der abzubildenden Informationen. Je komplizierter die dargestellten Sachverhalte sind, desto schwieriger kann es u. U. werden, diese korrekt in die XML-Repräsentation zu übertragen. Dies kann sich einerseits in Fehlern in der XML-Syntax äußern (Verletzung der Wohlgeformtheit) und andererseits in Fehlern des zu übertragenden semantischen Modells der Informationen. Während Fehler der ersten Art leicht durch XML-Verarbeitungsprogramme ermittelt werden können, ist dies bei semantischen Fehlern schwieriger, da diesen Programmen (standardmäßig) das Datenmodell nicht zur Verfügung steht und der Nutzer selber darüber urteilen muss, ob dessen Aspekte richtig dargestellt werden. Erreicht ein XML-Dokument eine gewisse Größe, ist es daher für den Nutzer schwierig, den Überblick über die Informationen und deren Korrektheit im Hinblick auf das angestrebte Datenmodell zu behalten.

Mit Schemabeschreibungssprachen wie XML-Schema (ebenfalls vom W3C entwickelt, siehe [FW04, GSMT12, PGM+12]) ist es möglich, eigene Datenmodelle für XML-Dokumente zu definieren. Diese Definitionen beschreiben, wie Markup und Inhalt von XML-Dokumenten, d. h. im Wesentlichen Elemente und Elementinhalt sowie Attribute, angeordnet sind. Entspricht ein XML-Dokument einem solchen Schema (die Datenmodelle stimmen also überein), wird es diesem gegenüber als gültig bezeichnet. Die Menge von XML-Dokumenten, die gültig gegenüber einem Schema sind, ist damit genau die Klasse an XML-Dokumenten, die dieses Schema beschreibt.

Obwohl dies ein wirkungsvoller Mechanismus ist, um die semantische Korrektheit von XML-Dokumenten bezüglich eines gegebenen Datenmodells zu gewährleisten, finden Schemasprachen für XML in der Praxis oftmals keinen Einsatz (vgl. [BMV05, MBV03]). Bex et al. haben in [BNVdB04, MNSB06] zudem gezeigt, dass im Speziellen eingesetzte XML-Schemata zudem noch einige Fehler beinhalten. Die Gründe hierfür sind nicht eindeutig auszumachen. Die Komplexität von XML-Schema spielt dabei allerdings mit hoher Wahrscheinlichkeit eine große Rolle [GIMN10]. Durch die Vielzahl von Konzepten und Möglichkeiten von XML-Schema ist es einem unerfahrenem Nutzer schwierig, diese effizient anzuwenden. Darüber hinaus ist die XML-Syntax von XML-Schema zwar (prinzipiell) vorteilhaft für die maschinelle Verarbeitung und u. U. vertrauter als etwa die DTD-Syntax, durch deren längliche Natur aber für den Menschen nicht einfach zu lesen und erschwert somit das Verständnis zusätzlich. Dies sind Faktoren, die vom eigentlichen Prozess der Modellierung der Anwendung bzw. Domäne ablenken.

Die konzeptionelle Modellierung von Informationen ist in vielen Bereichen der Informatik eine bewährte Methode von Implementierungsdetails zu abstrahieren. Damit ist es Nutzern möglich,

[1] siehe http://www.w3.org/TR/xml/#sec-origin-goals.

sich besser auf die eigentliche Semantik und die logischen Bedingungen einer Domäne zu konzentrieren. Es existieren eine Reihe von XML-Editoren, mit denen auch XML-Schemata grafisch modelliert werden können. Die Repräsentation der XML-Schemata bleibt dabei in den meisten Fällen relativ nahe an der baumartigen Syntax der Schemadokumente. Das logische Datenmodell eines Schemas wird in dieser Form allerdings nicht optimal dargestellt (siehe [KK03, RBG02]). Die Typhierarchie eines XML-Schemas ist strukturgebend für (gültige) XML-Dokumente. Eine geeignete konzeptionelle Repräsentation als Hilfestellung zum Verstehen sowie eine einfache Modifizierbarkeit der Typhierarchie wären daher vor allem wünschenswert. Letzteres ist zudem für das Gebiet der XML-Schemaevolution von Interesse. Ziel dieses Aufgabengebiets ist es, XML-Schemata sowie gültige XML-Dokumente an Veränderungen anzupassen. Für eine genauere Erläuterung sei an dieser Stelle auf Abschnitt 2.4 verwiesen.

Im Rahmen dieser Arbeit sollen verschiedene Ansätze für die konzeptionelle bzw. grafische Modellierung von XML-Schemata vorgestellt, verglichen und bewertet werden, wobei das Hauptaugenmerk auf der Darstellung und den Editiermöglichkeiten der Typen bzw. Typhierarchien liegt. Auf Grundlage der Resultate wird ein eigenes Konzept für das Handling von Typhierarchien in XML-Schemata entwickelt. Im Anschluss erfolgt die Implementierung dieses Konzepts in den Forschungsprototypen CodeX (Conceptual Design and Evolution for XML-Schema) für die XML-Schemaevolution der Universität Rostock. Neben der Visualisierung der Typhierarchie eines XML-Schematas werden außerdem Änderungsoperationen zu deren Modifikation realisiert. Somit soll es möglich sein, auf einfache Weise die Konsequenzen von verschiedenen Änderungen an den Typen aufzuzeigen und je nach Nutzerangaben das restliche Schema zur Kompensation entsprechend anzupassen.

Der Aufbau der nachfolgenden Arbeit gestaltet sich folgendermaßen:

- Kapitel 2 erläutert die Grundlagen von XML und XML-Schema und in diesem Zusammenhang den Begriff der Typhierarchie.
- Kapitel 3 stellt bestehende Ansätze für die konzeptionelle Modllierung bzw. Editierung von XML-Schema vor. Das Ziel der Arbeit wird dabei noch einmal klar definiert.
- Kapitel 4 erörtert das entwickelte Konzept für das Management von Typhierarchien.
- Kapitel 5 liefert eine detaillierte Analyse der erfolgten Implementation des Konzepts in den Prototypen CodeX.
- Kapitel 6 wertet die Fähigkeiten der Implementation anhand einer Kollektion von selbstdefinierten Beispielen aus.
- Kapitel 7 fasst die Arbeit abschließend zusammen.

2. Grundlagen

Nachdem die Konzepte von XML bzw. XML-Schema in der Einleitung angerissen wurden, sollen jene als Grundlagen des Themas der Arbeit in diesem Kapitel zunächst noch einmal genauer erläutert werden. Besondere Aufmerksamkeit findet dabei das Prinzip des Typs bzw. des Typsystems. Weiterhin wird ein kurzer Einblick in die XML-Schemaevolution gewährt und welche Bedeutung Typen für diese besitzen.

2.1. XML

Wie in Kapitel 1 erwähnt, ist die eXtensible Markup Language (XML) eine Auszeichnungssprache mit dem primären Ziel, Informationen in einer simplen, aber strukturierten Weise zu speichern. Damit lässt sich eine für Menschen akzeptable Lesbarkeit sowie ein einfacher Datenaustausch über das Internet oder zwischen Computersystem erreichen. Die Wurzeln von XML liegen in der Standard Generalized Markup Language (SGML), was bedeutet, dass die Informationsdarstellung in XML ähnlichen Prinzipien folgt wie der in SGML. Durch das Hinzufügen von Auszeichnungssymbolen (Markup) zu den Daten wird die Strukturierung der Informationen erzielt, wobei u. a. zwei wichtige Arten von Markup-Konzepten unterschieden werden: Elemente und Attribute.

Jedes Element besitzt einen sog. Start- bzw. End-Tag. Der Start-Tag besteht aus dem Datum in Form einer Zeichenkette (Elementname), die von einer öffnenden und schließenden spitzen Klammer umgeben ist. Der End-Tag folgt dem gleichen Prinzip, wobei dem Elementnamen zusätzlich ein Schrägstrich zur Markierung des Endes des Elements vorangestellt wird. Innerhalb der Tags eines Elements sind beliebige Zeichenketten (ohne Markup) oder weitere Elemente als sog. Elementinhalt unter der Prämisse platzierbar, dass Start- und End-Tag der verschiedenen Elemente nicht verzahnen bzw. überlappen. Alternativ sind Start- und End-Tag verschmelzbar, wenn ein Element keinen Inhalt besitzt. Mit dieser Methode lassen sich sowohl unstrukturierte als auch strukturierte, komplexe Informationen abbilden.

Das zweite Markup-Konzept, das Attribut, findet innerhalb des öffnenden Tags eines Elements Anwendung. Ein Attribut ist ein Name-Wert-Paar, das durch ein Gleichheitszeichen verbunden ist. Auf dessen linken Seite steht dabei der Attributname, während auf der rechten Seite der Attributwert umgeben von zwei Anführungszeichen steht. Es ist prinzipiell möglich, beliebig viele Attribute einem Element zuzuordnen. Auf diese Weise können Elemente mit weiterführenden Metainformationen versehen werden. Beispiel 2.1 zeigt ein einfaches XML-Dokument, das eine Person „John Doe“ mit Geburtstag 01.02.1983 und ID „123“ (Attribut) beschreibt.

```
<?xml version="1.0" encoding="utf-8"?>
<person id="123">
 <vorname>John</vorname>
 <nachname>Doe</nachname>
 <geburtsdatum>1983-02-01</geburtsdatum>
</person>
```

Beispiel 2.1: Ein XML-Dokument

2.2. XML-Schema

XML Schema Definition Language (XSD) ist ein Vertreter von Schemasprachen für XML. Das Ziel dieser Sprachen ist es, mittels der Spezifizierung eines Datenmodells XML-Dokumente zu strukturieren. Die dadurch resultierende Möglichkeit der Gültigkeitsprüfung von XML-Dokumenten hilft, die Fehlerrate bei der Abbildung von originaler Information zu XML-Dokument zu verringern. Je nach Schemasprache stehen unterschiedliche Mittel für eine solche Strukturdefinition zur Verfügung, was diese Sprachen durch eine Reihe von Vor- und Nachteilen charakterisiert.

Einer der Vorteile von XML-Schema ist die Verwendung der XML-Syntax zur äußeren Repräsentation der verschiedenen Komponenten. An dieser Stelle ist es wichtig zu erwähnen, dass die Definition von XML-Schema unabhängig von dessen Darstellung in XML ist. Die Definition beschreibt selbst ein eigenes, abstraktes Datenmodell konzeptioneller Natur. Die Überführung der einzelnen konzeptionellen Komponentendefinitionen in die XML-Darstellung wird unter der Verwendung von Informationseinheiten vollzogen, die wiederum der Spezifikation von XML Information Set (eine weitere Empfehlung des W3C) folgen.

Konkrete XML-Schemadateien sind also leidglich serialisierte Instanzen des abstrakten XML-Schema-Datenmodells bzw. des analog dazu definierten XML-Infosets. Der Einfachheit halber orientieren sich die nachfolgenden Erläuterungen der verschiedenen Komponenten an deren XML-Repräsentation. Der Standard teilt die Komponenten in drei Gruppen (primäre, sekundäre und Hilfskomponenten) ein. Ausgangspunkt bzw. Wurzel eines jeden XML-Schemas ist die <schema>-Komponente, die selbst nicht zum abstrakten Datenmodell gehört, aber sämtliche Komponenten dieser Gruppen direkt oder indirekt aufnimmt. Schemakomponenten, die direkt unter <schema>, also als Kind, auftauchen, werden als Top-Level-Komponenten bezeichnet und befinden sich auf der höchsten Schemaebene. Die genaue Anordnung einiger Komponenten wird nachfolgend erläutert.

2.2.1. Namensräume

Namen und Namensräume sind wichtige Konzepte, die die restlichen Komponenten betreffen. In XML-Schema existieren diverse Bestandteile, die einen Namen in der From eines Werts für ein Namensattribut tragen müssen. Anhand des Namens einer Komponente lassen sich jene später referenzieren. Um die Referenzierung eindeutig zu gestalten, gilt die Prämisse, dass die Namen innerhalb verschiedener Komponentenklassen nicht mehrfach vergeben werden dürfen, also ebenfalls eindeutig sind.

Innerhalb eines einzigen Schemas ist dies ohne große Schwierigkeiten umsetzbar. XML-Schema bietet über Modulkomponenten[1] zusätzlich Mechanismen an, ein einziges XML-Schema bzw. das beschriebene Datenmodell auf verschiedene XML-Schemadokumente zu verteilen oder bereits existierende Datenmodelle zu integrieren. Die Einhaltung von eindeutigen Namen über mehrere Dokumente hinweg gestaltet sich damit wesentlich problematischer.

Aus diesem Grund wurden parallel zu den Namen Namensräume eingeführt. Jedem Schema kann ein Namensraum zugewiesen werden, sodass die Namen benannter Komponenten (siehe die nächsten Abschnitte) eines Schemas mit diesem assoziiert werden. Dies geschieht durch die Angabe einer URI als Wert des „targetNamespace“-Attributs der <schema>-Komponente. Möchte man auf diese Komponenten zugreifen, wird die gleiche URI einem „xmlns“-Attribut zugewiesen und eine entsprechende Modulkomponente im Schema platziert. Bei Verwendung des Attributs in dieser Form werden standardmäßig alle referenzierten Namen in diesem Namensraum gesucht, der dann als „Default Namespace“ bezeichnet wird.

Während nur ein Default Namespace angegeben werden darf, können beliebig viele weitere Namensräume angegeben werden. Hierzu wird dem „xmlns“-Attribut ein nutzerdefiniertes Prä-

[1] an dieser Stelle sei auf den Standard verwiesen: http://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/#layer2.

fix nachgestellt, getrennt durch einen Doppelpunkt. Referenzen auf Komponentennamen, die in derartig spezifizierten Namensräumen liegen, müssen qualifiziert sein. Im Wesentlichen bedeutet dies, dass das entsprechende Präfix lediglich dem referenzierten Namen vorangestellt wird, ebenfalls mit einem Doppelpunkt getrennt.

Hieraus ergibt sich die Unterscheidung zwischen NCNamen (Non-Colonized-Name) und QNamen (Qualified Name). QNamen sind wie oben erwähnt qualifizierte Namen und sind bei Referenzen anzugeben, können aber bei entsprechender Namensraumangabe im Schema durchaus nicht präfixiert sein. NCNamen hingegen sind stets unpräfixiert, da sie Namen von neu eingeführten Komponenten eines Schemas darstellen. Entsprechend werden die verschiedenen Arten wie angedeutet an unterschiedlichen Stellen genutzt. Die anschließenden Ausführungen gehen auch auf diesen Aspekt ein. In XML-Schema werden beide Namensarten durch eigene Datentypen repräsentiert. [2]

2.2.2. Primäre Schemakomponenten

Diese stellen die grundlegensten Bausteine von XML-Schema dar, indem sie maßgebend für die (mögliche) Struktur von XML-Instanzen sind. Die Stellen eines Schemas, in denen diese Komponenten eingeführt werden, bestimmen dabei ihren Gültigkeitsbereich und damit die möglichen Strukturen von XML-Dokumenten. Es werden vier Arten von primären Komponenten unterschieden:

Definition von einfachen Typen Eine der Stärken von XML-Schema ist das Vorhandensein einer reichhaltigen Typisierung. Mit der Auswahl von 44 vordefinierten (built-in), atomaren Datentypen stehen vielfältige Möglichkeiten zur Verfügung, gültige Wertebereiche von Element-Textinhalten oder Attributwerten in XML-Dokumenten zu definieren. Beispiele für solche vordefinierten Datentypen sind u. a. „xs:integer“ oder „xs:string“, deren Bedeutung analog zu den Typsystemen anderer Sprachen ist.

Durch eine Ableitung der vordefinierten Datentypen mittels der <xs:simpleType>-Komponente lassen sich darüber hinaus nutzerdefinierte einfache Typen erstellen. Damit wird die Mächtigkeit des Typsystems von XML-Schema erhöht. Abschnitt 2.5 geht auf diesen Aspekt im Detail ein.

Weiterhin ist es möglich, Typen auf sog. lokaler oder globaler Art zu definieren. Global bedeutet dabei, dass eine Definition „frei“ im Schema als Top-Level-Komponente eingeführt wird. Sie besitzt somit einen globalen Gültigkeitsbereich und kann für die Typzuweisung von Attribut- und Elementdeklarationen per Referenz genutzt werden. Das heißt wiederum, dass globale Definitionen auch einen global eindeutigen Namen besitzen müssen, anhand dessen die Typdefinition referenziert wird. Realisiert wird dies durch die Angabe eines Wertes vom Datentyp NCName für das „name“-Attribut einer <simpleType>-Komponente.

Lokale Definitionen sind dagegen an ihren Kontext gebunden und können nicht wiederverwendet werden. Die <simpleType>-Komponente wird hierbei innerhalb einer Attribut- oder Elementdeklaration verwendet, womit die Typzuweisung dieser Komponenten unmittelbar geschieht. Ein Name ist daher überflüssig, d. h. das „name“-Attribut darf hier explizit nicht auftauchen. Dementsprechend werden lokale Typen auch als anonyme Typen bezeichnet.

Definition von komplexen Typen Einfache Typen sind ausschließlich zur Angabe von Wertebereichen bzw. zur Beschränkung der Ausprägung von Zeichenketten nutzbar. Sollen die Inhalte von Elementen eine tiefergehende Struktur erhalten oder Elemente selbst Attribute tragen, so ist hierfür die Verwendung von komplexen Typdefinitionen vorgesehen. Diese beschreiben, welche

[2] für die genaue Definition sei hier erneut auf den Standard verwiesen: http://www.w3.org/TR/2012/REC-xmlschema11-2-20120405/datatypes.html#NCName bzw. http://www.w3.org/TR/2012/REC-xmlschema11-2-20120405/datatypes.html#QName.

Elemente bzw. Attribute in einem Element auftreten dürfen und wie sie angeordnet sind. Sie bilden das sog. Inhaltsmodell eines Elements. Die Kontrolle über die Reihenfolge von Elementen wird dabei über Elementgruppen, einer Klasse von Hilfskomponenten in XML-Schema, gesteuert (siehe Paragraph 1 in Abschnitt 2.2.4.).

Im Unterschied zu einfachen Typen existieren keine vordefinierten komplexen Typen, womit diese immer vom Nutzer per <complexType>-Komponente zu definieren sind. Dennoch existieren, analog zu den einfachen Typen, Mechanismen zur Ableitung von bereits definierten komplexen Typen. In beiden Fällen stellen jene Mechanismen die Grundlage für die Entstehung von Typhierarchien in XML-Schemata dar. In diesem Zusammenhang sei noch mal auf Abschnitt 2.5 verwiesen.

Auch komplexe Typen lassen sich wie einfache Typen mit einer globalen oder lokalen Definition einführen und folgen diesbezüglich den gleichen Regeln wie die einfachen Typen (siehe oben). Wichtig hierbei ist, dass sich einfache und komplexe globale Typen die selbe Menge an Namen teilen. Die Namenseindeutigkeit umfasst daher die Namen beider Komponentenarten eines Schemas. Entsprechend ist darauf zu achten, dass Attributdeklarationen keine globalen komplexen Typen referenzieren.

Attributdeklaration Soll in einem XML-Dokument grundsätzlich nur ein gewisser Satz an verwendbaren Attributen erlaubt sein, sind dafür Attributdeklarationen bestimmt. Diese können mit mit der <attribute>-Komponente beschrieben werden. Ähnlich wie bei den Typdefinitionen existiert auch hier die Option, Attribute global oder lokal zu deklarieren. Globale Deklarationen erfolgen auf der obersten Schemaebene, während lokale Deklarationen innerhalb anderer Elementdeklarationen (bzw. komplexen Typdefinitionen) als Teil ihres Inhaltsmodells auftreten. Dementsprechend gestaltet sich der Gültigkeitsbereich.

Im Unterschied zu Typdefinitionen trägt eine Attributdeklaration in beiden Fällen ein „name“-Attribut mit einem eindeutigen Wert vom Typ NCName, es sei denn, eine bestehende Deklaration wird referenziert. In diesem Fall wird stattdessen das „ref“-Attribut genutzt, über das der Name einer globalen Deklaration referenziert werden kann. Dabei muss die Referenz als QName angegeben werden. Diese Option ist allerdings nur für lokale Deklarationen erlaubt.

Die Zuweisung eines Attributs bzw. eines Attributnamens zu einem einfachen Typ kann, wie oben erwähnt, auf zwei Wegen erfolgen. Lokale einfache Typen werden direkt in der Attributdeklaration anonym eingeführt, während sich globale einfache Typen durch die Angabe ihres QNamens als Wert des „type“-Attributs referenzieren lassen. Beide Formen schließen sich gegenseitig aus.

Elementdeklaration Mit der <element>-Komponente sind eigene Elemente zur Verwendung in XML-Dokumenten deklarierbar. Die Prinzipien von lokaler und globaler Deklaration sowie der Typzuweisung gelten hier analog zu denen der Attributdeklaration. Beide Komponenten besitzen zudem getrennte Mengen von Namen. Elemente, die auf der obersten Ebene eines Schemas deklariert werden, sind zudem mögliche Einstiegspunkte bzw. Wurzelelemente für XML-Dokumente.

Im Gegensatz zur Attributdeklaration sind die Namen von Elementdeklarationen sowohl mit einfachen Typen als auch komplexen Typen assoziierbar. Der erste Fall ergibt einen Elementinhalt, der nur aus einer Zeichenkette besteht, während der zweite Fall komplex geschachtelte Elementstrukturen ermöglicht.

2.2.3. Sekundäre Schemakomponenten

Aus den sechs sekundären Schemakomponenten werden nachfolgend nur jene vorgestellt, die die höchste Relevanz zum Thema der Arbeit besitzen. In der vorangegangenen Bachelorarbeit des Autors ([Kap13]) oder unter `http://www.w3.org/TR/xmlschema11-1/#concepts-data-model` können Erläuterungen zu den restlichen Komponenten nachgelesen werden.

Definition von Attributgruppen Attributgruppen sind ein Hilfsmittel, um mehrere Attribute einzukapseln. Dies geschieht, indem Attributdeklarationen oder -referenzen innerhalb einer <attributeGroup>-Komponente platziert werden. Die Neueinführung dieser Komponente geschieht immer auf globaler Ebene und muss einen NCNamen in der Form eines „name"-Attributs besitzen. Alternativ sind Attributgruppen anhand ihres Namens in anderen Attributgruppen oder in komplexen Typdefinitionen referenzierbar. Das bedeutet, dass sich gruppierte Attribute beliebig oft an anderen (passenden) Stellen eines Schemas wiederverwenden lassen. Dadurch ist es auf einfache Weise möglich, bestimmte Attributkombinationen im XML-Dokument mehrmals zu verwenden, ohne diese im Schema jedes mal explizit als Deklaration oder Referenz aufzuschreiben.

Definition von Elementgruppen Ein weiteres Werkzeug, um die Wiederverwendbarkeit von Schemakomponenten zu erhöhen, sind Elementgruppen. Ähnlich wie Attributgruppen kapseln sie Elementdeklarationen oder -referenzen in eine wiederverwendbare Einheit, der <group>-Komponente. Ebenso sind diese global zu definieren und besitzen einen Namen. Die Referenzierung von Elementgruppen mittels des „ref"-Attributs wird dabei innerhalb der Definition eines komplexen Typen oder einer anderen Modellgruppe angewandt. Somit lässt sich der gleiche „Elementanteil" eines Inhaltsmodells für andere komplexe Typen redundanzfrei nutzen.

Bei der Definition ist zwingend die Angabe der Art der Elementgruppe erforderlich, die im nächsten Paragraph erläutert wird.

2.2.4. Hilfskomponenten

Die letzte Kategorie besteht aus Komponenten, die innerhalb anderer Komponenten genutzt werden. Auch hier sollen nur die wichtigsten gezeigt werden[3].

Elementgruppen Die Wiederverwendung von Elementgruppen wird mit der <xs:group>-Komponente ermöglicht. Die eigentliche Elementgruppe bzw. die Beschreibung des Inhaltsmodells stellen allerdings eine Gruppe von Hilfskomponenten, den Kompositoren, dar. Es werden drei verschiedene Kompositoren unterschieden.

sequence Diese Elementgruppe gibt eine genaue Reihenfolge der Partikel eines Elementinhalts vor. In einem XML-Dokument dürfen die Unterelemente eines entsprechenden Elements prinzipiell nur in exakt dieser Reihenfolge auftreten.

choice Auch hier wird im Schema eine Partikelliste für ein Element angegeben. Im Unterschied zum <sequence>-Kompositor ist in einer XML-Instanz nur exakt eine Wahl aus dieser Liste als Unterelement erlaubt, die dennoch beliebig oft wiederholt werden kann.

all In diesem Fall ist die Reihenfolge der Unterelemente in einer XML-Instanz beliebig. Seit Version 1.1 von XML-Schema gilt dies zusätzlich für ihre Häufigkeit (siehe den nächsten Paragraph).

Partikel Als Partikel werden Elementdeklarationen und -referenzen sowie Elementgruppen bezeichnet, die in einem komplexen Typ auftreten. Zusätzlich umfasst der Begriff noch sog. Element-Wildcards. Diese sind ebenfalls eine weitere Hilfskomponente (<xs:any>) und ermöglichen in einem XML-Dokument das Einsetzen belieber Elemente an ihrer Stelle.

Wie oben angerissen, kann die Häufigkeit des Auftretens bzw. Kardinalität von Partikeln individuell bestimmt werden. Hierzu können jedem Partikel die optionalen Attribute „minOccurs"

[3] für weitere Details siehe http://www.w3.org/TR/xmlschema11-1/#Model_Group_Summary.

und „maxOccurs“ hinzugefügt werden, die die unteren bzw. oberen Grenzen bestimmen. Der Wert 0 für „minOccurs“ lässt das entsprechende Partikel optional werden. Wird „maxOccurs“ auf „unbounded“ gesetzt, entspricht dies einer Häufigkeit ohne obere Schranke.

2.3. Modellierungsstile

Die vorgestellten Komponenten reichen bereits aus, um komplexe Schemata zu konstruieren bzw. Dokumentstrukturen zu beschreiben. Interessant dabei ist, dass trotz verschiedener Vorgehensweisen bei der Schemamodellierung identische Dokumentmengen beschrieben werden können. Dies gewährt Nutzern einen gewissen Freiheitsgrad Modellierungsformen zu wählen, die individuelle Anforderungen an die Schemaeigenschaften erfüllen, ohne die gewünschte Semantik zu verletzen. Hauptverantwortlich hierfür ist die Option Definitionen und Deklarationen lokal oder global einzuführen. Prinzipiell ist es durchaus legitim, diese Formen nach Belieben zu vermischen. Dennoch gilt das Verwenden von außschließlich lokalen oder globalen Elementen, Attributen und Typen als bewährte Methode für das Schemadesign. Dementsprechend wurden in früheren Arbeiten von Costello in [Cos] und Maler in [Mal02] vier mögliche Modellierungsstile charakterisiert, die nachfolgend kurz vorgestellt werden. Tabelle 2.1 gibt zunächst einen Überblick über die verschiedenen Modellierungsstile.

		Definition	
		lokal	global
Deklaration	lokal	Russian Doll	Venetian Blind
	global	Salami Slice	Garden of Eden

Tabelle 2.1.: Deklarations- und Definitionsarten der einzelnen Modellierungsstile

2.3.1. Russian Doll

Die Modellierung in diesem Stil sieht die Verwendung von lokalen Deklarationen sowie Definitionen vor. Ausgehend von einer einzigen globalen Elementdeklaration werden weitere Typen und Elemente bzw. Attribute ineinander verschachtelt. Die äußere Form des Schemas nimmt damit eine baumartige Struktur an und entspricht quasi exakt der Struktur möglicher XML-Instanzen. Größter Nachteil dieses Stils ist die fehlende Wiederverwendbarkeit von Komponenten, was relativ hohe Redundanzen nach sich ziehen kann. Andererseits sind solche Schemata in sich abgeschlossen. Das heißt, dass keine Abhängigkeiten zu anderen Komponenten im Schema bestehen und Änderungen von einzelnen Komponenten auch nur diese betreffen und keine referenzierenden. Für einen Autor ist dieser Stil damit relativ einfach zu beherrschen.

In XML-Instanzdokumenten, die durch ein Russian Doll Schema beschrieben werden, ist es darüber hinaus möglich, Namensraumangaben aller lokaler Elemente zu verbergen. Kontrollierbar ist diese Option über das „elementFormDefault“-Attribut in der <schema>-Komponente. Der Wert „qualified“ sieht die standardmäßige Verwendung von qualifizierten Elementnamen für lokale Elemente vor, während diese mit „unqualified“ standardmäßig ohne Qualifikation angegeben werden. In Beispiel 2.2 ist ein XML-Schema im Russian Doll Stil dargestellt.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="person">
```

```
  <xs:complexType>
   <xs:sequence>
    <xs:element name="vorname" type="xs:string"/>
    <xs:element name="nachname" type="xs:string"/>
    <xs:element name="geburtsdatum" type="xs:date"/>
   </xs:sequence>
   <xs:attribute name="id" type="xs:integer"/>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Beispiel 2.2: Ein XML-Schema im Russian Doll Stil

Das Schema beschreibt eine Dokumentstruktur für das eingangs gezeigte XML-Dokument 2.1. An der Anordnung der Komponenten lässt sich direkt erkennen, dass dieses Dokument eine Instanz des Schemas ist. Genau wie im Instanzdokument enthält das globale Element „person“ die lokalen Elemente „vorname“, „nachname“ und „geburtsdatum“ sowie das lokale Attribut „id“.

2.3.2. Salami Slice

Die ausschließliche Nutzung anonymer Typen und globaler Deklarationen wird als Salami Slice Stil bezeichnet. Die Inhaltsmodelle der lokalen Typen bestehen dabei aus Referenzen vorhandener Deklarationen. Mit dieser Vorgehensweise wird ein Schema in kleine kompakte Stücke - wie Salamischeiben - eingeteilt und besitzt eine flache Hierarchie. Damit lässt sich ein Schema modularer gestalten, was sich allerdings nachteilig auf die Übersichtlichkeit auswirkt. Ebenso können sich (semantische) Fehler durch die höhere Kopplung der Komponenten an verschiedenen Stellen auswirken. Dennoch sind bei dieser Methode Redundanzen möglich: durch die nicht wiederverwendbaren Typen müssen diese u. U. mehrmals anonym definiert werden. Ein Umweg über Attribut- und Elementgruppenreferenzen innerhalb von anonymen Typen kann diesen Effekt mindern. Da alle Deklarationen global vorliegen, ist es nicht möglich die Namensräume der entsprechenden Komponenten in Instanzdokumenten zu verbergen. Nachfolgend zeigt Beispiel 2.3 ein Salami Slice Schema.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="vorname" type="xs:string"/>
 <xs:element name="nachname" type="xs:string"/>
 <xs:element name="geburtsdatum" type="xs:date"/>
 <xs:attribute name="id" type="xs:integer"/>
 <xs:element name="person">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="vorname"/>
    <xs:element ref="nachname"/>
    <xs:element ref="geburtsdatum"/>
   </xs:sequence>
   <xs:attribute ref="id"/>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Beispiel 2.3: Ein XML-Schema im Salami Slice Stil

Die ehemals lokalen Elemente „vorname“, „nachname“, „geburtsdatum“ und das Attribut „id“ wurden hier global deklariert. Die Dokumentstruktur ist zwar noch an den entsprechenden Referenzen in dem Element „person“ erkennbar, doch lassen sich die Typen der Elemente nicht mehr unmittelbar ablesen. Zusammen mit den „verdoppelten“ Schemazeilen vermindert dies die Übersichtlichkeit des Schemas.

2.3.3. Venetian Blind

Im Gegensatz zum Salami Slice Stil werden beim Venetian Blind Stil lokale Deklarationen und globale Typen eingesetzt. Durch die Typreferenzierung sind ganze Inhaltsmodelle leicht wiederverwendbar. Da Deklarationen nun lokal vorliegen (außer einigen ausgewählten, die als mögliche Einstiegspunkte bzw. Wurzeln für die XML-Dokumente dienen), bessert sich die Übersichtlichkeit des Schemas bei gleichbleibender flacher Hierarchie. Die Kopplung von Komponenten ist ähnlich hoch. Änderungen an Deklarationen betreffen zunächst nur sie selbst, durch die Einbettung in Typdefinitionen betreffen diese allerdings u. U. noch andere Komponenten, die von diesen Typen Gebrauch machen. Redundanzen treten hier entsprechend an der Stelle der Deklarationen auf, sollten diese mehrfach verwendet sein. Auch ist es wieder möglich, durch Umschalten des „elementFormDefault“-Attributs des Schemas, Namensraumangaben von Elementen in XML-Dokumenten zu verbergen oder anzuzeigen. Ein Schema in diesem Stil lässt sich in Beispiel 2.4 betrachten.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="person" type="personType"/>
 <xs:complexType name="personType">
  <xs:sequence>
   <xs:element name="vorname" type="xs:string"/>
   <xs:element name="nachname" type="xs:string"/>
   <xs:element name="geburtsdatum" type="xs:date"/>
  </xs:sequence>
  <xs:attribute name="id" type="xs:integer"/>
 </xs:complexType>
</xs:schema>
```

Beispiel 2.4: Ein XML-Schema im Venetian Blind Stil

Im Gegensatz zu den bisherigen Schemata referenziert „person“ hier den globalen Typ „personType“. Das Inhaltsmodell dieses Typs ist dabei wieder lokal definiert. Durch die Abtrennung der Typen von den globalen Elementen könnten bei größeren Schemata dennoch Probleme mit der Übersichtlichkeit entstehen.

2.3.4. Garden of Eden

Garden of Eden ist die konsequente Weiterführung der Wiederverwendung von Komponenten. Sowohl Deklarationen als auch Definitionen werden global auf der höchsten Schemaebene eingeführt und an anderen Stellen referenziert. Dieser Stil „erbt“ damit die Eigenschaften von Salami Slice und Venetian Blind. Wie der Salami Slice Stil ist dieser Stil relativ unübersichtlich, dafür lassen sich Redundanzen durch mehrfache identische Deklarationen vermeiden. Darüber hinaus sind nach Art des Venetian Blind Stils Typen wiederverwendbar, was die Möglichkeit der redundanten Definition von ein und demselben Inhaltsmodell ausschließt. Die Kopplung der Komponenten erreicht hier also ihr Maximum, mit der bekannten Konsequenz, die Effekte von Änderungen an

Komponenten (und damit u. U. Fehler) an mehrere Stellen zu propagieren. Bei der Fähigkeit, Namensräume zu verbergen, setzt sich die Eigenschaft des Salami Slice Stils durch, bedingt durch die ebenfalls vorhandenen globalen Deklarationen. Als Abschluss der Modellierungsstile ist in Beispiel 2.5 der Garden of Eden Stil präsentiert.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="vorname" type="xs:string"/>
 <xs:element name="nachname" type="xs:string"/>
 <xs:element name="geburtsdatum" type="xs:date"/>
 <xs:attribute name="id" type="xs:integer"/>
 <xs:element name="person" type="personType"/>
 <xs:complexType name="personType">
  <xs:sequence>
   <xs:element ref="vorname"/>
   <xs:element ref="nachname"/>
   <xs:element ref="geburtsdatum"/>
  </xs:sequence>
  <xs:attribute ref="id"/>
 </xs:complexType>
</xs:schema>
```

Beispiel 2.5: Ein XML-Schema im Garden of Eden Stil

In diesem Schema liegen die Elemente „vorname“, „nachname“, „geburtsdatum“ und das Attribut „id“ wie im Salami Slice Stil global vor. Nach der Art des Venetian Blind Stils ist zudem der Typ von „person“ global definiert. Die Möglichkeit der maximalen Wiederverwendbarkeit dieser Komponenten wird zum Preis einer redundanten bzw. unübersichtlichen äußeren Form des Schemas erzielt.

2.4. XML-Schemaevolution

Ist ein XML-Schema einmal (ohne syntaktische Fehler) niedergeschrieben, heißt dies nicht, dass es keinen Änderungen mehr unterliegt. XML-Schemata sind im Kern selbst Modelle von „Dingen“, die ein Schemaautor abbilden wollte. Hierbei können zum einen semantische Fehler bei der Übertragung in XML-Schema auftreten. Zum anderen ist es durch natürliche Änderungen des „realen“ Vorbilds wahrscheinlich, dass sich im Laufe der Zeit auch die Anforderungen an die Abbildung ändern. In beiden Fällen muss das Schema angepasst werden. Nach dieser Prozedur kann es mitunter passieren, dass die Gültigkeit von XML-Dokumenten nicht mehr gegeben ist. In einem zweiten Schritt sind daher die Änderungen eines XML-Schemas auf die zugehörigen XML-Dokumente zu übertragen. Beide Vorgänge werden mit dem Begriff XML-Schemaevolution umfasst.

Die manuelle Ausführung dieser Tätigkeiten kann eine langwierige Aufgabe bedeuten, weswegen Ansätze zur Automatisierung dieses Prozesses entwickelt werden. Neben der Evolution auf den XML-Schemadokumenten selbst, ist diese auch auf konzeptionellen Modellen der Schemata möglich. Bisher existieren allerdings nur wenige Prototypen für diese Art der Schemaevolution. In [NKMH13] stellen Nečaský et al. den Prototypen *eXolutio* vor, einen der aktuellsten Vertreter für die konzeptionelle Schemavolution. Nach den Prinzipien der Model-driven Architecture wird in fünf Ebenen von den Details der XML-Dokumente abstrahiert. Schemasprachen wie XML-Schema dienen hier als logische Ebene und werden ihrerseits in höheren Ebene durch UML-artige Diagramme repräsentiert.

Ein weiterer Prototyp ist *CodeX* (Conceptual Design and Evolution for XML-Schema), das von Nösinger et. al an der Universität Rostock entwickelt wurde [NKH12]. Die konzeptionelle Darstellung von XML-Schemata basiert hier im Unterschied zu anderen Ansätzen, nicht auf UML- oder (erweiterten) ERM-Diagrammen, sondern auf dem konzeptionellen Modell *Entity Model for XML-Schema* (EMX), dass ebenfalls an der Universität Rostock entworfen wurde [NKH13]. Als Grundlage für diese Arbeit wird sich in den folgenden Kapiteln ausführlich mit diesem Forschungsprototypen beschäftigt. Weitere Prototypen stellt z. B. Deffke in [Def12] vor.

2.5. Typsystem von XML-Schema

In den Ausführungen über einfache und komplexe Typen wurde nur die einfachste Form der Definition beschrieben. Mit verschiedenen Mitteln des Typsystems von XML-Schema können bestehende Typdefinitionen für neue genutzt werden. In diesem Abschnitt sollen die Möglichkeiten hierfür und das Typsystem von XML-Schema im Allgemeinen vorgestellt werden.

In der Entwicklung der Programmiersprachen war die Bildung des Typkonzepts ein wichtiger Schritt für die Anfertigung fehlerfreier Programme. Typen werden den Konstrukten einer Sprache zugewiesen und erlauben damit u. a. die Ermittlung von ungültigen Wertezuweisungen zu jenen Konstrukten. Darüber hinaus definieren sie, welche Operationen auf Werten zulässig und wie diese für die Ergebnisbildung zu interpretieren sind. Die Gesamtheit der Typen einer Programmiersprache fließt in dem sog. Typsystem zusammen, mit dessen Hilfe z. B. die Typprüfung zur Übersetzungszeit von Programmen (statische Typisierung) oder zur Laufzeit (dynamische Typisierung) ausgeführt wird.

2.5.1. Wertebereich und Repräsentation von Werten

Obwohl XML-Schema keine Programmiersprache ist, können dennoch einige Parallelen gezogen werden. Mit der Bereitstellung diverser atomarer Datentypen ist es ebenfalls möglich, zunächst die erlaubten Werte für einige Konstrukte von XML-Schema einzuschränken. Diese Konstrukte sind in diesem Fall Attribut- und Elementdeklarationen bzw. im Speziellen die Inhalte von „text-only“-Elementen und die Werte der Attributen in XML-Instanzen, wie in Abschnitt 2.2.2 beschrieben. Erst durch solche Einschränkungen können Element-Textinhalte und Attributwerte in XML-Dokumenten auf ihre Konformität zu den Definitionen in einem Schema feingranular überprüft werden. Wenn diese gegeben ist, ist der entsprechende Dokumentteil quasi eine Instanz der Typdefinition.

An dieser Stelle ist es wichtig, zwischen Wert und dessen Darstellung zu unterscheiden. Sowohl in den Programmiersprachen als auch in XML-Schema sind Datentypen als eine Zusammenfassung von Werten zu einer Menge, also einen Wertebereich (**Value Space**), zu verstehen. Demgegenüber steht die Repräsentation eines Wertes durch Literale des lexikalischen Raums (**Lexical Space**). Beispielsweise ist 815 ein möglicher Wert des Datentyps „float“. Für diesen Datentyp sind 815 selbst, 0.815E3 oder +8.15+e2 gültige Repräsentationen dieses Wertes. Werte können also durchaus verschiedene Darstellungen besitzen. Gerade für die Validierung spielt dies eine Rolle, da etwa die genanntenten Beispiele alle gültige Angaben für ein entsprechendes Attribut in einem XML-Dokument sind. Mit dieser Dynamik setzt sich XML-Schema erneut von anderen Schemasprachen wie z. B. DTD ab.

2.5.2. Konstruktion und Hierarchie von einfachen Typen

Ähnlich zu den Programmiersprachen existiert in XML-Schema die Möglichkeit, mittels Typkonstruktoren neue Datentypen aus bestehenden zu erzeugen. In Abschnitt 2.2.2 wurde dies bereits kurz erläutert. XML-Schema stellt zur Definition von einfachen Typen drei verschiedene Typkonstruktoren zur Verfügung:

- Restriction
- List
- Union

Restriction Im ersten Fall wird die Wertmenge eines einfachen Typen i. A. durch Untermengenbidung eingeschränkt. Das bedeutet, dass letzten Endes immer ein built-in Datentyp der XML-Schemaspezifikation als Ausgangspunkt bzw. Basisdatentyp für nutzerdefinierte Datentypen dient. Abbildung A.1 im Anhang stellt alle Datentypen der XML-Schemaspezifikation sowie deren Beziehung in der Typhierarchie im Überblick dar.

Sämtliche Typen, ob built-in oder nutzerdefinierte, einfache oder komplexe, stammen von dem Datentyp „anyType“ ab. Ausgehend von diesen ist speziell für einfache Typen der Typ „anySimpleType“ abgeleitet. Der Unterschied zwischen diesen beiden besteht darin, dass „anyType“ in XML-Dokumenten sowohl als komplexer als auch einfacher Typ dienen kann, während „anySimpleType“ ausschließlich für die Verwendung als einfacher Typ vorgesehen ist. Wertebereich und -repräsentation dieses Typs sind dabei prinzipiell unbeschränkt, d. h. (beliebige) Werte sind durch beliebige Unicode-Zeichenketten darstellbar, solange sie atomar oder endliche Listen sind (siehe unten).

Als Zusammenfassung der *primitiven Datentypen* leitet sich der Typ „anyAtomicType“ von „anySimpleType“ ab. Dieser ist bereits insofern eingeschränkt, als sein Wertebereich bzw. lexikalischer Bereich als Vereinigung der restlichen, von ihm abgeleiteten Datentypen definiert wird. „anyType“, „anySimpleType“, „anyAtomicType“ werden auch als *spezielle Datentypen* bezeichnet.

Die *primitiven Datentypen* stammen wie angedeutet von „anyAtomicType“ ab und stellen den eigentlichen „Kern“ der Typbibliothek von XML-Schema dar. Neben den primitiven Datentypen, die von Anfang an existieren, gibt es einige vordefinierte, von diesen *abgeleitete Datentypen*. Für jeden dieser Typen definiert die Spezifikation einen Wertebereich und eine passende Repräsentation. Die erste Angabe erfolgt durch eine rein textuelle Beschreibung des „Konzepts“ des Datentyps. Letzteres wird durch eine EBNF-ähnliche Grammatik angeben. Erwähnenswert ist dies deshalb, da diese Art der Einschränkung nicht der Art für die Ableitung von *nutzerdefinierte Datentypen* entspricht.

Diese werden nämlich über die <restriction>-Komponente in einer Schemadatei erzeugt. Wie der Name vermuten lässt, schränkt diese Komponente gewisse Eigenschaften eines bestehenden Typen ein, der so die Rolle des Basisdatentyps einnimmt. Diese Eigenschaften werden durch sog. Facetten repräsentiert, wobei für jeden primitiven Datentyp eine Reihe von verschiedenen Facetten definiert ist. Über Werteangaben in den einzelnen Facetten steuert der Autor, welche Eigenschaften wie eingeschränkt werden. Dies führt zwangsläufig zu einer Reduzierung der erlaubten Werte eines Datentyps bzw. ihrer möglichen Repräsentationen und damit zur eingangs erwähnten Untermengenbildung. Da für die speziellen Datentypen keine Facetten definiert sind, dürfen diese auch nicht als Basisdatentyp verwendet werden. Zur Erläuterung der einzelnen Ableitungskonzepte sind nachfolgend Auszüge aus einem Schema angegeben, wobei das vollständige Schema im Anhang unter B.1 betrachtet werden kann. In Beispiel 2.6 ist zunächst die Ableitung durch Restriktion für einfache Typen dargestellt.

```
...
<xs:simpleType name="hobbyType">
 <xs:restriction base="xs:NMTOKEN">
  <xs:maxLength value="20"/>
 </xs:restriction>
</xs:simpleType>

<xs:simpleType name="europaLandType">
 <xs:restriction base="xs:string">
  <xs:enumeration value="Deutschland"/>
  <xs:enumeration value="Schweiz"/>
 </xs:restriction>
</xs:simpleType>

<xs:simpleType name="amerikaLandType">
 <xs:restriction base="xs:string">
  <xs:enumeration value="USA"/>
  <xs:enumeration value="Kanada"/>
 </xs:restriction>
</xs:simpleType>
...
```

Beispiel 2.6: Ableitung von einfachen Typen durch Restriktion

Es werden drei einfache Typen „hobbyType“, „europaLandType“ und „amerikaLandType“ von den built-in Datentypen „NMTOKEN“ bzw. „string“ abgeleitet. Die Facette „maxLength“ begrenzt dabei die Länge möglicher Werte von Komponenten, denen als „hobbyType“ als Typ zugewiesen wurde. „europaLandType“ sowie „amerikaLandType“ definieren jeweils mit Hilfe der „enumeration“-Facette eine kleine Anzahl an Werten, die ausschließlich verwendet werden dürfen.

List Ein weiterer Typkonstruktor ist der Listenkonstruktor. In XML-Schema sind Listen eigene Datentypen und lassen sich durch die <list>-Komponente erzeugen. Listendatentypen verwenden prinzipiell, ähnlich wie die Ableitung durch Restriktion, einen bestehenden Datentyp als Basisdatentyp. Im Unterschied zu atomaren Datentypen umfasst der Wertebereich von Listendatentypen endliche Mengen von Werten seines Basistyps. Analog dazu besteht der lexikalische Bereich aus einer endlichen Liste aus atomaren Werten in der Form des lexikalischen Bereichs des Basistyps, wobei die Listenelemente mit Whitespace-Zeichen getrennt sind. Als Basisdatentypen kommen hierbei sowohl atomare Typen als auch Union-Typen (ohne Listendatentypen als Mitglieder) in Frage. Es gibt zwar einige vordefinierte Listendatentypen wie „IDREFS“ oder „NMTOKENS“, doch diese sind ebenfalls mittels Ableitung erzeugt und daher auch keine primitiven Datentypen. Beispiel 2.7 zeigt die Listenbildung auf Basis des vorherigen Beispiels.

```
...
<xs:simpleType name="hobbiesType">
 <xs:list itemType="hobbyType"/>
</xs:simpleType>
...
```

Beispiel 2.7: Ableitung eines einfachen Typen durch Listenbildung

„hobbyType“ aus Beispiel 2.6 dient hier als Basistyp für „hobbiesType“. Werte für diesen Typen bilden somit solche Zeichenketten, die jeweils maximal 20-stellige Strings (ohne Whitespaces) mit Whitespaces verbinden.

Union Der letzte Typkonstruktor wird durch die bezeichnende Vereinigungoperation (Union) charakterisiert. Ähnlich wie bei dem Listenkonstruktor umfasst der Wertebereich Mengen von Werten anderer Typen. Diese sind dabei nicht an einen einzigen Basistyp gebunden. Vielmehr entsteht der Wertebereich eines Union-Typen aus der Vereinigung der Wertebereiche mehrerer, vom Nutzer angegebenen Datentypen. Diese können in diesem Fall sowohl atomar als auch listenwertig sein. Entsprechend gestaltet sich auch der lexikalische Raum als endliche Liste aus den Wertepräsentationen verschiedener Datentypen. Im Schema wird dieser Konstruktor mittels der <union>-Komponente verwendet. Die Vereinigung von einfachen Typen ist in Beispiel 2.8 zu sehen.

```
...
<xs:simpleType name="landType">
 <xs:union memberTypes="europaLandType amerikaLandType"/>
</xs:simpleType>
...
```

Beispiel 2.8: Ableitung eines einfachen Typen durch Vereinigung

Die Wertebereiche von „europaLandType“ und „amerikaLandType“ werden in einem neuen Typ „landType“ vereinigt. Nach der Zuordnung dieses Typen zu einer XML-Schemakomponente, dürfen in einem Instanzdokument anschließend an entsprechenden Stellen die <enumeration>-Werte beider Mitglieds-Typen auftreten.

Die konkrete Nutzung aller drei Typkonstruktoren in XML-Schemata folgt dem gleichen Prinzip. Die jeweilige Komponente wird als direktes Kind in einer <simpleType>-Komponente platziert. An dieser Stelle besteht die Möglichkeit globale oder lokale Datentypen als Basistypen zu verwenden. Im globalen Fall wird entsprechender Komponente ein Attribut hinzugefügt, mit dem globale Typen referenzierbar sind („base“ für <restriction>, „itemType“ für <list> und „memberTypes“ für <union>, in den beiden letzteren Attributen können mehrere Typen angegeben werden). Sollen stattdessen anonyme Typen als Basistypen dienen, wird dies wie üblich umgesetzt: unterhalb der jeweiligen Komponenten wird ein einfacher Typ (<restriction>) oder mehrere (<list>, <union>) einfache Typen als direkte Kinder eingeführt.

Neue Datentypen werden also immer über eine Ableitung bzw. Einschränkung anderer erzeugt. Ohne einen jeweiligen Basisdatentyp könnten jene Datentypen nicht existieren. Aus dieser Tatsache wird deutlich, warum es zur Bildung einer Typhierarchie in XML-Schema kommt.

2.5.3. Konstruktion und Hierarchie von komplexen Typen

Komplexe Typen nehmen ebenfalls an Typhierarchien teil. Die Spezifikation sieht dabei nur einen einzigen komplexen Typ vor, und zwar den konzeptionellen Typ „anyType“. Dieser dient nutzerdefinierten komplexen Typen, die erstmals in einem Schema definiert werden, als impliziter Basistyp. Von jenen sind dann wiederum weitere komplexe Typen ableitbar. Die Ableitungsmöglichkeiten hängen dabei davon ab, nach welcher Kategorie der neue komplexe Typ konstruiert wird.

Komplexe Typen sind unterteilbar in solche mit einfachem oder komplexem Inhalt. In einer Schemadatei ist dies erkennbar an entweder dem <simpleContent>- oder <complexContent>-Kind einer <complexType>-Komponente. Fehlen diese völlig und es steht an ihrer Stelle direkt

eine Elementgruppe, handelt es sich um eine Kurzform eines Typs mit komplexem Inhalt, wobei „anyType“ als Basistyp fungiert. Der Unterschied zwischen den beiden Kategorien besteht darin, dass die erste Sorte nur Elemente mit reinem Textinhalt ermöglicht, während bei der zweiten Sorte auch komplexe Inhaltsmodelle bzw. Unterelemente in den XML-Instanzen denkbar sind. Beiden gemein ist die Möglichkeit für die Angabe von Attributen. Ebenfalls bieten beide Gruppen zwei Konstruktoren für die Typableitung an:

- Restriction
- Extension

Restriction Bei komplexen Typen mit einfachem Inhalt funktioniert die Ableitung durch Einschränkung ähnlich zu der entsprechenden Ableitungsform von einfachen Typen. Im Wesentlichen wird direkt unterhalb der <simpleContent>-Komponente eine <restriction>-Komponente eingefügt. Hier kann sich ein Schemaautor nun wieder entscheiden, ob er einen lokalen oder globalen einfachen Typ mit dem jeweils verschiedenen Vorgehen einschränken möchte. Zum Basistyp passende Facetten können anschließend ebenfalls manipuliert werden. Neu hinzu kommt die Möglichkeit, Attribute, Attributgruppen oder eine Attribut-Wildcard anzugeben. Diese signalisieren, dass entsprechend typisierte Elemente in XML-Dokumenten, neben einem konformen Wert als Inhalt, auch die angegebenen Attribute besitzen müssen oder können.

Liegt ein Typ mit komplexem Inhalt vor, verhält sich dieser Vorgang ähnlich, wobei keine lokalen Typen einschränkbar sind. Statt den Werten eines einfachen Typen wird hier allerdings das Inhaltsmodell eines komplexen Typen beschränkt. Dies äußert sich so, dass mit der Angabe von Partikeln selbst das Inhaltsmodell im abgeleiteten Typ quasi neu definiert wird. Allerdings müssen sämtliche Elementgruppen und Elemente des Basistypen im abgeleiteten Typ wieder auftreten, wobei die Attribute bzw. Attributwerte einiger der Partikel manipulierbar sind. Der Grund hierfür ist, dass Instanzen eines abgeleiteten Typen auch Instanzen des Basistypen sein müssen. Dementsprechend unterliegen die Änderungen der Partikelattribute gewissen Regeln, die die Entstehung einer Untermenge (im Bezug zum Inhaltsmodell) des Basistyps durch die Ableitung sicherstellen.

Attribute des Basistyps werden bei der Ableitung durch Einschränkung anders behandelt. Diese sind implizit in der Definition des neuen Typen vorhanden bzw. vererbt ohne sie aufzuschreiben. Erst wenn gewisse Eigenschaften der Attribute geändert werden sollen, müssen diese erneut mit den gewünschten Änderungen definiert werden. Auch hier greifen Regeln, die die möglichen Änderungen einschränken. In Beispiel 2.9 wird die Ableitung von komplexen Typen mit einfachem und komplexem Inhalt gezeigt.

```
...
<xs:complexType name="gehaltDollarType">
 <xs:simpleContent>
  <xs:restriction base="gehaltType">
   <xs:attriubte name="waehrung" type="waehrungType" fixed="Dollar"/>
  </xs:restriction>
 </xs:simpleContent>
</xs:complexType>

<xs:complexType name="USAEliteStudentType">
 <xs:complexContent>
  <xs:restriction base="studentType">
   <xs:sequence>
    <xs:sequence>
     <xs:element name="vorname" type="xs:string"/>
     <xs:element name="nachname" type="xs:string"/>
```

```
      <xs:element name="geburtsdatum" type="xs:date"/>
      <xs:element name="land" type="amerikaLandType" fixed="USA"/>
     </xs:sequence>
     <xs:sequence>
      <xs:element name="hobbies" type="hobbiesType" minOccurs="0" maxOccurs=
          "0"/>
      <xs:element name="gehalt" type="gehaltDollarType" minOccurs="1"
          maxOccurs="1"/>
     </xs:sequence>
    </xs:sequence>
    <xs:attribute name="matrikelnummer" type="xs:ID" use="prohibited"/>
   </xs:restriction>
  </xs:complexContent>
</xs:complexType>
...
```

Beispiel 2.9: Ableitung von komplexen Typen durch Restriktion

Die Beschränkung im komplexen Typ „gehaltDollarType“ auf Basis von „gehaltType“ (siehe Beispiel 2.10) geschieht, indem das Attribut „waehrung„ auf den Wert „Dollar“ fixiert wird. Zudem werden im Typ „USAEliteStudentType“ folgende Bestandteile des Inhaltsmodells seines Basistypen „studentType“ beschränkt: Element „land“ ist auf den Wert „USA“ fixiert, die Kardinalitäten von „hobbies“ und „gehalt“ wurden verringert und das Attribut „matrikelnummer“ darf nicht mehr auftauchen.

Extension Entgegen des bisherigen Paradigmas, einfache oder komplexe Typen durch Einschränkungen abzuleiten, arbeitet die Ableitung durch Erweiterung. Ziel ist hier nicht die Erzeugung von Untermengen, sondern von Obermengen der Inhaltsmodelle bzw. möglicher Instanzen der Typen. Diese Art der Typkonstruktion ist demnach exklusiv den komplexen Typen vorbehalten und wird mit der <extension>-Komponente eingeleitet. Die Unterscheidung zwischen den beiden Kategorien von komplexen Typen wird auch bei dieser Ableitungsform gemacht.

Den einfachste Fall stellen komplexe Typen mit einfachem Inhalt dar. Facetten finden hier keinen Einsatz, womit „lediglich“ die Angabe von Attributen, die ein Element tragen soll, übrig bleibt. Weiterhin ist es dieses Mal nicht möglich, anonyme Typen als Basistyp zu verwenden. Es muss daher immer ein globaler Typ über das „base“-Attribut referenziert werden.

Das Vorgehen bei komplexen Typen mit komplexem Inhalt ist quasi identisch zu dem für die Einschränkung jener. Erneut können Partikel spezifiziert werden, die das Inhaltsmodell des erweiterten Typen bilden sollen. Dabei verändern die notierten Partikel keine Komponenten des Basistyps, weswegen diese nicht doppelt aufgeschrieben werden müssen. Genau wie bei den Attributen ist das Inhaltsmodell des Basistyps dem neuen Typ bereits implizit vererbt. Neu hinzukommende Komponenten werden dann konzeptionell direkt hinter die letzte Komponente des Basistyps gehängt. Der Effekt ist dabei der gleiche, als würde man einen <sequence>-Kompositor um die jeweiligen Kompositor-Partikel des Basistyps und des abgeleiteten Typs platzieren. Das bedeutet, dass Instanzen des erweiterten Typs nicht zwangsläufig Instanzen des Basistyps sein müssen. Dieses Vorgehen birgt damit einige Risiken, die die Korrektheit des Schemas verletzen könnten. Dementsprechend sind auch hier einige Regeln für die Verwendung dieses Konstruktors einzuhalten.

Bei beiden Formen werden Attribute genauso wie bisher behandelt, d. h. sie werden ebenfalls vererbt und bei Bedarf neu definiert. Die Extension ist nachfolgend in Beispiel 2.10 einsehbar.

```
...
<xs:complexType name="gehaltType">
```

```
 <xs:simpleContent>
  <xs:extension base="xs:nonNegativeInteger">
   <xs:attribute name="waehrung" type="waehrungType"/>
  </xs:extension>
 </xs:simpleContent>
</xs:complexType>

<xs:complexType name="studentType">
 <xs:complexContent>
  <xs:extension base="personType">
   <xs:sequence>
    <xs:element name="hobbies" type="hobbiesType" minOccurs="0" maxOccurs="
       1"/>
    <xs:element name="gehalt" type="gehaltType" minOccurs="0" maxOccurs="1"
       />
   </xs:sequence>
   <xs:attribute name="matrikelnummer" type="xs:ID"/>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>
...
```

Beispiel 2.10: Ableitung von komplexen Typen durch Erweiterung

Der primitive Datentyp „nonNegativeInteger“ dient als Grundlage für die Erweiterung zu dem komplexen Typ „gehaltType“. Elemente in Instanzdokumenten dürfen als Werte weiterhin nur jene des Basistyps besitzen. Als Ergänzung kommt hier das Attribut „waehrung“ hinzu, sodass neben dem eigentlichen Elementinhalt noch die Angabe dieses Attributs möglich ist. Zu dem Inhaltsmodell von „personType“ fügt der Typ „studentType“ die Elemente „hobbies“ und „gehalt“ sowie das Attribut „matrikelnummer“ hinzu.

Als Abschluss fasst die Tabelle 2.2 die möglichen Konstruktoren zur Typableitung noch einmal zusammen.

Ableitung	**simpleType**	**complexType**	
		simpleContent	complexContent
Restriction	Facetten	Facetten Attribute	Inhaltsmodell Attribute
Extension	-	Attribute	Inhaltsmodell Attribute
List	Liste von Typwerten	-	-
Union	Vereinigung von Typwerten	-	-

Tabelle 2.2.: Typableitungsformen in XML-Schema

Ziel dieses Kapitels war zunächst die Erläuterung der Grundlagen von XML und XML-Schema, wobei das Hauptaugenmerk auf den strukturgebenden Komponenten wie Elemente und Typen lag. Anschließend wurden die Thematiken der Konstruktion bzw. Ableitung von Typen sowie der Bildung von Typhierarchien beleuchtet. Aufbauend auf diesen Erkenntnissen, soll nachfolgend untersucht werden, wie Werkzeuge aus der Praxis die Typhierarchien eines Schemas darstellen und handhaben.

3. Stand der Technik

Als Vorbereitung für die Entwicklung des Konzepts zum Management von Typhierarchien gibt dieses Kapitel zunächst eine Übersicht darüber, welche Methoden hierfür bereits existieren. Da es nach bestem Wissen des Autors keine Werkzeuge gibt, die ausschließlich diesen Aspekt beleuchten, werden zu diesem Zweck Ansätze aus verschiedenen Bereichen der XML-Schemaverarbeitung betrachtet. Diese setzen sich zum einen aus XML-Editoren mit der Fähigkeit zur Bearbeitung von XML-Schemata und zum anderen aus Lösungen der konzeptionellen Modellierung zusammen.

3.1. Testschema

Als Grundlage zur Bewertung der untersuchten Ansätze dient das „Person-Ableitungs-Schema“ aus Anhang B.1, was in den vorherigen Abschnitten teilweise vorgestellt wurde. Als akademisches Beispiel liegt der Fokus auf den verschiedenen Formen der Typableitung. Passenderweise kam daher ein Modellierungsstil mit globalen Typdefinitionen in Frage, wobei aus Gründen der Übersichtlichkeit Ausnahmen Venetian Blind als Basis mit wenigen Ausnahmen gewählt wurde.

Das Schema definiert eine Hierarchie aus komplexen Typen für Personen, Studenten und Professoren mit jeweils unterschiedlichen Inhaltsmodellen. Die Typen der Partikel referenzieren dabei u. a. auch nutzerdefinierte einfache Typen, die ihrerseits in flachen Hierarchiebeziehungen stehen. Es wurde darauf geachtet, dass im Schema jede Art der Typableitung mindestens einmal erscheint. Tabelle 3.1 gibt einen Überblick darüber, welche Komponenten welche Ableitungsformen realisieren.

Typen	**Ableitung**	**Typ**	**Basistyp**	**Bemerkung**
einfache	Restriction	hobbyType	xs:NMTOKEN	enumeration
		professorHobbiesType	hobbiesType	enumeration
		waehrungType	xs:string	enumeration
		amerikaLandType	xs:string	enumeration
		europaLandType	xs:string	enumeration
	List	hobbiesType	hobbyType	
	Union	landType	amerikaLandType europaLandType	
komplexe	Extension	gehaltType	xs:nonNegativeInteger	neues Attribut
		professorType	personType	neue Elemente
		studentType	personType	neue Elemente und Attribute
	Restriction	gehaltDollarType	gehaltType	Attributwertfixierung
		gehaltEuroType	gehaltType	Attributwertfixierung
		europaProfessorType	professorType	Untertypen der Partikel
		USAEliteStudentType	studentType	Beschränkung von Elementkardinalitäten und Attributauftreten

Tabelle 3.1.: Typableitungsformen im Testschema

In der Vergangenheit haben Felix Michel und Erik Wilde in [MW06] eine Reihe von XML-Editoren mit der Fähigkeit zur (grafischen) Darstellung und Bearbeitung von XML-Schemata analysiert. Hierfür haben die Autoren eine Liste aus Kriterien aufgestellt, an denen die Beurteilung der untersuchten Werkzeuge erfolgte. Nach reichlicher Überlegung haben sich einige der Kriterien auch bzw. weiterhin für die aktuelle Untersuchung als angemessen und sinnvoll ergeben, sodass eine Auswahl dieser auch hier als Grundlage dient. Während sich Michel und Wilde mehrere Aspekte Visualisierung verschiedener XML-Schemakomponenten widmeten, soll

diese Arbeit vor allem die Aspekte rund um Typen und Typhierarchien fokussieren. Im Zentrum stehen somit folgende Fragestellungen, die anhand des Testschemas geklärt werden:

- Perspektive: Auf welche Komponentenart konzentriert sich die Darstellung, d. h. stehen Typen oder Elemente im Vordergrund und kann zwischen den Darstellungen gewechselt werden? Sind versteckte Informationen auffindbar?
- Darstellungsform: Welche Form der Visualisierung verwendet der Ansatz? Textuelle Beschreibungen, Tabellen, Bäume, Graphen oder andere Formen? Werden mehrere alternative Sichten unterstützt?
- Abstraktion: Wie stark wird von der eigentlichen Schemasyntax abstrahiert? Orientiert sich die Visualisierung an dem Schemadokument oder an dem Datenmodell des Schemas?
- Typen: Wie werden diese dargestellt und gibt es eine Unterscheidung zwischen lokalen und globalen Definitionen? Ist der Wertebereich von einfachen Typen erkennbar?
- Typhierarchien: Wird die Visualisierung der Typhierarchie eines Schemas unterstützt und unterscheidet diese die einzelnen Ableitungsformen? Ist es möglich, innerhalb der Hierarchie zu navigieren? Sind eingeschränkte Eigenschaften des Basistyps wie z. B. Facetten oder verbotene Bestandteile besonders hervorgehoben?
- Inhaltsmodelle: Wie handhabt die Visualisierung die Inhaltsmodelle der Typen? Sind diese direkt und mit der vollständigen Nestung erkennbar oder abstrahiert die Darstellung zusätzlich?
- Referenzen: Lässt sich die Definition oder Deklaration einer Referenz suchen? Ist dies auch in die umgekehrte Richtung möglich, d. h. sind z. B. sämtliche Referenzen zu einer gegebenen Deklaration bzw. Definition auffindbar?
- Editierbarkeit: Dient die Visualisierung als reine Veranschaulichung oder werden auch Optionen zur Änderung der Schemakomponenten angeboten? Existieren dabei (im Hinblick auf die Typhierarchie) Ausgleichsmechanismen, sollte nach Änderungen die Korrektheit nicht mehr gegeben oder die semantische Bedeutung verletzt sein?

3.2. XML-Editoren

Das Fazit der Arbeit von Michel und Wilde ist, dass zur damaligen Zeit keines der untersuchten Werkezeuge eine ausreichende Visualisierung des Datenmodells eines XML-Schemas bot. Unter den analysierten Editoren befanden sich z. B. bereits auch frühere Versionen von XMLSpy und Eclipse WTP. Die gewählten Darstellungsformen abstrahierten vielmehr von der Syntax eines XML-Schemadokuments, als von seinem eigentlichen Datenmodell selbst.

Dieses Vorgehen verspricht allgemein nur wenig Besserung für das Verständnis eines XML-Schemas, da etwa die Navigation nicht entlang der Domänenkonzepte erfolgt, sondern im Prinzip immer noch an den Konstrukten von XML-Schema, bloß in einer anderen Darstellung. Eine Verringerung der Komplexität der XML-Schemaspezifikation wird damit also nicht in Angriff genommen. Mittlerweile ist einige Zeit vergangen, sodass eine erneute, ähnliche Untersuchung eventuell neue Erkenntnisse für die Schemamodellierung mit derartigen Editoren liefern könnte.

3.2.1. Altova XMLSpy 2014

Der erste Kandidat ist XMLSpy von Altova als einer der bekannteren XML-Editoren ([Alt14]). In einer integrierten Entwicklungsumgebung ist die Bearbeitung von XML-basierten Anwendungen möglich, wobei diverse XML-Technologien wie XML-Schema unterstützt werden.

Der Editor bietet drei wesentliche Sichten auf ein XML-Schema: eine Text-View, in welcher das Schemadokument selbst bearbeitet werden kann, eine Grid-View, die die XML-Syntax in einer baumartigen Darstellung wiedergibt und eine Schema-View, die eine vollständige Visualisierung des Schemas ist. Der Fokus soll hier auf der letzten Sicht liegen.

Prinzipiell stehen die Elemente in der Darstellung im Vordergrund, aber auch die Typen sind einsehbar. Hierzu sind zunächst alle globalen Deklarationen und Definitionen aufgelistet. In dieser Liste können die Attribute der jeweiligen Komponenten editiert werden. Weiterhin sind aus dieser Liste komplexe Typen und Elemente für die Visualisierung auswählbar. Per Knopfdruck wird die entsprechende Komponente in eine Diagrammansicht überführt. Eine Alternative hierzu bietet die Komponenten-Seitenleiste, in der auf alle globalen Komponenten schnell zugreifbar ist.

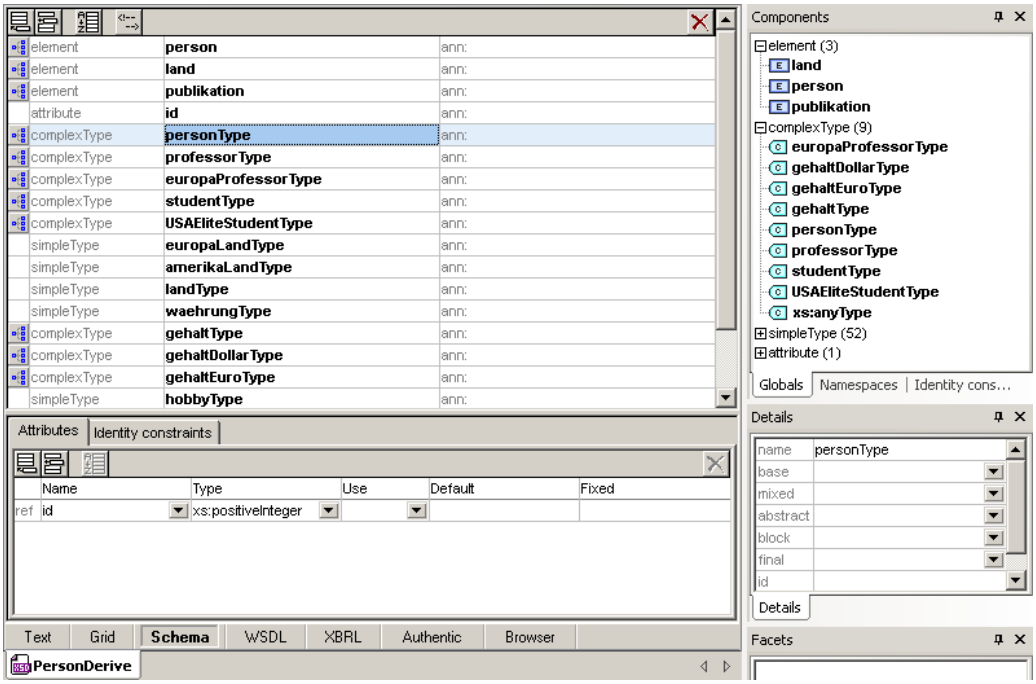


Abbildung 3.1.: Übersicht der Schema-View, Komponenten-Seitenleiste sowie Attribut- und Detailangaben von XMLSpy.

Abbildung 3.1 zeigt die Listen-Übersicht sowie die Komponenten-Seitenleiste. Für den Zweck des Typmanagements kann man diese Tatsache bereits negativ auslegen, da jeweils nur einzelne Schemakomponenten visualisiert werden. Innerhalb der Darstellung ist es jedoch möglich, die Typen von genesteten Elementen „aufzuklappen“ und so wiederum deren Inhaltsmodell zu betrachten, was sowohl für lokal deklarierte Elemente als auch für Referenzen funktioniert. Dazu kommt die Möglichkeit, per Kontextmenü die Definitionen und Deklarationen von referenzierten komplexen Typen und Elementen zu finden. Ebenso können von diesen alle Referenzen ermittelt werden. Diese Mechanismen ermöglichen eine einfache Navigation in der Schemastruktur.

Die Inhaltsmodelle von ausgewählten komplexen Typen wird direkt angezeigt, während das Inhaltsmodell eines referenzierten komplexen Typen von einer gestrichelten Linie umzogen und gelb markiert ist. Handelt es sich hierbei um eine Typableitung, wird zusätzlich die Art der Ableitung hinter dem Namen des Basistyps vermerkt. Das Inhaltsmodell des Basistypen ist in beiden Fällen vollständig abgebildet, was besonders für die Ableitung durch Erweiterung einen Verständnisgewinn bedeutet. Das geerbte Inhaltsmodell wird wie beschrieben markiert und neue Elemente und Attribute außerhalb der Markierung platziert, sodass eine klare Abtrennung von den ursprünglichen und den neu hinzugekommenen Partikeln erkennbar ist. Die Ableitung

durch Erweiterung wird in Abbildung 3.2 links gezeigt, während rechts die Ableitung durch Einschränkung zu sehen ist.

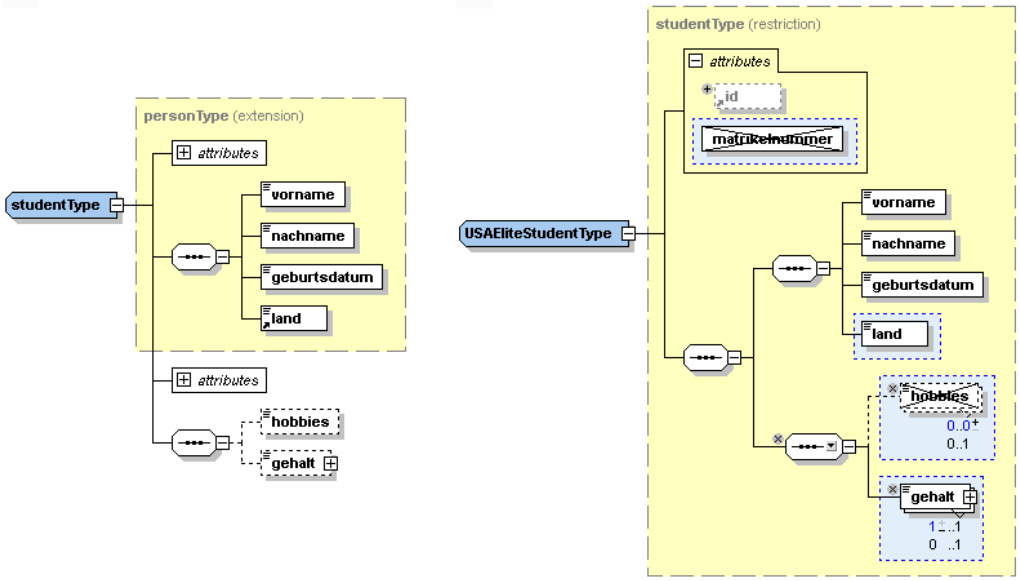


Abbildung 3.2.: Darstellung von komplexen Typen in XMLSpy, die durch Erweiterung (links) und Einschränkung (rechts) abgeleitet wurden.

Ebenso hilfreich ist die Darstellung des Inhaltsmodells eines durch Einschränkung abgeleiteten Typen. Attribute, deren Auftreten im neuen Typ durch den Wert „prohibited“ für die „use“-Angabe ausgeschlossen werden, sind mit einem durchstreichenden Kreuz markiert. Gleiches gilt für Elemente, deren Kardinalitäten auf minimal und maximal 0 gesetzt werden, wobei auch die veränderten Kardinalitäten von Basistyp zu abgeleiteten Typ zu erkennen sind.

Generell deutet die Markierung einer Komponente mit einer gestrichelten, blauen Linie auf eine Einschränkung verschiedener Eigenschaften hin, wie etwa die Verringerung von Kardinalitäten, die Attributwertfixierung, die Verschärfung über die Nutzung von Attributen oder der Beschränkung auf eine Zuweisung eines Untertypen zu einem Partikel. Leider ist nicht in allen Fällen sofort in den Diagrammen ersichtlich, welche Eigenschaft betroffen ist. Diese sind stattdessen aus der Details-Seitenleiste zu entnehmen.

Die Details von Attributen und (nutzerdefinierten) einfachen Typen sind nicht weiter visualisiert und werden ausschließlich in der Details-Seitenleiste angezeigt. Für einfache Typen können hier zusätzlich Ableitungsformen und entsprechende Basistypen eingesehen und per Dropdown-Liste bestimmt werden. Daneben zeigt eine weitere Seitenleiste Informationen über mögliche Facetten an. In Abbildung 3.3 sind die verschiedenen Ableitungsformen für einfache Typen in XMLSpy dargestellt. Von Elementen referenzierte einfache Typen sind ebenfalls nicht in der eigentlichen Visualisierung erkennbar, sondern tauchen unter der Rubrik „type“ in der Details-Seitenleiste auf. Wird ein nutzerdefinierter einfacher Typ referenziert, ist es allerdings nicht möglich, zur entsprechenden Definition zu springen. Diese muss manuell aufgesucht werden.

Weiterhin ist es möglich, viele Aspekte bereits in den Diagrammen selbst zu bearbeiten. Das Kontextmenü bietet etwa die Funktion, XML-Schemabausteine an Komponenten an- bzw. einzufügen, wobei stets nur zur Spezifikation valide Optionen vorgeschlagen werden. Für genauere Angaben stehen zudem die erwähnten Seitenleisten zur Verfügung.

Obwohl die Navigierbarkeit in der Schemastruktur bzw. in den Referenzbeziehungen sehr gut ausfällt, gibt es keinen Gesamtüberblick über die Hierarchien bzw. Vererbungsbeziehungen von komplexen und einfachen Typen eines Schemas. Damit ist die Navigierbarkeit in diesem Aspekt quasi nicht gegeben. Modifikationen an einem Typ werden an alle Referenzen bzw. an die Defi-

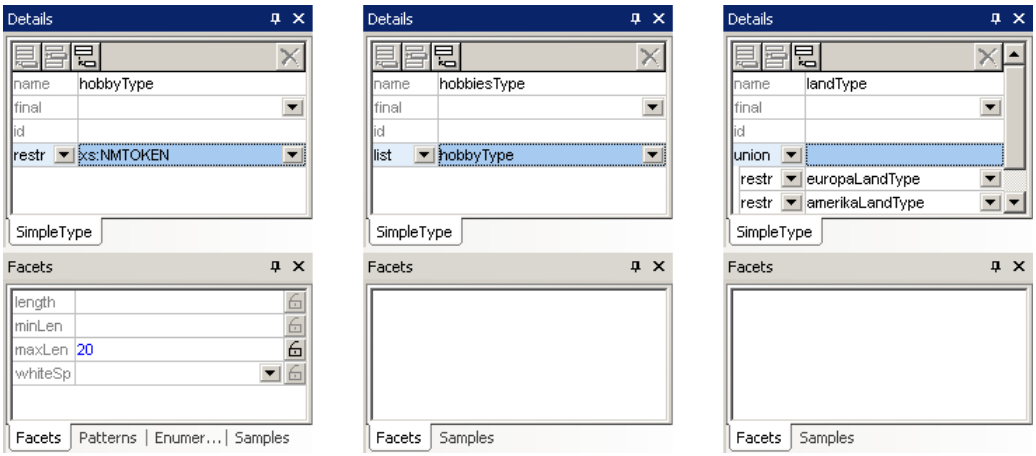


Abbildung 3.3.: Darstellung von abgeleiteten einfachen Typen in der Details-Seitenleiste. Die Ableitungsform ist in einer Zeile aufgeführt und kann manipuliert werden. Facetten sind in der separaten Facets-Seitenleiste zu entnehmen.

nition propagiert. Fehler werden dabei rot markiert. Es existiert allerdings keine Kompensation, sollten Typen z. B. aus der Hierarchie gelöscht werden.

Zusammenfassend kann festgehalten werden, dass XMLSpy ein mächtiges Werkzeug zur grafischen Bearbeitung von XML-Schema ist. Dabei bleibt die grafische Darstellung stets relativ nahe an der eigentlichen Syntax des Schemas, sodass XMLSpy als Werkzeug zur Modellierung von Domänen eher weniger geeignet ist. Dementsprechend stehen Apsekte wie die vollständige Darstellung der Beziehung zwischen den einzelnen Typen auch nicht im Vordergrund. Die Handhabung von eingeschränkten Schemakomponenten bei der Typableitung kann dennoch als positiv hervorgehoben werden.

3.2.2. Visual Studio 2013 XML-Schema-Designer

Ein weiteres Werkzeug ist der XML-Schema-Designer von Microsofts Visual Studio ([Mic13]). Diese Entwicklungsumgebung ist nicht primär auf den Umgang mit XML-Technologien ausgelegt, dennoch wird mit dem integrierten XML-Editor ein interessanter Ansatz zur Visualisierung von XML-Dokumenten und XML-Schemata verfolgt. Die Bearbeitung und die Darstellung von Schemadateien ist grundlegend voneinander getrennt: ersteres geschieht ausschließlich in einem Dokumenteditor, während für die Darstellung verschiedene Sichten angeboten werden. Die Direktmanipulation der grafischen Darstellung ist also nicht möglich. Der Dokumenteditor bietet aber Textvervollständigungsfunktionen, Warnungs- und Fehlerinformationen als Unterstützung an. Vorschläge zur Behebung von Fehlern während des Editiervorgangs (wie fehlende Typen) werden nicht angeboten.

Die Sichten teilen sich hier wie folgt auf: die Graph-View zeigt die reinen Beziehungen zwischen den verschiedenen Schemakomponenten in einem Graphen, während die Details in der der Content-Model-View angezeigt werden können. Beide Sichten arbeiten zusammen mit der Seitenleiste „XML Schema Explorer“. Hier wird, ähnlich wie die Components-Leiste von XMLSpy, eine Übersicht über alle globale Komponenten gegeben. Diese lassen sich aus der Leiste per Drag and Drop auf den Arbeitsbereich ziehen, der entweder die Graph-View oder die Content-Model-View anzeigt. Zwischen bedien Ansichten kann mit den rot markierten Symbolen in Abbildung 3.4 gewechselt werden. Alternativ dazu lässt auch das gesamte Schema (ebenfalls als Icon in der Explorer-Seitenleiste repräsentiert) auf den Arbeitsbereich ziehen, womit alle Komponenten auf einmal visualisiert werden.

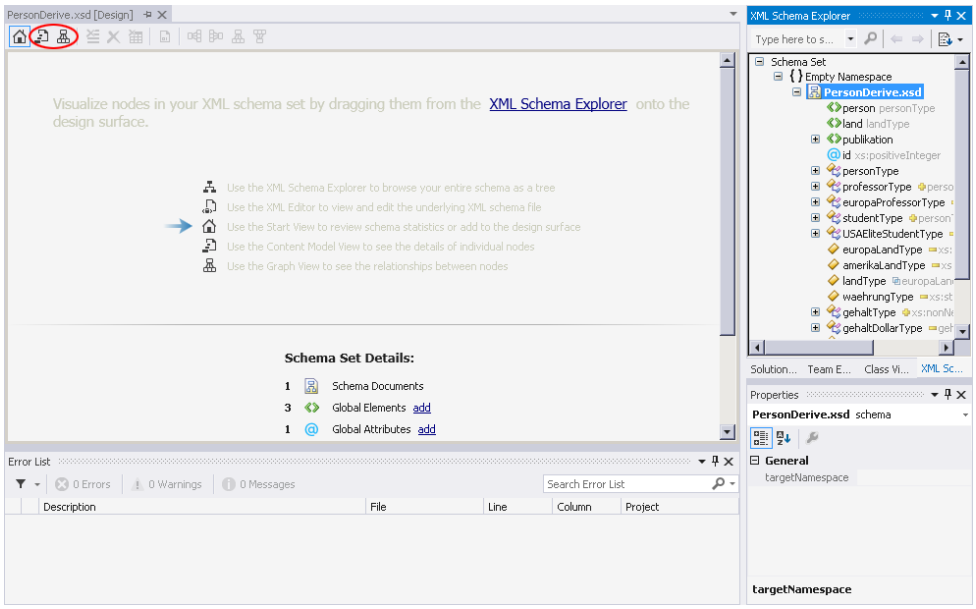


Abbildung 3.4.: Erste Ansicht im XSD-Designer vom Arbeitsbereich (Mitte) und XML Schema Explorer (rechts).

Die Graph-View (zu sehen in Abbildung 3.5) stellt sich als eine der Stärken dieses Werkzeugs heraus, da die Beziehungen einzelner Schemakomponenten ohne zusätzliche Klicks erkennbar ist. Dies umfasst sowohl Elemente und komplexe Typen als auch Attribute und einfache Typen. Komponenten, die in einer Vererbungsbeziehung (Typen) oder Referenzbeziehung (Elementereferenzen in Inhaltsmodellen oder Typreferenzen globaler Elemente) stehen, werden durch Pfeile verbundenen, sodass quasi eine Art Wald entsteht. Diese Ansicht ist damit bereits eine relativ hohe Abstraktion eines Schemadokuments, da die Typreferenzen von Partikeln im Inhaltsmodell eines komplexen Typen nicht dargestellt sind und generell viele Details ausgelassen werden. Somit steht hier nicht die Dokumentstruktur im Vordergrund, sondern die „Konzepte“ der Anwendung. Besonders nützlich ist dies für die Visualisierung der Typhierarchien, sowohl von komplexen als auch einfachen Typen.

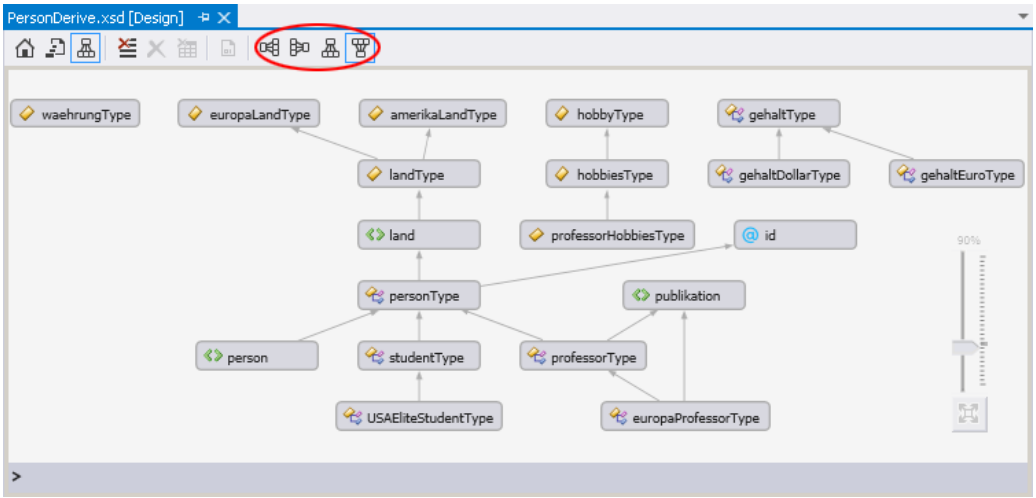


Abbildung 3.5.: Visualisierung der Beziehungen mit der Graph-View, wobei keine Details angezeigt werden. Mit den Symbolen innerhalb der Markierung lässt sich die Ausrichtung der Diagramme verändern.

Aus der grafischen Darstellung dieser Ansicht selbst geht nicht hervor, um welche Ableitungsform es sich handelt. Ausnahme bildet die Ableitung durch Vereinigung für einfache Typen, da zwischen abgeleiteten Typen und allen Mitgliedstypen eine Verbindung besteht. Im Allgemeinen müssen diese Informationen aus der Explorer-Seitenleiste oder der Content-Model-View entnommen werden. Hier wird hinter dem Namen eines Typen ein Symbol zusammen mit dem Basistypnamen angegeben. Mittels der Seitenleiste ist zusätzlich das Suchen und Auffinden von Referenzen, abgeleiteten Typen und Vorgängern in der Hierarchie von einfachen und komplexen (globalen) Typen möglich. Für Element- und Attributdeklarationen können zugewiesene Typen und im Falle von globalen Deklartionen ebenfalls alle Referenzen gesucht werden. Andersherum lassen sich ebenso die Deklarationen von referenzierten Attributen und Elementen ausfindig machen. Die Content-Model-View kann zudem dazu genutzt werden, die durch referenzierte komplexe Typen entstehende Nestung von Elementen und Attributen vollständig anzuzeigen.

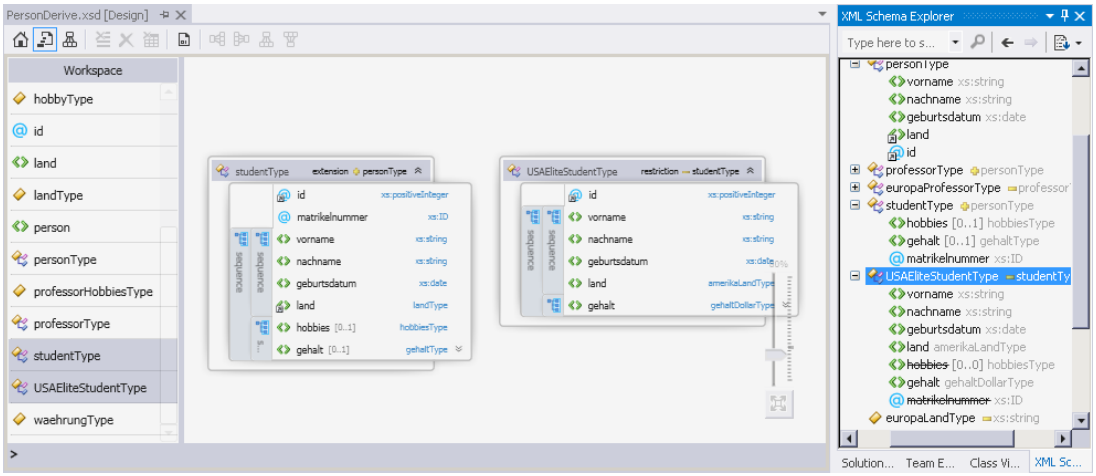


Abbildung 3.6.: Darstellung der Ableitung durch Erweiterung und Einschränkung komplexer Typen in der Content-Model-View und dem Schema-Explorer

Während diese Funktionen eine gute Navigierbarkeit in der Schemastruktur erzielen, kann die eigentliche Darstellung der Inhaltsmodelle komplexer Typen als „durchwachsen“ angesehen werden. Generell ist dafür die Nutzung der Explorer-Seitenleiste und die Content-Model-View vorgesehen (siehe Abbildung 3.6). Positiv ist die Tatsache, dass verbotene Attribute und Elemente in dem Inhaltsmodell eines durch Einschränkung abgeleiteten komplexen Typen mittels Durchstreichen hervorgehoben sind. Dies betrifft dabei allerdings nur den Explorer. Mit der Content-Model-View wird wahrscheinlich ein Prinzip verfolgt, nach dem die nur tatsächlich vorhandenen Partikel angezeigt werden sollen. Dafür spricht auch der Umstand, dass das Inhaltsmodell von Typen, die durch Erweiterung entstanden sind, in der Content-Model-View vollständig angezeigt wird, während die Explorer-Seitenleiste nur die neu hinzugekommenen Partikel zeigt. Die Kompositoren sind dabei exklusiv in der Content-Model-View zu sehen. Das Inhaltsmodell anonymer Typen ist direkt unterhalb der entsprechenden Elternkomponente platziert und durch ein Aufklappen dieser erkennbar. Alle weiterführenden Informationen bzw. Attribute, von XML-Schemakomponenten, die u. a. mögliche Werte einschränken, sind zudem in einer „Properties“-Seitenleiste vermerkt. Die verschiedenen Ableitungsarten von einfachen Typen werden in der Content-Model-View optisch nur durch verschiedene Symbole bzw. den Namen der Ableitung unterschieden, was in Abbildung 3.7 betrachtet werden kann.

Dadurch, dass verschiedene Informationen aus verschiedenen Bereichen der grafischen Oberfläche entnommen werden müssen, wirkt die Darstellung insgesamt ein wenig inkonsistent bzw. verstreut. Obwohl Manipulationsmöglichkeiten an der Visualisierung fehlen, ist zum Zweck der

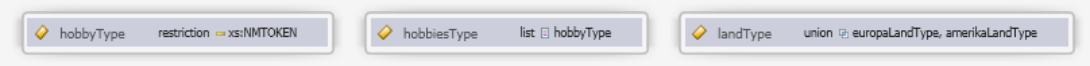


Abbildung 3.7.: Darstellung der Ableitung durch Einschränkung, Listenbildung und Vereinigung einfacher Typen in der Content-Model-View.

reinen Darstellung von Schemadokumenten dieser Ansatz dennoch interessant, da verschiedene Abstraktionsebenen existieren, mit denen vor allem die Typhierarchien erkennbar sind.

3.2.3. Oracle JDeveloper 12c

Diese Entwicklungsumgebung von Oracle ([Ora13b]) dient hauptsächlich der Entwicklung von Java Anwendungen, aber auch hier wird ein Editor zur Bearbeitung von XML-Schemata angeboten. Sobald ein XML-Schemadokument geöffnet wurde, stehen dem Nutzer in der Source-View ein Dokumenteditor und eine Visualisierung mittels Diagrammen in der Design-View zur Verfügung, zwischen denen nach belieben gewechselt werden kann.

Die Gestaltung und die Prinzipien der Design-View erinnert an die Schema-View aus XMLSpy, da zunächst alle globalen Komponenten untereinander aufgelistet werden. Im Unterschied zu XMLSpy lassen sich weitere Details bereits aus den Diagrammen in dieser Auflistung durch Aufklappen entnehmen, sodass ein Wechsel zwischen den Komponenten unnötig ist. Dieses Prinzip lässt sich in Abbildung 3.8 anschauen. Weiterhin werden neben den komplexen Typen und Elementen auch einfache Typen und Attribute soweit wie möglich grafisch dargestellt. Es werden zwar andere Symbole für die verschiedenen Komponentenarten verwendet, ein klarer Fokus auf Elemente oder Typen ist dabei aber nicht auszumachen.

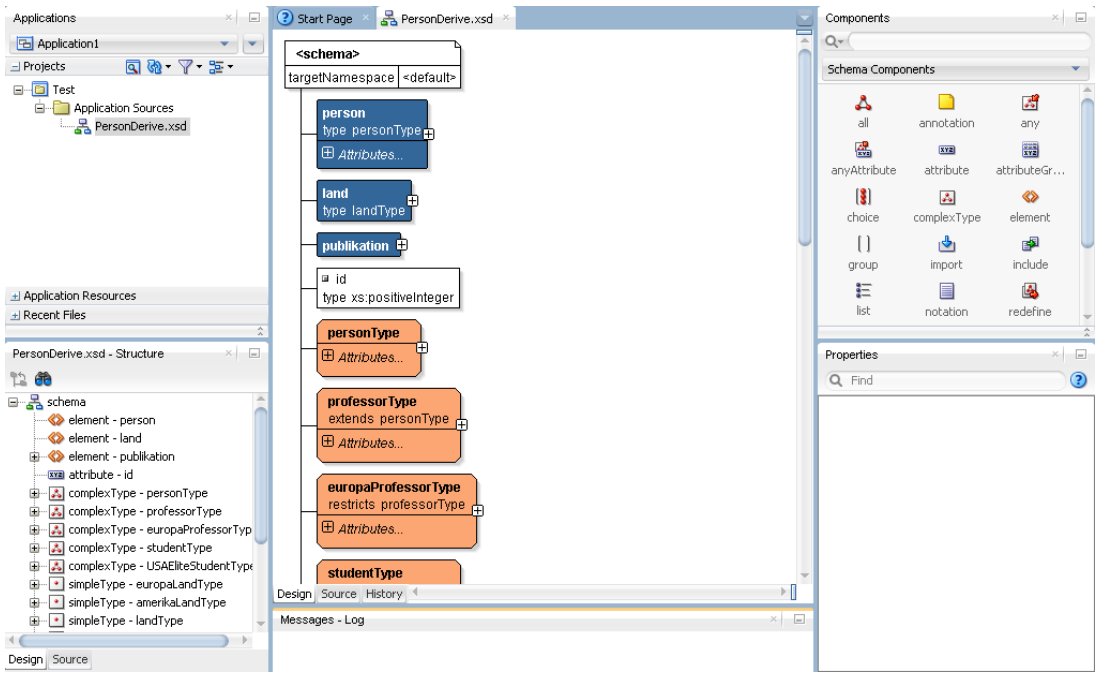


Abbildung 3.8.: Übersicht der grafischen Oberfläche in JDeveloper.

Die Struktur von komplexen Typdefinitionen ist direkt mitsamt der Kompositoren zu sehen. Wird der gleiche Typ von einem Element referenziert, so wird das Inhaltsmodell mit einem farblichen Hintergrund markiert, wobei der Name des referenzierten Typen ebenfalls vermerkt wird. Dieses Vorgehen wiederholt sich auch für tiefer verschachtelte Elemente. Elementreferen-

zen werden grafisch aufgelöst, sodass auch hier die Inhaltsmodelle der Deklarationen direkt zu sehen sind. Das Inhaltsmodell von lokalen komplexen Typen wird hingegen erneut direkt ohne Markierung angezeigt.

Die Darstellung von Typableitungen ist ebenfalls ähnlich wie in XMLSpy gelöst. Unter den Namen von Typdefinitionen, die per Ableitung konstruiert sind, wird der Basistyp und Ableitungsform vermerkt. Bei der Erweiterung von komplexen Typen wird zunächst das geerbte Inhaltsmodell markiert und neu hinzukommende Partikel außerhalb der Markierung angeordnet, sodass hier wieder eine Trennung des Ursprungs der Partikel zu erkennen ist. Attribute, die bei der Ableitung durch Einschränkung nicht mehr Teil der Typdefinition sind, werden durch ein rotes Kreuz markiert. Für Elemente, auf die das ebenfalls zutrifft, wird statt einer schwarzen Umrandung eine gestrichelte gewählt. Dieser Aspekt ist also relativ gut gelöst und kann in Abbildung 3.9 eingesehen werden.

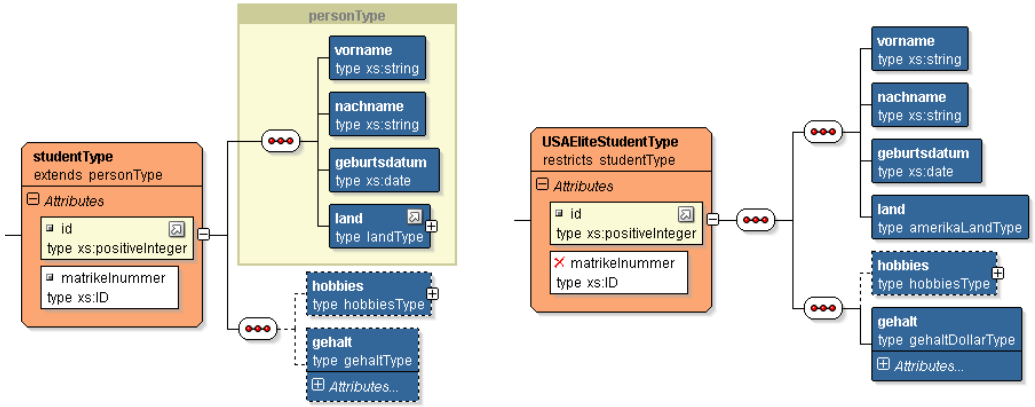


Abbildung 3.9.: Darstellung der Ableitung durch Erweiterung und Einschränkung komplexer Typen.

Eigenschaften bzw. Attribute von Komponenten sind in der Properties-Seitenleiste zu finden. Weniger strikte Wertänderungen von Eigenschaften wie die Auftretenshäufigkeiten oder Typänderungen in Partikeln sind allerdings nicht weiter im Diagramm des abgeleiteten Typen hervorgehoben. Stattdessen wird lediglich mit einem grünen Punkt an entsprechender Eigenschaft in der Properties-Seitenleiste auf eine Modifikation hingewiesen, wie in Abbildung 3.10 zu sehen. Sollte die obere Grenze eines entsprechenden Partikels im Basistyp größer als eins sein und auf eins im abgeleiteten Typ verringert werden, lässt sich diese Änderung immerhin durch den Vergleich der Diagrammformen erkennen, da Komponenten mit erhöhter Kardinalität mit „gestapelten“ Diagrammen dargestellt sind.

Die Repräsentation einfacher Typdefinitionen ist im Allgemeinen aufschlussreich, da die Facetten direkt im Diagramm aufgelistet werden (siehe Abbildung 3.11). Um zwischen Vereinigungs- und Listentypen zu unterscheiden, müssen diese aufgeklappt werden, sodass die Ableitungsart und die Mitgliedstypen oder der Listenbasistyp einsehbar sind. Referenzierte einfache Typen werden wie die komplexen Typen ebenfalls mit einer Markierung umfasst.

Weiterhin ist es möglich, Referenzen von verschiedenen Komponenten über den Aufruf des Kontextmenüs am jeweiligen Diagrammbaustein zurückzuverfolgen, umgekehrt scheint dies jedoch nicht möglich zu sein. Zusammen mit der Möglichkeit zur Darstellung des vollständigen Inhaltsmodells von (verschachtelten) Elementen ist damit erneut ein rasches Fortbewegen in und Verstehen der Schemastruktur gegeben.

Editiermöglichkeiten bestehen generell überall da, wo Werte, Namen oder Referenzen angezeigt werden, auch wenn dies nicht gleich ersichtlich ist. Durch einen Doppelklick ist es in den meisten Fällen möglich, entsprechende Angaben zu ändern. Zusätzlich dazu sind bei Referenzen

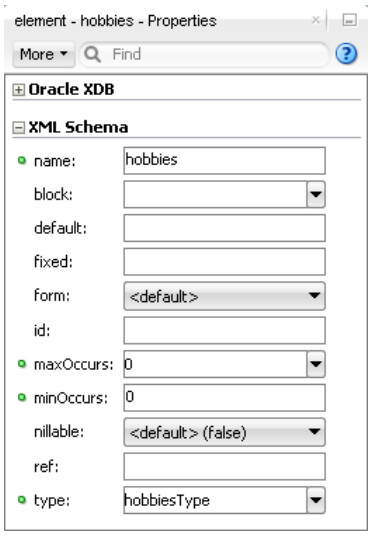


Abbildung 3.10.: Anzeige von Details in der Properties-Seitenleiste.

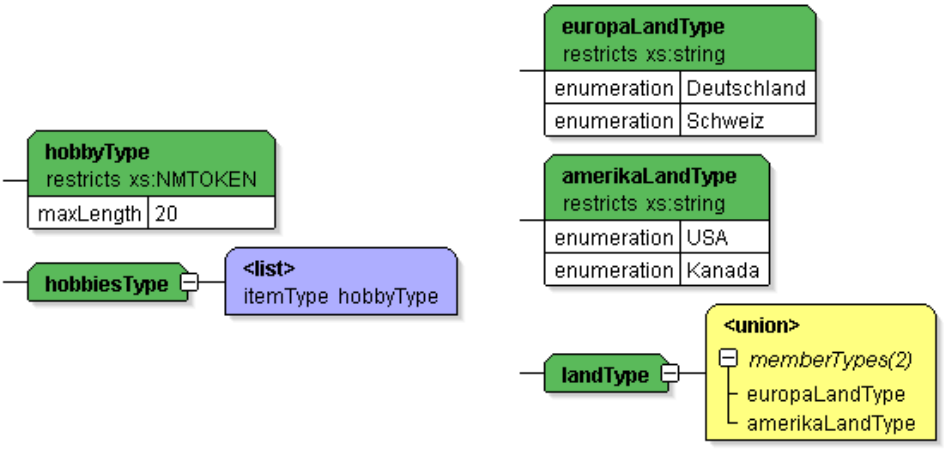


Abbildung 3.11.: Darstellung der Ableitung durch Einschränkung, Listenbildung und Vereinigung einfacher Typen. Facetten und Werte sind direkt einsehbar.

entsprechend passende globale Komponenten als Vorschlag aus einer Dropdown-Liste auswählbar. Weiterhin kann die bestehende Struktur durch Einfüge- und Löschoperationen aus dem Kontextmenü manipuliert werden, wobei beim Einfügen nur valide Komponenten auswählbar sind. Nutzer können zudem per Drag and Drop von oder Klick auf Bausteine aus der Schema-Components-Seitenleiste entsprechende Komponenten an passenden Stellen in der Visualisierung einfügen, um schnell neue Strukturen zu erstellen.

Änderungs- und Navigieroperationen sind jeweils noch einmal auf der Structure-Seitenleiste möglich, die in einer klassischen Baumansicht einen schnellen Überblick über ein Schema liefert, wobei einige Details ausgelassen werden.

Die Visualisierung geht dabei allerdings nicht wesentlich über die XML-Syntax hinaus, womit der Editor für die Darstellung der Typhierarchien ebenfalls nicht optimal ist. Auch Mechanismen für die Kompensation von fehlenden oder fehlerhaften Typen fehlen. Somit eignet sich der JDeveloper allgemein gut für die Editierung von XML-Schemata und weniger für die Darstellung der darin abgebildeten Konzepte.

3.2.4. Eclipse WTP 3.5.2

Auch die Entwicklungsumgebung Eclipse ([Ecl13]) enthält einen Editor für die Bearbeitung von XML-Dokumenten bzw. XML-Schemata. Dieser ist Teil des Projekts Web Tools Platform (WTP, [Ora13a]) und kann unter dem Namen „Eclipse XML Editors and Tools“ auch separat als Plugin heruntergeladen werden.

Sobald der Nutzer ein XML-Schemadokument öffnet, erhält dieser eine Übersicht über die globalen Elemente, Typen, Attribute und Elementgruppen. In der Outline-View von Eclipse sind diese Angaben ebenfalls als Baum dargestellt zu finden. Beide Komponenten werden in Abbildung 3.12 dargestellt.

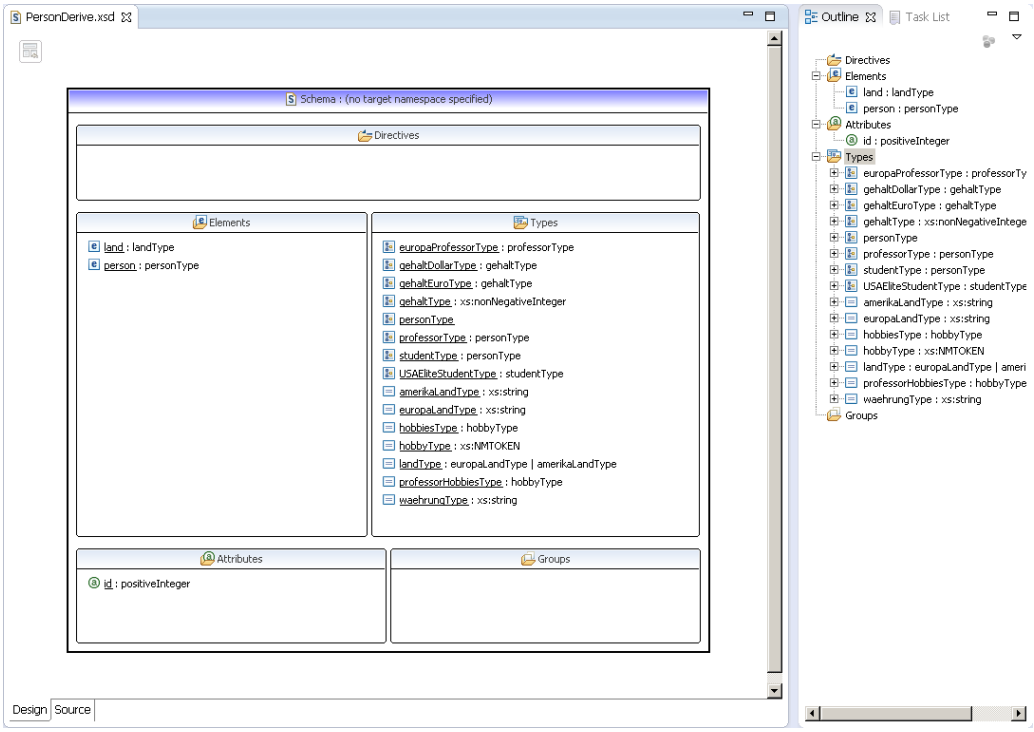


Abbildung 3.12.: Komponenten-Überblick in Eclipse in dem Darstellungsbereich (Mitte) und der Outline-View (rechts)

Außerdem ist das Wechseln zwischen einer Design-View für die Visualisierung und einer Source-View für die detaillierte Bearbeitung von XML-Schemadokumenten erneut möglich. Wird eine Komponente für die Visualisierung ausgewählt, wechselt die Ansicht von der Komponentenübersicht zu einer Detaildarstellung der ausgewählten Komponente. Dieses Vorgehen ähnelt somit dem von XMLSpy und teilt die Eigenschaft der mangelnden Gesamtdarstellung.

Elemente und Typen können gleichermaßen für die Darstellung ausgewählt werden, dennoch liegt das Hauptaugenmerk hier auf den Typen. Die Struktur einer komplexen Typdefinition wird in tabellenartigen Diagrammen aufgelistet, wobei Symbole für die Kompositoren wie gewohnt entsprechende Bestandteile verbinden und der Typname als Diagrammüberschrift dient. Liegen in der Struktur Elemente mit nutzerdefinierten Typen vor, werden entsprechende Inhaltsmodelle ohne weitere Klicks zum Aufklappen o. Ä. direkt ihrerseits als Tabelle angezeigt und mit Referenzen durch Pfeile verbunden. Das gleiche gilt auch für einfache sowie anonyme Typen. Letztere werden aus Konsistenzgründen benannt, wobei der Name der Elternkomponente verwendet und um das Suffix „Type“ erweitert wird. Zur Abgrenzung echter benannter Typen wird der so konstruierte Name von Klammern umschlossen.

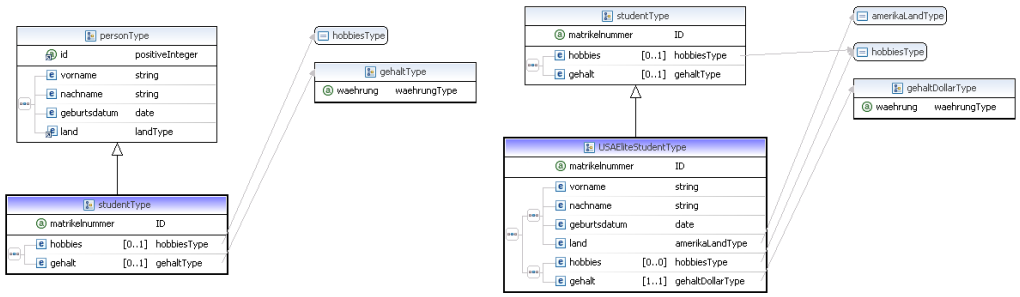


Abbildung 3.13.: Visualisierung der Ableitung komplexer Typen durch Erweiterung (links) und Einschränkung (rechts)

Ähnlich wird auch die Darstellung von Vererbungsbeziehungen zwischen Typen gehandhabt. Zusammen mit dem darzustellenden Typ, der durch eine Ableitung entstanden ist, wird sein Basistyp direkt über ihn mit angegeben. Basistyp und abgeleiteter Typ sind zudem durch einen Pfeil verbunden, was optisch an die Klassendiagramme aus UML erinnert. Diese Art der Darstellung wird sowohl für komplexe als auch einfache Typen verwendet (jeweils zu sehen in den Abbildungen 3.13 und 3.14), wobei auch built-in Datentypen als Basistyp angezeigt werden. Besonders nützlich ist dies für die Ableitung durch Erweiterung, da im erweiterten Typ nur das neue Inhaltsmodell dargestellt wird, während das ursprüngliche Inhaltsmodell im Basistyp eingesehen werden kann. Sobald die oberen und unteren Kardinalitäten von Partikeln andere Werte außer eins haben, werden diese an entsprechender Stelle in der Darstellung vermerkt. Ist das Auftreten durch die Beschränkung dieser Werte auf null ausgeschlossen, so wird dies nicht zusätzlich optisch vermerkt. Gleiches gilt für das Verbot von Attributen, was grafisch nicht erkennbar ist und erneut aus der Properties-View entnommen werden muss. Für ein einfaches Verständnis ist dies nicht förderlich.

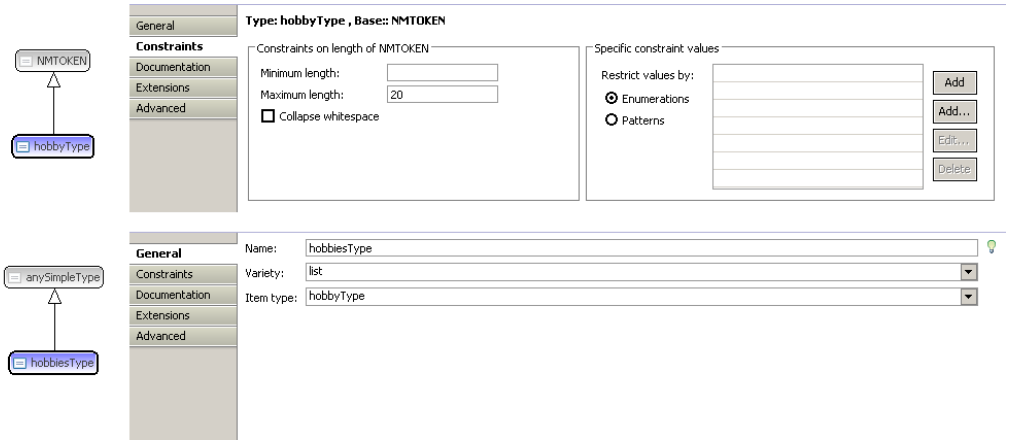


Abbildung 3.14.: Es existiert keine grafische Abgrenzung zwischen den verschiedenen Ableitungen für einfache Typen. Stattdessen sind weitere Informationen aus verschiedenen Reitern der Properties-View zu entnehmen.

Die grafische Oberfläche bietet verschiedene Möglichkeiten für die Manipulation von XML-Schemadokumenten. In dem Diagramm einer Komponente können Namen und Typen geändert werden. Lokale Attribute und Elemente sind frei benennbar, während für Referenzen von globalen Deklarationen oder Definitionen (als Teil der refernzierten Deklarationen) weitere passende globale Komponenten in einer Dropdown-Liste als Vorschlag erscheinen. Eine Typänderung in einer

Referenz ist gleichzusetzen mit der Änderung des Typen der zugehörigen Deklaration und betrifft somit alle weiteren Referenzen. Das Kontextmenü bietet zusätzliche Editieroptionen wie das Setzen von vordefinierten Kardinalitäten und das Einfügen oder Löschen von Elementen an. Die aufgeführten Eigenschaften sind zudem der Properties-View änderbar. Das textuelle Editieren in der Source-View wird ebenfalls durch die Properties-View bzw. der Outline-View unterstützt. Nach einem Klick auf einen Tag im Dokumenteditor wechseln diese GUI-Komponenten zur entsprechenden Komponente, sodass entsprechende Details auch hier eingetragen werden können. Zusätzlich dazu vereinfacht eine kontextabhängige Autovervollständig von Komponentennamen und Attributnamen das rein textliche Bearbeiten von Schemadokumenten.

Im Großen und Ganzen bietet Eclipse WTP eine interessante Sicht auf die Typen eines Schemas. Wie beschrieben werden alle Typen, die mit dem zu visualisierenden Typ durch Vererbung oder Referenz in den Partikeln Verbindung stehen, ebenfalls dargestellt. Dadurch ist die Hierarchie der Typen sowie deren Nutzungsbeziehungen stellenweise sehr gut zu erkennen. Stellenweise deshalb, da dieses Vorgehen zwei Nachteile mit sich bringt. Zum einen muss wie erläutert zwischen den verschiedenen Komponenten gewechselt werden und zum anderen sind nur die Typen, die in direkter Beziehung stehen, involviert. Die Darstellung endet also jeweils nach einer Stufe der Vererbung bzw. Referenz und ist auch nicht auf Wunsch weiter aufklappbar o. Ä. Aufgrund dieser Tatsachen lässt sich die Typhierarchie nicht vollständig betrachten. Ebenso wenig ist die Dokumentstruktur ausschließlich anhand der Grafiken explorierbar. Obwohl die Navigation in der Dokumentstruktur dank der Outline-View problemlos möglich ist, wären außerdem Optionen zum Auffinden von Definitionen bzw. Deklarationen zu Referenzen und umgekehrt wünschenswert gewesen.

3.3. Konzeptionelle Modelle

Im vorigen Abschnitt wurden einige Werkzeuge vorgestellt, deren primäres Einsatzgebiet die Bearbeitung von XML-Schemata ist. Anschließend folgt die Präsentation von zwei Ansätzen, die unterschiedliche Wege in der Handhabung von XML-Schema wählen. Anstatt die XML-Syntax zu einem großen Teil lediglich durch Symbole darzustellen, abstrahieren diese Werkzeuge zusätzlich, sodass hauptsächlich die Konzepte der Anwendung im Vordergrund stehen.

3.3.1. hyperModel 3.6

Mit diesem von Carlson entwickelten Werkzeug ([Car06,Car12]) lassen sich u. a. UML-Modelle für Domänenkonzepte erzeugen, die in XML-Schema abgebildet wurden. Diese Methode verspricht somit eine hohe Abstraktion sowie ein gutes Verständnis von XML-Schemata. hyperModel ist als Plugin für die Entwicklungsumgebung Eclipse realisiert, wird aber auch als vorgefertigtes Package angeboten. Grundlage dieser Implementierung ist ein vollständiges Mapping von UML2 zur XML Schema Definition Language, wobei Carlson zur Darstellung ein ebenfalls selbst konzipiertes UML-Profil mit Stereotypen verwendet, die entsprechende XML-Schemakomponenten repräsentieren sollen.

Um ein XML-Schema von hyperModel anzeigen zu lassen, muss dieses zuvor in ein Projekt importiert werden. In dem Projekt-Explorer von Eclipse erscheint danach eine *.uml Datei, mit dem Namen, der bei dem Import ausgewählt wurde. Unterhalb dieser Datei sind alle globalen Bestandteile des importierten Schemas zu finden. Dabei ist es durchaus möglich, mehrere Schemata in ein einziges Projekt zu importieren, da in diesen verschiedene Schemadokumente als UML-Pakete aufgeführt werden. Mit Hilfe des Kontextmenüs lassen sich sich entweder die ausgewählten Bestandteile (Mehrfachmarkierung ist möglich) oder gesamte Schemata direkt öffnen.

Abbildung 3.15 zeigt die Standardansicht nach dem Öffnen eines kompletten Schemas. Diese ist eine detaillierte, tabellarische Übersicht über entsprechend ausgewählte UML-Komponenten

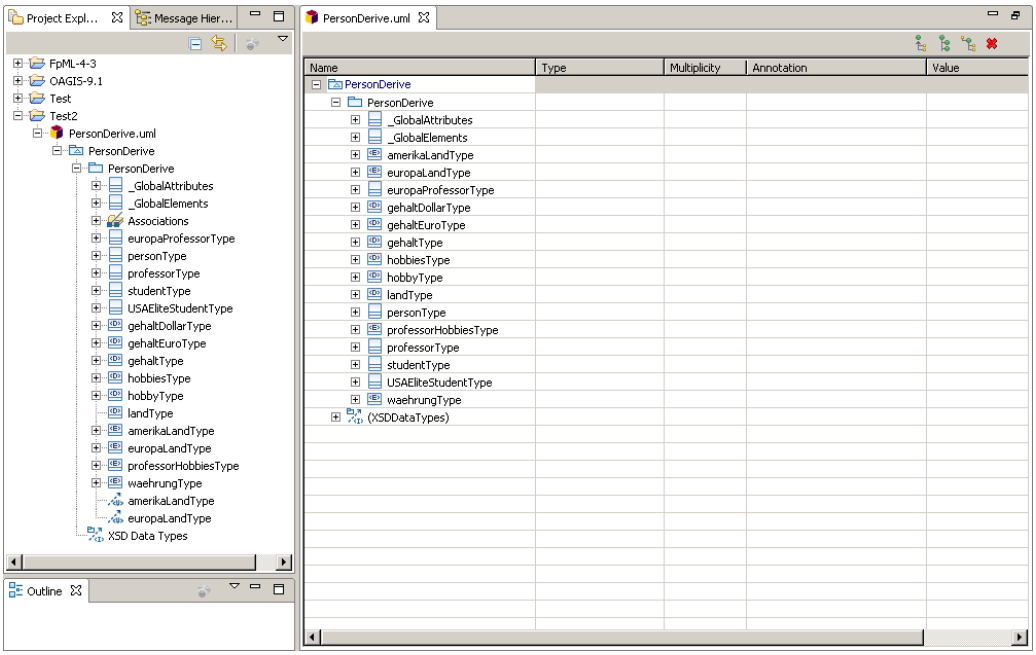


Abbildung 3.15.: Erste Ansicht der UML-Komponenten in hyperModel. Diese sind links im Project-Explorer permanent sichtbar, während die Tabellenübersicht zusätzliche Details angibt.

bzw. des gesamten Schemas. Das eigentliche Visualisieren der Auswahl ist über den Unterpunkt „Class Dynagram“ im Untermenü des Kontextmenüeintrags „Open with"möglich. Nach der Auswahl dieser Option wird ein neuer Reiter erzeugt, der entsprechende Komponenten in gewohnten UML-Klassendiagrammen visualisiert. Aus Platzgründen sei an dieser Stelle auf Abbildung A.2 im Anhang für eine vollständige Darstellung des Testschemas in hyperModel verwiesen. Sowohl die Visualisierung mittels Klassendiagrammen als auch der Projekt-Explorer bzw. die Tabellenübersicht zeigen, dass die gewählte Repräsentation Typ-zentrisch ist. Diese drei Sichten führen globale Typen nämlich als eigene Klassen auf, während globale Attribute und Elemente im Standardfall als Attribute der generierten Klassen „_Global Attributes“ und „_Global Elements“ erscheinen. Zudem ist es möglich, bei dem Import das Einfügen von globalen Elementen und Attributen verhindern, was die Darstellung allerdings kaum verändert. Die initiale Visualisierung zeigt hauptsächlich die komplexen Typen eines Schemas. Diese werden als UML-Klassen dargestellt und nach den Typnamen benannt. Interessanterweise trifft dies sowohl für globale als auch lokale komplexe Typen zu, da für lokale Typen einfach eine Klasse generiert wird, die den Namen des übergeordneten Elements trägt, wie in Abbildung 3.16 zu sehen ist.

Die Partikel eines Inhaltsmodells finden als Attribute dieser Klassen Verwendung, wobei die XML-Schemaattribute zusätzlich mit einem Stereotyp markiert sind. Liegen die Partikel in anderen Kardinalitäten außer „1..1“ vor, so wird dies durch eine passende UML-Multiplizität hinter dem Attribut notiert. Die Behandlung der Kompositoren variiert je nach deren Art. Besteht ein Inhaltsmodell aus einer einzigen <sequence>-Komponente, werden keine weiteren Schritte für die Visualisierung unternommen. Handelt es sich hingegen um <choice>- und <all>-Kompositoren wird über dem Klassennamen ein entsprechend benannter Stereotyp platziert. Die Darstellung von tiefer verschachtelten Kompositoren unterliegt ebenfalls speziellen Regeln. Für jeden verschachtelten Kompositor wird eine neue Klasse generiert, wobei eine Nummer im Klassennamen Aufschluss auf die Position des Kompositors im enthaltenden Inhaltsmodell gibt. Zusätzlich dazu wird eine solche Klasse mit einem «group, x»-Stereotyp versehen, wobei x den Namen des Kompositors wiedergibt. Zu guter Letzt stehen Klassen von komplexen Typen mit entsprechenden

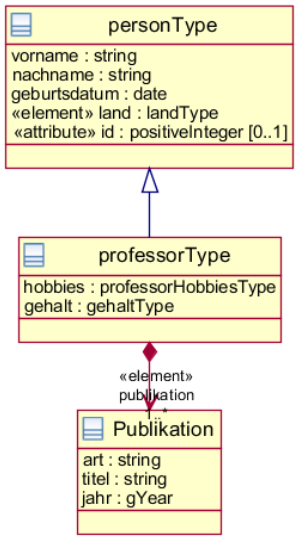


Abbildung 3.16.: UML-Diagramme für globale und lokale komplexe Typen.

Klassen von verschachtelten Kompositoren in einer Kompositionsbeziehung. Dieser Sachverhalt ist in Abbildung 3.17 dargestellt.

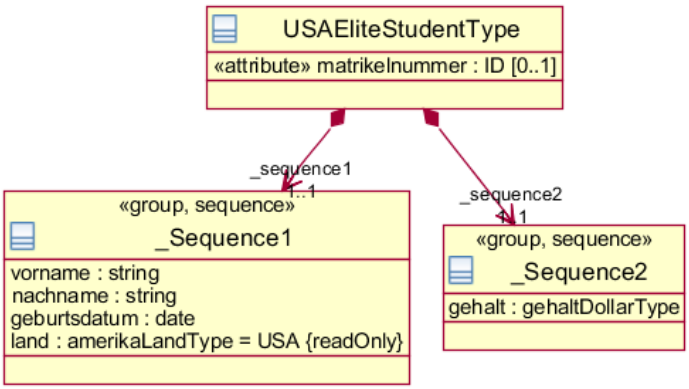


Abbildung 3.17.: Aufspaltung von verschachtelten Kompositoren eines Inhaltsmodells in eigene Klassen.

Bei einer größeren Anzahl von Nestungen wird auf eine Klasse nur noch verwiesen und weitere Pfade bzw. Beziehungen werden abgeschnitten. Symbolisiert wird solch eine Referenz anhand einer gestrichelten Umrandung und blauen statt gelben Einfärbung. Außerdem ist lediglich der Klassenname aufgeführt. Durch einen Doppelklick auf die entsprechende Klasse bzw. Referenz lassen sich aber die weiteren Informationen über die referenzierte Klasse sowie tiefere Beziehungen einholen. Um zurück zur vorherigen Visualisierung zu gelangen, bietet Class Dynagram eine Historie an. Außerdem kann die Referenz über den Kontextmenüeintrag „Expand Node“ auch an Ort und Stelle aufgelöst werden. Angezeigte Klassen lassen sich wieder über den Punkt „Collapse Node“ einklappen.

Typableitungen werden i. A. durch die UML-Generalisierung repräsentiert, was Abbildung 3.18 zeigt. Für komplexe Typen wird die Ableitung durch Erweiterung als Standardfall angenommen, da in diesem Fall keine weiteren Stereotype Anwendung finden. Entsprechend der Logik von UML werden in der Klasse, die den abgeleiteten Typ darstellt, auch nur die neu hinzugekommenen Elemente oder Attribute aufgelistet. Anders sieht es bei der Ableitung durch Einschränkung

aus. Der Vererbungspfeil wird mit einem «restriction»-Stereotyp annotiert und das komplette Inhaltsmodell erscheint als Attribute oder weiteren Klassen. Beide Methoden entsprechen also prinzipiell durchaus der Notation von abgeleiteten Inhaltsmodellen in XML-Schema selbst. Durch die Verwendung von verbundenen Diagrammen ist hier dennoch das Verstehen einer Typhierarchie vereinfacht.

Die Anzeige möglicher Veränderungen am Inhaltsmodell bzw. an Partikeln selbst ist bei der Ableitung durch Einschränkung allerdings nicht optimal. Wertefixierung werden durch eine entsprechende Zuweisung innerhalb des UML-Attributs und einer readOnly-Constraint deutlich. Verbotene Elemente werden hingegen erst gar nicht aufgeführt, was zwar korrekt, optisch aber nicht unbedingt direkt ersichtlich ist. Veränderungen des Typen von Partikeln sind außer mittels manuellem Vergleich ebenfalls nicht erkennbar. Zudem wird ein Verbot eines Attributs mittels der „use“-Eigenschaft in eine nicht korrekte Multiplizität übertragen.

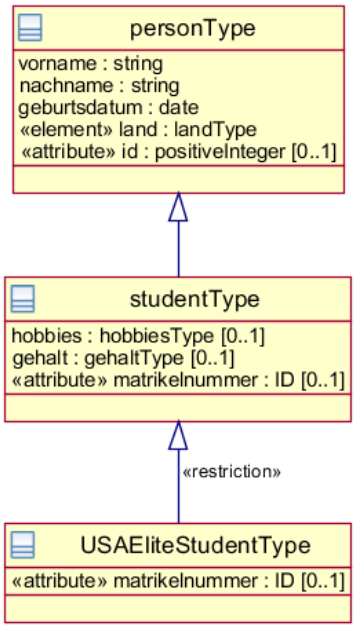


Abbildung 3.18.: Darstellung der Erweiterung und Einschränkung komplexer Typen durch die UML-Generalisierung.

Etwas umständlich ist die Handhabung von einfachen Typen bzw. komplexen Typen mit einfachem Inhalt. In der initialen Visualisierung eines Schemas sind Referenzen von diesen zwar als Typangabe in den Klassendiagramm zu finden, ihre Definitionen selbst sind aber nicht weiter visualisiert. Dennoch lassen sich auch diese mittels Kontextmenüaufruf im Project-Explorer bzw. in der UML-Tabelleübersicht in Class Dynagram darstellen. Wünschenswert wäre hier die Möglichkeit, dies direkt im Diagramm selbst zu erledigen, da Referenzen von einfachen Typen so jedes mal selbst aufzusuchen sind. Offenbar scheinen noch einige Probleme bei der Visualisierung zu existieren, da etwa <enumeration>-Facetten nicht dargestellt werden, wohl aber im Project-Explorer als Enumerations-Literale auftauchen.

Im Allgemeinen werden Facetten als stereotypisierte Attribute aufgeführt. Klassen, die durch Einschränkung abgeleitete einfache Typen darstellen, stehen mit dem Basistyp in einer Vererbungsbeziehung, wobei auf built-in Datentypen nach der oben beschriebenen Methode verwiesen wird. Wenn es sich um einen Listen- oder Vereinigungs-Typkonstruktor handelt, ist dies in entsprechenden Klassen mit einem Stereotyp vermerkt. Der Basistyp eines Listentypen erscheint allerdings nicht grafisch, da dieser nur als Datentyp des entsprechenden UML-Attributs „itemType“ erwähnt wird. Auch werden die Mitgliedstypen einer Vereinigung ebenfalls nicht angezeigt,

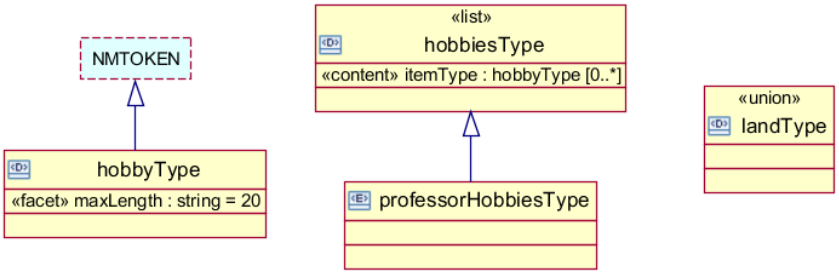


Abbildung 3.19.: Darstellung der Einschränkung, Listenbildung und Vereinigung einfacher Typen. Die Beziehung zum Basistyp der letzteren Ableitungsformen wird grafisch durch Attribute (Listenbildung) oder gar nicht (Vereinigung) angezeigt.

sodass lediglich ein „nacktes“ UML-Diagramm ohne jegliche Attribute für den Vereinigungs-Konstruktor verwendet wird. Die Darstellung der verschiedenen Ableitungsformen sind in Abbildung 3.19 wiedergegeben. Im Standardfall wird die Vererbungshierarchie des ausgewählten Typen nur in Richtung des Vorgängers verfolgt, durch einen Klick auf „Show Subclasses“ im Kontextmenü lassen sich dennoch alle Typen erkennen, die aus dem gegebenen Typ entstehen. Damit ist die vollständige Typhierarchie von einfachen und komplexen Typen erkennbar.

Enthält ein Inhaltsmodell eine Referenz auf eine Deklaration, so wird jene Referenz nicht direkt als Attribut der entsprechenden Klasse dargestellt. Stattdessen wird diese Klasse mit derjenigen Klasse durch eine Komposition verbunden, die für den Typ der Deklaration steht. Anschließend wird der Name der Referenz bzw. der Deklaration an die Pfeilspitze mitsamt der Kardinalität der Referenz angegeben. Referenzen von globalen Elementen werden ähnlich gehandhabt, mit dem Unterschied, dass statt einer Komposition eine einfache Assoziation verwendet und mit einem «globalElement»-Stereotyp versehen wird. Für referenzierte globale Attribute wird diese Methode nicht verfolgt, da entsprechende Attributreferenzen wie üblich als einfache (stereotypisierte) UML-Attribute einer Klasse auftreten. Abbildung 3.20 stellt dieses Vorgehen in hyperModel dar.

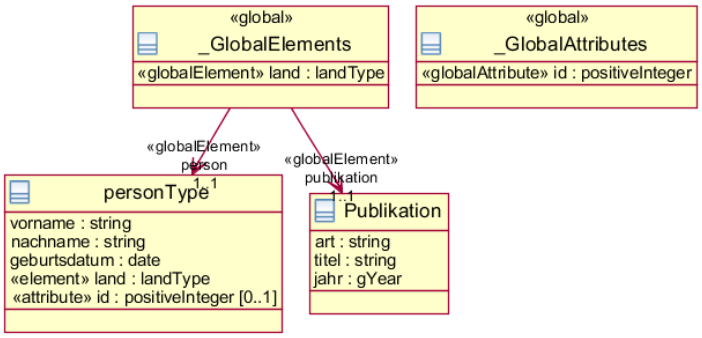


Abbildung 3.20.: Visualisierung von globalen Deklarationen sowie deren Nutzung. Im Gegensatz zu Elementen werden bei Attributreferenzen die beteiligten Klassendiagramme nicht verbunden.

Die Editiermöglichkeiten innerhalb der UML-Diagrammdarstellung ist eingeschränkt, da lediglich Ausschneide-, Kopier-, Einfüge-, und Löschfunktion im Kontextmenü angeboten werden. Zudem scheinen der Kopier-Anteil des Ausschneidens und das Kopieren bzw. Einfügen effektlos zu sein. Mehr Möglichkeiten stehen in der UML-Tabellenübersicht bzw. der Project-Explorer bereit. Zu bestehenden Strukturen können u. a. grundlegende XML-Schemakomponenten wie Typen, Elemente, Attribute oder Kompositoren hinzugefügt werden. Das Programm übersetzt diese Optionen automatisch in entsprechende UML-Komponenten. Weitere Aspekte wie Refe-

renzen und Ableitungen bzw. Vererbungen sind direkt über entsprechende UML-Komponenten realisierbar. Für das Setzen von Typen, Kardinalitäten oder Werten lassen sich die jeweiligen Spalten der Übersicht manipulieren. Bedingt durch die Tatsache, dass es sich um keinen Schemaeditor handelt, ist die (exakte) Unterstützung in diesem Bereich nicht so stark ausgeprägt wie in den bisher untersuchten Werkzeugen.

Die Ermittlung von Fehlern muss manuell im Project-Explorer oder in der Tabellenübersicht ausgelöst werden. Anschließend können diese in der Problems-View von Eclipse entnommen werden, wobei betroffene Komponenten auch in den erwähnten Sichten durch eine Markierung hervorgehoben sind. In der Visualisierung werden etwa fehlende Typen durch eine referenzierte Klasse dargestellt, die allerdings nicht aufklappbar ist. Eine Option für die Kompensation von fehlenden Typen innerhalb der Typhierarchie ist nicht vorhanden.

hyperModel stellt sich als ein sehr hilfreiches Werkzeug für die Visualisierung von XML-Schemata heraus. Die Syntax von XML ist dank der Nutzung von UML-Klassendiagrammen quasi nicht mehr zu erkennen, sodass tatsächlich die Domänenkonzepte im Vordergrund stehen. Darüber hinaus bietet die Darstellung in Class Dynagram eine gewisse Dynamik, mit der die Gesamtheit oder auch nur Teile einer Domäne betrachtet werden können. Das Fokussieren der Typhierarchien eines Schemas ist eine gelungene Herangehensweise an die Abstraktion, wobei auch die Anzeige von einfachen Typen in der initialen Ansicht wünschenswert wäre. Ebenfalls wäre eine bessere Darstellung von geänderten Eigenschaften in abgeleiteten Typen hilfreich. Zudem stehen für die Modellierung einer Anwendung einige Mittel von UML zur Verfügung, die allerdings ein entsprechendes Wissen im Bereich der UML-Modellierung voraussetzen.

3.3.2. CodeX 2

Der Prototyp CodeX wird an der Universität Rostock von Nösinger et al. ([NKH12]) entwickelt. Die Aufgabe dieses Editors dient primär nicht der XML-Schemamodellierung, sondern der Evolution von XML-Schemata (siehe 2.4). Schemakomponenten lassen sich im Garden of Eden Stil am einfachsten lokalisieren, was sich positiv auf die Evolution auswirkt [NKH13]. Deshalb haben die Autoren für CodeX einen Weg gewählt, in welchem XML-Schemata nur im entsprechenden Modellierungsstil erzeugt bzw. bearbeitet werden können. Importierte XML-Schemata werden dabei automatisch in den Garden of Eden Stil überführt. Als weitere Einschränkung sind nur Typen mit „flachem“ Inhaltsmodell, d. h. ohne die Schachtelung von Kompositoren erlaubt. Aus diesem Grund wurde das Testschema an diese Bedingungen angepasst. Die Bearbeitung selbst geschieht nicht auf Schemaebene sondern auf dem konzeptionellen Modell EMX (Entity Model for XML-Schema), welches XML-Schemakomponenten auf eindeutige Entitäten abbildet. Entwurfsschritte auf den Entitäten werden mit ELaX-Ausdrücken (Evolution Language for XML-Schema) dargestellt und für die spätere Evolution geloggt. Die EMX-Entitäten lassen sich dabei in zwei Kategorien einteilen, den sichtbaren und nicht sichtbaren Entitäten. Die sichtbaren Entitäten sind im Wesentlichen Element- und Attributgruppenreferenzen, Schlüsselbedingungen sowie Module, die in Beziehung miteinander stehen. Für den Nutzer ergibt sich dadurch bereits eine erste Ebene der Abstraktion, da die nicht sichtbaren Entitäten ausgelagert sind. Deren Modifikation geschieht innerhalb Formular-artiger Dialoge, welche über der Werkzeugleiste des Editors oder per Kontextmenü-Einträge von sichtbaren Entitäten aufgerufen werden können.

Dies betrifft auch die einfachen und komplexen Typen. Während erstere nur als Eigenschaft von Elementen und Attributen vorliegen, sind letztere implizit durch die Darstellung ihres Kompositors sowie dessen Elementreferenzen und Attributgruppenreferenzen visualisiert. Damit ergibt sich eine Element-zentrische Darstellungsform. Abbildung 3.21 zeigt die Hauptansicht des Editors mit den Typen „personType“, „professorType“ und „studentType“. Trotz der Element-zentrischen Ausrichtung sind Deklarationen unsichtbar und wie beschrieben nur ihre Referenzen visualisiert. Dieses Vorgehen bietet sich an, da im Garden of Eden Stil alle Deklarationen global vorliegen, sodass die Unterscheidung zwischen lokalen und globalen Deklarationen in der Visua-

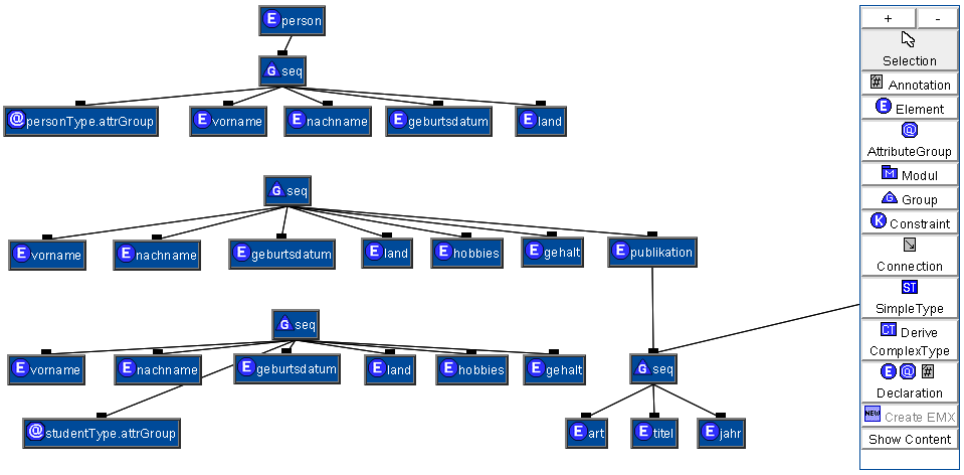


Abbildung 3.21.: Visualisierung von Komponentenreferenzen im Arbeitsbereich. An der rechten Seite befindet sich die Werkzeugleiste zur Modellierung.

lisierung nicht mehr nötig ist. Vorteil dieses Ansatzes ist, dass sich mit einem Blick die komplette Struktur von möglichen XML-Dokumenten überschauen lässt. Dies resultiert aus der Designentscheidung, Inhaltsmodelle der durch Deklarationen referenzierten komplexen Typen direkt an den entsprechenden Elementreferenzen anzuzeigen. In Abbildung 3.21 wird z. B. „publikation.Type“ direkt unter dem Element „publikation“ angezeigt. Somit stehen die Referenzbeziehungen im Vordergrund, was den manuellen Wechsel zwischen den Komponenten zur Erfassung der gesamten Struktur wie z. B. in Eclipse WTP oder auch XMLSpy unnötig macht.

Auffällig dabei ist jedoch, dass die Darstellung in CodeX neben der Referenzbeziehung relativ wenig Informationen preisgibt. Außer dem Namen einer (referenzierten) Elementdeklaration sowie Attributgruppendeklaration bieten die Diagrammbausteine selbst keine weiteren Angaben. Dieser Aspekt ähnelt der Graph-View des XSD-Designers in Visual Studio. Schlüsselinformationen wie die Kardinalität, Wertefixierung oder der Defaultwert und Typnamen müssen per Kontextmenü abgerufen werden. In der Abbildung 3.22 sind etwa Einschränkungen von „USAE-

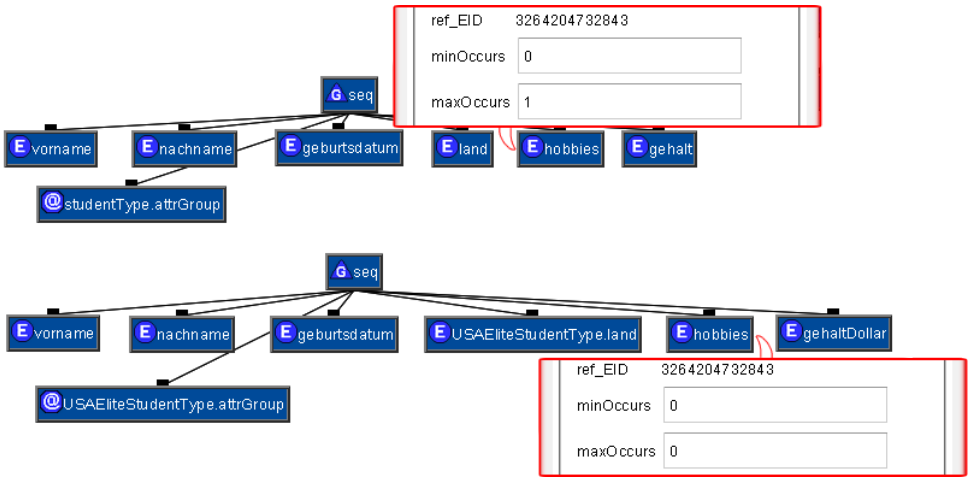


Abbildung 3.22.: Darstellung der Inhaltsmodelle von komplexen Typen. Partikeleigenschaften sind nicht an den Diagrammbausteinen vermerkt.

liteStudentType“ im Vergleich zu „studentType“ nur durch einen Aufruf der entsprechenden Dialoge über die Elementreferenzen zu erkennen (ein vollständiges Dialogfenster lässt sich in

Abbildung 3.23 betrachten). Die Unterscheidung der Art des Typs ist durch das Vorhandensein von Unterkomponenten im Diagramm gegeben. Die Zuordnung von Attributen zu komplexen Typen geschieht in CodeX über Attributgruppen. In der Werkzeugleiste ist eine Liste abrufbar, welche alle Attributdeklarationen des Schemas erfasst (auch für einfache Elementdeklarationen möglich). Um jedoch herauszufinden, welche Attribute ein komplexer Typ letzten Endes referenziert, müssen seine Attributgruppen per Kontextmenüeinträge und Dialogfenster geöffnet werden. In diesen Dialogfenstern sind entsprechende Attributreferenzen mit ihren Eigenschaften einsehbar. Zudem ist es an allen Dialogenfenstern von referenzierenden Komponenten möglich, auf die darunterliegende Deklaration zu gelangen (siehe ebenfalls Abbildung 3.23). Umgekehrt sind alle Referenzen einer Deklaration nur als Inhaltsmodellteile in der Hauptansicht erkennbar. Da durch den internen Garden of Eden Stil keine lokalen Deklarationen vorliegen, ist die Eindeutigkeit gesichert. Um Details über verschiedene Komponenten zu erhalten, muss ein Nutzer i. A. in mehreren Dialog-Ebenen navigieren. Die Nutzbarkeit ist in diesem Punkt etwas eingeschränkt.

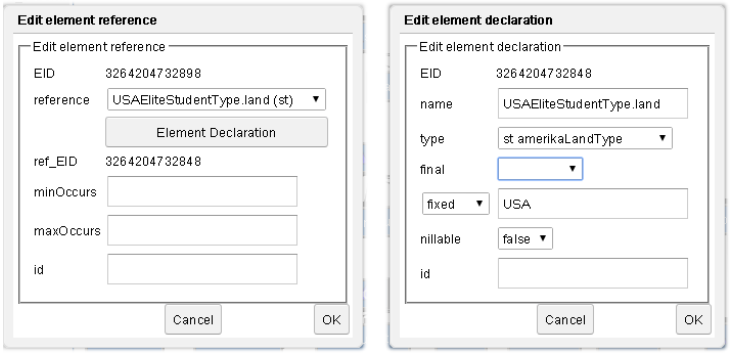


Abbildung 3.23.: Weiterführende Informationen über Referenzen sowie Deklarationen in separaten Dialogen.

Da die (implizite) Darstellung komplexer Typen an das Vorhandensein von Typreferenzen durch Deklarationen gekoppelt ist, hat diese Methode jedoch den Nachteil, dass die Hierarchie jener Typen nicht erkennbar ist. Dadurch ist auf konzeptioneller Ebene die Ableitungsart durch Erweiterung oder Einschränkung nicht zu erfassen. Um diese zu ermitteln, müsste ein Nutzer manuell in den Dialogfeldern der Kompositoren nach entsprechenden Angaben suchen, was relativ mühsam ist. Dieses ist nötig, da in der aktuellen Version die Ableitung von komplexen Typen zwar im Modell vorgesehen ist, jedoch nicht vollständig implementiert wurde. Das heißt, dass CodeX momentan keine Ableitungen von komplexen Typen unterstützt. Für einfache Typen sind jedoch alle Ableitungsformen auch in der Implementierung umgesetzt. Dennoch lässt sich die Beziehung zwischen zwei einfachen Typen beschwerlich erkennen, da diese nur als Übersicht in der Werkzeugleiste vorliegen oder wie oben erwähnt in einem Referenzbaustein abrufbar sind. Angaben über Facetten o. Ä. sind dann jeweils in den Formular-artigen Dialogen möglich. Möchte ein Nutzer zwischen zwei Typen wechseln, sind einige Klicks in der Oberfläche nötig. Positiv ist die Tatsache, dass Facetten der vorhergehenden Hierarchie direkt im entsprechenden Dialog angezeigt werden, was dem Verständnis über den Wertebereich der einfachen Typen zugute kommt. Abbildung 3.24 zeigt dies am Beispiel „professorHobbiesType“.

Obwohl die Dialogfenster-Lastigkeit sich etwas ungünstig auf die Navigation zwischen den Komponenten im Editor auswirkt, zeigt gerade dieser Punkt die Stärke dieses Vorgehens. Die Editiermöglichkeiten sind in CodeX sehr ausgeprägt. Mit der Verwendung von Dialogen wird der Nutzer zudem bei der XML-Schemamodellierung einerseits passiv unterstützt, da er jeweils nur vorgefertigte Formulare auszufüllen braucht, in welchen oftmals aus vorgefertigten, gültigen Werten ausgewählt werden kann. Es entsteht somit nicht die Situation, in der z. B. eine XML-Schemakomponente oder ein vordefiniertes Attribut an einer falschen Stelle verwendet wird.

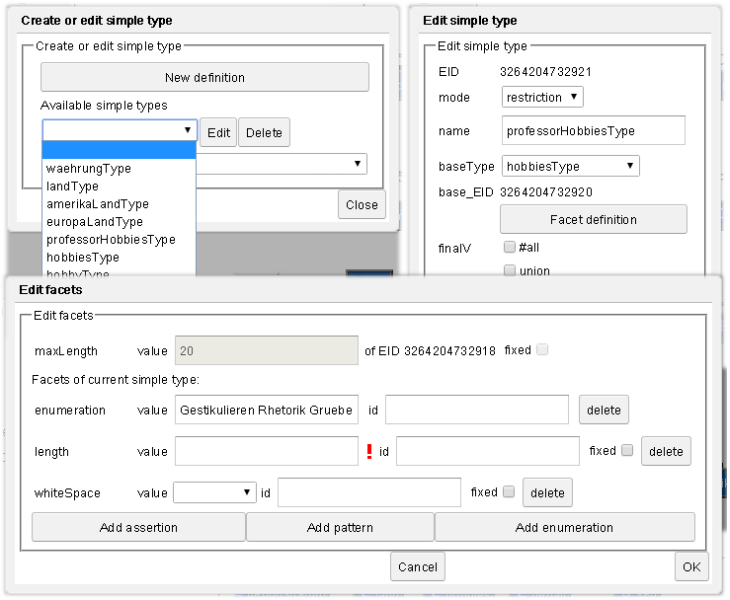


Abbildung 3.24.: Mehrere Dialogebenen zur Bearbeitung von einfachen Typen. Hier und in anderen Dialogen werden ungültige Angaben hervorgehoben.

Gerade für die Bearbeitung bzw. Eingabe von Zusicherungen und Schlüsselbedingungen bietet der Editor eine umfangreiche Unterstützung. Dadurch und durch die generell abstraktere Darstellung rückt XML-Syntax in den Hintergrund. Andererseits erfolgt eine aktive Unterstützung bereits während der Modellierung. Bevor die Änderungen innerhalb von Dialogen umgesetzt werden, überprüft CodeX nach dem Betätigen des Bestätigungsknopfs viele Angaben auf Konsistenz mit bestehenden Angaben im Schema. Fehler und fehlende, erforderliche Werte werden so direkt angezeigt, wie in Abbildung 3.24 und 3.25 zu sehen. CodeX erlaubt die Umsetzung fehlerhafter Operationen nicht. Bricht der Nutzer den Dialog daraufhin ab, wird die entsprechende Operation also erst gar nicht umgesetzt, sodass keine Fehlerbehebung erforderlich ist. Dennoch bietet CodeX derzeit wie die anderen Ansätze keine Möglichkeit, bei der Löschung von Typen einen Ausgleich für referenzierende Komponenten zu finden.

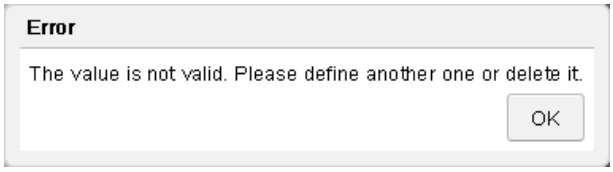


Abbildung 3.25.: Anzeige eines Hinweis-Dialogs bei ungültigen Eingaben.

CodeX wurde primär für die XML-Schemaevolution konzipiert. Trotzdem bietet dieser Prototyp umfangreiche Modellierungsfunktionalitäten sowie ein gutes Verständnis über die resultierende Dokumentstruktur, wobei Details in der Hauptansicht verborgen bleiben. Die Unterstützung für den Nutzer durch den Fokus auf Dialoge geht einher mit einer etwas umständlichen Navigation. Mögliche Fehler während der Bearbeitung sind sofort erkennbar. Im Mittelpunkt stehen die Referenzbeziehungen zwischen den einzelnen XML-Schemakomponenten, was das Erkennen der Typhierarchien jedoch erschwert. Da gerade die Typen sowie deren Beziehungen die Konzepte einer Domäne zusammenfassen, wäre eine Möglichkeit zur entsprechenden Erkennung eine vorteilhafte Ergänzung für diesen Prototyp.

3.4. Vergleich der Ansätze

Nachdem die verschiedenen Werkzeuge für die XML-Schemamodellierung im Detail präsentiert wurden, folgt nun ein Vergleich zwischen diesen, der die Editoren ähnlich wie in [MW06] in die Tabellen 3.2 und 3.3 einteilt. Als Vergleichskriterien dienen erneut die Fragestellungen aus Abschnitt 3.1, die in den jeweiligen Unterabschnitten zu den Werkzeugen beantwortet wurden und hier als Zusammengefassung erscheinen.

Allen untersuchten XSD-Editoren gemein ist der Umstand, dass sich die grafische Darstellung eines XML-Schemas relativ stark an der XML-Syntax orientiert, während die konzeptionellen Editoren hyperModel und CodeX in diesem Punkt besser abschneiden. Lediglich der XML Schema Designer von Visual Studio und teilweise der XML-Editor von Eclipse WTP bieten Ansichten innerhalb eines XML-Schema-Editors, die dem Nutzer eine etwas höhere Abstraktion des Schemas liefert. Im Gegensatz zu den restlichen Editoren gewährt besonders die Abstraktion des XML Schema Designers einen guten Einblick in die bestehenden Typhierarchien. Dies trifft auch auf hyperModel für die reine konzeptionelle Modellierung zu. Der Fokus auf die (komplexen) Typen in Kombination mit verschiedenen Möglichkeiten, die dargestellten Mitglieder einer Hierarchie aus oder einzublenden, sorgt für ein schnelles Erfassen der abgebildeten Konzepte.

Alle Werkzeuge zeigen syntaktische Fehler an. Vor allem in CodeX lassen sich viele Fehler durch eine präventive Kontrolle vermeiden. Eine Kompensation von gelöschten Typen für referenzierende Bestandteile erreicht allerdings keiner der verglichenen Ansätze. Gerade dieser Aspekt bedarf also einer gründlichen Auseinandersetzung in der weiteren Arbeit.

	XMLSpy 2014	**XSD-Designer**	**JDeveloper 12c**
Perspektive	Element-zentrisch	kein besonderer Fokus	kein besonderer Fokus
Darstellung	tabellarische Auflistung globaler Komponenten in Arbeitsfläche und Seitenleiste, Diagramme für Elemente und komplexe Typen mit manuellem Wechsel, alternative Ansichten (Dokumenteditor und Baumdarstellung)	Auflistung globaler Komponenten in Seitenleiste, Graph-Ansicht für Komponentenbeziehungen, unverbundene Diagramme für Inhaltsmodelle, Dokumenteditor	Darstellung globaler Komponenten in Diagrammen ohne Wechsel und in Seitenleiste, Dokumenteditor
Abstraktion	schwach, nahe an der XML-Syntax	sehr hoch in der Graph-Ansicht, schwach / Syntax-nahe sonst	schwach, nahe an der XML-Syntax
Typen	keine grafische Darstellung von einfachen Typen, Umrandung komplexer Typen bei Referenz, Kerneigenschaften in Diagrammen angezeigt, weitere in Seitenleiste (auch von einfachen Typen)	grafische Darstellung einfacher (ohne Details) und komplexer Typen, Kerneigenschaften in Diagramm-Ansicht angezeigt, weitere in Seitenleiste	grafische Darstellung einfacher (mit Facetten) und komplexer Typen, Umrandung von Typen bei Referenz, Kerneigenschaften in Diagramm-Ansicht angezeigt, weitere in Seitenleiste
Typhierarchien	textuelle Kennzeichnung der Ableitungsart, Darstellung des Basistyp-Inhaltsmodells, blaue Umrandung veränderter und Durchstreichen verbotener Bestandteile, keine Darstellung der Beziehungen zw. Typen	Kennzeichnung der Ableitungsart nur in Diagramm-Ansicht und Seitenleiste, Darstellung des tatsächlichen Inhaltsmodells ohne Hervorhebung des Basistyp-Anteils, veränderte Bestandteile nicht hervorgehoben, verbotene Bestandteile durchgestrichen (Seitenleiste), Beziehungen zw. Typen in Graph-Ansicht, Auffinden von Ableitungen und Vorgängern eines Typen in Seitenleiste	textuelle Kennzeichnung der Ableitungsart, Darstellung des Basistyp-Inhaltsmodells, Markierung verbotener Bestandteile, Markierung veränderter Bestandteile nur in Seitenleiste, keine Darstellung der Beziehungen zw. Typen
Inhaltsmodelle	genestete Darstellung, vollständig einsehbar durch Aufklappen	genestete Darstellung, vollständig einsehbar durch Aufklappen	genestete Darstellung, vollständig einsehbar durch Aufklappen
Referenzen	grafisch aufgelöst, Auffinden von Definitionen bzw. weiteren Referenzen möglich	grafisch aufgelöst, Auffinden von Definitionen bzw. weiteren Referenzen in Seitenleiste möglich	grafisch aufgelöst, Auffinden von Definitionen möglich, Auffinden von weiteren Referenzen nicht möglich
Editierbarkeit	sehr gut, einige Aspekte direkt in den Diagrammen manipulierbar, Anzeige von Fehlern, keine Kompensation	schwach, ausschließlich im Dokumenteditor möglich, Anzeige von Fehlern, keine Kompensation	sehr gut, Drag and Drop möglich, einige Aspekte direkt in den Diagrammen manipulierbar, Anzeige von Fehlern, keine Kompensation

Tabelle 3.2.: Vergleich der Ansätze Teil 1

	Eclipse WTP 3.5.2	**hyperModel 3.6**	**CodeX 2**
Perspektive	Typ-zentrisch	Typ-zentrisch	Element-zentrisch
Darstellung	Übersicht über globale Komponenten in Arbeitsfläche und Seitenleiste, Diagramme für globale Komponenten mit manuellem Wechsel, Dokumenteditor	Auflistung globaler Komponenten in Seitenleiste und UML-Tabelle, stereotypisierte UML-Klassendiagramme für globale und lokale Komponenten, manueller Wechsel nur bei einfachen Typen nötig	automatische Überführung in Garden of Eden, Visualisierung der Komponentenreferenzen in Bausteine, Auflistung globaler Deklarationen und Definitionen in Werkzeugleiste
Abstraktion	relativ hoch, XML-Syntax zu Beziehungen abstrahiert	sehr hoch, XML-Syntax nicht mehr erkennbar	relativ hoch, XML-Syntaxdetails nicht vorhanden, Dokumentstruktur erkennbar
Typen	grafische Darstellung einfacher (ohne Details) und komplexer Typen, Kerneigenschaften in Diagrammen angezeigt, weitere in Seitenleiste (umständlich angeordnet)	grafische Darstellung einfacher (teilweise fehlende Facetten) und komplexer Typen, Kerneigenschaften in Diagrammen angezeigt	implizit in der Hauptansicht für komplexe Typen, Eigenschaften dieser in Dialogen, einfache Typen nicht visualisiert (Eigenschaften ausschließlich in Dialogen)
Typhierarchien	Kennzeichnung der Ableitungsart nur in Seitenleiste, Darstellung des Basistyp-Inhaltsmodells in separatem Diagramm, veränderte und verbotene Bestandteile nicht hervorgehoben, Darstellung der Beziehungen zw. Typen über eine Ebene	Kennzeichnung der Ableitungsart durch Stereotyp, Darstellung des Basistyp-Inhaltsmodells in separatem Diagramm (Generalisierung), veränderte und verbotene Bestandteile nicht hervorgehoben, Darstellung der Beziehungen zw. Typen über beliebige Ebenen (auf Wunsch), Auffinden von Ableitungen eines Typs	Ableitungen für komplexe Typen nicht vollständig, für einfache Typen Ableitungsart in Dialog erkennbar, Darstellung aller Facetten der vorhergehenden Hierarchie
Inhaltsmodelle	Darstellung verbundener Diagramme ohne Aufklappen, nicht vollständig einsehbar	Darstellung per Komposition verbundener, aufklappbarer Diagramme, Aufspaltung von Kompositoren in eigene Diagramme, vollständig einsehbar	vollständig in der Hauptansicht einsehbar, keine Unterstützung von verschachtelten Kompositoren
Referenzen	grafisch aufgelöst über eine Ebene, Auffinden von Definitionen bzw. weiteren Referenzen nicht möglich	grafisch aufgelöst über beliebige Ebenen, Auffinden von Definitionen bzw. weiteren Referenzen durch manuelles Verfolgen der Beziehungen möglich	grafisch komplett aufgelöst, Auffinden von Deklarationen und Definitionen möglich, alle (Element-)Referenzen dargestellt (manuelles Auffinden), Auslagerung von Attributen in Attributgruppen (Attributreferenzen grafisch nicht erkennbar)
Editierbarkeit	gut, einige Aspekte direkt in den Diagrammen manipulierbar, Anzeige von Fehlern, keine Kompensation	schwach in der Diagramm-Ansicht, gut in der UML-Tabelle bzw. Seitenleiste, Anzeige von Fehlern, keine Kompensation	sehr gut, Anzeige von Fehlern vor Durchführung von Änderungen, keine Kompensation

Tabelle 3.3.: Vergleich der Ansätze Teil 2

4. Konzeption

In diesem Kapitel erfolgt die Überlegung zu einer weiteren Darstellung von Typhierarchien in XML-Schemata sowie möglichen Vorgehen als Konsequenz der Entfernung von Typen aus diesen Hierarchien. Dabei werden einfache und komplexe Typen gesondert behandelt.

4.1. Visualisierung

Im Folgenden werden Anforderungen an die gewünschte Visualisierung genannt und verschiedene Optionen zu deren Erfüllung aufgezeigt, wobei sich auch auf die untersuchten Ansätze bezogen wird. Gleichzeitig wird erörtert, welche Vor- und Nachteile diese Optionen in verschiedenen Aspekten besitzen. Als grober Rahmen dienen hierbei erneut einige Fragestellungen aus der Liste in Kapitel 3.1, wobei nicht streng nach den aufgeführten Kategorien vorgegangen wird.

4.1.1. Perspektive

Das Hauptaugenmerk der angestrebten Visualisierung sollen die Typen bzw. Typhierarchien sein, sodass eine Typ-zentrische Perspektive in der Darstellung vorteilhaft ist. Sowohl komplexe als auch insbesondere einfache Typen sind grafisch zu repräsentieren. Dementsprechend soll der Fokus ausdrücklich nicht auf den Elementen oder Attributen liegen. Das bedeut nicht, dass jene Komponenten gar keine Beachtung finden, denn als Teil von Inhaltsmodellen sollen diese auch in der zu entwickelnden Visualisierung erscheinen. Auf eine explizite Darstellung globaler Elemente etwa wird jedoch verzichtet. In diesem Sinne wird die Darstellungsform zwar als eigenständige Ansicht eines Schemas konzipiert, durch ihren spezialisierten Charakter ist es jedoch denkbar, sie auch in bestehende Ansätze als weitere Sicht umzusetzen. Diese Option bietet sich an, da z. B. das Wechseln der Ansicht von Typen und Elementen nicht Bestandteil des Umfangs der zu entwickelnden Darstellung sein soll.

4.1.2. Abstraktion

Das grundlegende Ziel ist eine möglichst hohe Abstraktion der XML-Syntax. Damit scheiden also Visualisierungen aus, wie sie die meisten grafischen XML-Schemaeditoren wie XMLSpy verfolgen, da sich diese eher an dem Schemadokument als an dem Schema selbst orientieren. Ein guter Ansatz ist die Graph-View des XSD-Designers von Visual Studio, in welchem die Beziehungen zwischen globalen Komponenten zu sehen sind. Im Unterschied zum XSD-Designer sollen allerdings ausschließlich die Beziehungen zwischen den Typen dargestellt werden. Bei größeren Schemata kann sich dies vorteilhaft auf die Übersicht auswirken, da so weniger Komponenten grafisch anzuzeigen und zu verbinden sind. Ein Nachteil dieser Ansicht des XSD-Designers ist, dass viele Informationen wegfallen, die für das Verständnis einer Schemakomponente bzw. eines Typs wichtig sind. Möchte ein Nutzer weitere Informationen über jene Komponente einholen, muss dieser zwangsweise in eine andere Ansicht des XSD-Designers wechseln. Demzufolge ist es erstrebenswert eine Form zu wählen, die zwar eine hohe Abstraktion bietet, gleichzeitig aber entscheidende Informationen preisgibt, ohne zusätzlich Klicks oder Wartezeiten zu benötigen. Dies sind durchaus gegenläufige Anforderungen, da ein höherer Detailgrad an dargestellten Informationen auch eine Erhöhung der nötigen Komplexität der Visualisierung bedeutet. Um Kompromisse in einer einzigen Darstellung zu umgehen deuten z. B. Michel und Wilde darauf hin, für verschiedene Aspekte verschiedene Sichten einzuführen. In dieser Hinsicht verfolgt der XSD-Designer mit seinen zwei verschiedenen Hauptansichten bereits dieses Konzept. Wie ausgeführt kann aber auch eine zu scharfe Trennung wiederum zu Mängeln in der Nutzbarkeit führen.

Diese Zusammenhänge sollen einerseits den „Sicht-Charakter“ (wie oben angerissen) der angestrebten Visualisierung unterstreichen. Um dennoch einen bedeutenden Ausschnitt eines Schemas zu gewährleisten, soll diese Sicht andererseits genügend Informationen enthalten. Dabei ist zu ermitteln, welche Informationen die wichtigen sind.

4.1.3. Darstellungsform

Um beim Beispiel des XSD-Designers zu bleiben, wäre es sinnvoll, z. B. die „abgekürzten“ Diagramme der Graph-View durch die vollständigen Diagramme der Content-Model-View zu ersetzen. Die Basis bei einem solchen Vorgehen wäre dabei immer noch die Darstellung der Beziehungen, die zusätzlich aber um gewisse Informationen der Komponente angereichert wird. Im Falle eines komplexen Typen bedeutet dies etwa das Anzeigen seines Inhaltsmodells.

Betrachtet man die Visualisierungen der Typ-zentrischen Ansätze Eclipse WTP und hyperModel unter diesem Aspekt, so realisieren beide Werkzeuge diesen Vorschlag bereits auf gewisse Weise. Eclipse WTP stellt die Inhaltsmodelle komplexer Typen in Diagrammen dar, die tabellarisch organisiert sind. Grafisch hervorgehoben werden nur die Kompositoren und Kardinalitäten. hyperModel wählt einen ähnlichen Weg, indem Partikel als Klassenattribute von UML-Klassendiagrammen erscheinen. Als Besonderheit werden verschachtelte Kompositoren hier als gesonderte Diagramme aufgeführt. Dies macht zwar die Aufteilung deutlich, kann visuell aber schnell überladen wirken, sodass von einem ähnlichen Vorgehen in dieser Arbeit abgesehen wird.

Neben der Auflistung von Informationen in Diagrammen stehen noch andere Optionen für ihre Darstellung zur Verfügung. Inhaltsmodelle und Attribute ließen sich z. B. durch eine DTD-ähnliche Notation beschreiben. Der frühere Ansatz „Turbo XML“, der von Michel und Wilde bereits untersucht wurde, wählt diese Darstellung in einer Seitenleiste. Daneben haben Bernauer et al. diese Methode in einem eigenen UML-Profil für die Darstellung von XML-Schema in UML umgesetzt. Hier werden Inhaltsmodelle als Constraints außerhalb des Klassendiagramms eines zugehörigen Typs vermerkt. Die Notation orientiert sich dabei direkt an der DTD-Grammatik. Für diese Art der Informationsdarstellung spricht die kompakte äußere Form. Andererseits können bei komplexen Inhaltsmodellen längliche, verschachtelte Ausdrücke entstehen, die optisch unansehnlich und in ihrer Gesamtheit schwer zu verstehen sind. Weiterhin setzt diese Form Kenntnisse über die Semantik der DTD-Notation voraus. Diese Variante kommt daher ebenfalls nicht in Frage.

Eine weitere Lösung besteht darin, Elemente eines Inhaltsmodells und Attribute jeweils in eigene kleinere Diagramme auszulagern, die mit dem Diagramm des enthaltenden Typen verbunden oder von diesem umgeben sind. Damit verschiebt sich allerdings die Perspektive wieder in eine Element-zentrische Richtung, da die Typen selbst in den Hintergrund geraten. Im Prinzip verwenden die meisten Element-zentrischen Werkzeuge wie XMLSpy eine ähnliche Visualisierung. Es lässt sich dabei beobachten, dass die Darstellung von Typbeziehungen in diesen Ansätzen nicht optimal ist, da etwa das Anzeigen eines Basistyps an die Referenzierung eines abgeleiteten Typs durch ein Elements gekoppelt ist. Die Typhierarchien sollen aber direkt erkenntlich sein, weswegen dieses Vorgehen nicht praktikabel ist. Alternativ dazu könnte man wie besprochen die Typen einer Hierarchie anzeigen und mit den Diagrammen der entsprechenden Elemente ihres Inhaltsmodells verbinden. Bei tiefen Hierarchien würde hierdurch allerdings erneut eine visuelle Überladung durch eine hohe Anzahl an Verbindungspfeilen entstehen, da alle Nachfolger eines gegebenen Typen mit den Elementen seines Inhaltsmodells zu verbinden wären. An dieser Stelle wäre ein Auslassen der Verbindungen zu geerbten Partikeln denkbar, was der Semantik von UML-Diagrammen entspräche. Damit würde quasi wiederum eine Kenntnis über die Bedeutung externer Darstellungsformen vorausgesetzt. Die Visualisierung soll allerdings ohne solche externen Kenntnisse auskommen, indem Informationen sofort erkenntlich dargestellt werden. Auch diese Option wird also nicht weiterverfolgt.

Unter den genannten Optionen scheint die erste für das Vorhaben am attraktivsten. Typen und die Typhierarchien stehen hierbei im Vordergrund, indem sie als tabellenartige Diagramme dargestellt und verbunden werden. Weiterführende Informationen wie Facetten, Partikel oder Kardinalitäten sind als Inhalte der Diagramme einsehbar. Dies ermöglicht eine visuell einfach zu erfassende Darstellung und betont den gewünschten, Typ-zentrischen Charakter ohne auf fundamentale Informationen über die Typen zu verzichten. Im Folgenden wird die genaue Visualisierung konkretisiert und dargelegt, welche Informationen Bestandteil dieser sein sollen.

4.1.4. Typen und Typhierarchien

Für die Ermittlung der groben Darstellungsform dienten Aspekte von komplexen Typen als Beispiele. Wie festgelegt, sollen natürlich auch einfache Typen grafisch angezeigt werden. Nachfolgend wird daher für beide Typarten definiert, welche Informationen in der Visualisierung erscheinen sollen.

Einfache Typen

Die Informationsdarstellung für einfache Typen fiel unter den betrachteten Ansätzen generell relativ schwach aus. Während einige Werkzeuge einfache Typen visuell gar nicht erst darstellen, zeigen andere oftmals leere Diagramme, in denen nur der Name des einfachen Typs zu sehen ist. Die darzustellenden Informationen richten sich hierbei nach der Ableitungsart.

Built-in Datentypen Die Essenz eines einfachen Typen ist sein Wertebereich, sodass die Visualisierung diesen Aspekt hervorheben sollte. Für built-in Datentypen sind ihre Wertebereiche bereits im Standard festgelegt. In den meisten Fällen werden Grammatiken oder reguläre Ausdrücke genutzt, um erlaubte Literale zu beschreiben, die entsprechende Werte repräsentieren. Die built-in Datentypen sind fundamental für die Ableitung von nutzerdefinierten einfachen Typen, sodass auch diese als Diagramme zu visualisieren sind. Durch die komplexe Definition möglicher Literale bzw. Werte werden diese aber bis auf ihren Namen nicht genauer aufgeführt. Stattdessen wird an dieser Stelle angenommen, dass der Nutzer in Kenntnis über mögliche Werte eines Datentypen ist. Die Diagramme von built-in Typen dienen somit als Einstiegspunkte bzw. Wurzeln für die weiteren Hierarchieebenen, wobei die Hierarchie der Datentypen selbst nur soweit dargestellt wird, wie es durch anderweitige Referenzen im Schema nötig ist.

Restriction Facetten sind die wichtigsten Informationen eines nutzerdefinierten einfachen Typen, der durch Einschränkung konstrukiert wird. Bei vielen Ansätzen werden diese in Seitenleisten ausgelagert, die sich abseits der eigentlichen Bildfläche der Visualisierung befinden. Damit weiß der Nutzer auf der einen Seite immer, wo sich entsprechende Angaben im Werkzeug befinden, da die Position der Seitenleisten i. A. fest sind. Auf der anderen Seite hängen die Informationen durch ihre Auslagerung nicht mehr zusammen. Dieser Aspekt wird zusätzlich dann verstärkt, wenn die Facetten in mehrere Reiter wie in XMLSpy oder Eclipse WTP verstreut sind. In diesem Fall bedarf es zusätzlicher Aktionen seitens des Nutzers, um an die entsprechenden Stellen zu gelangen. Obwohl es naheliegend ist, Facetten nach Kategorien (verschachtelt) zu ordnen, ist dies insgesamt dem schnellen Erfassen jener Facetten nicht zuträglich. Ein Grund, warum viele Ansätze diese Methode wählen, liegt vermutlich in der Art der Einbindung von einfachen Typen. Im Prinzip verwenden alle Ansätze (auch die Typ-zentrischen) einfache Typen grafisch ausschließlich als Eigenschaften für referenzierende Elemente oder Attribute. Würden an jeder dieser Referenzen weitere Angaben vermerkt, käme es zu unnötigen Redundanzen in der Darstellung.

Die Anzeige der Facetten macht demnach nur an den Diagrammen Sinn, die eine Definition selbst repräsentieren. Die Programme JDeveloper und teilweise hyperModel realisieren genau

diesen Ansatz. Vergleicht man die entsprechenden Diagramme mit denen der Ansätze ohne einer Visualisierung für Facetten, wird der Informationsgewinn deutlich. Im Grunde besteht die Angabe einer Facette in beiden Werkzeugen aus einer Auflistung von Facettennamen-Werte-Paaren. Weitere (XML-)Attribute („fixed“ und „id“) einer Facette sind im Fall des JDevelopers weiterhin aus einer Seitenleiste zu entnehmen, was zudem auch für die Attribute der einfachen Typen selbst gilt („final“ und „id“). Da diese Informationen nur einmal an den Diagrammen für die definierenden Komponenten erscheinen würden, wäre ihre Unterbringung an jenen Diagrammen überlegenswert. Diese Attribute sind in vielen XML-Schemata häufig nicht gesetzt oder mit einem Standardwert belegt, weswegen diese Einträge dann leer blieben. Dies hätte „dünnbesetzte“ Diagramme bzw. Tabellen zur Folge, sodass zumindest für diese Attribute die Verwendung eines Mechanismus ähnlich einer Seitenleiste zweckmäßig erscheint.

Was die Facetten betrifft, macht es hingegen durchaus Sinn, eine Visualisierung wie die von JDeveloper zu wählen. Dabei ist es möglich, ihre Darstellung in dem Sinne einer Kategorisierung kürzer zu fassen. Eine Auflistung von Name-Werte-Paaren orientiert sich noch relativ nahe an der XML-Syntax. Daher könnte man beispielsweise die Facetten „length“, „minLength“, „maxLength“ in eine Eigenschaft „LengthFacets“ zusammenfassen und als Wert eine Notation wählen, die der Kardinalitätsangabe in UML ähnlich ist. Aus den Facetten „minLength = 1“ sowie „maxLength = 5“ entstünde damit z. B. die Angabe „LengthFacets = <1..5>“. Es lässt sich argumentieren, dass es unvorteilhaft für das Verständnis ist, wenn eigene Notationen in die Visualisierung gebracht und mit „echten“ Facetten vermischt werden. Viele der untersuchten Werkzeuge bedienen sich aber bereits dieses Mittels, da z. B. die Partikelattribute „minOccurs“ und „maxOccurs“ häufig ebenfalls durch eine Angabe symbolisiert werden, die der UML-Kardinalität entspricht. Es liegt also nahe, dies auch für einige Facetten zu realisieren, unter der Voraussetzung, dass jene Eigenschaften als Zusammenfassung von mehreren Facetten erkennbar sind. Weitere Facetten, die komprimiert dargestellt werden können, sind „minInclusive“, „maxInclusive“, „minExclusive“ und „maxExclusive“. Hier würde sich als Notation von Werten die verschiedenen Intervallgrenzen aus der Mathematik anbieten. So wäre z. B. die Eigenschaft „InclusionFacets = [1,5[“ eine mögliche Abkürzung für die Facetten „minInclusive“ und „maxExclusive“ mit den Werten 1 und 5. Die Werte von „enumeration“-Facetten ließen sich ebenfalls in einer gemeinsamen Eigenschaft aufzählen. Eine Aufzählung der Werte 1, 2 und 3 würde dann statt durch mehrerer Name-Wert-Paare notiert in der einzelnen Eigenschaft „EnumerationFacets = (1,2,3)“ stehen.

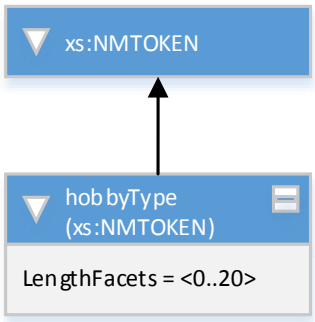


Abbildung 4.1.: Mögliches Diagramm für eine Ableitung durch Einschränkung.

Es ist naheliegend, in den Diagrammen für abgeleitete einfache Typen auch nur die direkten Facetten der Typdefinition anzuzeigen. Angenommen, es besteht eine tiefe Hierarchie aus eingeschränkten einfachen Typen, wird das manuelle Nachverfolgen der Hierarchie u. U. mühselig. Es wäre also vorteilhaft, wenn auch die Facetten aller Vorgänger (bis auf die built-in Datentypen) eines einfachen Typen an dessen Diagramm einsehbar wären. Wird ein Facettentyp verfeinert, zählt für die Darstellung immer nur die aktuellste. Um zu große Redundanzen in der Darstellung

zu vermeiden, sollten die „geerbten“ Facetten an einem Diagramm nur bei Bedarf durch eine Nutzeraktion angezeigt werden.

Union Einer besonderen Behandlung bedarf die Darstellung der Ableitung durch Vereinigung. In allen untersuchten Werkzeugen werden hierfür inhaltsleere Diagramme verwendet. Die Vereinigung selbst wird anschließend optisch durch das explizite Aufschreiben der Mitgliedstypen, oder wie im Falle der Graph-View im XSD-Designer, durch jeweils eine Verbindung zu den Diagrammen der Mitgliedstypen angezeigt. Das Wesen der Vereinigung ist die Kombination möglicher Wertebereiche zu einem neuem Wertebereich. Aus dieser Sicht macht das Vorgehen des XSD-Designers Sinn und übermittelt den Inhalt der Wertebereichsvereinigung schneller, als würden die Namen der vereinigten Typen manuell im Schema aufgesucht. Obwohl die Ansätze davon absehen, den Inhalt einer Vereinigung genauer darzustellen, wäre aber auch dies möglich. Einerseits könnte die bisherige Hierarchie der Vorgänger in dem Diagramm der Vereinigung selbst erneut vollständig (als Diagramme) abgebildet werden, was allerdings redundant ist und dadurch visuell schnell überladen wirkt. Andererseits könnten die Diagramme der Mitgliedstypen mit dem Diagramm des Vereinigungstypen umschlossen werden. Dies würde die Redundanz beseitigen, kann bei einem häufigen Einsatz der <union>-Komponente in einem Schema jedoch zu großen, unübersichtlichen Diagrammen führen. Für die Visualisierung wird daher eine Alternative zu dem ersten Vorschlag gewählt.

Anstatt die Hierarchie selbst zu wiederholen, wird diese abgeflacht dargestellt. Für jeden Mitgliedstypen wird ein Bereich in dem Diagramm bestimmt, die entsprechend der Namen der Mitgliedstypen benannt werden. Für das Erfassen des Wertebereichs sind im Prinzip nur die verwendeten built-in Datentypen am Anfang der Hierarchiepfade sowie die bis einschließlich zu den Mitgliedstypen aufgetretenen Facetten wichtig. Genau diese Angaben werden jeweils in den verschiedenen Bereichen des Diagramms für die Vereinigung notiert. Dies würde die Redundanzen durch die doppelte Angabe der Hierarchiebeziehungen vermeiden. Das mehrfache Auftreten der Facetten bliebe allerdings. Daher wäre auch hier das Verbergen dieser Informationen günstig, die dann auf Wunsch dargestellt werden können. Somit entstünde eine kompakte Darstellung, die den inhaltsleeren Diagrammen der verschiedenen anderen Ansätze ähnelt.

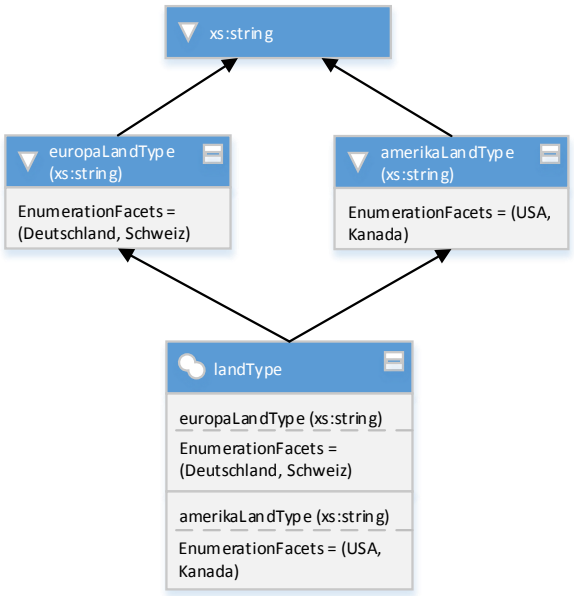


Abbildung 4.2.: Mögliches Diagramm für eine Ableitung durch Vereinigung.

Einen Bruch gäbe es bei diesem Vorgehen, wenn von einem Vereinigungstypen erneut durch Einschränkung abgeleitet wird. In diesem Fall sind nur die <enumartion>-, <pattern>- und <assertion>-Facetten erlaubt. Diese dürfen dabei nur mit Werten belegt sein, die bereits aus dem Wertebereich des Vereinigungstypen stammen. Das Weiterführen der Facetten, die aus fortlaufenden Typeinschränkungen entstehen, hätte an dieser Stelle ein Ende. Dennoch würde diese Methode eine Bereicherung für die Darstellung von Vereinigungen sein, sodass sie Bestandteil der angestrebten Visualisierung sein soll.

List Einfache Typen lassen sich weiterhin durch die Listenbildung erstellen. Auch diese Ableitungsart wird von den untersuchten Werkzeugen nur unzureichend visualisiert. Syntaktisch wie semantisch ähnelt der List-Konstruktor dem Union-Konstruktor. Daher wird auch ein ähnliches Vorgehen für die Darstellung vorgeschlagen. Die Listenbildung basiert nur auf einem einzigen einfachen Typen, sodass auch nur eine Verbindung zu entsprechendem Diagramm nötig ist. Um diese Ableitungsart von den bisherigen zu unterscheiden, wird das Diagramm durch ein Symbol oder eine Beschriftung markiert, die den UML-Stereotypen ähnlich ist. Der Rest der Darstellung folgt dann dem Prinzip der Darstellung von Vereinigungstypen. Das heißt, dass auch hier die Facetten des Hierarchiepfads der Vorgänger notiert und zunächst verborgen werden, sodass ein kompaktes, aufklappbares Diagramm entsteht. Da Listentypen nur durch die <enumeration>-, <pattern>-, <assertion>-Facetten und <minLength>-, <maxLength> sowie <length>-Facetten einschränkbar sind, ergibt sich auch hier der aufgezeigte Bruch in dem Vorgehen.

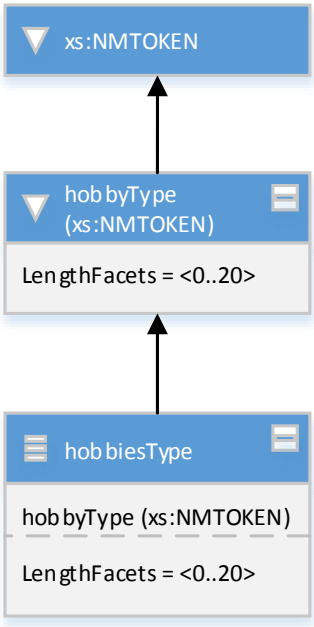


Abbildung 4.3.: Mögliches Diagramm für eine Ableitung durch Listenbildung.

Komplexe Typen

Im Gegensatz zu einfachen Typen beschreiben komplexe Typen (mit komplexem Inhalt) keine Wertebereiche, sondern Objektmengen. Die Struktur der Objekte solcher Mengen, wird durch die Kombination von Kompositoren, Elementen und Attributen sowie die von diesen genutzten (Daten-)Typen festgelegt. Für die Darstellung der Hierarchie komplexer Typen sind diese Informationen ebenfalls von Interesse, da nur durch diese Veränderungen der Objektmengen

erkennbar sind. Dementsprechend finden sich diese Angaben auch in den meisten Werkzeugen wieder. Entscheidend dabei ist, wie diese Komponenten grafisch angeordnet werden.

Kompositoren Kompositoren werden jeweils in einer eigenen Hierarchie pro komplexen Typ verwendet. Diese Hierarchie entspricht bereits dem äußeren eines Baumes, wobei die Blätter Elemente sind und die Wurzel derjenige Kompositor ist, der sich direkt unterhalb des <complexType>-Elements eines Schemas befindet. Dementsprechend wählen viele Ansätze eine Darstellung, die diesen „Kompositor-Baum“ direkt abbilden. Da dieser bei vielfach geschachtelten Kompositoren stark anwachsen kann, sollte darauf geachtet werden, diesen möglichst platzsparend unterzubringen. Eine explizite Darstellung der Kompositoren in eigenen Diagrammen wie in XMLSpy oder JDeveloper ist ungeeignet, da dieses Vorgehen zu viel Platz in Anspruch nimmt und sich von der Typ-Zentrierung entfernt.

Vorteilhaft sind also die Darstellungen von Eclipse WTP und der Content-Model-View in Visual Studio, da die Kompositoren direkt in dem Diagramm des enthaltenden komplexen Typen angezeigt werden. Gut zu sehen ist dabei ein Vorteil der grafischen Darstellung, da die Blätter bzw. Elemente in einer Ebene angeordnet sind. Dadurch werden diese hervorgehoben und die Kompositoren treten in den Hintergrund, womit die Elemente schneller zu sehen sind, als etwa in der üblichen Notation von XML-Schema. Dies täuscht nicht darüber hinweg, dass das direkte Abbilden der Kompositoren relativ Syntax-nahe ist.

Wie im Abschnitt über die allgemeine Darstellungsform erläutert, ließe sich der Kompositorbaum mitsamt der Blatt-Elemente in einen geklammerten Ausdruck ähnlich zu der DTD-Notation oder regulären Ausdrücken abflachen, wobei verschiedene Symbole bzw. „Operatoren“ für die verschiedenen Kompositoren stehen würden. Dies ist zwar platzsparend, dafür aber nicht unbedingt einfach zu entschlüsseln. Dies gilt besonders dann, wenn die Elemente um weitere Angaben wie Typen oder Kardinalitäten ergänzt werden. Ein weiterer Vorschlag ist, Elemente mit einer Angabe zu versehen, die dem Pfad entlang der Kompositoren eines Typen entspricht. Jedoch ist auch diese Darstellungsform nicht einfach zu erfassen. Zudem träten Redundanzen auf, da gemeinsame Pfadabschnitte von verschachtelten Kompositoren mehrfach notiert würden. Daher wird der Kompromiss mit der Syntaxnähe durch eine Abbildung der Kompositoren in den Diagrammen eingegangen. Die Abbildung der Kompositoren soll sich dabei an den Diagrammen der Content-Model-View im XSD-Designer orientieren, da diese Darstellung im Vergleich zu der von Eclipse WTP näher an der festgelegten tabellenartigen Ausrichtung liegt.

Elemente und Attribute Im nächsten Schritt ist zu klären, welche Informationen Elemente und Attribute tragen sollen. Die untersuchten Ansätze bieten in diesem Aspekt gute Vorschläge. Zunächst gilt es zu definieren, welche Eigenschaften direkt im Diagramm und welche in einer Seitenleiste o. Ä. ausgelagert werden sollen. Der wichtigste Bestandteil eines Elements oder Attributs ist sein Name. Im Stil von Name-Wert-Paaren bietet sich die Kombination des Namens mit dem zum Element oder Attribut zugewiesenen Typen an. Referenzierte Deklarationen werden zur Unterscheidung mit einem Symbol markiert. Falls die „default“- oder „fixed“-Attribute gesetzt sind, lässt sich diese Angabe durch eine Zuweisung nach dem Name-Wert-Paar ähnlich der UML-Notation abbilden, wobei hier unterschiedliche Symbole zur Unterscheidung verwendet werden müssen. Alternativ dazu könnte bei gesetztem „fixed“-Attribut eine zusätzliche Eigenschaft angegeben werden, die die Nicht-Änderbarkeit der Werte anzeigt.

Bernauer et al. verwenden in ihrem UML-Profil hierfür z. B. die „read only“-Constraint. Die Attribute „minOccurs“ und „maxOccurs“ an Elementen oder Kompositoren sind maßgebend für die Wiederholung von Elementen in XML-Instanzen. Es ist naheliegend, nach dem Vorbild der meisten Werkzeuge beide Eigenschaften mittels einer UML-ähnlichen Notation für Kardinaltitäen abzukürzen. Gleichermaßen ist jene Notation für die „use“-Eigenschaft von Attributen nutzbar. Der Wert „optional“ entspricht dabei „[0..1]“, „required“ steht für die Kardinalität „[1..1]“ und

„prohibited“ wird durch die Angabe „[0..0]“ repräsentiert. Weitere Attribute wie „final“, „form“ oder „id“ tragen weniger zur Struktur eines Typen bei, weswegen diese aus Gründen der Übersichtlichkeit in einer Seitenleiste o. Ä. angezeigt werden sollten. Für die Unterscheidung von Elementen und Attributen bieten sich optisch abgetrennte Bereiche oder verschiedene Symbole zur Markierung entsprechender Einträge im Diagramm an.

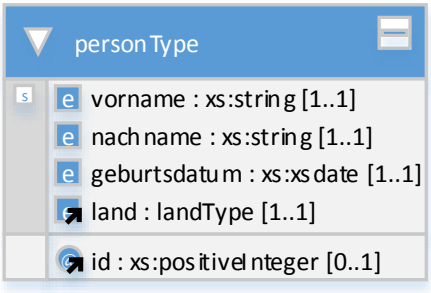


Abbildung 4.4.: Mögliches Diagramm für einen komplexen Typen mit komplexem Inhalt.

Attributgruppen und Elementgruppen Attribute sind weiterhin per Attributgruppen in die Definition eines komplexen Typen einbindbar. Dieser Mechanismus dient dabei lediglich der Einkapselung und Wiederverwendung von Attributen bei der XML-Schemamodellierung. Durch die Referenz von Attributgruppen wird also keine semantisch unterschiedliche Aussage getroffen, als würden die Attribute direkt in der Typdefinition notiert. Um die Darstellung möglichst einfach zu halten, werden dafür die verpackten Attribute direkt in dem vorgesehenen Bereich des Diagramms angezeigt und die Attributgruppen nicht weiter visualisiert.

Außerdem sind Elementgruppen nutzbar, um auch Elemente mitsamt der umschließenden Kompositoren einzukapseln. Mit dieser Komponente wird dasselbe Ziel verfolgt, wie mit den Attributgruppen. Durch die Einkapselung sollen ganze Strukturenbeschreibungen für verschiedene komplexe Typen ebenfalls wiederverwendbar sein. Im Vergleich zur direkten Notation des Inhaltsmodells wird im Endeffekt aber auch durch diese Form der Referenz die Semantik eines komplexen Typen nicht verändert. Um eine konsistente und klare äußere Form der Visualisierung zu gewährleisten, werden Elementgruppen genau wie Attributgruppen nicht in eigenen Diagramme angezeigt. Diese werden aufgelöst, indem das Inhaltsmodell direkt an den Diagrammen der komplexen Typen angezeigt wird.

Wildcards Neben der Angabe von Elementen und Attributen ist es möglich, sogenannte Element- bzw. Attribut-Wildcards als Komponenten in komplexen Typen zu verwenden. Diese erlauben es, in einem XML-Dokument an ihrer Stelle (prinzipiell) beliebige andere Elemente aus verschiedenen Namensräumen anzuzeigen. Die Angabe der Namensräume erfolgt über Attribute, die Namensräume zulassen oder ausschließen. Zudem können Elemente oder Attribute explizit über einen QNamen blockiert werden. Gemein ist diesen Eigenschaften, dass die Werte in Listen angegeben werden. Diese sind schwierig platzsparend in den Diagrammen unterzubringen, sodass jene Angaben in eine Seitenleiste o. Ä. ausgelagert werden. In dem Diagramm selbst wäre für Element- und Attribut-Wildcards die Anzeige des Attributs „processContents“ denkbar, da hier nur Werte eines vordefinierte Bereichs möglich sind. Für Elementwildcards ist es angebracht, zusätzlich die Attribute „minOccurs“ und „maxOccurs“ mit der erläuterten Methode anzugeben.

Referenzen In Referenzen sind stets Elemente oder Attribute involviert. Eine Form der Referenz ist die Angabe eines Verweis auf globale Deklarationen anstelle der Deklarierung von

Elementen oder Attributen innerhalb eines komplexen Typs selbst. Da Deklarationen nicht angezeigt werden sollen, wird diese Art Referenz wie besprochen nur durch eine Hervorhebung oder Markierung der entsprechenden Diagrammkomponente repräsentiert. Damit einher geht die zweite Referenzart, die Referenzierung von Typen. Die Zuweisung eines Typen zu Elementen oder Attributen erfolgt in XML-Schemata an den Deklarationen selbst. Da dies mit der angestrebten Visualisierung nicht darstellbar ist, wird diese Angabe stattdessen in der Referenz verschmolzen. In dem Diagramm wird also prinzipiell eine Referenz durch die Deklaration „ersetzt“ und speziell markiert. Ein ähnliches Vorgehen kann in anderen Ansätzen wie Eclipse WTP, JDeveloper oder Visual Studio beobachtet werden. Damit wird zwar auf der einen Seite der Redundanzminimierung durch den Referenzierungsprozess in XML-Schemata konzeptionell entgegengewirkt, eignet sich auf der anderen Seite am besten für die kompakte Darstellung des Sachverhalts in einer Typ-zentrischen Umgebung und wird daher ebenfalls zu einem Aspekt der zu entwickelnden Visualisierung.

Weiterhin ist zu überlegen, ob die Typreferenzierung grafisch weiter aufgelöst werden kann. In einigen Element-zentrischen Editoren ist es möglich, das Inhaltsmodell und Attribute eines referenzierten Typen direkt an dem Diagramm anzeigen zu lassen, welches für eine referenzierte Deklaration steht. Im Fall von XMLSpy etwa, ist dieses Vorgehen nötig, da die Typen in diesem Visualisierungsparadigma nicht durchgehend einsehbar sind und somit dem Nutzer das manuelle Aufsuchen oder Öffnen der entsprechenden Deklaration erspart bleibt. Durch die Anzeige von Typen und ihrer Beziehungen sind diese jederzeit erkennbar. Das manuelle Auffinden wird dadurch allerdings nicht vollständig beseitigt, sondern nur erleichtert. Aus Gründen der Übersichtlichkeit wird das Einbinden von Inhaltsmodellen anderer komplexer Typen in das Diagramm eines ausgewählten komplexen Typen, ähnlich wie in XMLSpy oder JDeveloper, nicht umgesetzt. Um das Auffinden der Diagramme der refenzierten Typen dennoch zu erleichtern, ist eine Verbindung des Diagrammeintrags für ein Partikel mit dem entsprechenden Diagramm des referenzierten Typen denkbar. Das Äußere dieser Verbindungsart sollte sich dabei von dem für die übliche Vererbungsbeziehung unterscheiden, z. B. durch eine andere Linienform oder Pfeilspitze. Eclipse WTP realisiert diese Methode bereits, aber nur über eine Ebene, sodass stets eine gute Übersicht gewährleistet ist. Bei einer zu großen Anzahl an Typen sowie Referenzen von diesen könnte allerdings durch eine Masse von Pfeilen eine visuelle Überladung entstehen, sodass dieser Aspekt als optionaler Anteil für die Visualisierung angesehen werden soll.

Weitere Angaben XML-Schema definiert eine Reihe weiterer Komponenten, die in dem Grundlagen-Kapitel ausgelassen wurden. Hierzu zählen Annotationen, Schlüsselbedingungen, Zusicherungen und Module. Es wurde bereits an verschiedenen Stellen angedeutet, dass einige Informationen nicht direkt in der Visualisierung darstellbar sind. Dies ergibt sich aus der geforderten Fokussierung eines Aspekts von XML-Schema, sodass bei der Darstellung anderer Informationen, die sich nicht in dem Fokus befinden, Abstriche gemacht werden müssen. Im Falle einer Typ-zentrischen Darstellung ist für die genannten Komponenten keine explizite Darstellung vorgesehen.

Annotationen können an fast jeder XML-Schemakomponente auftreten. Daher wäre es schwierig, geeignete Stellen in der Visualisierung für jede dieser Komponenten zu finden, die zudem auch noch wenig Platz in Anspruch nehmen. Zudem sind lange Einträge in den Annotation möglich, die selbst einiges an Raum einfordern. Weiterhin genießt diese Komponente keine hohe Priorität für das Verständnis der Typhierarchie, sodass entsprechende Werte in Seitenleisten o. Ä. ausgelagert werden. Sind für eine Diagrammkomponente jedoch gar keine Seitenleisteneinträge vorgesehen, wird auch auf diese Darstellung verzichtet.

Schlüsselbedingungen können in XML-Schema unterhalb von Elementen auftreten. Wie erörtert, werden keine Diagramme für Elemente eingeführt. Jedoch ist es vorgesehen, einige Informationen von Elementen in gesonderten Bereichen der Visualisierung anzuzeigen. Da die Struktur

bzw. Hierarchie der Schlüsselbedingungskomponenten realtiv flach ist, bietet es sich an, auch für diese Aspekte Einträge in jenem gesondertem Bereich vorzusehen.

XML-Schema 1.1 führt mit den Zusicherungen ein Konstrukt ein, welches logische Konsistenzbedingungen ausdrückt. Diese können für einfache und komplexe Typen angegeben werden. Im Fall von einfachen Typen geschieht dies über die <assertion>-Facette. Die Integration in die Visualisierung kann daher wie bei den restlichen Facetten erfolgen, wobei sich lediglich die Attributnamen unterscheiden. Für komplexe Typen lassen sich Konsistenzbedingungen mittels <assert>-Komponenten angeben. Da diese auf der gleichen Ebene wie die Attribute und der „Wurzel-Kompositor“ in einer Typdefinition notiert werden, wäre ein gesonderter Bereich im Diagramm des entsprechenden Typen denkbar, wie z. B. in XMLSpy zu sehen. Da nicht absehbar ist, wie lang die Testausdrücke sind und eine gewisse maximale Diagrammgröße eingehalten werden soll, müssten jene Angaben in einem solchen Fall abgekürzt werden. Ratsam ist es daher, wiederum in einem Seitenleisten-ähnlichen Bereich die vollen Ausdrücke des „test“ bzw. „xpathDefaultNamespace“-Attributs anzuzeigen.

Die letzte Kategorie sind Module. Modul-Komponenten ermöglichen es, Deklarationen und Definitionen auf verschiedene Weisen zusammenzuführen. Damit lässt sich ein Datenmodell auf mehrere Schemadateien zu verteilen. Das heißt, dass sich z. B. in einer Schemadatei Referenzen auf Bestandteile einer anderen Schemadatei beziehen. Dabei kann ebenso ein Basistyp in einer anderen Datei definiert sein, als ein von ihm abgeleiteter Typ. Die Visualisierung sollte daher nicht nur die Defintionen einer Schemadatei berücksichtigen, sondern auch alle weiteren, die durch Module integriert werden, da andernfalls die Hierarchie nicht vollständig abgebildet werden kann.

Extension Weiterhin muss die Visualisierung der Vererbungsbeziehungen diskutiert werden. Ziel soll es dabei sein, nicht nur das vollständige Inhaltsmodell abgeleiteter Typen anzuzeigen, sondern auch gleichzeitig die Veränderungen zu betonen. Für die Erweiterung bedeutet dies, dass die Partikel des Basistypen im abgeleiteten Typ mit angezeigt werden, ähnlich wie in XMLSpy oder JDeveloper. Eine Markierung von geerbten Partikeln wird dabei nicht eingesetzt, da durch die direkte Darstellung des Basistypen in der Hierarchie erkennbar ist, woher diese stammen. Um dennoch die Veränderung des Inhaltsmodells durch das Hinzukommen von neuen Partikeln anzuzeigen, werden diese stattdessen hervorgehoben. Dafür bieten sich z. B. farbliche Texte oder Symbole wie etwa ein vorangestelltes „+“-Zeichen an.

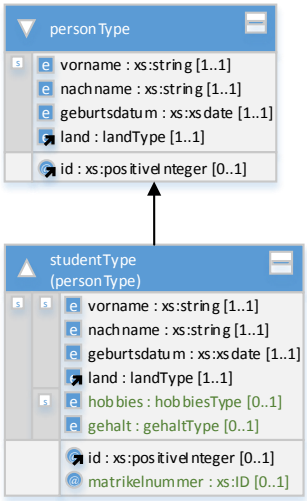


Abbildung 4.5.: Mögliches Diagramm für die Erweiterung von komplexen Typen.

Restriction Die Einschränkung von Inhaltsmodellen beinhaltet mehrere Punkte, die zu beachten sind. Grundsätzlich soll auch hier das komplette Inhaltsmodell des abgeleiteten Typen angezeigt werden. Dabei können Elemente durch das Setzen der Attribute „minOccurs“ bzw. „maxOccurs“ auf jeweils 0 vom Inhaltsmodell entfernt werden. Ebenso lassen sich Attribute durch den Wert „prohibited“ für die Eigenschaft „use“ ausschließen. Als Konsequenz zeigen einige Werkzeuge wie der XSD-Designer in Visual Studio diese Bestandteile nicht mehr in dem abgeleiteten Typen. Michel und Wilde konnten diese Methode in der Vergangenheit auch in anderen Editoren ausmachen. Dieses Vorgehen ist zwar prinzipiell berechtigt, jedoch wird die Veränderung nicht unmittelbar deutlich. Andere Ansätze wie XMLSpy oder JDeveloper zeigen hingegen entfernte Elemente oder Attribute an, wobei eine gesonderte Markierung zur Hervorhebung genutzt wird, häufig durch ein Durchstreichen des Namens. Für die angestrebte Visualisierung bedeutet dieses Vorgehen, dass die Löschung direkt im abgeleiteten Typ angezeigt wird und nicht durch einen Vergleich der beteiligten Diagramme vom Basistyp und abgeleiteten Typ vom Nutzer hergeleitet werden muss. Daher soll dieses Vorgehen für die Darstellung umgesetzt werden. Neben dem Durchstreichen von entsprechenden Diagrammkomponenten eignet sich auch hier das farbliche Markieren der Texte, wobei die Farbe möglichst gegenteilig zu der verwendeten Farbe bei der Ableitung durch Erweiterung sein sollte. Alternativ wären ebenfalls Symbole wie das „-“-Zeichen denkbar.

Neben dem totalen Ausschluss von Elementen oder Attributen ist auch eine einfache Beschränkung in der späteren Nutzung in XML-Dokumenten möglich. Diese umfassen dabei z. B. die Veränderung des zugewiesenen Typen eines Elements oder Attributs, wobei nur ein Untertyp eingesetzt werden darf. Daneben können Element- oder Attributwerte durch das Setzen des „fixed“-Attributs fixiert werden, die im Obertyp zuvor beliebig aus dem Wertebereich des zugewiesenen Typen wählbar waren. Zudem sind die unteren und oberen Multiplizitäten von Elementen bzw. die „use“-Eigenschaft von Attributen wie erläutert einschränkbar, ohne dass entsprechende Komponente aus dem Typ entfernt wird. In all diesen Fällen ist ebenfalls ein zusätzlicher optischer Vermerk über die Veränderung angebracht. Dafür bietet sich z. B. das Vorgehen von XMLSpy an, bei dem veränderte Bestandteile und Attribute bzw. Eigenschaften farblich und/oder durch eine Umrandung markiert werden. Im Vergleich zu der Darstellung von Eclipse WTP etwa sind derartige Veränderungen wesentlich schneller zu erkennen, sodass ein ähnliches Vorgehen in der geplanten Visualisierung umgesetzt werden soll.

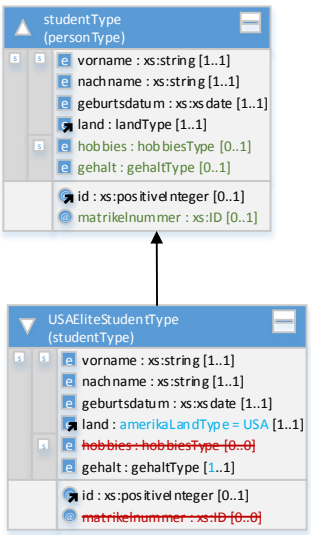


Abbildung 4.6.: Mögliches Diagramm für die Einschränkung von komplexen Typen.

Einfacher Inhalt Eine Besonderheit stellen komplexe Typen mit einfachem Inhalt dar. Im Prinzip funktioniert diese Art ähnlich zu den einfachen Typen. Inhalt dieser Typen sind erneut Facetten statt Partikel. Im Unterschied zu normalen einfachen Typen dürfen diese jedoch auch Attribute tragen und folgen den Ableitungsregeln der Erweiterung sowie Einschränkung komplexwertiger komplexer Typen. Im Fall der Erweiterung dürfen Attribute dem Basistyp hinzugefügt werden, was nach der vorgestellten Methode angezeigt wird. Die Einschränkung umfasst u. a. das Hinzufügen von verschiedenen Facetten, die in der Darstellung hervorgehoben werden. Attributtypen, -häufigkeiten und -wertefixierung werden wie erläutert gehandhabt. Die Diagramme von komplexen Typen sind somit quasi ein Hybrid aus den Diagrammen einfacher Typen und komplexer Typen mit komplexem Inhalt, wobei der Partikel-Anteil durch den Facetten-Anteil ersetzt wird. Demnach soll auch für diese Diagrammart das vorgestellte Prinzip der Darstellung der „Facetten-Fortpflanzung“ umgesetzt werden.

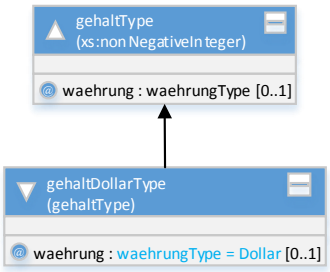


Abbildung 4.7.: Mögliches Diagramm für die Einschränkung von komplexen Typen mit komplexem Inhalt.

Lokale Typen

Die Bildung von tiefen Typhierarchien von einfachen und komplexen Typen basiert darauf, dass entsprechende Typen global vorliegen, da nur von solchen Typen abgeleitet werden kann. Dementsprechend soll auch die Darstellung genau diese Beziehung von globalen Typen abbilden. Daneben bietet XML-Schema die Möglichkeit, Typen lokal zu definieren, die dann einmalig für die entsprechend enthaltende Komponente gelten. Dennoch nehmen auch solche Typen an einer Hierarchie teil, da die genannte Voraussetzung, dass die Typableitung auf globalen Typen beruht, auch für lokale Typen gilt. Der Unterschied zu einem abgeleiteten globalen Typ besteht somit lediglich darin, dass abgeleitete lokale Typen selbst nicht mehr für die Ableitung neuer Typen nutzbar sind. Konzeptionell gesehen entsprechen lokale Typen also einem Ende eines möglichen Hierarchiepfades. Es wäre nicht konsistent, lokale Typen nicht zu visualisieren.

Das Hindernis für ihre Visualisierung besteht darin, dass sie in Elementen oder Attributen verschachtelt sind. Da Elemente und Attribute selbst nicht visualisiert werden sollen, würden entsprechende lokale Typen, die jenen Deklarationen zugewiesen sind, in der Darstellung fehlen. Andere Typ-zentrische Ansätze wie hyperModel oder Eclipse WTP umgehen dieses Problem, indem lokale Typen für die Visualisierung wie globale behandelt werden. Das bedeuet, dass Inhaltsmodelle oder die Facetten wie üblich in einem Diagramm angezeigt werden, was leicht in die bestehende Hierarchie integrierbar ist. Entsprechend dem Konzept des Endes eines Hierarchiepfads würden solche Diagramme mögliche Blätter eines Diagramm-Baums darstellen. Für den Titel solcher Diagramme müsste ein Name generiert werden, da lokale Typen unbenannt sind. Sowohl hyperModel als auch Eclipse WTP bedienen sich dafür des Namens der übergeordneten Komponente. Auch für tiefer verschachtelte Typen ließe sich durch das Rückverfolgen der Element- oder Attributhierarchie durch Verkettung der Bezeichner ein Name erzeugen. Dabei müsste darauf geachtet werden, trotzdem eindeutige Namen zu kreieren, um auch eine Eindeu-

tigkeit der Diagrammtitel zu sichern und Unklarheiten für Nutzer auszuschließen. In [Kap13] wurde dieser Aspekt näher beleuchtet.

Das beschriebene Vorgehen in Verbindung mit den Benennungsmöglichkeiten für derartige Diagramme soll daher als Lösung für die Darstellung lokaler einfacher bzw. komplexer Typen dienen und Bestandteil der Visualisierung sein. Diagramme von lokalen Typen sind demnach für einfache Typen immer ein Blatt eines Hierarchiepfades und für komplexe Typen nicht in der restlichen Hierarchie integriert.

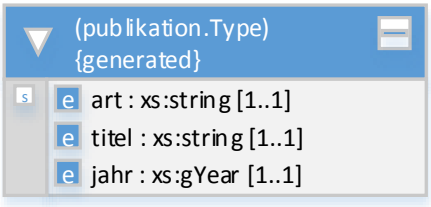


Abbildung 4.8.: Mögliches Diagramm für einen lokalen Typ.

Editierbarkeit

Im optimalen Fall bietet die Visualisierung Möglichkeiten per Direktmanipulation Aspekte zu verändern. Dafür müssten Grundoperationen zur Verfügung stehen wie das Erstellen und Löschen neuer Diagramme sowie das Ziehen von Verbindungen, um die Vererbungsbeziehungen anzugeben. Innerhalb der Diagramme müssten Optionen bereit stehen, um im Falle von einfachen Typen Facetten hinzuzufügen und zu löschen sowie entsprechende Werte zu setzen. Durch die vorgeschlagene zusammengefasste Darstellung einiger Facetten könnte für diese eine beschleunigte Wertemanipulation bestehen. Nicht erlaubt wäre die Manipulation von Facetten in Union- oder List-Diagrammen, da diese nur Repräsentationen einfließender Basistypen sind und entsprechend an den originalen Diagrammen zu editieren wären. Liegt ein Diagramm eines komplexen Typen vor, so müssten hier entsprechend Kompositoren, Elemente, Attribute sowie Zusicherungen anzugeben sein. Einige Aspekte wie Typen, fixierte Werte, Standardwerte oder Kardinalitäten ließen sich dabei direkt am Diagramm bearbeiten. Weitere Angaben müssten hingegen nach der Auswahl einer entsprechenden Komponente in einer Seitenleiste o. Ä. editierbar sein. Wichtig ist, dass bei jeglichen Änderungen die Konsistenz zum Standard geprüft wird. Idealerweise sollte der Nutzer durch kontextabhängige Editiervorschläge bzw. -Möglichkeiten nicht in der Lage sein, die Konsistenz des abgebildeten Schema zum Standard zu verletzen. Dies betrifft vor allem die Operationen rund um Aspekte von abgeleiteten Typen. Bei einer Einschränkung dürften demnach nur Facetten oder Eigenschaften von Partikeln oder Attributen manipuliert werden, sodass tatsächlich auch nur Untermengen an erlaubten Werten bzw. Objekten entstünden.

Für die Visualisierung wird allerdings keine Möglichkeit für die Manipulation angestrebt. Wie erläutert, soll sie ähnlich wie die verschiedenen Ansichten des XSD-Designers in Visual Studio als Veranschaulichung eines XML-Schemas dienen. Die Integration der vorgestellten Visualisierung als Sicht in eine andere Umgebung, die ihrerseits eine Bearbeitung von XML-Schemata ermöglicht, wäre für beide Komponenten eine Bereicherung. Somit ließe sich etwa die Schwäche der einen Komponente durch die Stärke der anderen ausgleichen.

4.2. Typmanagement

Dieses Kapitel stellt das Konzept zum Typmanagement vor. Dafür wird zunächst geklärt, was unter dem Begriff eigentlich zu verstehen ist. Als Typmanagement wird in dieser Arbeit der

Prozess aufgefasst, den Nutzer beim Umgang mit den Typen und den Typhierarchien eines XML-Schemas zu unterstützen. Dies setzt ein Werkzeug zur Bearbeitung von XML-Schemata voraus, in welchem dieser Prozess als Funktionalität zu integrieren wäre. Das Werkzeug muss dabei eine gewisse Abstraktion anbieten, die Typen als ganze Einheiten auffasst, sodass die Inhalte der jeweiligen Definitionen ihrerseits vollständig greifbar sind.

Im Detail ist das Einfügen, Löschen und Aktualisieren eines Typen Gegenstand des zu unterstützenden Prozesses. Wird ein Typ gelöscht, von dem andere Schemakomponenten abhängig sind, ist das entsprechende Schema nicht mehr gültig bzw. fehlerhaft. Üblicherweise muss der Nutzer in diesem Fall selbst die abhängigen Stellen ermitteln und Maßnahmen treffen, die die Korrektheit des Schemas wiederherstellen. Diese Maßnahmen unterscheiden sich je nach Art des Typen und seiner Beziehung zu den restlichen Typen eines Schemas. Als Maßnahmen stehen zwei Optionen zur Verfügung: das Löschen aller weiteren direkten und indirekten Abhängigkeiten oder das Zuweisen eines ähnlichen, im Schema vorhandenen Typen zu den betroffenen Komponenten als Kompensation. Der erste Fall bedeutet einen hohen Verlust an Informationen, da neben den direkten Referenzen des zu löschenden Typen auch Referenzen von abgeleiteten Typen betroffen sein könnten. Im zweiten Fall wird versucht, möglichst viele Informationen zu erhalten, sodass die Referenzen des zu entfernenden Typen weiter bestehen können und wenig Nachfolge-Änderungen nötig sind. Die Entscheidung über die Wahl der Maßnahme bleibt stets dem Nutzer überlassen, da auch der erste Fall mitsamt seiner Folgen durchaus beabsichtig sein kann. Oft ist sich aber ein Nutzer über mögliche Abhängigkeiten im Unklaren, sodass besonders hier die im zweiten Fall beschriebene Kompensation eine Hilfestellung bietet. Das heißt, dass das Typmanagement vornehmlich das Auffinden von Abhängigkeiten sowie die Umsetzung einer der beiden Maßnahme zum Erhalt der Schema-Korrektheit erleichtern bzw. automatisieren soll. Da der Nutzer die Entscheidung zwischen diesen beiden Optionen trifft, soll es ebenfalls Teil der Funktionalität sein, ermittelte Abhängigkeiten zur Entscheidungsfindung zunächst anzuzeigen.

Nachfolgend werden verschiedene Fälle für das Löschen von einfachen und komplexen Typen analyisiert und mögliche Konsequenzen für die Korrektheit des Schemas sowie Lösungen für deren Sicherstellung aufgezeit. Im Rahmen dieser Analyse wird auch untersucht, wie die Ähnlichkeit von Typen (auf konzeptioneller Ebene) bestimmt werden kann.

4.2.1. Einfügen von einfachen Typen

Das Einfügen eines einfachen Typen in ein XML-Schema ist prinzipiell problemlos möglich. Dabei muss unterschieden werden, ob der Typ außerhalb der bestehenden Hierarchie erstellt oder in diese integriert wird. Die erste Möglichkeit bedeutet, dass der neue Typ direkt von einem built-in Datentypen abgeleitet wird. Da an bestehenden Komponenten keinerlei Änderungen erfolgen, bedarf das XML-Schema bei dieser Möglichkeit keiner weiteren Anpassungen. Änderungen sind hingegen notwendig, sollte der neu erstellte Typ entsprechend der zweiten Möglichkeit in die Hierarchie eingefügt werden.

So ist es z. B. denkbar, dass der erstellte Typ als neue **Wurzel** eines Hierarchiepfades dienen soll. Hierbei muss der Typ, der ehemals die Wurzel repräsentierte angepasst werden. Im Detail heißt das, dass sein Obertyp auf den erstellten Typen gesetzt wird. Daneben lässt sich der neue Typ auch als **innerer Knoten** in eine Hierarchie einfügen, was ebenfalls das Umsetzen des Obertyps vorheriger Typen zur Folge hat. Weniger aufwendig gestaltet sich das Einfügen eines erstellten Typen als **Blatt** einer Hierarchie, da hier keine nachfolgenden Typen involviert sind. Weiterhin wäre es für alle drei Fälle ebenfalls möglich, dass ein neuer Typ einen alten ersetzen soll, d. h. dass die Länge des Hierarchiepfads nicht vergrößert wird. Im Prinzip umfasst diese Art des Einfügens lediglich zusätzlich das Löschen eines bestehenden Typen, worauf in Unterabschnit 4.2.3 eingegangen wird. Eine eventuelle Manipulation des Obertypen anderer, abgeleiteter Typen bleibt aber weiterhin bestehen.

Die Herausforderung an den Schemaautor besteht bei dem Einfügen zunächst also darin, den Wertebereich bzw. (fortgepflanzte) Facetten bis zum gewünschten Punkt des Einfügens zu berücksichtigen und dementsprechend richtige Angaben für einen neuen Typen zu setzen. Ist der eingefügte Typ z. B. eine Einschränkung, so müssen die Facetten kompatibel zu dem Ober- und Untertyp sein. Dies gilt auch andersherum, wenn der erstellte Typ selbst ein Listen- oder Vereinigungstyp ist, da hier Facetten von ableitetenden Untertypen besonderen Regeln unterliegen. Weiterhin muss auf die Art des Obertypen geachtet werden, wenn der einzufügende Typ selbst listenwertig ist, da XML-Schema die Listenbildung von Listen-Typen nicht vorsieht.

Das Einfügen ist also gerade bei komplexeren Hierarchien nicht unüberlegt auszuführen und umfasst i. A. auch das Umsetzen des Basistypen von bestehenden Untertypen, was einer Änderung an jenen Typen entspricht. Mitunter kann außerdem das Manipulieren von weiteren Eigenschaften wie den Facetten bestehender einfacher Typen nötig sein, damit der neu erstellte Typ an die gewünschte Stelle in der Hierarchie eines XML-Schemas passt. Da solche Änderungen an bestehenden Typen nicht mehr Teil der Einfügung eines neuen Typen selbst sind, sind Erläuterungen hierzu im nächsten Abschnitt 4.2.2 zu finden.

4.2.2. Aktualisieren von einfachen Typen

Ein Update eines Typen wird hier so definiert, dass der Typ selbst mit seinem Namen bestehen bleibt, allerdings verschiedene seiner Eigenschaften manipuliert, hinzugefügt oder entfernt werden. Somit können Updates auf mehrere Arten erfolgen, die sich jeweils unterschiedlich stark auf die Struktur der existierenden Typhierarchie auswirken. Nachfolgend werden die verschiedenen Updatemöglichkeiten nach Kategorien untersucht. Dabei wird erläutert, welche Updates in welchem Umfang in Abhängigkeit von Ober- und Untertypen realisierbar sind. Im idealen Fall sollte ein Werkzeug den Nutzer bei der Bearbeitung eines Typen insofern unterstützen, als solche Abhängigkeiten und Bedingungen kenntlich gemacht werden oder eine entsprechende ungültige Änderungsoperation nicht oder nur mit einer Fehlermeldung ausgeführt wird.

Ändern von Facetten Die Facetten eines Typen, der per Einschränkung abgeleitet wurde, stellen eine weitere Möglichkeit für mögliche Änderungen dar. Bei den Überlegungen wird davon ausgegangen, dass generell nur Facetten verwendet werden, die zum ursprünglichen built-in Datentypen bzw. zur Art des Obertypen (List- und Union) passen und entsprechend Werteangaben getätigt werden, die im Wertebereich liegen. Daneben existieren weitere Bedingungen, die sich aus der Beziehung zu direkten und indirekten Ober- und Untertypen ergeben, die anschließend aufgezeigt werden. Generell ist das Einfügen und Ändern einer Facette im Hinblick auf den Obertypen problemlos möglich, sofern dieser ein built-in Datentyp ist und der Wertebereich nicht überschritten wird. Schränkt der zu modifizierende Typ einen anderen nutzerdefinierten Typen ein, ist zusätzlich auf ein eventuelles Auftreten verschiedener Facettenarten (abhängig von der eingefügten bzw. manipulierten Facette) zu achten. Dies gilt für alle weiteren direkt und indirekt abgeleiteten, einschränkenden Typen. Der wesentliche Punkt ist, dass die Facettenwerte eines Obertyps nicht mit weniger strikten Werteangaben in einem Untertyp quasi „rückgängig“ gemacht werden können, sodass der Wertebereich maximal verkleinert werden kann. Beim Löschen einer Facette herrschen weniger strenge Bedingungen, da lediglich eine Einschränkung der möglichen Werte aufgehoben wird, was zudem unabhängig von der Position des einschränkenden Typen in der Hierarchie ist. Das bedeutet, dass das Löschen einer Facette keine Instanzanpassungen provoziert, im Gegensatz zum Einfügen oder Ändern von Facetten.

Weiterhin kann jede Facette das Attribut „fixed“ tragen, welches die Werte „true“ oder „false“ annehmen kann. Der Wert „true“ bedeutet, dass der eigentliche Wert der Facette in weiteren Ableitungsschritten nicht verändert werden kann, während „false“ dies ermöglicht. Sollte der Schemaautor dieses Attribut mit dem Wert „true“ setzen, muss also ermittelt werden, ob in der abgehenden Hierarchie Typen existieren, die eine weitere Einschränkung genau mittels der

fixierten Facette vornehmen. Dies wäre ungültig, sodass diese Änderungsoperation nicht durchgeführt wird und der Nutzer entsprechend zu informieren ist. Alternativ dazu wäre es denkbar, die weiteren betroffenen Facetten zu löschen, um das Update dennoch zu ermöglichen. Daneben sind für bestimmte built-in Datentypen bereits einige Facetten vordefiniert und teilweise auch fixiert. So ist beispielsweise für Typen in der Hierarchie unterhalb von „xs:decimal“ die fractionDigits-Facette auf „0“ fixiert. Für einen Nutzer ist es vorteilhaft, wenn ein Werkzeug auch solche vordefinierten, nicht änderbaren Facetten anzeigt, um einen fehlerhaften Gebrauch von vornherein auszuschließen.

Facetten: minExclusive, minInclusive, maxExclusive, maxInclusive Der Wert einer eingefügten **minExclusive**-Facette muss größer als oder gleich dem Wert der minExclusive-Facette des Obertyps sowie kleiner als oder gleich dem Wert der Facette des Untertyps sein. Dementsprechend darf der Wert nur bis zum Wert der Obertyp-Facette verkleinert sowie bis zum Wert der Untertyp-Facette vergrößert werden. Darüber hinaus darf diese Facette nicht eingefügt werden, wenn im selben Typ bereits die **minInclusive**-Facette vorhanden ist. Diese Facette unterliegt dabei den gleichen Regeln wie die der minExclusive-Facette. Weiterhin ist es notwendig, dass der minExclusive-Wert des betrachteten Typen echt größer bzw. echt kleiner als der Wert einer minInclusive-Facette im Ober- bzw. Untertyp ist.

Für die Facetten **maxExclusive** und **maxInclusive** gelten die Bedingungen in umgekehrter Reihenfolge. Beim Einfügen, Verkleinern und Vergrößern dieser Facetten in einen einschränkenden Typen ist darauf zu achten, dass die Werte kleiner als oder gleich dem Wert der Obertyp-Facette sowie größer als oder gleich dem Wert der Untertyp-Facette sind. Die gemeinsame Nutzung in einer Typdefinition ist ebenfalls ausgeschlossen.

Facetten: length, minLength, maxLength Diese Gruppe von Facetten beschränkt die Länge einer Zeichenkette im Fall von atomaren Typen sowie die Anzahl von Elementen von listenwertigen Typen. Wird eine **minLength**-Facette eingefügt, muss diese größer als oder gleich dem Wert der Obertyp-Facette sein sowie kleiner als oder gleich dem Wert der maxLength-Facette, was zudem auch zutrifft, wenn diese Facette auf dem modifizierten Typ vorhanden ist. Findet diese Facette zudem bereits in einem Untertypen Anwendung, muss der Wert größer als oder gleich sein sowie kleiner oder gleich dem Wert einer der maxLength-Facette eines Untertypen. Soll eine bestehende minLength-Facette verändert werden müssen die gleichen Begrenzungen des Wertes beachtet werden. Entsprechend umgekehrt verhält sich die Situation beim Einfügen der **maxLength**-Facette. Ihr Wert muss im Vergleich zum Obertyp kleiner werden und im Vergleich zum Untertyp größer werden oder in beiden Fällen gleich bleiben. Ebenso darf der Wert den Wert einer minLength-Facette eines Ober- oder Untertypen nicht unterschreiten.

Das Einfügen der **length**-Facette gleicht dem Verwenden der minLength- und maxLength-Facetten mit identischen Werten. Der Wert muss daher im Hinblick auf den Obertypen im Bereich zwischen vorhandenen minLength- und maxLength-Facetten liegen. Im Umkehrschluss bedeutet dies, dass für die Verwendung dieser Facette in Untertypen keine weiteren minLength- und maxLength-Facetten vorliegen dürfen, da der Wertebereich nach dem Einfügen einer length-Facette nicht mehr verändert werden darf. Das Einfügen der length-Facette ist auch dann gültig, wenn diese bereits im Ober- oder Untertyp vorhanden ist. Hierbei müssen die Werte allerdings identisch sein, was somit diese Semantik des veränderten Typen nicht verändert.

Facetten: totalDigits, fractionDigits Ähnlich zu den Längen-Facetten schränkt diese Kategorie von Facetten die Anzahl an erlaubten Ziffern von zahlenwertigen Typen ein. Die **totalDigits**-Facette gibt dabei die Anzahl an insgesamt möglichen Stellen an. Demnach darf der Wert einer neu eingefügten Facette nicht größer als der Wert einer Obertyp-Facette sein sowie den Wert der Facette eines Untertypen nicht unterschreiten. Ein Ändern eines bestehenden Wertes ist somit

im Bereich der Ober- und Untertyp-Facetten möglich, wobei der Wert 0 generell ausgeschlossen ist. Den gleichen Regelungen unterliegt die **fractionDigits**-Facette, die die Dezimalstellen einer Dezimalzahl angibt, wobei der Wert 0 prinzipiell erlaubt ist. Bei der Verwendung beider Facetten in Kombination miteinander darf der Wert der fractionDigits-Facette sowohl im Hinblick auf den selben Typen als auch auf Ober- und Untertypen den Wert einer totalDigits-Facette nicht überschreiten.

Facetten: enumeration, pattern, assertion Mit dieser Gruppe von Facetten lassen sich mögliche Werte für einen einfachen Typen festlegen. Durch ein Einfügen von **enumeration**-Facetten werden Werte spezifisch aufgelistet, die anschließend für den bearbeiteten Typen erlaubt sind. Auf Untertypen pflanzen sich diese Angaben implizit fort und gelten auch dort. Sollen also enumeration-Facetten in einen Typ eingefügt werden, dessen Obertyp ebenfalls enumeration-Facetten trägt, sind nur Werte von diesen für die neuen Facetten erlaubt, wodurch wieder eine kleinere oder maximal gleichgroße Wertmenge entsteht. Für die Bearbeitung von existierenden enumeration-Facetten muss daher immer die Hierarchie in Richtung des Obertyps betrachtet werden.

Das Einfügen von **pattern**-Facetten schränkt die Werte durch reguläre Ausdrücke für den manipulierten Typ sowie für seine Untertypen ein. Liegen mehrere pattern-Facetten in einem Typen vor, werden die beschriebenen Wertebereiche verodert, sodass es ausreicht wenn ein Wert in einer Instanz in einem der beschriebenen Bereiche liegt. Im Gegensatz zur enumeration-Facette ist es zunächst erlaubt in einem Untertypen gänzlich andere Wertebereiche zu definieren, die sich mit denen des Obertypen gegenseitig ausschließen. Für die Validierung von XML-Dokumenten stellt dies allerdings ein Problem dar, da die Wertebereiche von Obertyp- und Untertyp-Facetten verundet werden. Das bedeutet, dass so eine leere Menge für Instanzwerte geschaffen wird, was oftmals nicht im Sinne des Schemaautors ist.

Die **assertion**-Facette teilt eine ähnliche Charakteristik wie die pattern-Facette. Statt regulären Ausdrücken werden hier allerdings XPath-Ausdrücke verwendet, die gewisse Bedingungen beschreiben, um Wertebereiche einzuschränken. Werden mehrere assertion-Facetten in einen Typ eingefügt, müssen Werte im Unterschied zur pattern-Facette alle Bedingungen erfüllen, damit diese gültig sind. Dies wirkt sich auch auf abgeleitete Typen aus. Liegen also in einem Obertyp assertion-Facetten vor, müssen die Untertyp-Werte diese ebenfalls erfüllen. Durch die Komplexität von XPath-Ausdrücken ist es für die XML-Schemavalidierung schwierig, die Ausdrücke zu vergleichen und Widersprüche zu ermitteln (siehe auch [Sch04]). Prinzipiell können also erneut unerfüllbare Bedingungen geprüft werden, was nicht als Fehler erkannt wird, mögliche Werte für den entsprechenden Typen aber ausschließt.

Obige drei Facetten können auch in Kombination miteinander verwendet werden. Auch hier können durch unachtsame Angaben in pattern- und assertion-Facetten leere Wertebereiche erzeugt werden. Bei der Manipulierung dieser Facetten wäre also eine Unterstützung für einen Nutzer durch ein Werkzeug besonders vorteilhaft. Während dies für die pattern-Facetten mit Methoden der theoretischen Informatik bzw. formalen Sprachen noch vorstellbar wäre, ist ein Vergleich von XPath-Ausdrücken wie erwähnt, bedingt durch die Mächtigkeit für komplexere Ausdrücke, nicht effizient umzusetzen (siehe querycontainment).

Facetten: whiteSpace, explicitTimezone Eine gemeinsame Eigenschaft der letzten Kategorie von Facetten ist, dass nur vordefinierte Werte erlaubt sind. Die **whiteSpace**-Facette einer Typdefinition signalisiert dem XML-Prozessor, wie dieser verschiedene Whitespace-Zeichen in entsprechenden Attribut- oder Elementwerten zu handhaben hat. Der Wert „preserve“ gibt an, dass gesetzte Whitespace-Zeichen nicht verändert werden. „replace“ bedeutet, dass sämtliche Whitespaces, die kein Leerzeichen sind, durch eben dieses ersetzt werden. Zu guter Letzt werden mit dem Wert „collapse“ vorangehende und nachfolgende Whitespace-Zeichen gelöscht,

Whitespace-Zeichen, die kein Leerzeichen sind, durch dieses ersetzt und aufeinanderfolgende Leerzeichen zu einem einzigen reduziert. Für das Einfügen dieser Facetten in einen Typen und ihre Bearbeitung bedeutet dies, dass im Bezug zum Ober- und Untertyp der Wert der Facette in ausschließlich der genannten Reihenfolge wechseln darf. Zudem ist ein mehrfaches Verwenden nicht erlaubt.

Mit der **explicitTimezone**-Facette ist die Angabe der Zeitzone in Datums-Datentypen kontrollierbar. Der Wert „optional“ macht jene Angabe möglich, ohne verpflichtend zu sein. Zu einer Verpflichtung kommt es dann, wenn der Wert „required“ verwendet wird. Als Gegenteil dazu verbietet der Wert „prohibited“ die Angabe der Zeitzone. Wird diese Facette in einen Datentyp eingefügt oder bearbeitet, muss erneut auf die Ober- und Untertypen geachtet werden. Der Wert darf von „optional“ zu „required“ und „prohibited“ in Richtung der Untertypen wechseln, aber nicht in Richtung des Obertyps. Sobald diese Facette in einem Typen einen der letzten zwei Werte trägt, darf dieser nicht mehr geändert werden.

Ändern des Obertypen Das Ändern des Obertypen eines Typs ist nicht ohne Weiteres möglich, da durch die Konstellation von verschiedenen Facetten und Typkonstruktoren sowie beteiligten built-in Datentypen Abhängigkeiten bestehen. Zunächst muss geklärt werden, ob die Untertypen des Typen, dessen Obertyp verändert wird, weiterhin Untertypen bleiben sollen oder nicht. Im ersten Fall wird die komplette abgehende Hierarchie ab dem modifizierten Typ ebenfalls „umgehängt“. Der zweite Fall bedeutet, dass die Untertypen an den ehemaligen Obertypen des bearbeiteten Typen gehängt werden. Dieser Fall gleicht im Prinzip dem Löschen eines Typen aus einer Hierarchie. Abschnitt 4.2.3 betrachtet diese Operation im Detail, sodass hier vom ersten Fall ausgegangen wird.

Anschließend wird ein mögliches Vorgehen beschrieben, das die Eignung eines anderen Typen als neuen Obertypen ermittelt. In einem potentiellen Werkzeug könnte dieses eingesetzt werden, um den Nutzer entweder zu warnen, falls dieser einen unpassenden Typen als neuen Obertyp ausgewählt hat oder dazu, um von vornherein nur passende Obertypen anzuzeigen. Zur Vereinfachung werden nachfolgend der manipulierte Typ als T, sein alter Obertyp als OT_1 und der mögliche neue als OT_2 bezeichnet.

Schritt 1: Vergleich der built-in Datentypen Um eine möglichst zügige Entscheidung zu liefern, ist es von Vorteil, verschiedene Aspekte eines Typen getrennt voneinander zu betrachten. Fundamental für die Qualifizierung eines Typen sind die beteiligten built-in Datentypen. Im Prinzip muss die Wertmenge, die durch die built-in Datentypen von OT_1 beschrieben wird, eine Teilmenge von OT_2 sein. Dieser Vergleich ist relativ schnell ausführbar, indem die Hierarchie in Richtung der Obertypen zurückverfolgt wird. Für atomare Typen muss lediglich jeweils entlang eines einzigen Hierarchiepfades navigiert und nur ein Paar von built-in Datentypen betrachtet werden. Ist der von OT_2 zugrundeliegende built-in Typ der gleiche built-in Typ, auf dem OT_1 basiert oder ein Obertyp von diesem, lässt sich OT_2 als neuer Obertyp verwenden. Weiterhin ist es möglich, auch built-in Typen zuzulassen, die zwar laut XML-Schema kein wirklicher Obertyp sind, trotzdem aber Werte des ursprünglichen built-in Typen darstellen können. So könnte z. B. der Typ „xs:string“ auch die selben Werte von „xs:integer“ für Attribute und Elemente in einer XML-Instanz ermöglichen. Problematisch sind etwaige Datentyp-spezifische Facetten, die in den Hierarchiepfaden auftreten können, worauf Schritt 3 eingeht. Liegt OT_1 und/oder OT_2 direkt oder indirekt im Laufe der Hierchiepfade als Vereinigungen vor, erhöht sich die Anzahl an zu vergleichenden built-in Datentypen. Die Prämisse, dass die built-in Datentypen von OT_2 zunächst Werte der built-in Datentypen von OT_1 aufnehmen können müssen, bleibt dabei bestehen. Sollte diese nicht gelten, kann OT_2 bereits an diesem Punkt als neuer Obertyp ausgeschlossen werden, unabhängig von der Art der untersuchten Obertypen. Ein Update ist in diesem Fall also nicht bzw. nur mit Fehlern im XML-Schema oder Informationsverlust in der Instanzbasis umsetzbar.

Schritt 2: Vergleich der Art der Obertypen Sobald festgestellt ist, ob kompatible built-in Datentypen vorliegen (wovon nachfolgend ausgegangen wird), ist die Art der betrachteten Obertypen ein weiteres Kriterium, mit welchem die Eignung schnell ausgeschlossen werden kann. Die Art eines Obertypen ergibt sich aus möglichen Listen- oder Vereinigungskonstruktoren im Laufe der vorangehenden Hierarchien, die zur Ermittlung vollständig abzusuchen sind.

2.1 Atomic → Atomic Der Wechsel eines atomaren Obertypen zu einen anderen atomaren Obertypen ist prinzipiell ohne Schwierigkeiten möglich. Wie bei allen weiteren Unterfällen müssen passende Facetten vorliegen.

2.2 Atomic → List Konzeptionell ist dieser Wechsel vorstellbar, da Listen-Typen auch einelementige Werte erlauben. Ist T selbst oder weitere Untertypen eine **Restriction** kann der Wechsel jedoch i. A. nicht durchgeführt werden, da die Semantik für bestehende Facetten verändert wird. Liegen hingegen ausschließlich enumeration- oder pattern-Facetten vor, wäre ein Wechsel möglich. Nicht erlaubt ist der Wechsel, wenn T oder Untertypen **List**-Typ ist bzw. sind. Falls T eine **Union** ist, kann der Wechsel durchgeführt werden. Da in diesem Fall lediglich ein Mitgliedstyp ausgetauscht wird, ist es sogar denkbar, einen neuen Typen zu wählen, bei welchem der Vergleich in 1. fehlschlägt, sofern ein anderer Mitgliedstyp die Werte des alten Mitgliedstypen darstellen kann. Dieser gilt auch für die weiteren Fälle, bei dem T ein Vereinigungstyp ist.

2.3 Atomic → Union Die Integration der Werte von T in eine Vereinigung ist immer möglich, es sei denn T ist bzw. Untertypen sind eine **Restriction**. Für diesen Fall ist ähnlich wie in 2.2 zu überprüfen, ob die Facetten solcher Einschränkungen weiterhin erlaubt sind (nur enumeration und pattern). Der Wechsel ist ebenfalls möglich, wenn T eine **List** oder **Union** ist.

2.4 List → Atomic Dieser Wechsel ist mit einen potentiellen Informationsverlust behaftet, sollte OT_2 keine Whitespaces erlauben. Sofern diese Eigenschaft gegeben ist (separat zu prüfen), ist ein Wechsel in einigen Fällen denkbar. Ist T eine **Restriction** bzw. existieren einschränkende Untertypen, entscheiden die Facetten, da wie in 2.2 die Semantik verändert wird. Im Speziellen dürfen keine Längen-Facetten oder assertion-Facetten mit z. B. Listen-Funktionen verwendet worden sein. Da letzteres schwierig zu überprüfen ist, wird die assertion-Facette ebenfalls ausgeschlossen. Ist T eine **Union**, kann ein Wechsel vollzogen werden, wenn ein anderer Mitgliedstyp entsprechende Listenwerte von OT_1 aufnehmen kann.

2.5 List → List Wie in 2.1 ist ein Wechsel zwischen gleichartigen Obertypen möglich.

2.6 List → Union Sollte der Test aus Schritt 1 ergeben, dass der ausgewählte Union-Typ auch die Listenwerte repräsentieren kann, ist einem Obertypwechsel im Grunde nichts entgegenzusetzen. Ist T eine **Restriction**, so dürfen erneut keine Längen-Facetten notiert sein. Ebenso kann nicht ausgeschlossen werden, dass in assertion-Facetten Bezug zum Listen-Charakter genommen wird, sodass ein Wechsel in diesen Situationen nicht erlaubt ist. Betrifft die Update-Operation einen **Union**-Typen, kann ähnlich wie in 2.3 der Ersatz für mögliche Werte von OT_1 auch aus den restlichen Mitgliedstypen von T stammen. Der Test in 1. ist dann nicht unbedingt notwendig.

2.7 Union → Atomic Auch der Austausch eines Vereinigungs-Typen kann zu Informationsverlust führen. Da in Schritt 1 nur auf die built-in Datentypen geachtet wird, muss zudem ermittelt werden, ob OT_1 listenwertig-vereinigt ist. Ist ein atomarer OT_2 also laut dem Test aus Schritt 1 allgemein genug, um zunächst Werte jedes built-in Datentyp aus OT_2 aufzunehmen,

muss dieser zusätzlich noch Whitespaces ermöglichen. Kann dies sichergestellt werden, steht einem Wechsel nichts im Wege. Wenn T eine **Restricion** ist, können in dem Fall auch mögliche enumeration- und pattern-Facetten bestehen bleiben. Lediglich assertion-Facetten mit Listen-Funktionen wie count() verhindern einen Obertypwechsel. Kann OT_2 nicht alle Werte darstellen, ist von einem Wechsel generell abzusehen. Wurde T als **List**-Typ abgeleitet, ist ein Wechsel unter denselben Voraussetzungen möglich, wobei auf listenwertige Mitgliedstypen von OT_1 bzw. die Fähigkeit von OT_2 Whitespaces darzustellen keine Rücksicht genommen werden muss, da der Listen-Charakter erst durch T entsteht. Bei einem Wechsel in dem Fall, dass T eine **Union** ist, müssen ebenfalls keine weiteren Besonderheiten untersucht werden, außer, dass auch hier der Test in Schritt 1 nicht zwangsweise ein positives Ergebnis liefern muss, sofern andere Mitgliedstypen die Werte von OT_1 ermöglichen.

2.8 Union → List Falls T als **Restriction** vorliegt, ist dieser Fall weniger strikt als Fall 2.7. und Facetten von T bereiten keine Probleme, da Listentypen die gleichen Facetten (und mehr) erlauben. Wurde T hingegen als **List**-Typ eingeführt, ist ein derartiger Wechsel nicht erlaubt. Für **Union**-Typen ist ein Wechsel wie gehabt ohne Schwierigkeiten durchführbar.

2.9 Union → Union Wie in 2.1 und 2.5 ist ein Wechsel zwischen gleichartigen Obertypen möglich.

Schritt 3: Vergleich der Facetten Nachdem in Schritt 1 und 2 die groben Aspekte eines Typen untersucht wurden, fehlt für die endgültige Entscheidung über die Eignung die Betrachtung der Facetten. In einigen spezielleren Fällen aus Schritt 2 wurde dies bereits angesprochen. Dennoch ist die Feststellung auch für einfache Fälle wie 2.1 (Atomic → Atomic) nötig. Im Wesentlichen müssen für T bzw. der abgehenden Hierarchie sowie für OT_1 und OT_2 bzw. der jeweils vorangehenden Hierarchie alle Facetten „eingesammelt“ und in einer Datenstruktur festgehalten werden. Das heißt, dass jeder beteiligte Restriction-Typ in der besagten Richtung zu betrachten ist. Die Rückverfolgung der Obertyphierarchien kann abgebrochen werden, sobald auf Restriction-Typen getroffen wird, die Werte mittels enumeration- oder pattern-Facetten einschränken.

Nachdem für alle drei Typen die Facetten gesammelt wurden, müssen die aktuell wirksamen Facetten ermittelt werden. Das bedeutet, dass für einen Vergleich nur die in einem Hierarchiepfad zuletzt notierte Facette einer Facetten-Art herangezogen wird. Anschließend werden die effektiven Facetten auf Kompabilität geprüft. Um Zeit zu sparen ist es ausreichend, zunächst nur zu testen, ob die effektiven Facetten von OT_2 die selben Werte oder mehr ermöglichen wie OT_2. Ist dies gegeben, muss OT_2 auf jeden Fall einen Wertebereich besitzen, der auch die Werte von T aufnehmen kann, sodass die Überprüfung erfolgreich beendet werden kann. Sind die Facetten nicht kompatibel, so kann OT_2 dennoch eine Wert-Obermenge von T beschreiben, die lediglich nicht vollständig in der Wertmenge von OT_1 liegt. Das heißt, dass dann die effektiven Facetten von OT_2 und die Facetten auf Kompatibilität zu untersuchen sind, die aus der abgehenden Hierarchie von T gesammelt wurden. Schlägt dieser Test erneut fehl, ist die Zuweisung des gewählten Typen OT_2 nicht möglich. Ein Erfolg hingegen drückt aus, dass der Wechsel des Obertypen ohne Fehler bzw. Informationsverlust vollzogen werden kann.

Das Testen der Kompatibilität von (effektiven) Facetten meint genau die Bedingungen, die im Paragraph zum Ändern der Facetten beleuchtet wurden. Es sollten also generell keine widersprüchlichen Aussagen durch z. B. maxInclusive- und minInclusive-Facetten kommen. Es wäre jedoch denkbar, einen Wechsel auch bei Nicht-Kompatibilität zu ermöglichen, indem entsprechend unpassende Facetten entfernt werden, da dies keine Instanzanpassungen verursacht. Nachteilig würde sich dies allerdings für das XML-Schema selbst auswirken, da so nach vielmaligen Änderungen der Zuweisung von Obertypen nach und nach gesetzte Facetten entfernt werden könnten

und am Ende die ursprüngliche Modellierungsarbeit des Autors quasi „entkräftigt“ wird. Wie erwähnt, ist der Vergleich bzw. das Abgleichen von pattern- und assertion-Facetten schwierig bzw. nicht effizient umsetzbar. Sollten diese Facetten für Vergleiche ignoriert werden, bietet sich besonders hier ihre Entfernung an, um sicher zu gehen, dass die Instanzbasis weiterhin nur gültige Werte beinhaltet.

Ändern der Art Das Verändern der Art eines Typen kann weitreichende Folgen haben und unterliegt erneut Bedingungen, die von möglichen direkten und direkten Ober- sowie Untertypen ausgehen. Eine solche Änderungen ist daher nicht in allen Fällen möglich. Entscheidend ist im Speziellem die Art des Obertypen OT selbst sowie die Ableitungsform der Untertypen UT (stellvertretend auch für indirekte Untertypen).

Restriction → List Durch diese Änderungen wird die Anzahl an möglichen Werten sogar noch erhöht, da einerseits Facetten gelöscht und andererseits Listen-Typen neben einelementige auch mehrelementige Werten erlauben. Für **atomare** sowie **Union**-Obertypen ist diese Änderung ohne Weiteres möglich. Ist OT ein **List**-Typ ist das Update nicht erlaubt. Existieren **Restriction**-Untertypen ist die Umsetzung nur bei gültigen Facetten möglich, deren Aussagen zum Listen-Charakter passen. Geht der modifizierte Typ in einen **Union**-Untertyp als Mitgliedstyp ein, kommt es zu keinen Komplikationen.

Restriction → Union Ob OT direkt oder weniger eingeschränkt als Mitgliedstyp in T wiederholt wird, hat keine Auswirkungen auf die XML-Instanzen. Diese Änderung ist aus Sicht von OT also möglich, egal ob es ein **atomarer**, **List**- oder **Union**-Typ ist. Für einen UT, der eine **Restriction** ist, müssen wieder die Facetten stimmen, sodass die Änderung für solche Fälle nicht allgemein gültig umsetzbar ist. Ist UT ein **List**- oder **Union**-Typ, stellt die Änderung kein Problem dar. Allerdings muss im ersten Fall darauf geachtet werden, dass keine listenwertigen Mitgliedstypen in der Vereinigung vorhanden sind.

List → Restriction Wie beim Update des Obertypen ausgeführt, liegt dieser Richtung potentieller Verlust an Informationen zugrunde. Es wird von mehrelementigen Werten auf einelementige reduziert. Für einen **atomaren** Obertyp ist dies dann realisierbar, wenn dieser Whitespaces zulässt. Ist der OT ein **Union**-Obertyp wird das Update nur selten möglich sein, da ein Mitgliedstyp erstens die Werte der anderen Mitgliedstypen aufnehmen können muss und zweitens Whitespaces ermöglichen muss. Sind von T **Restriction**-Typen abgeleitet, machen Längen-Facetten bzw. assertion-Facetten mit Listenfunktionen das Update unmöglich. Für abgeleitete **Union**-Typen muss ein Mitgliedstyp die Listenwerte aufnehmen können, damit die Änderung umsetzbar ist. Zusätzlich muss der Autor darauf achten, keine Facetten zu verwenden, die die Gültigkeit verletzen, was für sich selbst genommen dem Update durch das Einfügen von Facetten entspricht.

List → Union Im Unterschied zur vorigen Änderung wird der Obertyp von T hier nicht durch Facetten eingeschränkt, sondern wird unverändert als Mitgliedstyp einer Vereinigung hinzugefügt. Es gelten daher ähnliche Ausführungen. Je nachdem, welche anderen Mitgliedstypen beteiligt sind, kann zusätzlich auch innerhalb dieser ein Typ-Ersatz für die ehemaligen Listewerte gesucht werden, sollte der **atomare** Obertyp keine Whitespaces zulassen oder kein Mitgliedstyp eines **Union**-Obertypen diese Aufgabe erfüllen. Dieses Mehr an Möglichkeiten gilt auch für abgeleitete **Union**-Typen. Was jedoch bleibt, ist die Beschränkung auf **Restriction**-Untertypen, die keine Längen-Facetten enthalten dürfen, damit ein derartiges Update ausführbar ist.

Union → Restriction Beim Wechsel von einer Vereinigung sind mehrere Faktoren wichtig. Wenn nur ein einziger Mitgliedstyp existiert, ist die Wahl des Basistypen für die Einschränkung offensichtlich. Sind mehrere Typen vereinigt, muss der allgemeinste gewählt werden. Sind auch **List**-Typen als Mitglieder beteiligt, ist es ideal, wenn ihr grundlegender Wertebereich die Werte der anderen Typen aufnehmen kann. Sollte diese Situation nicht gegeben sein, muss umgekehrt ein ausgewählter allgemeiner **atomarer** oder **Union**-Typ neben jenen grundlegenden Werten der Listentyp-Mitglieder auch Whitespaces zulassen. Untertypen, die von der Vereinigung per **Restriction** abgeleitet sind, bereiten nur dann Schwierigkeiten, wenn assertion-Facetten verwendet wurden. Hier könnten z. B. Listenfunktionen verwendet worden sein, die Bezug zu Listen-Mitgliedstypen genommen haben, aber auf einen atomaren Ersatz nicht mehr passen. Weiterhin ist das Update nicht anwendbar, wenn ein listenwertiger Mitgliedstyp als neuer Basistyp ermittelt wird und gleichzeitig **List**-Untertypen von T direkt oder indirekt abgeleitet sind. Durch **Union**-Untertypen werden durch diesen Wechsel keine Bedingungen gestellt, sofern wie beschrieben ein allgemeiner Typ gefunden werden kann.

Union → List Hier gelten erneut ähnliche Überlegungen wie im vorigen Fall. Zur Bestimmung des neuen Basistype von T dürfen nun allerdings keine **List**-Typen verwendet werden. Ein **atomarer** oder **Union**-Mitgliedstyp, der zwar die grundlegende Werte von anderen Listen-Mitgliedstypen darstellen kann, jedoch keine Whitespaces, erhält diese Fähigkeit durch den neuen Listenkonstruktor. Dieses Update gestaltet sich also erleichternd in diesem Aspekt. Existiert ein UT als **Restriction**, werden wieder höchstens assertion-Facetten zu einem Problem. So könnte etwa in Listenfunktionen auf konkrete Werte eines Listen-Mitgliedstypen mit anderen Datentyp-spezifischen Funktionen zugegriffen werden. Lässt sich ein allgemeiner neuer Typ ermitteln, der zwar die Werte solcher Listen-Mitgliedstypen darstellen kann, aber von einem anderen Datentyp abstammt (z. B. Liste von „xs:NMTOKEN“ und Liste von „xs:integer“), käme es bei der Validierung zu Typfehlern. **List**-Untertypen verhindern wie oben auch hier die Änderung. **Union**-Typen hingegen sind erneut nicht beeinträchtigt, falls ein passender, allgemeiner Mitgliedstyp auffinderbar ist.

4.2.3. Löschen von einfachen Typen

Werte und Wertebereiche sind Mittelpunkt von einfachen Typen. Auf Grundlage der built-in Datentypen können Nutzer eigene einfache Typen definieren. Durch die Zuweisung von einfachen Typen zu Element- oder Attributdeklarationen ist der Textinhalt von Elementen bzw. Attributen in XML-Dokumenten regulierbar, da so nur die Werte des zugewiesenen Wertebereichs erlaubt sind. Die Werte selbst können dabei durch unterschiedliche Literale repräsentiert werden. Wird nun ein nutzerdefinierter Typ gelöscht, den Deklarationen referenzieren, verliert das Schema seine Korrektheit. Weiterhin ist unklar, welche Werte die Menge an Elementen und Attributen der Instanzbasis annehmen können. Aus diesem Grund muss für die betroffenen Deklarationen ein neuer Typ als Ersatz gefunden werden. Dies ist ein Teil der Kompensation für die Löschung von einfachen Typen. Der zweite Teil besteht darin, die Auswirkungen auf die weitere Hierarchie zu behandeln. Diese Auswirkungen entstehen, wenn ein gelöschter einfacher Typ Ableitungen besitzt. Diesen fehlt nach der Löschung der Basistyp, was wiederum die Gültigkeit des Schemas und von XML-Dokumenten verletzt. Der Ausgleich äußert sich so, indem ein neuer Basistyp zu ermitteln ist. Bei beiden Vorgängen ist es das Ziel, trotz der Zuweisung eines anderen Typen keine Änderungen in den XML-Dokumenten auszulösen. Je nachdem, nach welcher Ableitungsart der zu löschende Typ und seine direkten und indirekten Ableitungen konstruiert wurden, gilt es verschiedene Gesichtspunkte zu beachten. Diese werden in den folgenden Fällen beleuchtet.

Fall e1: der gelöschte Typ befindet sich in keiner Hierarchie Im Prinzip befindet sich jeder nutzerdefinierte einfache Typ in einer Hierarchie, da diese am Ende immer von einem built-in

Datentypen abstammen. Keine Hierarchie meint hier also, dass der Typ direkt von einem built-in Datentyp abgeleitet ist, er selbst aber nicht als Basistyp für weitere einfache Typen dient. Es sind daher keine weiteren, abgeleiteten Typen mitsamt der jeweiligen Objektmenge zu modifizieren. Entsprechend unkompliziert gestalten sich hier die nötigen Schritte zur Sicherung der Gültigkeit des XML-Schemas sowie der zugehörigen XML-Dokumente, da im XML-Schema nur auf die direkten Referenzen geachtet werden muss.

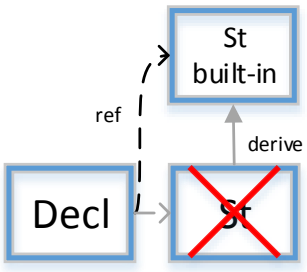


Abbildung 4.9.: Gelöschter Typ ist in keiner Hierarchie.

Am einfachsten ist eine gelöschte **Restriction** zu behandeln. Mit dieser Ableitungsform wird durch das Hinzufügen oder Verfeinern von Facetten stets ein sich verkleinernder Wertebereich beschrieben. Objekte, die Werte dieses Bereichs annehmen können, müssen diese Eigenschaft auch für die Werte des nächsten, allgemeineren Wertebereichs erfüllen. Es kann daher mit Sicherheit auf den built-in Datentyp zurückgegriffen werden, auf dem der gelöschte einschränkende Typ basiert, indem die Referenzen im Schema entsprechend angepasst werden. Mit dieser Aktion werden die Objekte in die nächsthöhere Menge integriert, was keine Anpassungen der XML-Dokumente bedeutet.

Komplizierter wird das Vorgehen bei **List**-Typen. Die Art der Werte ändert sich hier von atomaren zu listenwertigen. Es ist daher nicht immer möglich (z. B. für „xs:NMTOKEN“), den built-in Datentyp des gelöschten Listen-Typs für die referenzierenden Deklarationen zu verwenden. Diese Möglichkeit wird dann verfolgt, wenn der Basistyp Whitespaces zulässt. Sollte der Datentyp dies nicht erlauben, wird auf den Datentyp „xs:string“ zurückgegriffen, da dieser eine allgemeine Lösung für zahlen- und textwertige Datentypen ist, der zudem Whitespaces erlaubt. Damit sind auch hier keine Instanzanpassungen nötig, da die Listenwerte weiterhin erlaubt sind.

Als dritte Ableitungs-Möglichkeit existieren **Union**-Typen. Hier werden verschiedene Wertebereiche zu einem neuen vereinigt. Die Vereinigung äußert sich hier so, dass Elemente und Attribute in den XML-Dokumenten in mindestens einen Wertebereich der aufgeführten Mitgliedstypen fallen. Um keine Instanzanpassungen hervorzurufen, ist es daher ratsam, nach der Löschung des Vereinigungstypen den betroffenen Deklarationen den allgemeinsten Mitgliedstypen zuzuweisen, da der von ihm beschriebene Wertebereich die besten Chancen bietet, auch die Werte der anderen Typen aufzunehmen. Analog gilt dies für die entsprechend vorhandenen Objektmengen in den XML-Dokumenten. Kann keiner der beteiligten Typen die Werte der restlichen aufnehmen, bietet sich erneut eine Zuweisung der betroffenen Deklaration auf den Datentyp „xs:string“ an, sodass sichergestellt ist, dass die Elemente und Attribute in den XML-Dokumenten weiterhin ihren Wert behalten können.

Als Alternative kann (sofern vorhanden) für alle Ableitungsarten auch ein anderer, nutzerdefinierter Typ als Ersatz verwendet werden, der einen geeigneten Wertebereich beschreibt, sollte dieser einen besser passenden Wertebereich beschreiben.

Fall e2: der gelöschte Typ befindet sich in einer Hierarchie Wird der zu löschende einfache Typ auf Grundlage eines anderen nutzerdefinierten Typs eingeführt oder sind weitere Typen di-

rekt oder indirekt von diesem abgeleitet, befindet sich der Typ in einer Hierarchie. Je nachdem, welche Ableitungsarten involviert sind, müssen unterschiedliche Folgeaktionen unternommen werden, um die Korrektheit des Schemas sowie die Gültigkeit der Instanzen zu sichern. Dies wird in den folgenden Unterfällen analysiert.

Fall e2.1: der gelöschte Typ ist Blatt eines Hierarchiepfades Ein einfacher Typ wird hier als Blatt einer Hierarchie bezeichnet, wenn er von einem nutzerdefinierten einfachen Typ abgeleitet wird, selbst allerdings keine Untertypen besitzt. Damit ergibt sich eine Situation, die Fall e1 ähnelt, da keine direkten oder indirekten Ableitungen vom gelöschten Typ bestehen. Nach der Löschung ist also ebenfalls nur für die direkt referenzierenden Deklarationen ein neuer Typ zu ermitteln. Ziel ist es dabei wieder, möglichst wenig Instanzanpassungen zu provozieren. Im Wesentlichen ist hierfür das Vorgehen ähnlich zu dem aus Fall e1.

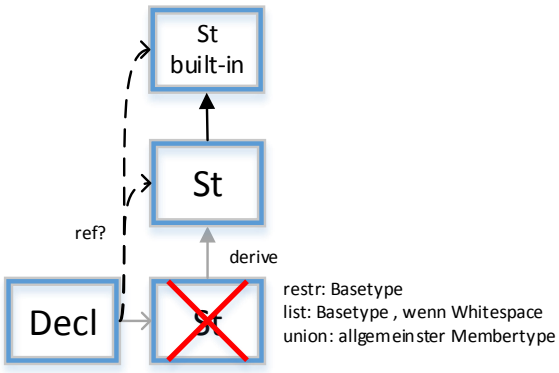


Abbildung 4.10.: Gelöschter Typ ist Blatt.

Ist der gelöschte Typ eine **Restriction** wird der nutzerdefinierte Basistyp als neuer Typ für entsprechende Deklarationen verwendet, was die XML-Dokumente nicht verändert. Der Basistyp darf dabei selbst eine Restriction, List oder Union sein, da die Wertebereiche weniger eingeschränkt sind, als die des gelöschten Typs.

Liegt ein **List**-Typ vor, ist mehr zu beachten. Generell dürfen Listentypen in XML-Schema nur von atomaren oder atomar-vereinigten Typen abgeleitet werden. Ist der nutzerdefinierte Basistyp nach dem vorhergehenden Hierarchiepfad noch immer atomar, ist es für seine Tauglichkeit als Typersatz für die betroffenen Deklarationen entscheidend, ob dieser noch Whitespaces erlaubt oder nicht. Im ersten Fall ist der Basistyp als neuer Typ für die Deklarationen verwendbar. Im zweiten Fall muss der Hierarchiepfad solange verfolgt werden, bis ein Typ gefunden wird, der Whitespaces erlaubt. Durch einen Blick auf den ursprünglichen built-in Typ kann hier Zeit eingespart werden, da bei rein atomaren Typen ein Zuwachs des Wertebereichs ausgeschlossen ist. Anders sieht es aus, wenn der Basistyp im Laufe der vorherigen Hierarchie einen atomar-vereinigten Charakter angenommen hat. Hier müssen alle Vereinigungs-Abzweigungen aller möglichen Ebenen in Richtung der beteiligten built-in Datentypen verfolgt werden. Als Vereinfachung soll es ausreichen, ausschließlich diese built-in Typen zu betrachten und einen passenden auszuwählen. Als allgemeinste Lösung wird erneut „xs:string“ als neuer Typ verwendet, sollten etwa alle Typen keine Whitespaces zulassen.

Als Mitglieder eines **Union**-Typen kommen atomare, listenwertige, atomar-vereinigte und listenwertig-vereinigte Typen in Frage. Da nicht entschieden werden kann, welche Elemente und Attribute eines XML-Dokuments welchen Teil eines solchen Vereinigungstypen für ihre Werte benutzen, muss als Lösung der kleinste gemeinsame Nenner gefunden werden. Sind als Mitgliedstyp ausschließlich atomare oder atomar-vereinigte Typen vorhanden, reicht es aus den allgemeins-

ten auszuwählen oder wahlweise die Hierarchien in Richtung der beteiligten built-in Datentypen zu verfolgen, sollten die Mitgliedstypen einzeln nicht alle Werte abdecken. Sobald listenwertige oder listenwertig-vereinigte Typen vorliegen, muss dabei als weiteres Kriterium zusätzlich auf die Möglichkeit zur Angabe von Whitespaces geachtet werden. Um eine konsistente Lösung anzubieten, sollen auch in diesen beiden Fällen als Vereinfachung ausschließlich die beteiligten built-in Datentypen verglichen werden. Bei einer Nicht-Eignung aller untersuchten Typen bietet sich wieder der Typ „xs:string“ für die Zuweisung zu den jeweiligen Deklarationen an.

Fall e2.2: der gelöschte Typ ist innerer Knoten eines Hierarchiepfades Im Unterschied zu Fall e2.1 dient der gelöschte Typ hier selbst als Basistyp für weitere einfache Typen. Betrachtet man zunächst den gelöschten Typ und dessen Basistyp bzw. Mitgliedstypen, ergibt sich eine identische Situation zu der aus Fall e2.1. Es gelten daher auch hier die oben genannten Ausführungen über die Typ-Kompensation von Deklarationen, die diesen direkt referenzieren. In einem weiteren Schritt sind allerdings zusätzlich Typen zu behandeln, die vom gelöschten Typ abgeleitet sind, um etwaige Anpassungen von Deklarationen bzw. Instanzen zu vermeiden.

Fall e2.2.1: der gelöschte Typ ist eine Einschränkung Dieser Fall ist relativ unkompliziert für die Kompensation. Weitere **Restriction**-Typen, die von diesem abgeleitet sind, müssen lediglich auf dessen Basistyp gesetzt werden. Die XML-Elemente in den Instanzen unterliegen keinen Änderungen, da die Wertebereiche vergrößert werden. Es ist daher zu überlegen, die Facetten des gelöschten Typen auf seine einschränkenden Untertypen zu übertragen, um den Wertebereich dieser Typen nicht zu vergrößern, was sich u. U. vorteilhaft auf die weitere XML-Schemamodellierung auswirken kann.

Sind vom gelöschten Typ **List**-Typen abgeleitet, lassen sich diese ebenfalls auf den Basistyp des gelöschten Typen setzen. Der Wertebereich von Listen aus weniger stark eingeschränkten Listen kann ohne Probleme Werte eines stärker eingeschränkten Bereich aufnehmen, sodass erneut keine Modifikationen an den XML-Dokumenten nötig sind.

Das Gleiche trifft auch für abgeleitete **Union**-Typen zu. Die Verallgemeinerung einer seiner Mitgliedstypen verursacht keinen Informationsverlust, sodass auch hier der Basistyp des gelöschten Typen als neue Grundlage dienen kann.

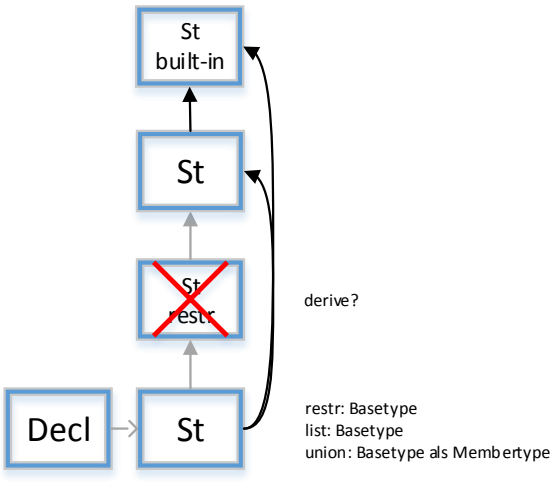


Abbildung 4.11.: Gelöschter Typ ist Knoten (Restriction).

Fall e2.2.2: der gelöschte Typ ist eine Liste In diesem Fall gelten ähnliche Überlegungen, wie die über die direkte Anpassung von Deklarationen aus Fall e2.1. Durch das Löschen eines Lis-

tentypen wird der Charakter der weiteren Hierarchie von einem listenwertigen zu einem atomaren verändert. Für abgeleitete **Restriction**-Typen verringern sich die erlaubten Facetten auf <enumeration>, <pattern>, <length>, <maxLength>, <minLength> und <assertion>. Weiterhin beziehen sich konkrete Werteangaben für diese Facetten ebenfalls auf die Semantik einer Liste in XML-Schema. Wird für die abgeleiteten Restriction-Typen der Basistyp des gelöschten Listen-Typ verwendet (der selbst nicht listenwertig sein darf), treffen die Facetten i. A. andere und womöglich nicht erlaubte Aussagen. Diese Aussagen müssen daher entfernt werden, wofür zwei Lösungen existieren. Zum einen können solche Restriction-Typen einfach selbst entfernt werden, wobei auch solche auf tieferen Ebenen betroffen sind. Für referenzierende Deklarationen wird anschließend ein Typ aus der vorherigen Hierarchie, der Whitespaces zulässt, oder „xs:string“ als neuer Typ verwendet. Zum anderen wären diese Typen auch geeignet, um als Basis für die Definitionen der einschränkenden Typen zu dienen, wobei sämtliche Facetten zu entfernen sind, sodass quasi leere Restriction-Typen entstehen. Vorteil dieser Methode ist, dass die Typnamen bestehen bleiben, wodurch die Deklarationen selbst nicht zu bearbeiten sind.

Findet der gelöschte Listentyp selbst direkt oder indirekt über eine Einschränkung von diesem als Mitgliedstyp in einer **Union** Anwendung, hängt die Folgeaktion erneut von der Fähigkeit des Basistypen ab, Whitespaces zuzulassen. Ist diese gegeben, kann dieser als Ersatz für den Mitgliedstypen genutzt werden. Als Alternative kommt ein Typ der vorherigen Hierarchie mit dieser Fähigkeit oder „xs:string“ in Frage.

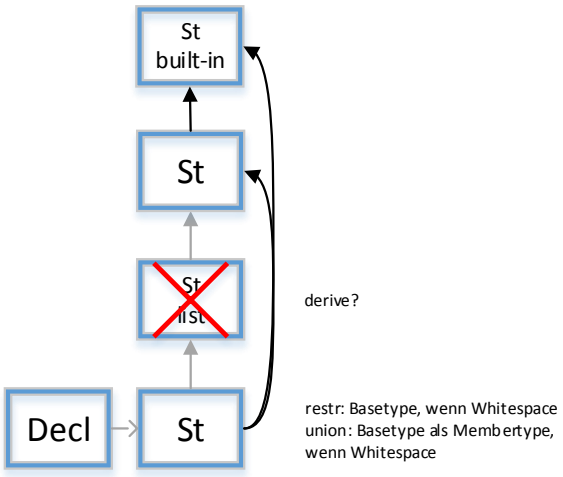


Abbildung 4.12.: Gelöschter Typ ist Knoten (List).

Fall e2.2.3: der gelöschte Typ ist eine Vereinigung Zuletzt ist der Fall zu untersuchen, in dem der entfernte Typ ein Vereinigungstyp ist. Werden weitere Typen von diesem per **Restriction** abgeleitet, ergibt sich eine ähnliche Situation wie in Fall e2.2.2. Erlaubt sind für solche Typen nur die Facetten <enumeration>, <pattern> und <assertion>. Dementsprechend ist es möglich, dass in diesen Facetten Angaben auftreten, die sich ausschließlich auf den Wertebereich eines beteiligten Mitgliedstypen beziehen. Nach der Löschung des Vereinigungstypen muss daher derjenige Mitgliedstyp als neuer Basistyp für einschränkend abgeleitete Typen ausgewählt werden, der den allgemeinsten Wertebereich beschreibt, auf den alle Facetten zutreffen können. Dies ist nötig, damit entsprechende Facetten weiterhin gültig sind. Dabei müssen prinzipiell alle abgehenden Hierarchiepfade auf etwaige Facetten untersucht werden, da auch in tieferen Ebenen einschränkende Typen Bezug zum Wertebereich des gelöschten Vereinigungstypen nehmen können. Sobald ein listenwertiger Typ als Mitgliedstyp involviert ist, muss erneut vor allem auf

die Möglichkeit geachtet werden, Whitespaces innerhalb der Werte in den XML-Dokumenten anzugeben.

Liegen abgeleitete **List**-Typen vor, ist es bei korrekten XML-Schemata ausgeschlossen, dass der gelöschte Vereinigungstyp listenwertige Typen als Mitglieder enthält. Dennoch ist nicht absehbar, ob Elemente oder Attribute in Dokumenten Listenwerte besitzen, die die Wertebereiche vermischen. Es muss daher auch hier der allgemeinste Mitgliedstyp oder mögliche Vorgänger von diesem als Ersatz für den Basistyp der abgeleiteten Listentypen ermittelt werden.

Im Prinzip gilt dies auch ähnlich für abgeleitete **Union**-Typen. Nachdem ein Mitgliedstyp (in der Form einer Union) entfernt wurde, ist unklar, ob die Instanzbasis weiterhin gültig ist, da in den weiteren Vereinigungen ebenfalls andere Typen und damit Wertebereiche einfließen. Sollte ein anderer Mitgliedstyp einer abgeleiteten Vereinigung so allgemein sein, dass er für den kompletten Wertebereich der gelöschten Vereinigung stehen kann, ist nichts weiter zu unternehmen. Ist dies hingegen nicht der Fall, muss der allgemeinste Mitgliedstyp der gelöschten Vereinigung als neuer Mitgliedstyp für die abgeleiteten Vereinigungstypen dienen. Dies ist natürlich wieder nur dann ausreichend, sofern der ausgewählte Typ allgemein genug ist und alleine einen Wertebereich beschreibt, der die Werte der restlichen Mitgliedstypen aufnehmen kann. Insbesondere müssen wieder beteiligte Listentypen beachtet werden.

Um auch in diesem Fall die Komplexität zu verringern, kann hier für alle drei Ableitungsarten auf den jeweils beteiligten built-in Typen zurückgegriffen werden. Diese lassen sich schneller ermitteln und bedeuten auch bei der direkten Verwendung als Mitgliedstypen für Vereinigungen tieferer Ebenen keinen Informationsverlust. Wie gehabt bietet sich der Datentyp „xs:string“ als allgemeinste Alternative an.

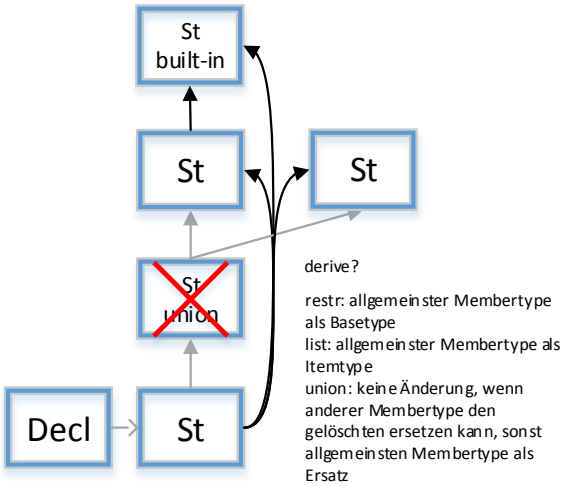


Abbildung 4.13.: Gelöschter Typ ist Knoten (Union).

Fall e2.3: der gelöschte Typ ist Wurzel eines Hierarchiepfades Die dritte mögliche Position eines einfachen Typen ist an der „Wurzel“ einer Hierarchie. Diese wird hier so aufgefasst, dass der betrachtete Typ direkt von einem built-in Datentyp abgeleitet ist und von weiteren nutzerdefinierten einfachen Typen als Basistyp verwendet wird. Bei der Entfernung des Typen sind einerseits die Deklarationen zu betrachten, die diesen Typ direkt referenzieren. Dieser Aspekt gleicht dem in Fall e2.1 beschriebenen, sodass auf die bereits vorgeschlagenen Lösungen für die verschiedenen Ableitungsarten verwiesen ist. Andererseits muss auf die Deklaration geachtet werden, die abgeleitete Typen auf verschiedenen Ebenen referenzieren. Vergleicht man diese Aufgabe mit der, die hauptsächlicher Bestandteil von Fall e2.2 ist, lässt sich auch in diesem Aspekt eine Übereinstimmung feststellen. Die Analyse mitsamt der Lösungsvorschläge kann daher hier für

den zweiten Aspekt angewendet werden. Einziger Unterschied ist der, dass keine anderen, nutzerdefinierten Typen zwischen dem gelöschten Typ und dem ursprünglichen built-in Datentypen liegen. Damit verringern sich etwaige Typvergleiche zur Ermittlung eines neuen Basis- oder Mitgliedstypen sofort auf den Vergleich der beteiligten built-in Datentypen. Sind diese ungenügend, wird auch in diesem Fall „xs:string“ als neue Basis für entsprechende Ableitungen genutzt.

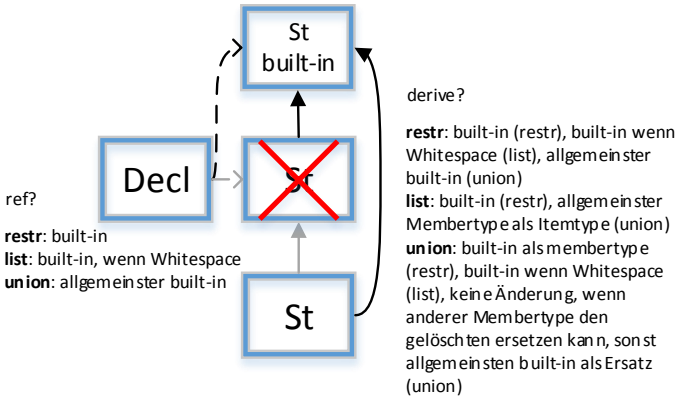


Abbildung 4.14.: Gelöschter Typ ist Wurzel.

4.2.4. Einfügen von komplexen Typen

Die Einfügeopeartion unterliegt ähnlichen Bedingungen, die auch für das Einfügen von einfachen Typen gelten (siehe 4.2.1). Ausführungen über den Inhalt komplexer Typen beziehen sich in diesem und den folgenden Unterabschnitten auf die komplexe Variante, da die einfache Variante bereits durch die Untersuchung von einfachen Typen behandelt wurde. Überlegungen über verwendete Attribute schließen beide Möglichkeiten ein. Es treten keine Komplikationen auf, wenn ein neuer komplexer Typ direkt vom Typ „xs:anyType“ abgeleitet wird, da hier keine Hierarchiebeziehungen vorliegen. Existiert ein Hierarchiepfad, in dem eine neue Definition integriert werden soll, sind erneut drei Einfügepositionen zu beachten.

Soll der erstellte Typ die neue **Wurzel** sein, muss die Hierarchie in Richtung der Untertypen untersucht werden. Von Interesse sind einerseits die direkt und indirekt einschränkenden Untertypen, da das Inhaltsmodell explizit vorliegt. Ein Vergleich des Inhaltsmodells des neuen Wurzel-Typs mit solchen Untertypen muss zeigen, dass diese weiterhin gültige Einschränkungen sind. Das gleiche trifft auch für die definierten Attribute zu, allerdings sind nur jene einschränkende Untertypen von Bedeutung, die ihrerseits expliziten Bezug auf zuvor definierte Attribute nehmen. Andererseits sind ebenfalls erweiternde Untertypen für das Einfügen zu betrachten, da Widersprüche im Inhaltsmodell und bezüglich der verwendeten Attribute auftreten können.

Diese Untersuchung ist ebenfalls durchzuführen, falls der neue Typ als **innerer Knoten** eingefügt wird. Zusätzlich muss hier noch die Richtung entlang des Obertypen verfolgt werden. Es dürfen keine Inhaltsmodellerweiterungen von erweiternden Obertypen fehlen. Ebenso darf das Inhaltsmodell des eingefügten Typen nicht allgemeiner als das vorheriger Einschränkungen sein. Daneben dürfen auch hier explizit verwendete Attribute nicht im Konflikt mit den (expliziten und impliziten) Attributvorkommen der Obertypen stehen. Dies gilt auch für Typen, die am Ende einer Hierarchie als **Blatt** stehen.

Für das Einfügen eines komplexen Typen sind die gegebenen Hierarchiebedingungen also erneut entscheidend. Letzten Endes kann das Einfügen in eine Hierarchie wieder als Manipulation existierender Typen interpretiert werden bzw. diese erfordern. Einerseits muss der Basistyp existierender Untertypen umgesetzt werden und andererseits müssen Änderungen am Inhaltsmodell

oder an Attributen vorgenommen werden, um das Einfügen umzusetzen. Soll der eingefügte Typ zusätzlich einen anderen ersetzen, ist weiterhin eine Löschoperation nötig. Auf diese Aspekte gehen die Abschnitte 4.2.5 sowie 4.2.6 ein.

4.2.5. Aktualisieren von komplexen Typen

Die Betrachtung der Aktualisierungsmöglichkeiten orientiert sich wie schon bei den einfachen Typen an den verschiedenen Eigenschaften und Unterkomponenten komplexer Typdefinitionen. Es wird mit den fundamentalen Bestandteilen, den Attributen und Elementen, begonnen und bis zu den Typattributen selbst fortgefahren. Mögliche Änderungen werden dabei jeweils in Bezug zu möglichen Hierarchiebedingungen gestellt. Als Abkürzung für erweiternde (E) und einschränkende (R) direkte und indirekte Ober- bzw. Untertypen sowie den veränderten Typen wird die Notation $\{T, OT, UT\}_{\{R,E\}}$ verwendet. Zum Beispiel bedeutet OT_E, dass ein erweiternder Obertyp betrachtet wird.

Einfügen von Attributen Es existierenden (spezielle) Kombinationsmöglichkeiten für die Angaben über die verschiedenen Attributeigenschaften wie das use- oder fixed-Attribut. Diese werden im nächsten Punkt betrachtet. Generell wirkt sich das Einfügen von Attributen in einen erweiternden Typen T_E implizit sowohl auf abgeleitete UT_E als auch UT_R aus. Im Allgemeinen gilt hier der Vorsatz, dass das Attribut nicht bereits in der Wurzel oder einem OT_E definiert sein darf. Ebeneso darf das Attribut nicht in einem UT_E definiert worden sein. Ist der bearbeitete Typ eine Einschränkung T_R, so ist ein eingefügtes Attribut prinzipiell nur erlaubt, wenn es ebenfalls in der Wurzel oder einer Erweiterung definiert wurde oder eine Attribt-Wildcard vorliegt. Soll das Einfügen in eine Einschränkung bei zuvor nicht existierendem Attribut tatsächlich durchgeführt werden, müsste im nächsten OT_E bzw. der Wurzel ein entsprechendes Attribut eingefügt werden. Bezüglich der Instanzen ergeben sich nur dann Anpassungen, wenn Attribute mit „use=required“ eingefügt werden. Entsprechend müssten alle Elemente angepasst werden, die sowohl den manipulierten Typen als auch Untertypen von diesem realisieren.

Ändern von Attributen Nachfolgende Unterparagraphen untersuchen die möglichen Eigenschaften von Attributen sowie die Auswirkung ihrer Einfügung, Änderung und Löschung auf Typdefinitionen in Hierarchien.

default- und fixed-Attribut Sowohl das Einfügen und Ändern als auch das Löschen des default-Attributs erzeugen keine Instanzanpassungen. Für die Gültigkeit ist es allerdings entscheidend, dass das enthaltende Attribut nur optional ist. Weiterhin darf das „fixed“-Attribut im entsprechenden Attribut eines Obertyp noch nicht verwendet worden sein, sollte der default-Wert in das Attribut eines UT_R eingefügt werden.

Das Einfügen bzw. das Ändern des Wertes des fixed-Attributs wirkt sich immer auf die Instanzen aus, während das Löschen ohne Weiteres durchführbar ist. Zudem darf das Attribut nicht bereits „prohibited“ sein. Das Ändern des Wertes ist nicht erlaubt, wenn in einem Obertyp der Wert bereits fixiert ist. Dementsprechend muss beim Einfügen des Attributs sowohl auf ein eventuelles Vorkommen in einem OT_R als auch in einem UT_R geachtet werden.

form-Attribut Die Verwendung dieses Attributs kann prinzipiell Instanzanpassungen in der Form des Hinzufügens oder Entfernens von Namensraum-Präfixen hervorrufen. Beschränkungen durch die Typhierarchie entstehen dabei nicht.

name- und ref-Attribut Wird der Name das Attributs gewechselt, so sind i. A. Instanzanpassungen nötig, es sei denn, eine Attribut-Wildcard existiert. Sollte dies nicht der Fall sein,

ist diese Änderung in allen UT_R, die sich explizit auf das betroffene Attribut beziehen, ebenfalls umzusetzen. Wird der Namenswechsel nicht in einem T_E durchgeführt, sind zusätzlich OT_R sowie der Ort der Einfügung des manipulierten Attributs zu bearbeiten. Derartige Folgeänderungen resultieren in weiteren Instanzanpassungen.

targetNamespace-Attribut Mit diesem Attribut wird eine lokale Attributdeklaration einem (anderen) Zielnamensraum zugeordnet. Gedacht ist dieser Mechanismus dafür, um komplexe Typen von Schemata eines anderen Namensraumes im aktuellen Schema zu nutzen und einzuschränken oder zu erweitern. Die Einfügung in eine übliche Typdefinition wäre daher zwecklos und ungültig. Damit dieses Attribut genutzt werden kann, muss das XML-Schema des entsprechenden Namensraumes in das aktuelle Schema zunächst importiert werden. Ordnet das importierte Schema lokale Attributnamen seinem Namensraum zu (über das attributeFormDefault-Attribut im Schema oder das form-Attribut im Attribut), ist dies im abgeleiteten Typen des betrachteten Schemas ebenfalls zu tun. Dies wird durch die Zuweisung des Zielnamensraums des importierten Schemas zu dem targetNamespace-Attribut des betroffenen lokalen Attributs realisiert. Sind lokale Attributnamen im importierten Schema hingegen unqualifiziert, sind sie keinem Namensraum zugeordnet, sodass dieses Attribut im aktuellen Schema auch nicht verwendet werden darf.

Diese Regelung trifft auch auf weitere Typen im aktuellen Schema zu, die von Typen des importierten Schemas über mehrere Ableitungsschritte hinweg indirekt abgeleitet sind. Weiterhin gelten die gewöhnlichen Bedingungen für Änderungen von Attributen, die einen Blick in die Typhierarchie erfordern. Operationen, die eine Modifikation des Obertyps erfordern, sind allerdings nicht mehr umsetzbar.

type-Attribut Liegt noch keine Typangabe in einem Attribut vor, muss beim Einfügen darauf geachtet werden, dass der Typ kompatibel zu existierenden fixed- oder default-Werten ist. Default-Werte eines Attributs, die ausschließlich in einem Obertyp vorliegen, sind jedoch von keiner Bedeutung für die Typangabe des Attributs in einem Untertypen. Bei einer Änderung des Typen muss sichergestellt sein, dass der neue Typ selbst weiterhin ein Untertyp des zum Attribut zugeweisenen Typ in OT ist. Außerdem muss er ein Obertyp bezüglich UT sein. Es entstehen also keine indirekten Instanzanpassungen. Wird der Wertebereich des Attributs durch die Typänderung in T selbst verkleinert, folgt für dieses Typattribut ein Informationsverlust. Das Löschen des type-Attributs entspricht dem Vergrößern des Wertebereichs auf den größt möglichen Fall und unterliegt somit den gleichen Bedingungen wie das Ändern.

use-Attribut Standardmäßig ist die Verwendung eines Attributs optional. Wird diese Eigenschaft in einem T_E oder der Wurzel verändert, muss diese Angabe konform mit Einschränkungen auf „required“ oder „prohibited“ in UT_R sein. Gleiches gilt für die Änderung in einem T_R selbst. Das Attribut muss so verwendet werden, dass keine weiteren Untertypen UT_R eine gegensätzliche Einschränkung beschreiben. In Richtung der Obertypen gelten diese Aussagen analog. Als Ausnahme darf ein Attribut in einem T_E mit beliebigem use-Wert neu eingefügt werden, sollte im letzten vorhergehenden OT_R der use-Wert auf „prohibited“ gesetzt worden sein. Für die Instanzen ist bei einem Wechsel von „optional“ auf „prohibited“ mit hohem Informationsverlust zu rechnen. Bei dem Wechsel hin zu „required“ müssen neue Komponenten eingefügt werden. Dies betrifft vor allem Instanzelemente bzw. deren Attribute, die ein Typ oder Untertypen von diesem realisieren, bei welchem das Auftreten der Attribute erstmals durch das Einfügen eingeschränkt wird. Gleiches gilt bei einem Austausch von „required“ mit „prohibited“ oder umgekehrt. Das Löschen gleicht dem Setzen auf „optional“ und muss daher auch mit den Einschränkungen von Ober- und Untertypen abgeglichen werden. Sofern dies gültig ist, sind für Instanzen dabei aber keine Anpassungen nötig.

inheritable-Attribut Durch das Einfügen dieses Attributs entstehen keine Instanzanpassungen, was auch für das Ändern des Wertes von „false“ auf „true“ gilt. Anders sieht es aus, wenn in der Definition von „true“ auf „false“ gewechselt wird oder das Attribut mit dem Wert „true“ gelöscht wird. Sollten etwa in Typalternativen Bezug auf vererbte Attribute genommen werden, würde die bedingte Zuweisung bei solchen Änderungen fehlschlagen und bislang gültige Dokumente nicht mehr validiert werden, was auch abgeleitete Typen betrifft. Außerdem darf entlang eines Hierarchiepfades ein einmal gesetzter Wert nicht durch einen UT_R verändert werden.

Löschen von Attributen Attribute können nur in Erweiterungen bzw. der Ursprungsdefinition gelöscht werden. Beträgt der use-Wert „prohibited“ entsteht kein Informationsverlust im Gegensatz zum Wert „required“ oder „optional“ (wenn das Attribut in Instanzen genutzt wird). In der abgehenden Hierarchie müssen Vorkommen in UT_R gelöscht werden, die das betrachtete Attribut explizit verwenden. Ausnahme bildet auch hier wieder das Vorkommen einer Attribut-Wildcard im Obertyp. Sofern die verwendeten Namensräume der Wildcard auch das gelöschte Attribut enthalten, wären Anpassungen der Instanzen nicht erforderlich. Ansonsten sind erneut jene Deklarationen bzw. Referenzen in den Dokumenten zu entfernen, die Typen der betroffenen Hierarchie verwenden.

Ändern von Attributgruppen Das Einfügen, Ändern und Löschen von Attributgruppen innerhalb einer Typdefinition kann als Zusammenfassung der jeweiligen Operation auf bzw. mit einem einzigen Attribut gesehen werden, sodass hier auf die drei vorherigen Paragraphen verwiesen ist. Als Besonderheit besteht der Fall, in welchem die Attributgruppe manipuliert wird, die als Default-Attributgruppe im Schema per defaultAttributes definiert wurde. Änderungen an jener würden so sämtliche Typen und somit möglicherweise weite Teile der Dokumente betreffen, in denen das defaultAttributesApply-Attribut auf „true“ gesetzt ist (Standardfall).

Einfügen von Elementen Das Einfügen darf prinzipiell nicht ausgeführt werden, wenn es im bearbeiteten Typ zu uneindeutigem Inhalt käme. Das heißt, dass in Richtung der Ober- und Untertypen auf bereits vorliegende Elementnamen geachtet werden muss. Wird ein Element in ein T_E oder die Wurzel eingefügt, so muss diese Änderung zunächst in alle UT_R übertragen werden. Um das Einfügen neuer Elemente in einen T_R zu ermöglichen, muss entsprechend umgekehrt in Richtung der Obertypen derjenige OT_E oder die Wurzel aufgesucht werden, dessen Kompositor betroffen ist. Das eingefügte Element ist dann hier zu übertragen, womit erneut weitere UT_R zu bearbeiten sind. Als Ausnahme ergibt sich die Situation, in welcher die Kardinalität [0..0] beträgt, was die Gültigkeit nicht verletzt. Ob Instanzanpassungen notwendig sind, entscheidet der übergeordnete Kompositor sowie die Kardinalität des Elements. Ist dieses optional, sind Instanzanpassungen nicht erforderlich. Bei nicht gegebener Optionalität ist das Anpassen nur dann nötig, wenn das Element unter dem <sequence>- oder <all>-Kompositor eingefügt wurde. Für den ersten Fall ist beim Bearbeiten der betroffenen Inhaltsmodelle die Position zu beachten. Üblicherweise sind neben den Elementinhalten, die den bearbeiteten Typ darstellen, auch die Inhalte der Elemente zu ergänzen, die jeweilige Untertypen repräsentieren.

Ändern von Elementen Nachfolgende Unterparagraphen untersuchen die möglichen Eigenschaften von Elementen sowie die Auswirkung ihrer Einfügung, Änderung und Löschung auf Typdefinitionen in Hierarchien.

abstract-Attribut Der Wert ist standardmäßig auf „false“ gesetzt. Durch das Setzen auf „true“ dürfen Elementreferenzen der Inhaltsmodelle der gesamten Typhierarchie nicht mehr in Instanzen verwendet werden. Beim Wechsel zurück auf „false“ besteht kein Handlungsbedarf.

block- und final-Attribut Beim Einfügen dieser Attribute muss auf die Elemente geschaut werden, die in Substitutionsgruppen teilnehmen. Verwenden Gruppenmitglieder Typen, deren Ableitungsart in der Gruppenwurzel durch das Einfügen dieser Attribute blockiert wird, wären Instanzen an entsprechenden Stellen nicht mehr gültig. Der Unterschied zwischen beiden Attributen besteht darin, dass beim final-Attribut bereits die Validierung des Schemas fehlschlägt. Die bestehende Typhierarchie ist also gerade in Verbindung mit gesetzten substitutionGroup-Attributen von Bedeutung. Durch das Löschen dieses Attributs sowie das Entfernen von Werten sind keine Komplikationen zu erwarten. Werden neue Angaben hinzugefügt, müssen erneut die Gegebenheiten der Typhierarchie für die Durchführung miteinbezogen werden.

default- und fixed-Attribut Die Regeln folgen denen der entsprechenden Attribute für Attribute, weswegen an dieser Stelle auf Paragraph *4.2.5 default- und fixed-Attribut* verwiesen ist. Die Kardinalität des Elements hat hier allerdings keine Auswirkungen.

form-Attribut Die Verwendung dieses Attributs gleicht ebenfalls der, die bereits in *4.2.5 form-Attribut* beschrieben wurde.

minOccurs- und max-Occurs-Attribute Diese Attribute können in einem T_E oder der Wurzel bereits eingeführt worden sein oder in einem UT_R eingeschränkt werden. In einem einschränkenden Ableitungsschritt darf der minOccurs-Wert prinzipiell nur vergrößert sowie der maxOccurs-Wert verkleinert werden. Der Standardwert beträgt für beide Attribute 1, daher ist sowohl beim Einfügen als auch beim Löschen darauf zu achten, dass in der vorhergehenden und nachfolgenden Hierarchie keine Unstimmigkeiten entstehen. Es wäre denkbar, ähnlich wie das Einfügen von Elementen selbst, auch die Änderung bzw. Einfügung dieser Attribute in Richtung der UT_R beim Manipulieren eines T_E zu propagieren. Beim Manipulieren eines T_R müsste dies in Richtung des definierenden OT_E ebenfalls getan werden. Damit wäre die entsprechende Operation möglich. Bei einer Lockerung der Werte würden keine Instanzanpassung hervorgeruft, bei einer Verschärfung hingegen schon.

name- und ref-Attribut Ähnlich wie die Namensänderung von Attributen (siehe *4.2.5 name- und ref-Attribut*), müssten auch hier alle OT_R bis einschließlich zum Definitionstypen und/oder alle UT_R bearbeitet werden. Die Eindeutigkeit von Inhaltsmodellen darf dadurch nicht verletzt werden. Instanzanpassungen sind nur dann unumgänglich, wenn der maxOccurs-Wert nicht 0 beträgt. Dabei wären erneut alle realisierten Hierarchietypen zu aktualisieren.

nillable-Attribut Da der Standardwert für dieses Attribut „false“ ist, sind bei einem Wechsel auf „true“ keine Änderungen an bestehenden XML-Dokumenten erforderlich. Dieser Wechsel muss in entsprechenden T_E oder dem Wurzeltypen vollzogen oder an diese entlang der vorhergehenden Hierarchie propagiert werden, da in einem einschränkenden Ableitungsschritt dieser Wechsel nicht gestattet ist. Die Rückrichtung ist hingegen erlaubt bzw. wirkt auch ohne Angabe des Attributs, da im direkten UT_R der Wert wieder den Standardwert „false“ annimmt. Sollte also an einem Punkt des Hierarchiepfads von „true“ auf “false“ gewechselt werden, müsste diese Änderung gleichzeitig auf alle weiteren Untertypen übernommen werden, die den Wert selbst noch explizit verwenden. Auf Instanzebene erfordert dies die Modifikation der Komponenten, welche entsprechende Komponenten der Typen realisieren.

substitutionGroup-Attribut Elemente, die dieses Attribut tragen, können anstelle des als Attributwert angegebenen Elements in dessen enthaltenden Inhaltsmodell erscheinen. Das Einfügen bzw. Hinzufügen von weiteren Substitutionsgruppen erzeugt also keine Instanzanpassungen. Auf Schemaebene ist die bestehende Typhierarchie zu beachten, denn der Typ des Elements

muss eine Einschränkung oder Erweiterung sein (sofern nicht blockiert). Umgekehrt entstehen Instanzanpassungen beim Entfernen von Substitutionsgruppen, da das Gruppen-Wurzelelement um alternativen Inhalt gekürzt wird, der sich aus den jeweils zugewiesenen, verschiedenen Typen ergibt. Das Attribut ist in lokalen Elementen oder Referenzen innerhalb einer Typdefinition nicht nutzbar, wodurch sich auch keine Bedingungen für die Verwendung in einer Hierarchie selbst ergeben.

targetNamespace-Attribut Die Ausführungen über die Verwendung dieses Attributs für lokale Elemente gelten analog zu denen, die in Paragraph *4.2.5 targetNamespace-Attribut* über lokale Attribute beschrieben wurden.

type-Attribut Wie schon beim type-Attribut für Attribute (siehe Paragraph *4.2.5 type-Attribut*) darf der angegebene Typ in einem UT_R nur ein Untertyp desjenigen Typs des betrachteten Elements in T sein. Eine Vergrößerung der möglichen Werte ist nicht erlaubt. Neben dem üblichen Werteverlust ist zusätzlicher Informationsverlust zu erwarten, wenn der neue Untertyp kein Mixed Content zulässt. Erläuterungen hierzu sind im Paragraph *4.2.5 mixed-Attribut* zu finden.

Typalternativen Durch das Hinzufügen von weiteren Typalternativen entstehen keine Instanzanpassungen im Gegensatz zum Entfernen bestehender Alternativen oder beim Verschärfen des jeweiligen Testausdrucks. Dennoch ist für das Verwenden der bedingten Typisierung die Hierarchie zu beachten, da als Alternativen ausschließlich vom Standardtyp abgeleitete Typen in Frage kommen. Dementsprechend muss beim Löschen von Typen (siehe Abschnitt 4.2.6) auch auf deren Verwendung als Typalternativen geachtet werden.

Schlüsselbedingungen Schlüssel sichern die Integrität gewisser Werte innerhalb von XML-Dokumenten. Das Einfügen von Komponenten dieser Kategorie erzeugt üblicherweise Instanzanpassungen, sollten existierende Werte nicht schon den Integritätsbedingungen entsprechen. Werden diese Komponenten in Elementen eines Typs einer Hierarchie erstmalig eingefügt oder gelöscht, müssen diese Änderungen wieder bis zum jeweiligen T_E bzw. der Wurzel und weiteren einschränkenden Ober- und Untertypen propagiert werden. Entsprechend viele Instanzkomponenten könnten anschließend eine Ungültigkeit hervorrufen.

Löschen von Elementen Sollte die Kardinalität eines Elements nicht [0..0] betragen, sind unabhängig vom übergeordneten Kompositor Instanzanpassungen zu erwarten, da z. B. bei einer choice-Gruppe nicht davon ausgegangen werden kann, dass das Element nicht verwendet worden ist. Wird das Element in einem T_E oder der Wurzel gelöscht, sind alle UT_R zu bearbeiten. In weiteren UT_E ist diese Änderung implizit vorhanden. Um die Gültigkeit des Schemas zu bewahren, müsste bei einer Löschung in einem T_R entsprechend der OT_E bzw. die Wurzel aufgesucht werden, die das Element in das Inhaltsmodell einführt. Instanzanpassungen folgen also erneut wieder für den betroffenen Hierarchieteil.

Einfügen von Kompositoren Beim Einfügen eines Kompositors wird hier davon ausgegangen, dass dieser ohne neue Elemente hinzugefügt wird, da dies eigene Operationen bedeuten würde. Weiterhin meint das Einfügen hier sowohl das Anfügen eines leeren Kompositors als auch das Umschließen von existierenden Partikeln. Da die Struktur des Inhaltsmodells verändert wird, ist diese Änderung wieder auf weitere Typen der Hierarchie zu übertragen. Nach dem Einfügen eines zusätzlichen Kompositors in ein T_E muss in allen weiteren UT_R am entsprechenden Teil der Kompositor ebenfalls eingefügt werden. Wird stattdessen in ein T_R eingefügt, so muss der OT_E ermittelt werden, dessen Inhaltsmodellteil um den neuen Kompositor ergänzt wird, um

anschließend die Änderung hier auszuführen. Ausgehend von diesem Punkt sind wieder weitere UT_R zu manipulieren. Ob Anpassungen der Instanzen folgen, ist abhängig vom eingefügten und übergeordneten (sofern vorhanden) Kompositor sowie von den Kardinalitäten der beteiligten Partikel.

Sequence in Sequence Wird in eine Sequenz eine weitere eingefügt oder mit dieser umschlossen, darf der „minOccurs"-Wert nicht größer als 1 sein, sonst müssen zusätzliche Elemente in die Dokumenten eingefügt werden. Bei der äußeren Umschließung darf der neue maxOccurs-Wert den alten nicht unterschreiten. Werden hingegen untergeordnete Partikel mit einer neuen Sequenz umfasst, darf die Kardinalität nicht [0..0] sein, sofern mindestens ein Partikel vorliegt, das diese Kardinalität nicht besitzt.

Choice in Sequence Werden Partikel einer Sequenz mit einer Alternative (choice-Kompositor) umfasst, muss ihr maximales Vorkommen die Summe aller maximalen Vorkommen der umschlossenen Partikel ermöglichen, da sonst nicht alle Elemente darstellbar wären. Gleichzeitig sollte das minimale Vorkommen der Alternative diese Zahl nicht übersteigen, um Zusatzeinfügungen von Elementen in XML-Dokumenten zu verhindern.

Sequence in Choice Sollte in umgekehrter Richtung eine Sequenz in eine Alternative eingefügt werden und Partikel zusammenfassen, folgt i. A. immer eine Instanzanpassung, da die effektiven Kombinationsmöglichkeiten des <choice>-Kompositors verringert werden (es sei denn, es ist effektiv nur ein Partikel betroffen).

Choice in Choice Durch die Einschließung von choice-Partikeln mit einer weiteren Alternative entstehen zunächst keine Instanzanpassungen. Hat die innere Alternative einen minOccurs-Wert der größer als 1 ist, kann das Element ehemalige Werte nicht mehr annehmen.

All in All Verschachtelungen mit dem <all>-Kompositor sind (direkt) nicht gestattet. Sollte die all-all-Verschachtelung über den Umweg mit einer <group>-Komponente vollzogen werden, treten dennoch keine Instanzanpassungen auf, es sei denn, die innere <all>-Komponente hat eine Kardinalität von [0..0] und wenigstens eines der zusammengefassten Partikel nicht.

Ändern von Kompositoren Änderungen können am Typ des Kompositors und an der Kardinalität vorgenommen werden. Sollen bei einem solchen Typwechsel keine Instanzanpassungen erzeugt werden, ist es in einigen Fällen unumgänglich, die vorliegende Kardinalität des (neuen) Kompositors im gleichen Schritt mit zu verändern. Ein Wechsel in eine ungültige Verschachtelung durch den <all>-Kompositor ist nicht erlaubt und bei entsprechenden Fällen zusätzlich zu überprüfen. Ähnlich wie beim Einfügen müsste Sorge dafür getragen werden, solche Änderungen in einem T_E oder der Wurzel auszuführen und bei entsprechenden UT_R zu aktualisieren. Instanzanpassungen können so wieder für die betroffenen, realisierten Hierarchieteile nötig sein.

Es sei angemerkt, dass seit XML-Schema 1.1 die Regelungen für erlaubte Ableitungen im Vergleich zu Version 1.0 verändert wurden [1]. Dadurch sind nun eigentlich logisch korrekte Einschränkungen in einem Ableitungsschritt möglich, die zuvor ausgeschlossen waren. So ist es durchaus möglich, dass ein UT_R mit einem <sequence>-Kompositor einen Obertypen mit einem <choice>-Kompositor bei geeigneten Kardinalitäten einschränkt. Da sich die Menge der möglichen Strukturen verkleinert, wäre dies erlaubt. Eine detaillierte Analyse aller Spezialfälle

[1] Die strikte 25-Fall-Tabelle aus http://www.w3.org/TR/xmlschema-1/#coss-particle, welche mögliche Einschränkungen beschrieben hat, wurde durch einen allgemeineren Algorithmus in http://www.w3.org/TR/xmlschema11-1/#sec-derivation-ok-restriction ausgetauscht. Dieser stellt fest, ob die Partikelbindung des Obertypen die des Untertypen zusammenfasst.

würde an dieser Stelle über den Rahmen hinaus gehen, sodass davon ausgegangen wird, dass die Änderung des Kompositors auch die Änderung von Ober- und/oder Untertypen nach sich zieht.

Die anschließend verwendete Notation hat folgende Bedeutung: c - choice, s - sequence, a - all, n - # Partikel im modifizierten Kompositor, min - minOccurs, max - maxOccurs.

Sequence → Choice Wird eine Sequenz mit einer Alternative ausgetauscht, entsteht Informationsverlust, wenn die Kardinalität gleich bleibt. Um jedes Partikel weiterhin mindestens ein mal mit dem maximalen Vorkommen der ursprünglichen Sequenz auswählen zu können, muss für den maxOccurs-Wert der Alternative die Ungleichung $c_{max} \geq s_{max} \cdot n$ gelten. Ebenso sollte der neue minOccurs-Wert die Ungleichung $c_{min} \leq s_{min} \cdot n$ erfüllen, da sonst Elemente hinzugefügt werden müssen.

Sequence → All Das maximal erlaubte Vorkommen einer Menge (<all>-Kompositor) beträgt 1. Sollte also bei der Transformation einer Sequenz zu einer Menge $1 < s_{min} \leq s_{max}$ gelten, sind Anpassungen der Instanzbasis immer erforderlich. Übersteigt das maximale Vorkommen der Sequenz 1 jedoch nicht, sind keine Modifikationen der Dokumente notwendig.

Choice → Sequence Der Wechsel von einer Alternative zu einer Sequenz führt immer zu Informationsverlust. Bleiben die Kardinaltitäten unverändert, wird die Sequenz anschließend i. A. längere Werte erfordern, als die Alternative zuvor zuließ. Daneben gehen die verschiedenen Partikelkombinationen der Alternative verloren. Dies trifft auch trotz der Verringerung der minOccurs- und maxOccurs-Werte der neuen Sequenz zu. Eine Ausnahme dieser Regel ist die Situation, in welcher alle bis auf ein Partikel die Kardinalität [0..0] besitzen oder überhaupt nur ein einziges Partikel im Kompositor vorhanden ist ($n = 1$).

Choice → All Die Kardinalitäten sind auch bei der Transformation einer Alternative zu einer Menge gegenbenfalls anzupassen. Im Allgemeinen ist mit Instanzanpassungen zu rechnen. Als Ausnahme ergibt sich hier der Fall, in welchem die Kardinalität der Alternative vor der Transformation $c_{min} = c_{max} = n$ beträgt.

All → Sequence Das Auswechseln einer Menge mit einer Sequenz erzeugt i. A. immer Informationsverlust, welcher nicht behoben werden kann. Ausnahme ist auch hier wieder der Fall, dass in der Menge effektiv nur ein Partikel vorkommt ($n = 1$).

All → Choice Ohne Modifikation der Kardinalität ist auch der Tausch einer Menge mit einer Alternative mit Informationsverlust behaftet. Dieser lässt sich vermeiden, indem die minOccurs- und maxOccurs-Werte der neuen Alternative auf $c_{min} \leq n \leq c_{max}$ gesetzt werden, sodass weiterhin alle Partikelkombinationen möglich sind.

Außerdem lassen sich die Kardinalitäten selbst bearbeiten. Die Überlegungen dazu entsprechen denen für das Bearbeiten der minOccurs- und maxOccurs-Werte von Elementen, weswegen an dieser Stelle auf Paragraph *4.2.5 minOccurs- und max-Occurs-Attribute* verwiesen wird.

Löschen von Kompositoren Das Löschen eines Kompositors ist so gemeint, dass dieser zwar entfernt wird, seine Kind-Partikel aber erhalten bleiben, indem diese an gleicher Stelle dem nächst höheren Kompositor zugeordnet werden. Das Entfernen eines Kompositors ist unzulässig, wenn dieser der Wurzel-Kompositor ist und wird daher nur für innere bzw. verschachtelte Kompositoren betrachtet. Auch diese Änderungen müssen an die entsprechenden Stellen in der Typhierarchie propagiert werden.

Sequence in Sequence Die Entfernung einer Sequenz aus einer Sequenz erzeugt nur dann eine Veränderung in der Objektstruktur, wenn das maximale Vorkommen der entfernten Sequenz größer als 1 ist und weitere Partikel in der übergeordneten Sequenz vorhanden sind. Liegen hingegen keine anderen Partikel außer der gelöschten Sequenz vor (oder haben jeweils eine Kardinalität von [0..0]), lassen sich deren minOccurs- und maxOccurs-Werte zur übergeordneten Sequenz hinzuaddieren, um die ursprüngliche Struktur wiederzuerlangen.

Choice in Sequence Beim Löschen einer Alternative aus einer Sequenz entscheidet der minOccurs-Wert über die Notwendigkeit von Instanzanpassungen. Beträgt dieser 0 oder 1, können Instanzanpassugnen vermieden werden, indem der minOccurs-Wert aller Kind-Partikel auf 0 gesetzt wird (erfordert einen Vergleich mit Ober- und Untertypen). Sobald c_{min} größer als 1 ist, kann durch den Wegfall an Kombinationsmöglichkeiten die Gültigkeit der Instanzen nicht mehr gesichert werden.

Sequence in Choice Wird hingegen eine Sequenz aus einer Alternative entfernt, ist die Gültigkeit herstellbar, indem der minOccurs- und maxOccurs-Wert der Alternative so manipuliert wird, dass $c_{min} \leq s_{min}$ bzw. $c_{max} \geq s_{max} \cdot n$ gilt. Durch die Lockerung der Kardinalität fallen existierende Partikelkombinationen nicht weg und der Sequenz-Charakter für die betroffenen Partikel bleibt erhalten. Diese Änderung ist nicht erforderlich, wenn die gelöschte Sequenz eine Kardinalität von [1..1] besitzt und nur ein einziges Partikel enthält oder alle Partikel bis auf eines maximal null mal vorkommen können.

Choice in Choice Für das Löschen einer Alternative aus einer Alternative sind Instanzanpassungen vermeidbar, wenn der maxOccurs-Wert der gelöschten zu dem der äußeren addiert wird.

Ändern von Elementgruppen Als Alternative zur direkten Verwendung von Kompositoren innerhalb eines Typen können diese auch mitsamt Inhalt in globale <group>-Komponenten ausgelagert werden. Mögliche Modifikationen wie das Einfügen, Ändern und Löschen von weiteren Kompositoren oder Elementen sind rückführbar auf die Modifikation entsprechender Komponenten, die direkt in einer Typdefinition genutzt werden, womit sich bereits die Paragraphen *4.2.5 Einfügen von Kompositoren*, *4.2.5 Ändern von Kompositoren* und *4.2.5 Löschen von Kompositoren* auseinandersetzen. Mögliche Komplikationen wirken sich hier nicht nur lokal in einer Hierarchie aus, sondern können auch weitere Schemateile umfassen.

Ändern von Wildcards Durch das Einfügen einer Wildcard in eine Typdefinition sind prinzipiell mehr Attribut- und Elementkombinationen innerhalb der Instanzen in entsprechenden Elementen möglich. Die Propagation des Einfügens muss nicht unbedingt mittels weiteren (u. U. eingeschränkten) Wildcards geschehen. Der Grund hierfür ist, dass sich Wildcards insofern auf abgeleitete UT_R auswirken, dass auch Attribute und Elemente angegeben bzw. eingeschränkt werden können, die nicht explizit im bearbeiteten T_E bzw. der Wurzel definiert wurden. In dieser Situation ist beim Löschen einer Wildcard mit der Verletzung der Gültigkeit des Schemas zu rechnen. Es wäre denkbar, alle Elemente und Attribute in Untertypen, die nach der Löschung einer Wildcard kein Pendant im Obertyp besitzen ebenfalls zu löschen, die Änderung also wieder entlang der Hierarchie auszuführen. Dies ist auch im allgemeinen Fall erforderlich, in welchem die Wildcard üblicherweise in abgeleiteten Inhaltsmodellen wiederholt wird. In beiden Fällen entstehen somit Instanzanpassungen für das Löschen. Wildcards bieten weiterhin einige Ansatzpunkte zur Änderung. Der allgemeinste Wert für das namespace-Attribut ist „##any“. Wird dieser auf ‘##other“ gesetzt, können beispielsweise keine Komponenten des Zielnamensraumes mehr erscheinen. Ebenso ist die detaillierte Angabe von URIs bzw. von ‘##targetNamespace“ und

‘##local“ potentiell mehr einschränkend. Die letzten drei Werte sind kombinierbar, sodass für jeden Wert (bzw. für jede weitere URI) eine Wertebereichsvergrößerung entsteht. Genau in die andere Richtung wirkt das Attribut notNamespace. Mit diesem sind explizite QNamen sowie globale Deklarationen („##defined“) und bereits im Inhaltsmodell vorhandene Elemente („##definedSiblings“) ausschließbar. Je mehr Angaben für dieses Attribut getätigt werden, desto kleiner wird der Wertebereich für die entsprechende Wildcard. Obwohl durch das processContents-Attribut keine Erweiterungen oder Einschränkungen im klassischen Sinne entstehen, kann sich die Anzahl an gültigen Dokumenten verringern. Weiterhin sind für Element-Wildcards Kardinalitäten bestimmbar.

Wird eine Wildcard durch einen UT_R wiederholt bzw. durch eine weitere konkrete Wildcard eingeschränkt, müssen die Werte aller beschriebenen Attribute mit denen der Wildcard im Obertyp abgeglichen werden. So darf im UT_R durch das namespace-Attribut keine Wertebereichsvergrößerung entstehen. Andererseits ist die Lockerung der Angaben in den notNamespacesowie notQName-Attributen ebenfalls nicht erlaubt. Dies trifft auch auf das processContents-Attribut zu, wobei die Rangfolge $strict > lax > skip$ gilt. Die Regeln für die Element-Wildcard-Kardinalität orientieren sich an denen der Partikel-Kardinalität (siehe Paragraph *4.2.5 minOccurs- und max-Occurs-Attribute*). Sollte zudem eine Wildcard, wie oben beschrieben, in einem Ableitungsschritt durch eine explizite Komponente ersetzt (Element) bzw. genutzt (Attribut) werden, muss deren Namensraum und Name ebenfalls mit der betroffenen Wildcard verglichen werden. Schließt die Wildcard diese Angaben aus, so ist jene Änderung bzw. Ableitung unzulässig.

Eine Besonderheit ergibt sich, wenn mehrere Attribut-Wildcards durch verschiedene Attributgruppen-Referenzen in einem T verwendet werden. In diesem Fall werden Angaben über erlaubte und verbotene Namensräume sowie QNamen durch die Schnittoperation in eine vollständige Wildcard zusammengeführt. Der Wert für das processContents-Attribut wird dabei von der ersten Wildcard entnommen. Definiert T selbst noch eine Attribut-Wildcard hat deren processContents-Attribut Vorrang und ihre Angaben werden ebenfalls mit weiteren Wildcards geschnitten. Diese vollständige Wildcard wird anschließend für weitere UT_R und UT_E direkt genutzt. Sollten die UT_E jedoch weitere Attribut-Wildcards hinzufügen, entsteht die neue vollständige Wildcard durch die Vereinigung entsprechender Namensraumwerte und QNamen der vorherigen vollständigen Wildcard und der in UT_E direkt geschnittenen Werte. Das processContents-Attribut wird dabei nach obiger Methode vom UT_E entnommen. Gerade bei unüberlegter Nutzung von Attribut-Wildcards durch Einfügung und Veränderung mehrerer Attributgruppen-Referenzen können somit leere Attributmengen entstehen, die sich über einen gesamten Hierarchiepfad erstrecken.

Ändern von Erweiterbaren Inhalt Mit der <openContent>-Komponente lassen sich Typen mit offenen Inhaltsmodellen erzeugen. Im Grunde bedeutet das, dass quasi Element-Wildcards an beliebigen Stellen des Inhaltsmodells vorliegen. Abgeleitete UT_R können solche Typen wieder durch die Angabe expliziter Partikel einschränken. Diese Komponente ist daher ideal, um mehrere verschiedene Erweiterungspunkte innerhalb eines Inhaltsmodells für abgeleitete Typen zu realisieren. Wichtig dabei ist, dass der Inhalt eines T_R nicht offen sein darf, wenn es der Inhalt des Obertypen nicht ist. Der Ort der Einfügung dieser Komponente muss daher immer die Wurzel oder ein T_E sein. Die Fähigkeit für offenen Inhalt wird nur für UT_E vererbt, während die Komponente für UT_R separat angegeben werden muss. Soll also ein UT_R einer Hierarchie erstmals mit dieser Fähigkeit ausgestattet werden, müsste entsprechende Komponente bis zum nächsten OT_E oder der Wurzel übertragen werden.

Da die möglichen Erweiterungen letzten Endes über eine Element-Wildcard innerhalb der <openContent>-Komponente gesteuert werden, können bei Änderungen an der Wildcard wie im vorherigen Paragraph *4.2.5 Ändern von Wildcards* erläutert Instanzanpassungen sowie Gültigkeitsverletzungen auftreten. Daneben entscheided das mode-Attribut darüber, an welchen Stellen

der Inhalt erweiterbar ist. „interleave“ (Standardwert) erlaubt die beliebige Einfügung während „suffix“ das Ende des Inhaltsmodells vorsieht. In einem einschränkenden Ableitungsschritt darf dabei i. A. nicht von „suffix“ auf „interleave“ gewechselt werden (Verallgemeinerung). Gleichermaßen ist in einer Erweiterung der Wechsel in die andere Richtung unzulässig (Verschärfung).

Standardmäßiger erweiterbarer Inhalt In Verbindung mit einer Wildcard-Unterkomponente lässt sich zudem anhand der <defaultOpenContent>-Komponente die Erweiterbarkeit aller komplexer Typen direkt auf (globaler) Schemaebene steuern. Das applyToEmpty-Attribut gibt an, ob auch Typen mit leerem Inhalt betroffen sein sollen. Für eine Typhierarchie wird diese Komponente dann interessant, wenn die Erweiterbarkeit in einem Typen T explizit ausgeschlossen werden soll. Dies geschieht über das Setzen des mode-Attributs der <openContent>-Komponente auf „none“. Befindet sich T in einer Hierarchie, sind normalerweise weitere UT_R standardmäßig noch erweiterbar, was ungültig ist. Um diese Änderung durchzuführen, muss die Fähigkeit für erweiterbaren Inhalt in jenen UT_R ebenfalls ausgeschlossen werden. Dies ist dabei nur bis zum nächsten UT_E nötig, da in Erweiterungen die (implizite) Verallgemeinerung erlaubt ist.

Weiterhin darf der Standardmodus und die Wildcard in einem T_E verallgemeinert und in einem T_R durch die explizite Verwendung der <openContent>-Komponente verschärft werden, aber nicht umgekehrt. Dabei müssen einerseits die Angaben erneut mit eventuellen <openContent>-Komponenten von Ober- und Untertypen verglichen werden, sofern vorhanden. Wenn in weiteren UT_R diese Komponente nicht vorhanden ist, muss sie auf diese bis zum nächsten UT_E übertragen werden, da für den Hierarchieteil die lokale <openContent>-Komponente die höhere Priorität besitzt. Ein UT_E nimmt bei Nicht-Vorhandensein der <openConten>-Komponente erneut den Standardmodus an. Sollte dieser strikter sein als der Modus der <openContent>-Komponente im Obertyp von UT_E, muss jedoch entsprechende Komponente auch auf den UT_E sowie weitere Untertypen von diesem übertragen werden.

Ändern von Zusicherungen Auf Schemaebene können <assert>-Komponenten im Prinzip beliebig eingefügt, verändert und gelöscht werden. Beim Einfügen und Verändern (sofern die Bedingungen durch den Testausdruck verschärft werden) ist es wahrscheinlich, dass Instanzanpassungen nötig sind. Da Zusicherungen über die Ableitungsschritte hinweg verundet werden, kann eine Änderung sich wieder mehrfach auf die XML-Dokumente auswirken. Im Gegensatz dazu provoziert das Löschen keine Modifikationen der Instanzen.

Ändern der Ableitungsart Die Ableitungsart ist ein weiterer Aspekt eines komplexen Typen, der verändert werden kann. Die Angabe der <restriction>- und <extension>-Komponenten ist verpflichtend, sodass für die Untersuchung nur eine Änderung in Betracht kommt. Diese wird so aufgefasst, dass tatsächlich nur eine Erweiterung in eine Einschränkung oder umgekehrt gewandelt wird, der Basistyp aber unverändert bleibt. Anschließend wird zwischen komplexen Typen mit einfachem und komplexem Inhalt unterschieden.

Einfacher Inhalt Gegenstand sind hier Facetten und Attribute. Ändert man eine **Einschränkung in eine Erweiterung**, fallen lediglich Einschränkungen weg. Da sich der Wertebereich vergrößert, folgen keine Instanzanpassungen. Es ist darauf zu achten, dass in der nachfolgenden Hierarchie kein Attribut durch einen UT_E neu eingeführt wird, dessen Nutzung im umgewandelten T_R ursprünglich auf „prohibited“ gesetzt wurde, da das Attribut sonst doppelt vorliegt. Anschließend sind Aktualisierungen durch das Einfügen von Attributen möglich, was jedoch eine separate (Folge-)Änderung ist und bereits in Paragraph *4.2.5 Einfügen von Attributen* untersucht wurde.

Die Rückrichtung in der Form des Wechsels einer **Erweiterung in eine Einschränkung** ist mit Informationsverlust behaftet, sofern Attribute in dem T_E definiert waren. Diese wären, je

nachdem was mit dieser Operation erzielt werden soll, zu entfernen oder auf den nächst höheren OT_E oder die Wurzel zu übertragen. Im Fall des Entfernens ist anschließend die Löschung der Attribute in solchen UT_R zu propagieren, die das Attribut explizit verwenden. War das Attribut nicht „prohibited“, müssen wieder die Instanzen an den Typen entsprechenden Teilen modifiziert werden. Zudem muss der Inhalt der Typdefinition des nächst höheren OT_R als neuer Inhalt eingefügt werden. Weitere Einschränkungen durch die Angabe in bzw. von Facetten unterliegen den Bedingungen, die Paragraph *4.2.2 Ändern von Facetten* schon erläutert hat.

Komplexer Inhalt Die Überlegungen für komplexe Typen mit komplexem Inhalt sind ähnlich. Da die Attribut-Mechanik zudem identisch ist, wird diese nicht noch einmal betrachtet. Auch hier entsteht bei der Umwandlung einer **Einschränkung in eine Erweiterung** kein Informationsverlust. Da Elemente und Kompositoren die zentrale Rolle spielen, müssen bei einer eventuellen Nachbearbeitung durch das Einfügen dieser Komponenten wieder auf die korrekte Verwendung geachtet sowie Schemaanteile angepasst werden, wie u. a. in den Paragraphen *4.2.5 Ändern von Elementen* und *4.2.5 Ändern von Kompositoren* erläutert.

Wird dagegen eine **Erweiterung in eine Einschränkung** transformiert, fällt i. A. ein Teil des Inhaltsmodells der Untertypen weg. Prinzipiell entspricht dies also zunächst dem Löschen von Partikeln in einem T_E einer Typhierarchie, was in Paragraph *4.2.5 Löschen von Elementen* und *4.2.5 Löschen von Kompositoren* beschrieben wurde. Da die Erweiterung für komplexe Inhalte neben Attributen auch Partikel enthalten kann, müssen in dem neu entstandenen T_R die Inhaltsmodellerweiterungen durch OT_E „eingesammelt“ werden und mit dem expliziten Inhaltsmodell des nächsten OT_R korrekt kombiniert werden.

Ändern der Inhaltsart Befindet sich ein Typ in einer Hierarchie, darf die Art seines Inhalts nicht verändert werden. Ist dies jedoch nicht der Fall, müssen bei einem Wechsel von **einfachen zu komplexen** Inhalt sämtliche Facetten entfernt werden. Da es sich bei dem bearbeiteten Typen so nur um eine Erweiterung eines built-in Datentypen handeln kann, ist diese zudem in eine Einschränkung von „xs:anyType“ zu tauschen, da sonst später keine Elemente eingefügt werden können und die Definition von komplexen Inhalt auf Basis eines einfachen Typen generell ungültig ist. Um existierende Textinhalte von Elementen in den bestehenden XML-Dokumenten weiterhin zu ermöglichen, sollte das mixed-Attribut auf „true“ gesetzt werden.

Im Allgemeinen führt der Tausch von **komplexen mit einfachen Inhalt** zu Instanzanpassungen, da alle Partikel zu entfernen sind. Ausnahme ist wieder der Fall, in welchem keine Typ-Partikel in den XML-Dokumenten erscheinen können. Dies kann durch das Vorliegen der Kardinalität [0..0] für die jeweiligen Komponenten entstehen oder wenn z. B. ausschließlich abstrakte Elemente im Inhaltsmodell vorhanden sind. Im Prinzip wäre so ein Typ gar nicht nutzbar (im Falle von referenzierten abstrakten Elementen) bzw. wären durch diesen keinerlei Aussagen getroffen, sodass überhaupt keine Informationen vorliegen. Der sinnvollste Anwendungsfall dieser Operation wäre also der, wenn das mixed-Attribut zuvor auf „true“ gesetzt war, sodass wenigstens Textinhalte möglich sind. Bei dem Wechsel ist dieses Attribut aus Gründen der Schemagültigkeit zu entfernen. Weiterhin ist für Typen mit einfachen Inhalt die Einschränkung von „xs:anyType“ nicht erlaubt. Um die Gültigkeit zu wahren und eventuelle Textinhalte weiterhin zu ermöglichen, sollte die Einschränkung in eine Erweiterung von „xs:string“ gewandelt werden.

Sämtliche angegebenen Attribute können bei dem Wechsel in beide Richtungen bestehen bleiben, da es sich auf jeden Fall um einen „definierenden“ komplexen Typen handelt.

Ändern des Obertypen Die Änderung des Obertypen soll wieder so verstanden werden, dass neben dem bearbeiteten Typen auch mit diesem seine abgehende Hierarchie umgesetzt wird. Das heißt, es wird nur das baseType-Attribut bearbeitet. Es folgt anschließend eine Unterscheidung zwischen einfachen und komplexen Inhalt.

Einfacher Inhalt Wird der Basistyp eines Typen verändert, muss auf Schemaebene lediglich das entsprechende Attribut verändert werden. Das Vorgehen, um zu bestimmen, ob Instanzanpassungen bzw. ob der neue Basistyp BT_N überhaupt erlaubt ist, gleicht dem Vorgehen aus Paragraph *4.2.2 Ändern des Obertypen*. Im Wesentlichen müssen die Facetten von Ober- und Untertypen in der Hierarchie sowie von T, falls dieser eine Einschränkung ist, verglichen werden. Beschreibt der neue Basistyp einen kleineren Wertebereich, sind Instanzanpassungen bis hin zu den Blatt-Typen nicht auszuschließen, sodass die Operation evtl. nicht ausgeführt werden sollte. Zusätzlich dazu dürfen einerseits doppelte Attributvorkommen nicht auftreten. Andererseits wären Attribute, die in der ehemaligen, vorhergehenden Hierarchie eingeführt wurden, aus solchen UT_R zu entfernen, die diese noch explizit verwenden. Da dies i. A. mit einem Informationsverlust einher geht, ist ein Ablehnen der Aktualisierung auch hier angemessen. Können fehlende Attribute durch eine Attribut-Wildcard ersetzt werden, entstehen keine Komplikationen. Auf <assert>-Komponenten muss aus Sicht der Schemagültigkeit keine Rücksicht genommen werden, jedoch bergen auch diese das Potenzial für Instanzanpassungen, was auch für den nächsten Paragraph gilt.

Komplexer Inhalt Ähnlich sieht es für Typen mit komplexen Inhalt aus. Sollte der neue Basistyp nicht zufällig ein exakt passendes Inhaltsmodell beschreiben, sind Instanzanpassungen die Folge. Diese entstehen durch die notwendige Veränderung bzw. Anpassung der Inhaltsmodelle. Es sind sowohl die einschränkenden und erweiternden (hier allerdings impliziten) Unterypen als auch der bearbeitete Typ selbst betroffen. Diese Manipulationen können die Veränderung einfacher Werte für Attribute wie das fixed-, minOccurs- oder maxOccurs-Attribut bedeuten, aber auch die Entfernung ganzer Strukturteile in der Form von Partikeln umfassen. Im schlimmsten Fall würde also durch diese Änderung eine komplett andere Objektstruktur für alle Typen einer Hierarchie ab dem modifizierten Typ beschrieben. Es wäre daher ratsam, auch hier die Eignung des neuen Basistypen zu testen und bei negativem Resultat diese Operation abzulehnen.

Das Vorgehen könnte ähnlich aussehen, wie das zur Bestimmung der Eignung eines Typen als neuen Basistyp für einfache Typen. Ist der bearbeitete Typ eine **Einschränkung**, liegt das ursprüngliche Inhaltsmodell direkt vor. Wenn BT_N ebenfalls ein Einschränkung ist, können beide Inhaltsmodelle sofort verglichen werden. Da die Vorschriften für die Ableitung komplexer Typen relativ streng bzw. klar sind, muss also im Wesentlichen auf die syntaktische Anordnung von Partikeln sowie deren Attributwerte geschaut werden. Für Sequenzen muss die Anordnung erhalten bleiben, während sie für die Alternative und die Menge ignoriert werden kann. Ebenso haben Partikel mit einem maxOccurs-Wert von 0 keinen Effekt und können in beide Richtungen ignoriert werden. Sollten Elemente in BT_N fehlen, kann dies u. U. durch das Vorhandensein von Element-Wildcards an entsprechenden Stellen kompensiert werden. Weiterhin müssen verwendete Typalternativen und Schlüssel praktisch identisch sein. Bei den Attributwerten gilt wieder die Prämisse, dass in BT_N nicht strengere Angaben als in T_R stehen dürfen. Durch die Nutzung solcher Typen käme es mitunter erneut zu Anpassungen von Dokumentteilen, die Typen der Hierarchie ab dem veränderten Typen darstellen. Dieser Vergleich muss auch für BT_N durchgeführt werden, wenn dieser eine Erweiterung ist. Das Inhaltsmodell liegt hier nicht explizit vor, weswegen dieses vor dem Vergleich intern mittels Rückverfolgung der Oberhierarchie von BT_N bis zum nächsten OT_R zusammengesetzt werden muss.

Dies ist zudem nötig, wenn der manipulierte Typ selbst eine **Erweiterung** ist. Nach der Zusammenfügung des vollständigen Inhaltsmodells von T_E lässt sich derselbe Vergleich mit BT_N realisieren. Unabhängig von der Ableitungsart muss wie im Fall des einfachen Inhalts auch auf die Kompatibilität der vorhandenen Attribute geachtet werden. Als letztes Kriterium ist die Fähigkeit zur Darstellung von gemischtem Inhalt ausschlaggebend. Sollte diese der bearbeitete Typ nicht besitzen, BT_N hingegen schon, scheidet dieser auch bei sonst passendem Inhalt als neuer Basistyp aus.

Ändern von Typ-Attributen Nachfolgend werden die möglichen Attribute der <complexType>-Komponente untersucht.

abstract-Attribut Dieses Attribut wirkt immer nur lokal für eine Typdefinition, unabhängig von seiner Hierarchieposition. Demnach müssen Ober- und Untertypen bei der Veränderung nicht beachtet werden. Der Standardwert ist „false“. Problematisch wird eine Änderung auf „true“ dann, wenn Elemente im Schema den Typen bereits referenzieren, da Instanzen u. U. nicht mehr validierbar sind. Diese müssten so angepasst werden, dass betroffene Elemente das xsi:type-Attribut (aus dem Namensraum „xsi=http://www.w3.org/2001/XMLSchema-instance“) tragen, welchem wiederum der Name einer Ableitung des nun abstrakten Typen zugewiesen wird. Dies zeigt auch gleich das Risiko für die Ungültigkeit von Dokumenten bei dem Wechsel von „true“ auf „false“. Sollte der gezeigte Mechanismus in Dokumenten genutzt worden sein, müssten der Inhalt und/oder die Attribute von Elementen an das Inhaltsmodell des ehemals abstrakten Typen angepasst werden.

block und final-Attribut Für Typen funktionieren die block- und final-Attribute ähnlich wie für Elemente (siehe Paragraph *4.2.5 block- und final-Attribut*), wirken sich allerdings an anderer Stelle aus. Generell bedeutet das Setzen eines dieser Attribute, dass vom enthaltenden Typen in den angegebenen Formen nicht mehr abgeleitet werden darf. Auf Schemaebene hat das block-Attribut jedoch keine Auswirkungen. Dies wirkt sich erst dann aus, wenn erneut per „xsi:type“ in einem Dokumentelement auf ein Untertyp des der Deklaration zugewiesenen Typen zugegriffen wird. Wird das block-Attribut also unüberlegt gesetzt, müssten jene Elemente angepasst werden.

Das final-Attribut arbeitet hingegen wieder auf der Schemaebene, sodass das Einfügen in Typen einer Hierarchie die Ungültigkeit verletzen kann. Ist die Hierarchie so beschaffen, sollte von einer Einfügung bzw. Änderung dieser Attribute abgesehen werden. Um solche Änderung dennoch durchzuführen, wäre die einzige Alternative ein Hierarchie-Neustart für betroffene Typen, worauf der Abschnitt 4.2.6 über das Löschen von komplexen Typen genauer eingeht.

mixed-Attribut Normalerweise hat dieses Attribut keine Belegung, sodass Textinhalte nicht angegeben werden können. Um Textinhalte für Typen zu ermöglichen, die sich innerhalb einer Hierarchie befinden, ist das Setzen dieses Attributs (mit dem Wert „true“) am Wurzel-Typ zwingend erforderlich. Wird das Attribut in nachfolgenden UT_R nicht ebenfalls explizit auf „true“ gesetzt, ist ab dem entsprechenden Typ kein Textinhalt mehr möglich. Für UT_E bleibt die Angabe des Basistypen auch ohne die direkte Verwendung des mixed-Attributs bestehen. Sobald einmal von „true“ auf „false“ gewechselt wurde, kann in keinem weiteren Ableitungsschritt zurück gewechselt werden. Wünscht der Nutzer einen Typ innerhalb einer Hierarchie so zu verändern, dass er gemischten Inhalt darstellen kann, müssen daher alle OT_R sowie der Typ selbst, falls dieser ein T_R ist, um das mixed-Attribut mit dem Wert „true“ ergänzt werden. Genauso ist dieses Attribut bei der Entfernung oder dem Wechsel auf „false“ in allen UT_R zu löschen, die es noch mit „true“ verwenden,

defaultAttributesApply-Attribut Dieses Attribut arbeitet mit dem defaultAttributesApply-Attribut der <schema>-Komponente zusammen, welchem der Name einer Attributgruppe zugewiesen werden kann. Typen, die anschließend das defaultAttributesApply-Attribut mit dem Wert „true“ verwenden (Standardwert), fügen die Attribute der Standard-Attributgruppe implizit ihrer Definition hinzu. Wichtig ist, dass in keinem Typ entsprechende Attributnamen bereits explizit vorkommen. Es ist dann nicht ohne Weiteres (siehe unten) möglich, beispielsweise in T_R die Standard-Attribute einzuschränken. Ebenso dürfen Standard-Attribute in T_E oder die Wurzel nicht eingefügt werden.

Das Vorkommen der Standard-Attribute in Typen einer Hierarchie ist anfänglich nur über den Wurzel-Typ kontrollierbar. Durch die Zuweisung des Attributs zu „false“ in diesem Typen wird das Vorkommen solange blockiert, bis ein UT_E den Wert wieder auf „true“ setzt. Für die Gültigkeit des Schemas ist es zwingend erforderlich, das Attribut in allen UT_R auf explizit „false“ zu setzen, die zwischen dem Wurzel-Typen und einem solchen UT_E liegen. Ansonsten hätte das implizite Attributvorkommen dieser UT_R kein Gegenpart im jeweiligen Obertyp. Sobald defaultAttributesApply (in der Wurzel oder einem UT_E) einmal auf „true“ gesetzt wurde, gilt das Attributvorkommen ab jenem Typen bis hin zu den Blättern der Hierarchie (prinzipiell) unabhängig davon, ob in etwaigen UT_R der Wert auf „false“ gesetzt wird (in XML-Dokumenten können Elemente solcher Typen trotzdem die Standard-Attribute verwenden).

Für diese Typen ist „false“ erst dann nützlich bzw. erforderlich, wenn Attribute der Standard-Attributgruppe explizit eingeschränkt werden sollen. Genau in diesem Fall müssen allerdings weitere UT_R ebenfalls „false“ für ihre defaultAttributesApply-Eigenschaft annehmen, da die Einschränkung sonst rückgängig gemacht würde. Gleiches gilt für UT_E unter der Bedingung, dass die vorherige Einschränkung das Attribut nicht ausschließt (use = „prohibited“). In diesem Fall wäre das erneute Setzen von defaultAttributesApply auf „true“ erlaubt.

4.2.6. Löschen von komplexen Typen

Die Definition eines komplexen Typen entspricht der Strukturbeschreibung für eine Menge von Elementen in gültigen XML-Dokumenten, wobei die Strukturbeschreibung erst durch die Zuweisung des betrachteten komplexen Typen zu Elementdeklarationen im entsprechenden XML-Schema nutzbar ist. Wird im Schema ein solcher komplexer Typ gelöscht, wird die Strukturbeschreibung von den referenzierenden Elementdeklarationen entfernt. Wie beschrieben, existieren zwei Maßnahmen, um in diesem Fall zunächst die Gültigkeit des XML-Schemas sicherzustellen. Die erste besteht in der Löschung aller betroffenen Referenzen. Aus der Sicht der XML-Schemaevolution hat diese Methode weitreichende Folgen für die Instanzdokumente des betroffenen XML-Schemas. Im Zweiten Schritt sind nämlich in den XML-Dokumenten jegliche Elemente zu löschen, die die entfernte Elementdeklaration realisieren, was einen hohen Informationsverlust bedeutet. Die Kompensation durch die Zuweisung eines anderen, ähnlichen Typen zu jenen Elementdeklarationen stellt die zweite Möglichkeit dar. Folge der Kompensation ist, dass betroffene Elemente eines XML-Schemas nicht entfernt werden müssen. Der Informationsverlust fällt also prinzipiell wesentlich geringer aus. Je nachdem, wie ähnlich die Strukturbeschreibung des neuen Typen zu der des gelöschten Typen ist und welche weiteren Typen abhängig von diesem sind, können dennoch Instanzanpassungen nötig sein. Um diese systematisch aufzuschlüsseln wird nachfolgend eine Fallunterscheidung durchgeführt, in denen jeweils die Folgen einer Typlöschung und Konsequenzen für die Kompensation diskutiert werden.

Fall k1: der gelöschte Typ befindet sich in keiner Hierarchie Der einfachste Fall ist der, wenn der zu löschende komplexe Typ auf herkömmliche Art definiert wurde, d. h. ohne Ableitung von einem anderen, nutzerdefinierten komplexen Typ. Hier sind ausschließlich die referenzierenden Deklarationen selbst betroffen. Indirekte Manipulationen anderer Typen bzw. Elementdeklarationen durch eine Veränderung der Typhierarchie treten nicht auf. Die hauptsächliche Schwierigkeit besteht hier darin, einen ähnlichen Typen im Schema zu ermitteln, der den entsprechenden Deklarationen zugewiesen wird. Kann keiner gefunden werden oder wird die Abweichung in der beschriebenen Struktur und damit die entstehenden Änderungen in den XML-Dokument zu groß, wird dem Nutzer als letzte Alternative die Zuweisung des Typen „anyType“ vorgeschlagen. Mit diesem allgemeinsten Typ sind keinerlei Anpassungen an bestehenden Dokumenten nötig, da Deklarationen, denen dieser Typ zugewiesen wurde, beliebigen Inhalt besitzen können. Bestehende Elemente in XML-Dokumenten werden damit der größtmöglichen Objektmenge zugeordnet. Für die weitere XML-Schemamodellierung kann diese Lösung u. U. zu allgemein sein, da eine

gewünschte Struktur an den jeweiligen Stellen der Instanzen nicht mehr vorausgesetzt werden kann. Diese Lösung sollte daher nur in den oben genannten „Notfällen“ genutzt werden. Im Allgemeinen ist es eher angebracht, eine möglichst kleine Obermenge oder eine möglichst große Untermenge trotz Informationsverlust zu verwenden, sofern vorhanden. Dieser Leitsatz gilt auch in den kommenden Fällen.

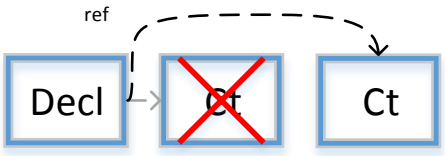


Abbildung 4.15.: Gelöschter Typ ist in keiner Hierarchie.

Fall k2: der gelöschte Typ befindet sich in einer Hierarchie Befindet sich der zu löschende komplexe Typ in einer Ableitungsbeziehung zu anderen Typen, sind weitere Aktionen für den Erhalt der Schemakorrektheit nötig. Entsprechend den verschiedenen Beziehungsarten werden anschließend Unterfälle betrachtet.

Fall k2.1: der gelöschte Typ ist Blatt eines Hierarchiepfades Innerhalb der Unterfälle gestaltet sich dieser am einfachsten. Gegenstand sind hier Typen, die durch eine Ableitung von einem anderen nutzerdefinierten komplexen Typen entstehen und selbst nicht weiter als Basistyp im Schema dienen. Dadurch besteht ähnlich wie in Fall k1 keine Gefahr, dass durch die Löschung solcher Typen weitere, von diesen abgeleitete Typen beeinträchtigt werden. Daneben wird ein Basistyp durch die Löschung einer seiner abgeleiteten Typen nicht beeinflusst. Das bedeutet, dass für die Kompensation im Kern die gleichen Ausführungen wie in Fall k1 gelten. Im Unterschied zu diesem Fall ist hier ein Basistyp vorhanden, der für die Kompensation zur Verfügung steht, wobei zwischen den Ableitungsformen für komplexe Typen zu unterschieden ist.

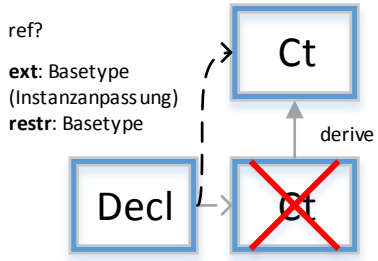


Abbildung 4.16.: Gelöschter Typ ist Blatt.

Wurde der zu löschende Typ als **Extension** definiert, beschreibt sein Basistyp eine Untermenge an Objekten. Durch die Zuweisung des Basistyps zu den Deklarationen, denen der nun gelöschte Typ zugewiesen war, entsteht ein Informationsverlust. Grund hierfür ist die Tatsache, dass es sich bei dem Basistyp um einen Untertyp handelt und die Erweiterung des Inhaltsmodells oder der Attribute nicht Teil des Basistyps sein kann. Als Folge müssen in den XML-Instanzen entsprechende Elemente um den betroffenen Teil eingekürzt werden. Durch die UPA-Regel (Unique Particel Attribution) von XML-Schema ist dabei immer sichergestellt, dass nur eindeutige Inhaltsmodelle existieren, sodass die richtigen Abschnitte der XML-Elemente gefunden werden

können. Gleiches gilt für Attribute, da die Namen der Attribute innerhalb eines Elements i. A. eindeutig sein müssen.

Weniger aufwendig ist die Kompensation von gelöschten Typen, die per **Restriction** definiert sind. Hier ist es möglich, den Basistyp direkt zu verwenden, ohne Instanzen anzupassen. Dies ergibt sich daraus, dass der Basistyp ein Obertyp ist und demnach auch eine Obermenge an Elementen beschreibt. Durch die Zuweisung des Obertyps zu den referenzierenden Deklarationen werden die Elemente aus der Elementmenge des gelöschten Typs in jene Obermenge integriert. Es kommt zu keinem Informationsverlust, sodass diese Lösung i. A. optimal ist.

Wie in Fall k1 beschrieben, bietet es sich natürlich trotzdem für beide Ableitungsarten an, außerhalb des direkten Hierarchiepfades des gelöschten Typen nach ähnlichen Typen im Schema zu suchen, die evtl. eine geeignetere Zielmenge als der Unter- aber auch Obertyp beschreiben.

Fall k2.2: der gelöschte Typ ist innerer Knoten eines Hierarchiepfades Aufwendiger ist der Fall, wenn der zu löschende Typ sich innerhalb einer Hierarchie befindet. Im Unterschied zu Fall k2.1 besitzt der betrachtete Typ dann neben einem nutzerdefinierten Basistyp auch einen oder mehrere abgeleitete Typen. Dementsprechend gilt es, bei der Löschung also zwei Richtungen zu betrachten. Die Überlegungen in Richtung des Basistyps sind dabei analog zu denen von Fall k2.1. Hier treten allerdings zusätzliche Effekte in Richtung der abgeleiteten Typen auf, da von dem gelöschten Typ neben den direkt referenzierenden Elementdeklarationen auch jene direkt oder indirekt abgeleitete Typen abhängig sind. Dies betrifft wiederum Deklarationen, die diese Typen referenzieren. Es sind also ganze Hierarchieabschnitte unterhalb des gelöschten Typs mitsamt der jeweils dazugehörigen Elementmengen in den XML-Instanzen betroffen. Es muss daher neben der üblichen Anpassung der Deklarationen besonders darauf geachtet werden, die unterbrochene Hierarchie ohne Informationsverlust zu „reparieren“. Auch hier hängen die nötigen Schritte von den beteiligten Ableitungsarten ab.

Fall k2.2.1: der gelöschte Typ ist eine Einschränkung Es stellt sich heraus, dass Typen, die durch eine Einschränkung des gelöschten Typen entstehen, relativ einfach zu handhaben sind. Handelt es sich bei dem entfernten Typ selbst um eine **Restriction**, kann auf Grund der Mechanik dieser Ableitungsform in XML-Schema sicher gesagt werden, dass auf beiden Ebenen eine monotone Verkleinerung der Objekt- bzw. Elementmenge in Richtung des jeweils abgeleiteten Typen stattfindet. Durch das Setzen des „baseType“-Attributs der eingeschränkten Un-

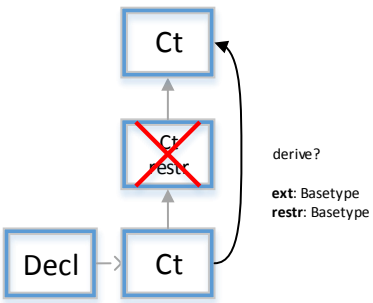


Abbildung 4.17.: Gelöschter Typ ist Knoten (Restriction).

tertypen auf den Basistyp des gelöschten Typen werden die betroffenen XML-Elemente in die nächst höhere Elementmenge ohne Informationsverlust integriert. Das gleiche Vorgehen kann für Typen genutzt werden, die vom gelöschten Typ durch eine Erweiterung abgeleitet sind. Eine Einschränkung wirkt sich zunächst auf die Elementmengen der direkt und indirekt abgeleiteten Typen aus. Wird nun der einschränkende Typ gelöscht, sind die Auswirkungen immer noch in

den bestehenden Elementmengen „aktiv“. Dies umfasst auch die Elementmengen von Erweiterungen des gelöschten einschränkenden Typen, da in dieser Ableitungsform keine Änderungen der Einschränkungen der vorherigen Ebene möglich sind. Damit genügt es hier ebenfalls, die „baseType“-Attribute der erweiternden Typen auf den Basistyp des entfernten einschränkenden Typen zu setzen.

Fall k2.2.2: der gelöschte Typ ist eine Erweiterung Mehr Aufwand muss hingegen betrieben werden, wenn es sich bei dem gelöschten Typ um eine **Extension** handelt. Dieser beschreibt anstatt einer Untermenge seines Basistyps eine neue Obermenge, die quasi „parallel“ zu der des Basistyps liegt. Weitere Typen, die durch eine Einschränkung abgeleitet sind, sind nicht ohne Weiteres auf den Basistyp zu setzen. Da in XML-Schema bei einer Einschränkung das komplette Inhaltsmodell des eingeschränkten Typen wiederholt wird, sind die Bestandteile des gelöschten, erweiternden Typen noch als „Spuren“ vorhanden. Dies betrifft wieder die komplette Hierarchie unterhalb des entfernten Typen. Da der besagte Bestandteil der gelöschten Erweiterung nicht mehr existiert, wäre einerseits das Schema nicht mehr korrekt und andererseits die Instanzbasis an vielen Stellen möglicherweise nicht mehr valide. Um dies auszugleichen, gibt es erneut zwei Optionen: die Hierarchie wird repariert oder neu gestartet.

Die erste Option lässt sich umsetzen, indem von den einschränkenden Typen auf den tieferen Hierarchieebenen die verbleibenden Anteile des gelöschten, erweiternden Typen ebenfalls entfernt werden. Diese Anteile sind aufgrund der UPA-Regel eindeutig identifizierbar. Es müssen also lediglich das beschriebene Inhaltsmodell und die <sequence>-Elternkomponente jeweils entfernt werden. Anschließend ist nur noch der Basistyp der direkt abgeleiteten, einschränkenden Typen auf den Basistyp des entfernten Typs zu setzen. Typen, die aus dem entfernten Typen per Erweiterung selbst entstehen sowie in tieferen Ebenen indirekt abhängig sind, müssen bei dieser Option nicht bearbeitet werden. Dies bedingt die Tatsache, dass das Inhaltsmodell hier nur implizit bzw. intern vorhanden ist. Es reicht also aus, den Basistyp der direkten Erweiterungen auf den Basistyp des entfernten, erweiternden Typen zu setzen, um die Änderung des Inhaltsmodells an diese Typen zu propagieren. Für beide Ableitungsarten ist mit diesem Vorgehen die Hierarchiebeziehung im Schema wiederhergestellt. Was bisher nicht betrachtet wurde, sind die betroffenen Elemente in den XML-Dokumenten. Durch das Löschen einer Erweiterung geht mit diesem Vorgehen der entsprechende Teil des Inhaltsmodells aller weiteren direkt oder indirekt abgeleiteten Typen verloren. Dies bedeutet, dass die Menge der erlaubten Elemente für jeden dieser Typen um genau den gelöschten Anteil verringert wird. Ein hoher Informationsverlust kann dann die Folge sein, wenn etwa viele Elemente diesen Anteil in den Dokumenten realisieren und eingekürzt werden müssen, um weiterhin dem veränderten Hierarchieabschnitt zu entsprechen.

Die zweite Option verfolgt ein Prinzip, nach dem die Hierarchie bei einer gelöschten Erweiterung neu gestartet wird. Für alle Typen die mittels Einschränkung direkt von der gelöschten Erweiterung abgeleitet werden bedeutet dies, dass sie in eine reguläre Typdefinition umgewandelt werden. Hierzu sind im ersten Schritt lediglich die <complexContent>- bzw. <simpleContent>- und <restriction>-Tags zu entfernen. Im zweiten Schritt müssen noch die Attribute und Attributgruppen aller Vorgänger übertragen werden, da diese bei der Restriction auch implizit vorhanden sind und daher noch einmal explizit notiert werden müssen. Typen, die per Erweiterung von der gelöschten Erweiterung konstruiert wurden, bedürfen neben der Tag-Entfernung und Attributübertragung weiterer Maßnahmen. Da die kompletten Inhaltsmodelle nur implizit vorhanden sind, muss rekursiv in Richtung des Basistyps nach einem Typen gesucht werden, der erstmals das vollständige Inhaltsmodell beschreibt. Geeignet hierfür ist die nächstliegende Restriction oder, im Falle einer Kette aus reinen Extensions, die Typdefinition an der Wurzel des Hierarchiepfades. Sobald dieser Typ gefunden wurde, kann das Inhaltsmodell übertragen und nach den Regeln des Erweiterungs-Mechanismus um den Teil der Erweiterung ergänzt sowie mit einer <sequence>-Komponente umschlossen werden. Mit diesen beiden Vorgehen lassen sich alle

vom gelöschten Typ abgeleiteten Typen in eigene Definitionen transformieren. Für die Resthierarchie bedeutet dies, dass die ehemalige Verzweigung nun in mehrere, eigenständige Hierarchiepfade gespalten wird, wobei die transformierten Definitionen als Wurzeln dienen. Vorteil dieser Variante ist, dass die indirekt betroffenen XML-Elemente nicht angepasst werden müssen, da die jeweiligen Objektmengen keinen Veränderungen unterliegen. Für Elemente, die die gelöschte Erweiterung realisieren, muss wie in den anderen Fällen ein neuer Typ bzw. eine neue Menge gefunden werden. Da es sich um eine Erweiterung des Inhaltsmodells handelt, ist die Richtung des Basistyps mit Informationsverlust behaftet. Im Prinzip stehen nun auch keine abgeleiteten Typen mehr zur Verfügung. Da die entstandenen Hierarchiepfade allerdings aus ehemaligen Ableitungen stammen, lässt sich hier ein Kandidat ermitteln, der die kleinst mögliche Obermenge (ehemalige Extension) oder am wenigsten einschränkende Untermenge (ehemalige Restriction) beschreibt, wobei letzteres erneut Informationsverlust mit nötigen Instanzanpassungen bedeuten kann. Alternativ kann wie gehabt nach einem anderen Typen gesucht werden, der womöglich eine besser passende Elementmenge beschreibt.

Die zweite Option verpsricht i. A. weniger Änderungen an den XML-Dokumenten, sodass für eine etwaige Umsetzung diese Methode vorgeschlagen wird.

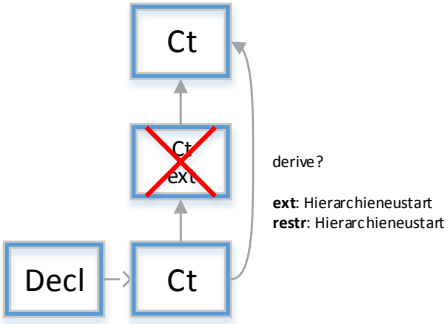


Abbildung 4.18.: Gelöschter Typ ist Knoten (Extension).

Fall k2.3: der gelöschte Typ ist Wurzel eines Hierarchiepfades Als letzter Fall wird die Konstellation betrachtet, bei welcher der zu löschende komplexe Typ selbst Wurzel einer Hierarchie ist. Damit stellt sich dieser Fall als Gegenteil von Fall k2.1 dar, da hier nicht in Richtung des Basistyps nach einem Ersatz geschaut werden kann, wohl aber in Richtung der abgeleiteten Typen. Letzterer Aspekt resultiert dabei im Hinblick auf die Löschung in eine Problematik, die identisch zu der in Fall k2.2 beschriebenen ist, womit die Ausführungen auch hier auf ähnliche Weise gelten. Generell müssen hier alle direkt abgeleiteten Typen in eine eigene Defintion transformiert werden. Für einschränkende Typen lässt sich dies erneut unkompliziert per Tag-Entfernung und eventuelle Attributübertragung durchführen. Um die Transformation im Falle von erweiternden Typen umzusetzen, sind das Inhaltsmodell und die Attribute des Basistyps auf die jeweiligen Erweiterungen zu kopieren. Für beide Arten von direkt oder indirekt abgeleiteten Typen müssen damit keine Elemente in XML-Dokumenten angepasst werden.

Für die verbleibenden Elemente, die zu der Elementmenge des gelöschten Typen gehören, muss eine neue gefunden werden. Dabei gelten wieder die Ausführungen über die zweite Option in Fall k2.2.2. Liegen ehemalige Erweiterungen vor, bieten sich diese als neuer Typ an. Aber auch die Zuordnung zu einer möglichst wenig einschränkenden Untermenge durch die Zuweisung der Deklaration zu einer ehemaligen Einschränkung ist denkbar. Die Möglichkeit andere, ähnliche Typen zu verwenden besteht ebenfalls für diesen Fall.

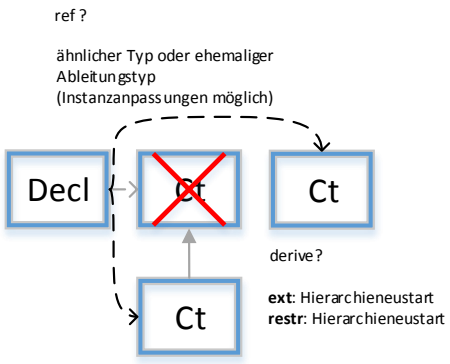


Abbildung 4.19.: Gelöschter Typ ist Wurzel.

Bestimmung der Ähnlichkeit Um einen gelöschten komplexen Typen durch einen anderen zu ersetzen, der dabei nicht in der selben oder generell in keiner Hierarchiebeziehung steht, muss seine Eignung als Basistyp untersucht werden. Im Wesentlichen müssen dazu die Inhaltsmodelle und Attribute des potentiellen Ersatzes sowie der vorhandenen Ableitungen verglichen werden. Am besten geeignet ist ein Typ dann, wenn sein Inhaltsmodell und seine Attributmenge mit den jeweiligen Aspekten des gelöschten Typen übereinstimmen oder allgemeiner sind. Je mehr Abweichungen in diesen Punkten existieren, desto geringer ist die Übereinstimmung, was tendenziell mehr Änderungen in den XML-Dokumenten zur Folge hat. Für Attribute bedeutet dies, dass zunächst alle Attributnamen des alten Typen im neuen Typen auftreten können. Für den Zweck der Herstellung der Gültigkeit des Schemas sind auch Synonyme als Namen denkbar, was jedoch Änderungen an den XML-Dokumenten nach sich zieht. Zudem setzt dies Synonymwörterbücher o. Ä. voraus. Anschließend müssen die Angaben „type“, „fixed“ und „use“ betrachtet werden. Generell muss zu dem neuen, gleichnamigen Attribut ein einfacher Typ zugewiesen sein, der einen möglichst deckungsgleichen oder allgemeineren Wertebereich beschreibt, wobei der gleiche Typ optimal wäre. Sind die Wertebereiche kompatibel, darf die „fixed“-Angabe im neuen Typen im idealen Fall nicht auftreten, wenn sie im zu ersetzenden Typen nicht aufgetreten ist. Wurde die Attributwertfixierung in beiden Typen vollzogen, sollte der Wert nicht abweichen. Ebenso ist für die „use“-Eigenschaft keine Verschärfung erlaubt, d. h. das Wechseln des Wertes von „optional“ zu „required“ oder „prohibited“ sowie jeweils der Wechsel der zuletzt genannten Werte.

Weiterhin muss die jeweilige Kombination von Kompositoren und Elementen betrachtet werden. Je nachdem, welche Kompositor-Art, Elementnamen und Kardinalitäten in den Inhaltsmodellen involviert sind, kann durchaus dieselbe Struktur beschrieben werden, auch wenn die Anordnung der Partikel syntaktisch anders ist. Da diese Analyse relativ aufwändig ist, soll hier als Vereinfachung die reine Syntax verglichen werden. Dies bedeutet, dass zunächst der jeweilige Kompositorbaum identisch sein muss. Im nächsten Schritt sind die Elemente selbst zu vergleichen. Primär wichtig ist hier erneut der Name. Es sollten alle Elementnamen im neuen Typen auftreten, die zuvor im gelöschten Typen deklariert oder referenziert waren. Als Alternative steht wie bei den Attributen die Verwendung von Elementen mit Synonymen als Namen zur Verfügung, mit den bekannten Konsequenzen. Daneben müssen die Elementtypen kompatibel sein. Das heißt, dass vorerst nur einfache Typen durch einfache und komplexe durch komplexe Typen ersetzt werden können. Liegen jeweils einfache Typen vor, gelten die selben Verhältnisse, die oben für die Attribute genannt wurden. Für komplexe Typen gilt dasselbe Prinzip, d. h. dass der neue Typ am besten allgemeiner oder identisch ist. Der Aufwand für den Vergleich ist hier allerdings höher, da rekursiv jeweils alle beteiligten komplexen und einfachen Typen verglichen werden müssen. Weiterhin sind die Angaben „abstract“, „fixed“, „minOccurs“, „maxOccurs“ und „nillable“ der Referenz sowie referenzierten Deklaration von Bedeutung. Ist die Deklaration eines

Elements im neuen Typ abstrakt und im gelöschten Typ nicht, bedeutet dies abhängig von den beteiligten Substitutionsgruppen einen Informationsverlust oder -gewinn. In beiden Fällen müssen u. U. die XML-Dokumente bearbeitet werden, da zwingende Elemente oder Werte wegfallen oder hinzukommen können. Für die „fixed“-Eigenschaft gelten die Ausführungen, die bereits für die Attribute getätigt wurden. Ebenso sollten die Kardinalitäten keine Elemente oder Kompositoren erzwingen bzw. zwingend häufiger machen oder gänzlich ausschließen, da dies wie die übliche Ableitung durch Einschränkung die Menge an Objekten einschränkt und die Änderung der betroffenen Stellen in den XML-Dokumenten erfordert. Zuletzt ist es vorteilhaft, wenn die „nillable“-Angabe bei beiden Typen gleich ist oder diese beim gelöschten Typ nicht bzw. auf „false“ und beim neuen Typ auf „true“ gesetzt ist. Der umgekehrte Fall bedeutet, dass für ehemals leere Elemente zwangsweise ein Inhalt konstruiert werden muss.

Wie beschrieben, sind dies lediglich „Empfehlungen“ für einen Typersatz. Im Allgemeinen kann nicht davon ausgegangen werden, einen perfekt passenden Typen im Schema zu finden, sodass mit dieser Methode mit Instanzanpassungen zu rechnen ist. Eine ähnliche Situation entsteht, wenn von dem gelöschten Typ weitere Typen abgeleitet sind. Es ist relativ unwahrscheinlich, dass die Einschränkung oder Erweiterung eines lediglich ähnlichen Typen dieselbe Semantik besitzt bzw. Struktur beschreibt, wie die entsprechende Ableitung vom originalen Typ. Daher ist es angebracht, generell alle beteiligten Hierarchien wie in Fall k2.2.2 neuzustarten, um möglichst wenig Folgeänderungen auszulösen.

5. Implementation in CodeX

Dieses Kapitel veranschaulicht die Implementation von Aspekten des Typmanagements in den Forschungsprototypen CodeX. Dazu wird CodeX zunächst noch einmal von der technischen Seite betrachtet. Anschließend folgt eine Vorüberlegung zu den zu implementierenden Funktionen, wobei Besonderheiten im Hinblick auf die Gegebenheiten in CodeX hervorgehoben werden. Als Abschluss des Kapitels wird die erfolgte Implementation vorgestellt.

5.1. Grundlagen von CodeX

In der aktuellen Version liegt der Prototyp als Client-Server-Anwendung in der Programmiersprache Java vor. Realisiert wird dies durch die Nutzung von Google Web Toolkit (GWT[1]). Mit diesem Werkzeug lässt sich sowohl Server-Code als auch Client-Code direkt in Java verfassen. Ein interner Compiler sorgt dafür, dass der Client-Code in JavaScript-Code zur Darstellung in Webbrowsern transformiert wird. Die Modellierung von XML-Schemata bzw. der EMX-Modelle in CodeX geschieht auf Clientseite. Aus einer relationalen Datenbank werden hierzu bislang gespeicherte EMX-Entitäten des Modells gelesen und nach der Änderung wieder gespeichert. Für solche Client-Server-Kommunikationen werden Remote Procedure Calls (RPC) mittels GWT-RPC verwendet. Diese werden auch dann eingesetzt, wenn komplexere Operationen auf der Serverseite ausgeführt und anschließend auf Client lediglich angezeigt werden. Die Implementation ist zwangsläufig auf der Client-Seite anzusiedeln. Für die Visualisierung ist dies selbsterklärend, aber auch die Kompensation von Typen benötigt die aktuellsten Typinformationen, die sich durch die Modellierung seitens des Nutzers ergeben.

5.2. Vorüberlegung zur Implementation

Als Besonderheit wurden in CodeX im Rahmen der Masterarbeit von Hannes Grunert in [Gru13] Integritätsbedingungen für die Modellierung der EMX-Modelle implementiert. Diese umfassen neben der Schlüsselintegrität und strukturellen Integrität auch die Wertebereichsintegrität, wobei ein Integritätsmonitor permanent auf die Korrektheit von möglichen Eingaben achtet. Gerade die letzte Art von Integritätsbedingungen ist auch für diese Arbeit interessant. Mit Hilfe der Wertebereichsintegrität werden einerseits Eingaben für Elemente und Attribute überprüft, denen ein built-in oder nutzerdefinierter einfacher Typ zugewiesen ist. Bei Nichtübereinstimmung der eingegebenen Werte mit dem Wertebereich des Datentypen wird ein Fehler angezeigt. Andererseits werden auch die Facetten eines nutzerdefinierten Datentypen miteinbezogen. Interessant ist dies deshalb, da auch schon bei der Erstellung sowie Bearbeitung einer einfachen Typdefinition auf die Korrektheit der Facetten bzw. ihrer Werte im Vergleich zum built-in Datentyp geachtet wird. Außerdem unterliegen Änderungen der Facetten eines Typen einer Kontrolle auf Konsistenz in Richtung des Obertypen, ähnlich wie in Abschnitt 4.2.2 beschrieben. Die Kontrolle in Richtung der Untertypen fehlt jedoch. Wird der der Basistyp einer Einschränkung verändert, erfolgt zusätzlich die Entfernung aller im Bezug zum built-in Datentypen ungültigen Facetten in dem veränderten Typen sowie seiner Untertypen.

In CodeX sind also einige Aspekte der Änderung sowie die korrekte Einfügung einfacher Typen bereits umgesetzt. Was momentan gänzlich fehlt ist das Löschen von einfachen Typen sowie die damit einhergehende Kompensation für referenzierende Deklarationen und abgeleitete Typen. Dies betrifft auch die komplexen Typen, wobei wie in Abschnitt 3.3.2 ausgeführt die Typhierarchie durch Erweiterung und Einschränkung momentan nicht vollständig implementiert

[1] http://www.gwtproject.org/

ist. Für die Kompensation nach der Löschoperation bedeutet dies, dass einerseits im Gegensatz zu einfachen Typen auf keinen klaren Obertypen zurückgegriffen werden kann. Andererseits erspart sich damit auch die Bearbeitung der abgeleiteten Typen (siehe Abschnitt 4.2.6). Weiterhin gibt es bislang keine Möglichkeit, die Beziehung bzw. die Hierarchie zwischen den Typen zu betrachten. Der Fokus in der Implementation soll folglich auf der Realisierung der Löschoperation für Typen mitsamt der Kompensation liegen. Daneben ist eine Möglichkeit zur Visualisierung der Typhierarchien zu schaffen, durch welche die Beziehungen einfacher und komplexer Typen besser zu erfassen sind. Die Editiermöglichkeiten sind in CodeX bereits in großem Maße vorhanden, sodass die Visualisierung ausdrücklich nicht zur Modellierung dienen soll. Vielmehr wird ein ergänzender Sicht-Charakter angestrebt, ähnlich wie z. B. die Graph-View des XSD-Designers in Visual Studio (siehe Abbildung 3.5). Zur klaren Unterscheidung zwischen einfachen und komplexen Typen soll die Visualisierung aber getrennt voneinander, d. h. in zwei verschiedenen Sichten jeweils für einfache und komplexe Typen implementiert werden. Als Grundlage für die Visualisierung und der Löschung von Typen dienen die entsprechenden Abschnitte des vorgestellten Konzepts. Durch die Gegebenheiten in CodeX sind u. U. Anpassungen des Konzepts vorzunehmen, worauf in den nachfolgenden Abschnitten eingegangen wird. In der Implementierung sind sämtliche verarbeitenden Klassen auf der Clientseite in dem Paket `de.uni_rostock.dbis.codex2.client.type_manager` untergebracht. Die Klassen für die Dialoge befinden sich im Paket `de.uni_rostock.dbis.codex2.client.dialog.menu-Dialogs`. Ein Klassendiagramm kann auf der beigelegten CD gefunden werden.

5.3. Visualisierung

In diesem Abschnitt wird die programmiertechnische Umsetzung der Visualisierung von Typen genauer betrachtet. Es erfolgt erneut eine Trennung zwischen einfachen und komplexen Typen.

5.3.1. Einfache Typen

Der Name und der Inhalt, d. h. die Facetten, von einfachen Typen werden durch die Klasse `St-Diagram` dargestellt. Zwecks der Unterscheidung der Ableitungsart wird dem Namen ein spezifischer Icon vorangestellt. Jede Klasse enthält eine Referenz auf eine `EmxSimpleType`-Instanz. Dadurch ist es möglich, die entprechenden Facetten mittels der Methode `findStFacets()` einzulesen. In einem zweiten Schritt werden im Hinblick auf den Obertypen die aktuell wirksamen Facetten ermittelt, was über die Methode `determineEffectiveFacets()` geschieht. Für Wurzeltypen sind dies genau die Facetten, die er selbst definiert. Für einschränkende Typen, die sich als innerer Knoten in einer Hierarchie befinden, ergeben sich die wirksamen Facetten aus einem Vergleich der eigens definierten Facetten und der Facetten des (ebenfalls einschränkenden) Basistypen. Damit sind quasi durch einen Typen überschriebene Aussagen seines Basistypen auffindbar. So stellt z. B. ein Diagramm immer nur die aktuellste Facette der Kategorie max-/minInclusive/-Exclusive dar. Sollte der Basistyp jedoch ein Listen- oder Vereinigungstyp sein, ergeben sich die wirksamen Facetten wieder vollständig aus der eigenen Facettendefinition.

Anhand dieses Vergleichs, lässt sich in der Methode `layoutFacets()` eine Färbung für den Wert der Facetten ermitteln. Grau steht dabei für vererbte, schwarz für noch nicht vorhandene und neu eingeführte und blau für veränderte Facetten des Basistypen. Um auch für Listen- und Vereinigungstypen eine Aussage über den Wertebereich treffen zu können, wird für beide die Menge der wirksamen Facetten des Basistypen bzw. der Mitgliedstypen angezeigt. Zur leichteren Identifizierung des Ursprungs der jeweiligen Facetten ist für letztere zusätzlich der Name der Mitgliedstypen vermerkt. Weiterhin wurden die Längen- und Aufzählungsfacetten jeweils zusammengefasst.

Die Positionierung sowie die endgültige Darstellung der Diagramme übernimmt die Klasse `StViewer`. Ausgehend von den built-in Datentypen durchläuft die Methode `iterateType()` Schritt für Schritt die `EmxSimpleType`-Instanzen eines EMX-Modells und erstellt für jede ein Diagramm. Anschließend werden die Typ-Diagramme entsprechend der vorliegenden Beziehung miteinander verbunden. Für die Darstellung von Vereinigungs-Typen werden lediglich statt einer Verbindung mehrere genutzt, die in Richtung der Mitgliedstypen zeigen. Es gäbe zwar auch andere Alternativen zur Visualisierung der Mitglieds-Beziehung, aber wie bereits in Paragraph *4.1.4 Union* des Konzept-Kapitels beschrieben erscheint die gewählte Form am geeignetsten.

5.3.2. Komplexe Typen

Das Vorgehen bei der Visualisierung komplexer Typen ist ähnlich dem der Visualisierung einfacher Typen. Name, Kompositor und Element- sowie Attributvorkommen komplexer Typen werden durch die Klasse `CtDiagram` repräsentiert. Jeder dieser Komponentengruppen ist ein eigener Bereich im Diagramm zugeordnet, wobei der Kompositor die Elementangaben umfässt. Statt einer Referenz auf eine `EmxComplexType`-Instanz enthält diese Klasse eine Referenz auf eine Instanz der Klasse `CtContent`, welche als eine Art Container agiert und die oben genannten Komponenten zusammenfasst. Das Platzieren der verschiedenen Komponenten in ihre vorgesehenen Bereiche übernehmen die Methoden `layoutElements()` sowie `layoutAttributes()`.

In CodeX sind die Attribute standardmäßig in Attributgruppen zusammengefasst, sodass diese zunächst aus jenen Gruppen zu extrahieren sind. Ziel dieses Prozess ist, einen besseren Überblick über die tatsächlich vorhandenen Attribute zu geben. Nach der Extraktion erfolgt ihre Platzierung unterhalb des Elementbereichs. Zusätzlich dazu werden die Default-Attribute im Diagramm angezeigt, sollte ein entsprechender Typ Gebrauch von einer eventuell definierten Default-Attributgruppe machen. Angaben für Elemente und Attribute gestalten sich nach dem folgenden, UML-ähnlichen Muster: <Name> ":" <Typ> [= <default-Wert> |<fixed-Wert> "{req}"] ["["<minOccurs>".."<maxOccurs>"]"]. Für Attribute wird wie im Konzept vorgestellt die use-Eigenschaft auf die Kardinalitäten [0..0] (prohibited), [0..1] (optional) und [1..1] (required) abgebildet.

Pro Typ ist in EMX nur eine Element-Wildcard vorgesehen. Bei der Erstellung von XML-Schemata wird diese zudem am Ende des jeweiligen Inhaltsmodells platziert. Dementsprechend zeigt ein Diagramm eine Element-Wildcard auch am Ende des Elementbereichs an. Ähnliches gilt für die Attribut-Wildcard. Da es auch in CodeX möglich ist, pro Attributgruppe eine Attribut-Wildcard zu spezifizieren, berechnet die Methode `createCompleteWildcard()` der `Utils`-Klasse die vollständige Attribut-Wildcard mittels der Durchschnittsbildung der jeweiligen Angaben. Statt einer Auflistung mehrerer Wildcards ist so nur noch die vollständige Wildcard im Attributbereich zu sehen. Aus platztechnischen Gründen sind im Diagramm selbst nur die vordefinierten Werte „##any„, „##other" oder „##targetNamespace und „##local" vermerkt. Für eine Element-Wildcard erscheint zudem ihre Kardinalität. Weitere Angaben lassen sich per Mouseover-Funktion über den Wildcard-Eintrag entnehmen.

Bestimmung der Beziehung Grundsätzlich funktioniert die endgültige Darstellung und Positionierung der Typ-Diagramme ähnlich wie die Darstellung der Diagramme für einfache Typen. Die Klasse `CtViewer` sucht zunächst mithilfe der Methode `findCts()` in der Klasse `Utils` nach den `EmxComplexType`-Instanzen eines EMX-Modells. Da die konkreten Beziehungen „Einschränkung" sowie „Erweiterung" von komplexen Typen in CodeX wie erwähnt noch nicht vollständig implementiert sind, wird anschließend jeder Typ mit jedem ein mal in der Methode `compare()` *verglichen*. Das Resultat dieses Vergleichs ist eine approximierte Beziehung zwischen den jeweils verglichenen Typen. Auf Grundlage dieser Beziehungen ordnet dann die Methode `iterateType()` den Diagrammen eine Position im `CtViewer` zu und verbindet

diese mit einer Linie. Zuständig für den Vergleich zweier komplexer Typen ist die Klasse `CtComparator`. Diese agiert ähnlich wie schon die Klasse `CtDiagram` auf Instanzen der Klasse `CtContent`.

In der Umsetzung werden ausschließlich Attribut- und Elementvorkommen verglichen. Dies stellt eine Vereinfachung dar, da in XML-Schema 1.1 wie in *4.2.5 Ändern von Wildcards* angemerkt etwa Elemente in einem UR_R nutzbar sind um eine Elementwildcard einzuschränken. Wollte man Wildcards in den Vergleich miteinbeziehen, so wäre der Aufwand relativ groß, da z. B. die Kardinalität aller Wildcards mit etwaigen Elementkardinalitäten des Untertypen in Verbindung gebracht werden müssten. Liegt ein endlicher maxOccurs-Wert für alle Komponenten vor, wäre es nötig für jede Wildcard die durch die Elemente des Untertypen „verbrauchten" Vorkommen zu speichern und für weitere Vergleiche heranzuziehen. Durch die Beschränkung in CodeX auf eine Elementwildcard am Ende eines Inhaltsmodells, wäre dieser Aspekt noch im Bereich des Möglichen. Allerdings müssten auch die Angaben der Wildcard überprüft werden. Eine Einschränkung durch konkrete Elemente, die nicht in etwaigen Namensräumen bzw. in verbotenen Namensräumen liegen, wäre nicht gestattet. Bedingt durch diese Komplexität ist es daher nicht das Ziel, eine Art XML-Schemaprozessor auf EMX-Ebene zu konstruieren. Die Bestimmung der Beziehung soll vielmehr eine Form der Annährung verstanden werden.

Dies trifft auch auf den reinen Vergleich der Elemente und Attribute zu. In *4.2.5 Ändern von Kompositoren* wurde bereits angemerkt, dass XML-Schema 1.1 eine Vielzahl an logischen Einschränkungen zulässt, die etwa mit dem alleinigen, von anderen Partikeln isolierten Vergleich von z. B. minOccurs- und maxOccurs-Werten nicht erkennbar sind. Ebenso sind Fälle möglich, in denen der Kompositor in einen einschränkenden Ableitungsschritt wechseln kann. Die exakte Implementierung des Algorithmus gemäß http://www.w3.org/TR/xmlschema11-1/#sec-derivation-ok-restriction zur Bestimmung ob z. B. ein Typ eine gültige Einschränkung eines anderen ist, würde jedoch ebenfalls über den Rahmen der Arbeit hinausgehen. Zudem bietet CodeX durch die Reduktion der Anzahl an verschachtelten Kompositorebenen eine gute Grundlage zur Approximation der möglichen Hierarchiebeziehungen.

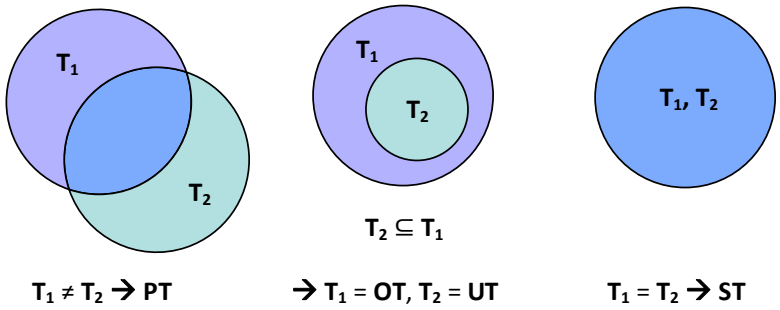


Abbildung 5.1.: Darstellung der Beziehungen Paralleltyp, Obertyp, Untertyp und Gleicher Typ.

Die Prämisse des Vergleichs lautet: „Was durch einen Untertyp darstellbar ist, muss auch der Obertyp darstellen können". Optional kann ein Obertyp auch mehr darstellen. Der Begriff „Darstellen" meint hier den Wertebereich bzw. die Menge an möglichen XML-Instanzstrukturen in der Form von Element- und Attributkombinationen. Die Klasse `CtComparator` unterscheidet zwischen vier grundlegenden Beziehungen: Untertyp (UT), Obertyp (OT), Gleicher Typ (ST) und Parelleltyp (PT), deren konzeptionelle Bedeutung Abbildung 5.1 beispielhaft zeigt. Daneben existieren zwei spezielle Beziehungen $ET1$ und $ET2$, die für die Erweiterung stehen. Der Unterschied zwischen beiden wird in Paragraph *5.3.2 Vergleich einer Sequenz mit einer Sequenz* erläutert, nachfolgend werden zur Vereinfachung beide mit ET zusammengefasst. In der Umsetzung dienen einfache, statische `int`-Variablen zur Unterscheidung. Die Beziehungen UT und OT beschreiben im Gegensatz zur bisherigen Notation immer das Verhätlnis der **Einschrän-**

kung und sind dabei jeweils „symmetrisch“. D. h. das Vertauschen der zwei Typ-Parameter der `determineRelation()`-Methode ergibt je nach „Blickrichtung“ entweder *UT* oder *OT*, sofern eine Einschränkung feststellbar ist. *ST* wird immer dann verwendet, wenn die Wertebereiche exakt gleich sind. Dahingegen soll *PT* den Fall ausdrücken, in welchem ein Vergleich sowohl die Beziehung *OT* als auch *UT* ermittelt. Dies bedeutet, dass die Werte eines Typen mit denen eines anderen in einigen Bereichen übereinstimmen, in anderen jedoch nicht und zusätzlich exklusive Wertmengen vorliegen können. Mathematisch gesehen umfasst quasi die Schnittmenge beider Objektmengen nicht die Mengen beider einzelnen Typen. Gleichermaßen findet das Ergebnis *PT* auch dann Anwendung, wenn keinerlei Übereinstimmung der Wertebereiche auszumachen ist und der Durchschnitt demnach leer ist.

Um möglichst nah am eigentlichen Verhalten von XML-Schema zu bleiben, kommt die Beziehung *ET* nur als mögliches Resultat beim Vergleich zweier Sequenz-Typen zum Einsatz. Begründen lässt sich diese Entscheidung damit, da in XML-Schema Erweiterungen sich nur über eine implizite Umschließung des bisherigen und erweiterten Inhaltsmodells mit einer Sequenz äußern (siehe Abschnitt 2.5.3). Da EMX keine verschachtelten Kompositoren vorsieht, wäre die Erweiterung in diesem Sinne quasi unmöglich. Als Ausnahme wird der Fall behandelt, in dem eine Sequenz den maxOccurs-Wert 1 nicht übersteigt. Das Anfügen von Elementen am Ende dieser Sequenz würde so der Erweiterung durch eine Sequenz mit diesem Element gleichen, die ebenfalls einen maxOccurs-Wert von 1 hat. Für sonstige Kompositoren wird die Erweiterung, d. h. das Ergebnis *ET* ausgeschlossen.

Die Bestimmung der Beziehung erfolgt über ein Entscheidungssystem mit verschiedenen Methodenaufrufen. In den Methoden werden verschiedene Aspekte wie die Reihenfolge, das Vorkommen und die Häufigkeit von Elementen sowie das Attributvorkommen getrennt voneinander betrachtet. Element- und Attributvergleiche geschehen immer auf Grundlage des eindeutigen EMX-Entitäten-Identifiers (EID) anstatt über den Namen der entsprechenden Komponenten. Jede dieser Methode liefert eine der oben genannten Entscheidungen. Anschließend werden sämtliche Entscheidungen paarweise mit der Methode `decide()` zu einer endgültigen Entscheidung zusammengeführt. Die folgende Tabelle 5.1 gibt einen Überblick über die Priorisierung der einzelnen Entscheidungsergebnisse. Dieses Vorgehen bietet den Vorteil, dass zukünftige Erweiterungen

Entscheidung	**PT**	**ST**	**UT**	**OT**	**ET**
PT	PT	PT	PT	PT	PT
ST	PT	ST	UT	OT	ET
UT	PT	UT	UT	PT	PT
OT	PT	OT	PT	OT	PT
ET	PT	ET	PT	PT	ET

Tabelle 5.1.: Wahl der Beziehung in Abhängigkeit von zwei Entscheidungen

des Algorithmus leicht in eigenen Methoden hinzugefügt und in die Ermittlung der Beziehung eingebunden werden können. Die nachfolgenden Paragraphen erläutern die Bestimmung der Beziehung für komplexe Typen mit verschiedenen Kompositoren sowie Attributen im Detail. Obwohl die Parameter der vorgestellten Methoden auf `CtContent`-Instanzen arbeiten, werden die untersuchten Typen bzw. ihre Inhalte nachfolgend mit T_1 und T_2 abgekürzt. Die Implementierung erfolgte so, dass immer T_2 gegen T_1 getestet wird, der also eine Art „Pivotelement“ darstellt. Außerdem werden bereits bei der Erstellung der `CtContent`-Instanzen nicht existente Elemente (maxOccurs=0) entfernt, sodass auf normalisierte Elementlisten gearbeitet wird.

Für die nachfolgende Notation werden Typen T_1 und T_2 als Mengen von Elementen $p_i \in T_1$ sowie $p_j \in T_2$ verstanden. Die Indizes sind wie folgt definiert: $1 \leq i \leq |T_1|$ und $1 \leq j \leq |T_2|$. Oftmals wird einfachheitshalber anstelle der Bezeichnung T_1 oder T_2 auch der jeweilige abgekürzte Kompositorname (s - sequence, c - choice, a - all) des Typs selbst verwendet, was

aber dieselbe Menge an Elementen bezeichnet. Dementsprechend wird die Anzahl an Elementen z. B. $|T_1| = n_s$ abgekürzt. In Fällen in denen der Kompositor identisch ist, wird erneut z. B. mit s_1 und s_2 unterschieden. Weiterhin wird an einigen Stellen die Schnittmenge mit $Int = T_1 \cap T_2$ bezeichnet. Die Elementnamen $p_i \in Int$ sowie $p_j \in Int$ in dieser sind für $i = j$ identisch, sodass für diese $p_i = p_j$ gilt. Der Index signalisiert dabei weiterhin, auf welchen Typ sich das Element bezieht, um die minOccurs- und maxOccurs-Werte in der Notation zu unterscheiden. In einer Schnittmenge befinden sich somit $1 \leq i = j \leq n_{Int}$ Elemente. Sollte der Begriff $Diff_1$ bzw. $Diff_2$ auftauchen, meint dies Differenzmengen mit $Diff_1 = T_1 \backslash T_2$ und $Diff_2 = T_2 \backslash T_1$. Für diese gilt immer $p_i \in Diff_1 \neq p_j \in Diff_2$. Die Anzahlen von Elementen in diesen Mengen sind mit $1 \leq i \leq n_{Diff_1} = |Diff_1|$ sowie $1 \leq j \leq n_{Diff_2} = |Diff_2|$ festgelegt.

Vergleich einer Sequenz mit einer Sequenz Den Vergleich zweier Sequenz-Typen übernimmt die Methode `handleSeqSeq()`. Im ersten Schritt werden die Elementnamen beider Typen in der Methode `elementNameRequiredRelation()` auf das Vorhandensein im jeweils anderen Typen getestet. Für beide Typen wird hierzu die Elementmenge des einen Typen vom jeweils anderen abgezogen, wodurch für beide Typen Differenzmengen entstehen. Sind beide Mengen leer, lautet die Entscheidung für T_2 zunächst ST. Hat die Menge von T_1 noch Elemente und T_2 nicht, entscheidet die Optionalität (minOccurs=0) der Elemente. Ist mindestens ein Element nicht optional, lautet die Entscheidung über die Elementnamen PT. Sind hingegen alle verbleibenden Elemente in T_1 optional, ist T_2 im Aspekt der Elementnamen ein UT. Im umgekehrten Fall wäre T_2 entsprechend ein OT. Dieses Urteil ist gerechtfertigt, da das Weglassen eines Elements in einer Einschränkung genauso durch die Verringerung des maxOccurs-Wert auf 0 darstellbar ist. Sollten jedoch beide Mengen nicht leer sein, haben beide Typen exklusive Elemente und sind dementsprechend Paralleltypen. Im oben beschriebenen Fall, dass die Elemente der Differenzmenge einer der beiden Typen am Ende des Inhaltsmodells auftauchen, liegt eine Erweiterung ET vor, sofern die maximale Kompositor-Kardinalität 1 beträgt. Ansonsten wird dieser Fall gleich behandelt. Da die Beurteilung der Beziehung immer aus der Sicht von T_2 geschieht, werden für ET zwei einzelne Ergebnisse $ET1$ und $ET2$ genutzt. Ersteres beschreibt den Fall, dass T_1 eine Erweiterung ist, während das zweite Ergebnise T_2 als Erweiterung auszeichnet. Diese Trennung dient lediglich der Vereinfachung der Visualisierung. Außerdem kann bei dem Ergebnis $ET1$ noch mal überprüft werden, ob T_2 nicht doch noch ein üblicher Untertyp von T_1 ist, sollten in den folgenden Entscheidungen gegensätzliche Ergebnisse liefern.

Anschließend erfolgt im zweiten Schritt die Analyse der Reihenfolge der Elemente. Dazu werden beide Elementmengen geschnitten. Sollte die Position mindestens eines Elements nicht in beiden Typen übereinstimmen, liegt mit T_2 gemäß der Natur der Sequenz ein Paralleltyp vor. Im besten Fall kann das Ergebnis nur ST lauten. Diese Überprüfung vollzieht die Methode `elementOrderRelation()`.

Als finaler Schritt findet die Beurteilung der Häufigkeit der Elemente statt, die sich in der Schnittmenge befinden. Lautet die Elementnamen-Entscheidung ST, UT, OT oder ET und es liegt in einer Sequenz nur ein Element vor (die Schnittmenge besitzt also auch nur ein Element), wird dessen Häufigkeit zusammen in Verbindung mit dem <sequence>-Kompositor nach den Formeln $s_{min} \cdot p_{min}$ und $s_{max} \cdot p_{max}$ durch die Methode `elementGroupOccRelation()` berechnet. Dies wird getan, da es für die Anzahl nur eines Element unerheblich ist, wie oft das Element selbst oder es in einer Sequenz wiederholt wird. Im normalen Fall, dass die Schnittmenge mehr als ein Element besitzt, nutzt die Methode `elementLocalOccRelation()` die lokalen Elementhäufigkeiten. In beiden Methoden wird nach Bestimmung der Elementhäufigkeit die Methode `determineOccurrenceRelation()` aufgerufen, welche nach dem klassischen Vergleich der unteren und oberen Häufigkeiten feststellt, ob das entsprechende Element verschärft oder verallgemeinert wurde. Dies wird zusätzlich auch für die <sequence>-Kompositoren beider Typen durchgeführt. Die möglichen Ergebnisse sind daher auch wieder ST, UT und OT. Sollten einmal

widersprüchliche Aussgen durch die Ergebnisse UT und OT für verschiedene Elemente in beiden Typen auftreten, wird der Vergleich der Häufigkeiten mit dem Ergebnis PT abgebrochen.

Zwecks der endgültigen Fällung einer Entscheidung werden alle drei Einzelentscheidungen mit der `decide()`-Methode zusammengeführt.

Vergleich einer Sequenz mit einer Alternative Zuständig für diesen Vergleich ist die Methode `handleSeqChoi()`, wobei T_1 immer die Sequenz ist und T_2 immer die Alternative. Die zwei wesentlichen Entscheidungskriterien sind einerseits erneut die Mengen der vorhandenen Elementnamen bzw. EIDs (Variable `elementNameRelation`) und andererseits die Fähigkeit der Sequenz oder Alternative, die Elemente der jeweils anderen Elementgruppe in vollem Maße darzustellen (Variable `canDisplayRelation`). Die Methode `elementNameRequiredRelation()` testet die Nicht-Optionalität von Elementen nur für die Sequenz. Ist also die Differenzmenge von T_2 nicht leer und die von T_1 schon, stehen die Anzeichen dafür, dass die Alternative ein OT ist. In diesem Fall ist es kein Problem, wenn deren Restelemente einen minOccurs-Wert besitzen, der größer als 0 ist, da die Elemente sowieso optional sind. Liegt ein umgekehrtes Verhältnis der Differenzmengen vor (die Sequenz wäre also ein OT) und es fehlt ein nötiges Element der Sequenz in der Alternative, so lautet das Ergebnis für den Vergleich erneut ein PT.

Im zweiten Schritt wird nacheinander geschaut wird, ob von der Anzahl der Elemente her entweder die Sequenz oder die Alternative ein Ober-, Unter- bzw. Paralleltyp ist. Stellt die Methode `seqCanDisplayChoiElements()` fest, dass die Sequenz die Elemente der Alternative darstellen kann, ist letztere laut dieser ein UT. Fehlen z. B. Elemente ist das Ergebnis PT, sodass anschließend noch mal in der Methode `choiCanDisplaySeqElements()` geschaut, ob eventuell die Alternative ein Obertyp der Sequenz ist. Damit die Sequenz überhaupt als Obertyp in Betracht kommt, müssen alle ihrer Partikel optional sein. Kann dies sichergestellt werden, ist danach nur noch das maximale Vorkommen der jeweiligen Elemente von Interesse. Aus diesem Grund wird für alle Elemente p der Schnittmenge $p_{i_{max}}$ mit s_{max} sowie $p_{j_{max}}$ mit c_{max} multipliziert. Gilt danach mindestens einmal $p_{i_{max}} < p_{j_{max}}$, so hat die Alternative mehr Möglichkeiten im entsprechenden Element und die Sequenz kann kein OT mehr sein. Spezialfälle die sich z. B. durch „maxOccurs = unbounded“ für einen Wert ergeben, werden abgefangen und nach der gleichen Devise behandelt.

Zur Bestimmung der Beziehung in `choiCanDisplaySeqElements()` kommt ein anderes Verfahren zum Einsatz. Zunächst wird überprüft, ob bereits ein Element die Ungleichung $p_{j_{min}} > p_{i_{min}}$ erfüllt. Sollte dies so sein, kann mit einer einzigen Wahl des betrachteten Elements in der Alternative nicht das minimale Vorkommen in der Sequenz erreicht werden, wodurch die Alternative kein OT ist. Dies gilt nicht für den Spezialfall $p_{j_{min}} = 1$ und $p_{i_{min}} = 0$, was lediglich bedeutet, dass das Element in beiden Gruppen optional ist (von der Alternative also darstellbar). Daraufhin werden erneut die maxOccurs-Werte verglichen. In einer Schleife wird mit jeder Iteration zunächst verglichen, wieviel Wahlen in der Alternative mit $p_{j_{max}}$ nötig sind, um $p_{i_{max}} \cdot s_{max}$ darzustellen. Das Ergebnis wird vom maxOccurs-Wert der Alternative abgezogen. Reichen bei einem späteren Schleifendurchlauf die verbleibenden Wahlen der Alternative nicht mehr aus, kann diese die Sequenz nicht vollkommen darstellen und ist somit kein OT. Auch hier erfahren Sonderfällle, die den Wert „unbounded“ für das maxOccurs-Attribut eines Kompositors oder Elements beinhalten, eine eigene Behandlung. Ebenso dürfen nicht wie oben alle $p_{i_{min}} = 0$ oder $s_{min} = 0$ sein, wenn die Alternative nicht auch selbst einen minOccurs-Wert von 0 besitzt oder wenigstens ein $p_{j_{min}} = 0$ existiert. Somit könnte die Alternative niemals leeren Inhalt haben, die Sequenz allerdings schon.

Nach der Bestimmung der `elementNameRelation` und der `canDisplayRelation` werden diese Ergebnisse wieder der `decide()`-Methode übergeben, wo die finale Entscheidung gemäß Tabelle 5.1 ermittelt wird. Weiterhin kümmert sich `handleSeqChoi` auch um den Vergleich einer Alternative mit einer Sequenz (die erwartete Beziehung ist vom Sinn her genau umgekehrt

zu den oben beschriebenen Fällen). Vor dem Aufruf werden die Parameter dafür so getauscht, dass die Alternative wieder T_2 ist und die Sequenz T_1. Anschließend wird das Ergebnis mittels Aufruf von `reverseRelation()` ebenfalls umgekehrt.

Vergleich einer Sequenz mit einer Menge Dieser Vergleich ähnelt dem Vergleich einer Sequenz mit einer weiteren. Der Unterschied besteht hier darin, dass die Menge potentiell mehr Möglichkeiten bietet, Elemente anzuordnen. Es wird davon ausgegangen, dass die Sequenz T_1 ist und die Menge T_2. Die Methode `handleSeqAll()` überprüft zuerst die Beziehung der Elementnamen mittels Aufruf von `elementNameRequiredRelation()`. Liegen in mindestens einer Element-Differenzmenge nicht optionale Elemente vor, ist die Beziehung zwischen beiden Tyoen zwangsläufig PT. Stellt sich heraus, dass die Menge prinzipiell ein Untertyp ist, d. h. weniger Elemente als die Sequenz besitzt und die fehlenden Elemente optional sind, ist die Anzahl ihrer Elemente n_a für das Urteil entscheidend. Ist diese nämlich größer als 1, erlaubt die Menge im Bereich der überlappenden Elemente mehr Kombinationsmöglichkeiten als die Sequenz und ist somit ein Paralleltyp. Hat die Menge jedoch nur ein einziges Element (Anzahl der Elemente in der Schnittmenge ist auch 1), ist sie diesbezüglich ein echter Untertyp. Sollten jedoch die Elementnamen exakt übereinstimmen und nur leere Differenzmengen entstehen, wird bei $n_a > 1$ die Menge als OT deklariert, während für den Fall $n_a = 1$ tatsächlich ST vorliegt.

Im Anschluss ist noch die Beziehung der Elementhäufigkeiten `eleOccRelation` mittels Aufruf von `elementSeqAllOccRelation()` zu bestimmen. Hier wird die lokale Häufigkeit eines Elements mit der Häufigkeit der Sequenz nach den Formeln $p_{i_{min}} \cdot s_{min}$ sowie $p_{i_{max}} \cdot s_{max}$ multipliziert. Diese Werte und die minOccurs- und maxOccurs-Werte der jeweiligen Elemente in der Menge werden der `determineOccurrenceRelation()` übergeben. Ergibt diese für jedes Element ein eindeutiges Ergebnis, ist die Sequenz oder die Menge in diesem Punkt entsprechend ein UT oder OT oder umgekehrt. Zusätzlich gilt wieder, dass der durch den Vergleich ermittelte Untertyp keinen leeren Inhalt erlauben darf (etwa durch mindestens ein $p_{min} = 0$ oder $s_{min} = 0$ bzw. $a_{min} = 0$), wenn es der korrespondierende Obertyp nicht auch kann.

Zum Schluss erhält die `decide()`-Methode beide Teilentscheidungen. Das Ergebnis ST kann hier nur entstehen, wenn beide Typen nur ein Element tragen, welches zudem jeweils insgesamt gleich häufig auftritt. Für den umgekehrten Vergleich einer Menge mit einer Sequenz werden die Typ-Parameter wieder in den vorgestellten Normalfall T_1 = Sequenz und T_2 = Menge gebracht und das Ergebnis durch die `reverseRelation()`-Methode vertauscht.

Vergleich einer Alternative mit einer Alternative Für diesen Fall kommen drei Entscheidungsstufen zum Einsatz. Die erste beinhaltet erneut die Überprüfung der vorhandenen Elementnamen bzw. EIDs in beiden Typen durch die Methode `elementNameRequiredRelation()`. Diese wird hier so konfiguriert, dass nicht optionale Elemente als optionale behandelt werden, was konform zur Natur der Alternative ist. Ist die Differenzmenge von T_1 leer und die von T_2 nicht, bedeutet dies also unabhängig von den minOccurs-Werten der zweiten Differenzmenge, dass T_2 ein OT ist. Umgekehrt ist T_2 ein UT. Haben beide Differenzmengen noch Elemente übrig, gilt auch in diesem Vergleich, dass T_2 ein PT ist, während ST indiziert ist, sollten beide Differenzmengen leer sein.

Darauf folgt die Bewertung der Elementhäufigkeiten. Diese werden jeweils mit dem Einfluss der Häufigkeit des <choice>-Kompositors mittels `elementGroupOccRelation()` nach den Formeln $p_{min} \cdot c_{min}$ sowie $p_{max} \cdot c_{max}$ bestimmt. Kann einheitlich für alle Elemente gesagt werden, dass sie eingeschränkter oder allgemeiner sind, lautet das Urteil für T_2 entsprechend UT oder OT, während eine exakte Übereinstimmung in ST und Widersprüche in PT resultieren.

Beide Entscheidungen werden per `decide()`-Aufruf in die vorläufige Entscheidung `eleRelation` zusammengeführt. Ergibt diese weiterhin ein eindeutiges Erebnis (nicht PT), so werden als letzte Stufe des Vergleichs die Mächtigkeiten der Sequenzen untersucht. Prinzipiell müsste

dazu die Anzahl aller möglichen Wahlen an minOccurs- und maxOccurs-Werten sämtlicher Elemente sowie der <choice>-Kompositoren bestimmt werden. Ist diese für T_2 z. B. größer und das Ergebnis der `eleRelation` ist UT, kann T_2 die maximale und minimale Kombination von T_1 nur mit Abstrichen darstellen, aber dafür noch andere Werte, die T_1 wiederum nicht darstellen kann, sodass sich ein Widerspruch ergibt.

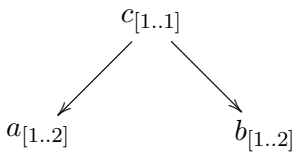


Abbildung 5.2.: Beispielhafte Darstellung des Inhaltsmodells einer Alternative.

Eine Alternative wie in Abbildung 5.2 ermöglicht die vier verschiedene Entscheidungsmöglichkeiten. Erhöht man das maximale Vorkommen der Alternative auf [1..2], ist bei zwei Wahlen jeder der Ausgangsmöglichkeiten von nur einer Wahl noch mal für die zweite Wahl der Alternative möglich, was in dem Beispiel $4 \cdot 4 = 4^2 = 16$ Wahlkombinationen ergibt. Insgesamt bietet so eine Alternative also $16+4 = 20$ Wege, aus den Elementen auszuwählen. Verschiedene Wahlmöglichkeiten ergeben mitunter identische Elementkombinationen (z. B. für $c = 1$ und $a = 2$ oder $c = 2$ und $a = 1$). Für die Mächtigkeit einer Alternative ist dies allerdings unerheblich, sodass die Kombinationsmöglichkeiten als das entscheidende Kriterium angesehen werden. Die Anzahl der Kombinationen ergibt sich also aus der Summe über der Anzahl aller möglichen Elementkardinalitäten potenziert mit einem Wert für die Kardinalität der Alternative, was nochmals für alle Kardinalitäten der Alternative summiert wird. Dies resultiert in Formel 5.1:

$$|\text{Kombinationen}| = \sum_{k=c_{min}}^{c_{max}} (\sum_{i=1}^{n_c} p_{i_{max}} - p_{i_{min}} + 1)^k \tag{5.1}$$

Ist c_{max} von T_2 bereits um eins größer als der entsprechende Wert von T_1, hat T_2 ein Vielfaches mehr an Kombinationen, unabhängig davon wie die minOccurs-Werte aussehen.

Als Vereinfachung wird daher angenommen, dass derjenige Typ Obertyp des anderen ist, der den höheren c_{max}-Wert besitzt, was die Methode `choiChoiCombinationRelation()` testet. Es ist lediglich c_{max} wichtig, da c_{min} bereits für die Beurteilung der Elementhäufigkeit eingeflossen ist. Sonderfälle die durch den Wert „unbounded“ entstehen werden entsprechend abgefangen. So werden z. B. beide Alternativen als gleichmächtig (Resultat ST) angesehen, sollten diese jeweils „unbounded“ sein.

Die zusammengefasste `eleRelation` und die `choiCombiRelation` müssen sich zum Abschluss der Beurteilung durch die `decide()`-Methode unterziehen, deren Ergebnis auch hier der finale Rückgabewert der Methode ist.

Vergleich einer Menge mit einer Alternative Das Vorgehen für diesen Vergleich ist vom Prinzip her identisch zu dem für den Vergleich einer Sequenz mit einer Alternative. Als Besonderheit ergibt sich hier der Umstand, dass die Elemente, die im XML-Schema durch eine all-Elemengtgruppe beschrieben werden, in XML-Dokumenten beliebig im angeordnet werden können. Ein Element mit der Kardinalität von [2..3] muss im Gegensatz zur Sequenz und Alternative z. B. nicht zwingend zwei oder drei Mal hintereinander erscheinen, sondern kann mit anderen Elementen verwoben sein. Diese Gegebenheit beachtet auch die Methode `handleAllChoi()`, die mit der Belegung T_1 = Menge und T_2 = Alternative arbeitet.

Als Erstes wird mit der Bildung der Element-Differenzmengen überprüft, ob die Elementnamen bzw. EIDs von T_1 überhaupt im Inhaltsmodell von T_2 auftauchen. Ist die Differenz der Menge

bereits mit mindestens einem nicht optionalen Element gefüllt, handelt es sich bei T_2 um einen PT. Sind jedoch alle verbleibenden Elemente von T_1 optional, wäre eine Einschränkung durch das Weglassen dieser in T_2 erlaubt (Ergebnis ist UT). Ist das Resultat der `elementNameRelation` ST, UT oder OT, muss diese Aussage noch gegen die involvierten Elementhäufigkeiten geprüft werden. Diese Analyse erfolgt in zwei Schritten, wobei zuerst geschaut wird, ob die Alternative evtl. ein Untertyp der Menge ist. Sollte dies nicht gegeben sein, wird danach getestet, ob die Alternative mächtig genug ist ein Obertyp zu sein. Beide Tests betrachten ausschließlich die Schnittmenge beider Elementmengen.

Der erste Test geschieht in der Methode `allCanDisplayChoiElements()`. Ist die Schnittmenge größer als 1, so muss die Menge alle Elemente optional halten, indem diese einen minOccurs-Wert von 0 besitzen, denn nur so kann diese ein Obertyp der Alternative sein. Es reicht aus, wenn dies für ein Element nicht mehr gilt, um die Menge als Obertyp zu disqualifizieren, da beim Realisieren der Menge in einem XML-Dokument jenes Element auf jeden Fall mit erscheinen muss. Befindet sich in der Schnittmenge jedoch nur ein Element, darf dieses prinzipiell einen beliebigen minOccurs-Wert besitzen. Dieser darf aber weiterhin nicht strenger als der minOccurs-Wert des entsprechenden Elements in der Alternative sein. Zudem darf die Anzahl an minimalen Wahlen der Alternative nicht zu groß sein, sodass $p_{i_{min}} \cdot a_{min} < p_{j_{min}} \cdot c_{min}$ gilt. Sind mehrere Elemente im Schnitt enthalten, muss wie beschrieben jedes $p_{i_{min}}$ 0 sein, sodass nur noch das maximale Vorkommen interessant ist. In diesem Fall wird mit der Methode `determineOccurrenceRelation()` untersucht, ob die Elemente der Alternative den üblichen Regeln der Einschränkung folgen. Hat ein Element z. B. ein höheres maximales Vorkommen, so ist die Menge kein Obertyp der Alternative und das Ergebnis lautet PT. Sollten alle Vorkommen tatsächlich identisch sein (die minOccurs-Werte der Elemente in der Alternative sind auch 0), würde das Ergebnis ST lauten, da die Menge jedoch beliebige Elementanordnungen zulässt, wird das Ergebnis auf UT für die Alternative korrigiert.

Ist das Ergebnis dieses Tests PT, betrachtet die Methode `choiCanDisplayAllElements()` die Rückrichtung. Der wesentliche Punkt ist hier der, dass der minOccurs-Wert keines Elements der Alternative größer als 1 sein darf. Ist $p_{i_{max}}$ größer als 1, so kann die Menge dieses Element wieder auf verschiedene Weisen anordnen, während die Alternative gezwungen ist, bei der Wahl von $p_{j_{max}}$ zur Realisierung die Elemente hintereinander aufzuführen. Um den gleichen Wertebereich zu beschreiben besteht die einzige Chance für die Alternative also darin, die maximalen Vorkommen der Elemente in der Menge mit dem Wert $p_{j_{min}} = 1$, also individuellen Wahlen zu „simulieren“. Dieses Verhalten wird in der Methode durch das Abziehen der $p_{i_{max}}$-Werte von c_{max} in Schleifeniterationen umgesetzt. Reichen die maximalen Wahlen nicht mehr aus, ist T_2 also auch kein Obertyp. Sonderfälle durch „unbounded“ werden abgefangen. Weiterhin summiert die Schleife alle $p_{i_{min}}$. Ist c_{max} größer als diese Summe, kann die Alternative zu kleine Werte bzw. Strukturen nicht mehr darstellen und ist auch ein PT. Dies trifft allerdings nur dann zu, wenn für kein einziges Element $p_{i_{min}} = 0$ gilt. Sollte dies so sein, kann die Alternative überflüssige Wahlen für dieses Element aufwenden.

Das Ergebnis dieser beiden Tests wird in der Variablen `canDisplayRelation` festgehalten. Zusammen mit der `elementNameRelation` liefert die `decide()` eine abschließende Entscheidung über die Beziehung von T_1 und T_2. Sollte die Belegung mit T_1 = Alternative und T_2 = Menge vertauscht sein, werden auch hier beide Parameter sowie das Ergebnis mit `reverseRelation()` ebenfalls vertauscht, sodass das Vorgehen selbst identisch bleibt.

Vergleich einer Menge mit einer Menge Da die umgekehrten Fälle durch die Umkehrung der Parameter abgedeckt sind, bleibt einzig dieser Fall zu untersuchen. Dieser Vergleich gestaltet sich dabei dem Vergleich von zwei Sequenz-Typen sehr ähnlich. Die erste Entscheidung über die verwendeten Elementnamen bzw. EIDs liefert die Methode `elementNameRequiredRelation()`,

die dieses mal wieder so konfiguriert wird, dass fehlende, nicht optionale Elemente in mindestens einer von beiden Differenzmengen zum Ergebnis PT führt.

Die Entscheidung über die Elementhäufigkeiten fällt die Methode `elementGroupOccRelation()`. Auch bei der klassischen Einschränkung einer Menge mit einer anderen dürfen keine Widersprüche entstehen. Beide Entscheidungen werden mit `eleRelation` per `decide()` zusammengefasst. Zum Schluss erfolgt der Vergleich der Kardinalitäten der <all>-Kompositoren. Diese werden noch mal extra betrachtet, um sicher zu gehen, dass z. B. das Produkt 0 nach der Multiplikation $p_{i_{min}} \cdot a_{i_{min}}$ in der Methode `determineOccurrenceRelation()` im potentiellen Untertypen nicht durch $a_{i_{min}} = 0$ entsteht, während $a_{j_{min}}$ im potentiellen Obertypen 1 lautet und die 0 von dessen $p_{j_{min}}$ stammt. Das Ergebnis dieses Vergleichs dient dann mit `eleRelation` als Parameter des nachfolgenden Aufrufs von `decide()`. Der Rückgabewert ist bereits die finale Entscheidung für diesen Fall.

Vergleich des Attributvorkommens Attribute werden in der Methode `compareAttributes()` verglichen. Zu Beginn betrachtet diese Methode alle Attribute, die im komplexen Typen T_1 vorhanden sind. Für jedes Attribut werden dessen use- sowie fixed-Attribute extrahiert und als jeweils erster Parameter für die Methodenaufrufe `attUseRelation()` und `attFixedRelation()` eingesetzt. Die entsprechenden zweiten Parameter stammen vom gleichen Attribut in T_2, sofern dieses dort definiert ist. Es wird automatisch der Wert „prohibited“ als zweiter Parameter für den Aufruf von `attUseRelation()` genutzt, sollte das Attribut nicht in T_2 vorliegen.

Ist der use-Wert in T_1 „optional“ und in T_2 nicht, liegt ein UT vor, während das Ergebnis in umgekehrter Reihenfolge OT lautet. Sind beide Werte identisch, stimmen beide Typen in diesem Attribut mit dem Resultat ST überein. In allen anderen Fällen, bei denen sich die use-Werte unterscheiden (Kombinationen aus „prohibited“ und „required“) folgt die Rückgabe der Konstante PT.

Die Beziehung der fixed-Werte ist auf einer ähnlichen Weise feststellbar. War ein Wert für das Attribut in T_1 gegeben und in T_2 nicht, ist letzterer allgemeiner und damit ein OT. Entsprechend umgekehrt wäre T_2 strikter und ein UT. Sind beide Werte nicht gesetzt oder identisch, gleichen sich beide Typen im betrachteten Attribut, sodass das Ergebnis ST ist.

Ergab die Element-Beziehung $ET1$ wird der Fall abgefangen, in welchem ein Attribut zwar in T_1, aber nicht bzw. nur mit dem Wert „prohibited“ in T_2 vorhanden ist. Das Resultat fällt mit ST für das betrachtete Attribut neutral aus. Sollte der Vergleich des fixed-Wertes und/oder der des use-Wertes entweder OT oder UT ergeben, liegen Widersprüche im Attributvorkommen vor, sodass T_1 keine Erweiterung sein kann, das Ergebnis lautet dann PT. Analog gilt dies, wenn T_2 andersherum laut der Element-Beziehung selber eine Erweiterung ist ($ET2$).

Daraufhin folgt die Untersuchung der exklusiv in T_2 vorhandenen Attribute mit der Methode `attUseRelation()`. Hier wird einfach der erste Parameter auf „prohibited“ gesetzt, wodurch das Resultat wieder durch die oben erwähnten Fälle bestimmt wird. Die beiden einzelnen Entscheidungen über die Schnitt- und T_2-exklusiven Attribute führt abschließend ein Aufruf von `decide()` zusammen.

Vergleich von weiteren Typkomponenten Die Attribute defaultAttributesApply und mixed sind zwei weitere wichtige Eigenschaften von komplexen Typen. Das erste Attribut wird bereits beim normalen Attributvergleich abgehandelt, da die Erstellung eines `CtContent`-Objekts automatisch nach den Standard-Attributen sucht. Die zweite Eigenschaft analysiert die Methode `compareMixed()`. Ist das mixed-Attribut in T_1 „true“ und T_2 „false“, so ist dieser ein UT. Umgekehrt ergibt sich das Ergebnis OT. Das Resultat lautet ST, wenn beide den gleichen Wert haben. Generell darf eine Erweiterung nicht die Varietät verändern, es sei denn sie fügt z. B. nur neue Attribute hinzu. War die Elementrelation $ET1$ oder $ET2$ und das Ergebnis hier ist

OT oder UT, folgt somit PT. Daneben ließen sich noch Zusicherung vergleichen, inwiefern diese die Beziehungen beeinflussen. Aufgrund der Komplexität dieser Thematik, die weit über den Rahmen der Arbeit geht, wurde davon abgesehen.

5.4. Löschen von Typen

Dieser Abschnitt befasst sich mit der Löschung von Typen aus einem vorliegenden EMX-Modell, wobei die Bestimmung eines Kompensations-Typen für referenzierende Komponenten vom primären Interesse ist. Dieser Prozess wird nachfolgend für einfache und komplexe Typen getrennt voneinander betrachtet.

5.4.1. Einfache Typen

Die hauptsächlich involvierten Klassen sind `StDeleter`, `StLexicalSpace` und `StDeleterDialog` im Package `de.uni_rostock.dbis.codex2.client.dialog.menuDialogs`. Der Löschvorgang wird durch eine `StDeleter`-Instanz initialisiert. Nach der Konstruktion der Instanz, die stets eine Referenz auf den zu löschenden Typen hält, lässt sich der Vorgang mittels Aufruf der Methode `start()` in Gang setzen.

Vorbereitungsphase Wie schon bei der Visualisierung von einfachen Typen wird zunächst beim Auslösen der Erzeugen des `StDeleters` mittels `Utils.findSts()` sowie `Utils.populateParentStMap()` die komplette Hierarchiebeziehung zur einfacheren Handhabung in `TreeMap`-Datenstrukturen abgebildet. Anschließend überprüft die Methode `determineAffectedEntities()`, welche EMX-Komponenten überhaupt betroffen sind. Die `Maps affectedDefinitions` und `affectedDeclarations` vermerken einfache Typen und Elemente bzw. Attribute getrennt voneinander, da für Definitionen auf jeden Fall eine Kompensation stattfinden soll, auch wenn der Nutzer eventuell die Löschung aller direkt referenzierenden Deklarationen wünscht. Die Bestimmung eines Kompensations-Typen basiert hauptsächlich auf den Vergleich der built-in Datentypen, wie im Konzept-Teil vorgeschlagen. Deswegen wird anschließend in der Methode `setSubst()` eine Art Verzeichnis erstellt, welche für jeden built-in Datentypen festlegt, durch welche Menge an anderen built-in Datentypen der betrachtete substituierbar ist. Die Beziehungen sind direkt aus dem Standard entnommen (siehe Abbildung A.1). Sobald dies erledigt ist, wird jedem einfachen Typ im EMX-Modell eine eigene `StLexicalSpace`-Instanz per rekursiven Aufruf von `setLexicalSpaces()` zugeordnet.

Ermittlung von lexikalischen Bereichen und beteiligten built-in Datentypen Diese Hilfsklasse enthält einerseits eine Referenz auf den entsprechenden Typen und andererseits einen Vermerk über dessen `variety`. Zusätzlich dazu werden alle beteiligten built-in Datentypen durch ein Verzeichnis über `InvolvedBuiltin`-Instanzen vermerkt. Für Restriction- und List-Typen enthält dieses für gewöhnlich nur einen Eintrag. Basieren diese auf einem Union-Typen, kann es wie für diesen selbst passieren, dass mehrere built-in Datentypen bei der Typdefinition beteiligt sind und dementsprechend in diesem Verzeichnis auftreten. Daneben existieren Variablen für alle Facettenarten, um etwa bei der Löschung eines Union-Typen, der über verschiedene Wege der Hierarchie dieselben built-in Datentypen mehrmals als Mitgliedstypen besitzt, den allgemeineren als Kompensation auszuwählen[2]. Zu guter Letzt enthält jede `StLexicalSpace`-Instanz noch Verweise auf mögliche `StLexicalSpace`-Instanzen von anderen nutzerdefinierten Basis- oder Mitgliedstypen. Die `variety` eines lexikalischen Bereichs ergibt sich immer auf Grundlage der übergeordneten Bereiche, sodass zwischen „atomic“, „list“ und „atomic-union“ bzw. „list-union“ unterschieden wird. Eine `InvolvedBuiltin`-Instanz beinhaltet neben dem Namen des built-in

[2] In der momentanen Umsetzung allerdings nicht realisiert.

Datentypen auch Verweise auf den aktuellen einfachen Typen und dessen Vorgänger, welche den built-in Datentyp entlang eines Hierarchiepfades verwendet haben. Zudem wird notiert, in welcher Variety der built-in Datentyp zuletzt vorliegt. Dadurch ist der Rückschluss möglich, welcher Typ innerhalb eines Hierarchiepfades z. B. die Variety des Typen „xs:decimal“ von „atomic“ auf „list“ verändert hat. Weiterhin kann nun gesagt werden, ob ein nutzerdefinierter Datentyp Whitespaces erlaubt, da einfach im zugeordneten `StLexicalSpace` nach den `InvolvedBuiltin`-Typen geschaut werden muss, was grundlegend für das weitere Vorgehen ist.

Löschphase Nach der Vorbereitung ruft die `start()`-Methode nun die Methode `determineHierachyCase()` auf. In zwei `boolean`-Variablen wird notiert, ob der zu löschende Typ ein Wurzel-Typ ist und ob weitere Typen von diesen abgeleitet sind. Mit diesen Merkmalen ist seine Position in der Hierarchie gemäß Abschnitt 4.2.3 bestimmbar.

Bestimmung des Kompensations-Typen Vor der eigentlichen Löschung schlägt die Methode `proposeType()` einen Typen zur Kompensation vor. Für **einschränkende Typen** ist dies immer der Basistyp.

Ist der gelöschte Typ eine **Liste**, entscheidet die Variety des Basistypen, die dem zugeordneten `StLexicalSpace` entnommen wird. Ist diese atomar, so wird die Whitespace-Fähigkeit des `InvolvedBuiltin`-Datentyp in der Methode `allowsSingleWhitespace()` geteset. Kann dieser Whitespaces darstellen (der built-in Datentyp ist „xs:string“, „xs:normalizedString“ oder „xs:token“), ist der Basistyp des Listen-Typen der perfekter Kandidat für die Kompensation. Sollte die Variety des Basistypen jedoch atomar-vereinigt sein, ermittelt die Methode `determineMostCommonBuiltinType()` den allgemeinsten beteiligten built-in Datentypen. Der übergeordnete Typ muss nicht zwangsläufig selbst eine Vereinigung sein, sodass stattdessen auf das Verzeichnis über die `InvolvedBuiltin`-Instanzen des bisherigen Hierarchiepfades zurückgegriffen wird. Ein built-in Datentyp ist dann allgemeiner als alle anderen in der Vereinigung beteiligten, wenn diese im Substitutions-Verzeichnis des betrachteten Typen vermerkt wurden. Besteht der allgemeinste built-in Datentyp noch den Whitespace-Test, kann der Basistyp des zu löschenden Listen-Typ genauso alle Wertebereiche in einer Zeichenkette (bzw. ehemaligen Listenwerte) gleichzeitig darstellen, und ist damit der optimale Typ für die Kompensation.

Die Suche nach dem allgemeinsten Mitgliedstypen erfolgt auch bei der Löschung eines **Vereinigungs-Typen**. Hier kann es allerdings auch passieren, dass der gelöschte Typ eine listenwertig-vereinigte Varietät besitzt. Der allgemeinste built-in Datentyp muss also auch Whitespaces erlauben. Liegt so ein Datentyp vor, extrahiert die Methode `getLatestType()` die zuletzt vermerkte Typdefinition, die den entsprechenden built-in Typ realisiert. Das Ergebnis für die Kompensation ist in diesem Fall erneut eindeutig.

Kann kein Kompensations-Typ ermittelt werden, da entweder die Varietät unpassend ist oder Whitespaces nicht möglich sind, wird „xs:string“ ausgewählt. Ist dieser Typ noch nicht vorhanden, dann wird dieser dem EMX-Modell zuvor noch hinzugefügt.

Nutzerinteraktion Der ermittelte Kompensations-Typ wird dem Nutzer anschließend in einem `StDeleterDialog` präsentiert. Hier hat dieser auch einen Einblick darüber, welche Deklarationen und Definitionen von der Löschung betroffen wären. Dem Nutzer bietet sich nun die Möglichkeit, den vorgeschlagenen Typen für die Kompensation oder keinen zu wählen oder auch die ganze Aktion abzubrechen. Die zweite Option resultiert in der Entfernung der Deklarationen. Unabhängig von der Entscheidung wird möglichen abgeleiteten Typen der Kompensations-Typ als neue Grundlage zugewiesen. Das Modell wird also konsistent gehalten. Dieser Prozess findet wiederum im `StDeleter`, welcher dem `StDeleterDialog` bei seiner Erstellung übergeben wurde, durch den Aufruf von `deleteType()` statt.

Nachbearbeitung Die Aktionen dieser Methode sind abhängig vom eingangs ermittelten Hierarchie-Fall. In allen Fällen wird auf jeden Fall die Methode `deleteOrUpdateAffectedEntities()` aufgerufen. Ist der gelöschte Typ ein innerer Knoten oder die Wurzel eines Hierarchiepfades, so werden in Abhängigkeit seines Modus Längen-Facetten (List-Typ) und/oder assertion-Facetten (List-Typ, Union-Typ) rekursiv in allen abgehenden einschränkenden Typen per `removeFacets()` gesucht und ebenfalls gelöscht (siehe auch Fall k2.2 und k2.3 in 4.2.3). Je nachdem wie die Entscheidung des Nutzers ist, ruft die Methode `deleteOrUpdateAffectedEntities()` nun `deleteDeclarations()` oder `updateDeclarations()` sowie immer `updateDefinitions()` auf. Auf Grundlage der beiden `Maps` `affectedDefinitions` und `affectedDeclarations` werden die betroffenen Entitäten nacheinander entweder der `update()` oder der `delete()` Methode übergeben. Hier kommt es zur Erzeugung des nötigen ELaX-Statements sowie der entsprechenden Operation am EMX-Modell. Die Entfernung des Typen selbst erfolgt dabei mit Hilfe der bereits in CodeX implementierten Methode `deleteSimpleType()` der Klasse `de.uni_rostock.dbis.codex2.client.emx_editor.model.EmxSimpleTypeModel`. Diese sorgt auch dafür, dass alle abhängigen Entitäten wie Facetten oder Annotationen mitgelöscht werden.

5.4.2. Komplexe Typen

Die Organisation der beteiligten Klassen zur Löschung komplexer Typen gleicht der zur Entfernung einfacher Typen. Verantwortlich für den Löschprozess sind vor allem die Klassen `CtDeleter` und `CtDeleterDialog` in `de.uni_rostock.dbis.codex2.client.dialog.menuDialogs`. Außerdem kommt eine Hilfsklasse `CtCostCalculator` zum Einsatz. Veranlasst der Nutzer die Entfernung eines komplexen Typen, wird zuerst eine `CtDeleter`-Instanz initalisiert, bevor die Interaktion im Dialog stattfindet.

Vorbereitungsphase Das Auslösen der Entfernung erfolgt auch im `CtDeleter` wieder durch den Aufruf der Methode `start()`. Dieses mal sind allerdings weniger Schritte zur Vorbereitung nötig. Hauptsächlich besteht diese Phase darin, die betroffenen EMX-Entitäten zu finden, die den komplexen Typen referenzieren. Dies können nur Elementdeklarationen sein, welche mittels der Methode `determineAffectedEntities()` in der Ergebnis-Variable `affectedDeclarations` gespeichert werden. Die Ermittlung von abhängigen Definitionen ist unnötig, da keine Typhierarchie vorliegt. Das einzige was eintreten kann, ist dass für andere Typen u. U. ein leeres Inhaltsmodell entsteht, sollte der Nutzer auch die Entfernung von Elementen wünschen, die den zu löschenden Typen referenzieren.

Löschphase Sobald die referenzierenden Elemente bestimmt wurden, beginnt die Löschphase. Als erster Schritt ist erneut die Ermittlung eines Kompensations-Typen durchzuführen, was die Methode `proposeType()` erledigt.

Bestimmung von Kompensations-Typen In CodeX sind sämtliche komplexe Typdefinitionen eigenständig und bezüglich möglicher Ableitungen unabhängig voneinander. Im Gegensatz zu den einfachen Typen ist somit ein optimaler Kandidat für die Kompensation nicht einfach greifbar. Aus diesem Grund ermittelt die Methode `proposeType()` mehrere Vorschläge. Hierzu wird jeder weiterer komplexer Typ im EMX-Modell mit dem zu löschenden verglichen. Der Vergleich beruht auf einer Kostenabschätzung durch eine Instanz der Klasse `CtCostCalculator`. Im Rahmen der Entwicklung von CodeX wurde durch Hannes Grunert in [Gru11] bereits eine Form der Kostenabschätzung für alle möglichen XML-Schemaevolutionsschritte konzipiert und implementiert. Nach einem grundlegenden Wechsel der Architektur von CodeX, ist diese Funktionalität momentan leider nicht mehr vorhanden. Daher wird eine Form der Kostenabschätzung

implementiert, die spezifisch auf den Vergleich komplexer Typen zugeschnitten ist. Der Begriff Kosten meint hier die nötigen Veränderungen an XML-Instanzomponenten, die den gelöschten Typen realisieren, sodass diese einem neuen Kompensations-Typen entsprechen. Die genaue Realisierung dieses Aspekts ist am Ende des Abschnitts in Paragraph *5.4.2 Bestimmung der Kosten* zu finden.

Nutzerinteraktion Das Resultat eines solchen Vergleichs ist ein `Cost`-Objekt, welches lediglich als Container für Teilkosten dient, wobei im Wesentlichen zwischen Lösch- und Einfüge- sowie Sortierkosten unterschieden wird. Jedem Kompensations-Kandidaten wird ein solches Objekt in der `Map` zugeordnet. Ein erzeugter `CtDeleterDialog` hat Zugriff auf die `CtDeleter`-Instanz und somit auch auf diese Abbildung. Ähnlich wie der Dialog für einfache Typen zeigt auch dieser als Erstes eine Auflistung der betroffenen Elemente. Daneben sind alle weiteren komplexen Typen dargestellt, aufsteigend nach ihren individuellen Kosten sortiert. Die Sortierung erfolgt nach der Reihenfolge Gesamtkosten > Löschkosten > Einfügekosten > Sortierkosten, sollten z. B. einige Teilkosten zwischen zwei Kandidaen identisch sein. Ein Mouseover-Tooltip schlüsselt dabei für einen Eintrag die Gesamtkosten in die Einzelkosten auf. Um die Kosten aller Typen auf einen Blick zu vergleichen, kann der Nutzer mit dem Button „Show Cost“ eine Kostentabelle öffnen. Hat der Nutzer eine Entscheidung getroffen, wird diese der Methode `deleteType()` im `CtDeleter` mitgeteilt.

Nachbearbeitung Je nachdem wie die Entscheidung ausgefallen ist, werden die in der Map `affectedDeclarations` befindlichen Elementdeklarationen durch die Methoden `updateDeclarations()` per Zuweisung des neuen Typen aktualisiert oder mittels `deleteDeclarations()`-Aufruf gelöscht. Aufrufe von `update()` und `delete()` erstellen die nötigen ELaX-Statements. Da neben eventuellen referenzierenden Elementdeklarationen und diesen zugeordneten Elementreferenzen noch weitere Komponenten der gelöschten Typdefinition selbst zu löschen sind, werden vorerst alle abhängigen Entitäten in der `Map scrapMap` gesammelt. Dies umfasst Elementreferenzen, Attributgruppenreferenzen, Elementwildcards, Assertions, die Elementgruppe des Typen und die Typdefinition selbst sowie Annotationen an diesen Komponenten. Im EMX-Modell sind Schlüsselbedingungen und diesen zugeordnete Pfadausdrücke eigene Entitäten, die unterhalb von Elementreferenzen auftreten, statt wie in XML-Schema eigentlich üblich unterhalb der darunterliegenden Elementdeklaration. Daher wird versucht für jene Entitäten, die abhängig von einer gelöschten Elementreferenz sind, eine neue Elementreferenz der gleichen Deklaration zu finden, die diese aufnehmen kann. Im `CtDeleter` stehen hierfür eine Reihe von `scrap*()` Methoden bereit, die nach den jeweiligen Komponenten Ausschau halten und der `scrapMap` hinzufügen. Nachdem alle gefunden wurden, entsorgt der Aufruf von `deleteScrap()` diese endgültig aus dem vorliegenden EMX-Modell.

Bestimmung der Kosten Wird einer Deklaration ein neuer Typ als Kompensation zugewiesen, können Instanzanpassungen folgen. Schränkt der neue Typ z. B. ein Element stärker ein oder sieht dieses in seinem Inhaltsmodell gar nicht vor, müssen diese an den jeweiligen Stellen in den XML-Instanzen verringert oder komplett gelöscht werden. Genauso ist es möglich, dass neue, notwendige Elemente (minOccurs > 0) im Inhaltsmodell auftreten, sodass Einfügung an den passenden Stellen der Dokumente verpflichtend ist. Jede einzelne solcher Manipulationen sind als Kosten auffassbar, die nötig sind, um eine vorliegende Struktur in einem XML-Dokument so zu transformieren, dass es dem neuen Inhaltsmodell entspricht. Diese Art von Kosten sind exakt bestimmbar, wäre die vollständige Instanzbasis des XMl-Schemas verfügbar. Somit ließe sich in jedem XML-Dokument nachschauen, wieviele Elemente, Attribute und andere Bestandteile z. B. zu löschen oder noch einzufügen sind, woraus eine genaue Anzahl an erforderlichen Änderungsoperationen ableitbar wäre. In CodeX ist auf die Instanzbasis für ein EMX-Modell

nicht zugreifbar, sodass diese Informationen nicht zur Verfügung stehen. Aber auch in „realen“ Szenarien ist oftmals unklar, ob weitere XML-Dokumente außerhalb der gesicherten Instanzbasis existieren, indem diese Bezug zum Namensraum des XML-Schemas nehmen. Allerdings könnte die Kostenanalyse bei großen Datenbeständen schon teurer als die Operation selbst sein.

Definition von Kosten Daher kommt statt einer genauen Berechnung eine Abschätzung der Kosten zum Einsatz, die ausschließlich auf den Informationen des Schemas bzw. EMX-Modells arbeitet, wie das Vorhandensein von Element- und Attributnamen bzw. EIDs sowie deren Anzahl im Inhaltsmodell. Um eine Änderung durchzuführen, wird davon ausgegangen, dass grundsätzlich *eine* eigenständige Operation notwendig ist, die auf der betrachteten Entität arbeitet. Demzufolge werden sowohl das Einfügen als auch das Aktualisieren und Löschen mit Kosten in der Höhe von 1 sanktioniert. Es wäre denkbar, je nach informationserweiternder, -reduzierender oder erhaltender Operation (siehe auch [Kle]) noch zusätzliche Gewichte zu verwenden, da z. B. das Löschen gravierendere Auswirkungen als das Einfügen hat. Dadurch wird allerdings unklar *wieviele* Elemente oder Attribute wirklich betroffen sind, was eine Auswertung seitens des Nutzers mitunter schwieriger macht. Deswegen wurde sich in der Umsetzung dagegen entschieden wurde.

Auf den exakten Vergleich von Wildcards und vorhandenen, konkreten Elementen und Attributen wurde verzichtet. Stattdessen wird einfachheitshalber angenommen, dass „unbekannte“ Elemente oder Attribute fehlen oder hinzuzufügen sind. Müssen Elemente im Inhaltsmodell umsortiert werden, wird dies ebenfalls mit einer allgemeinen Sortierpauschale abgeschätzt. In den nachfolgenden Paragraphen wird die programmiertechnische Umsetzung für jeden einzelnen Fälle in der Klasse `CtCostCalculator` gesondert betrachet. Vor dem Aufruf der einzelnen Methoden dieser Klasse wird ein `Cost`-Objekt erstellt. Dieses wird jenen Methoden nacheinander überreicht, die dann ihre Einzelkosten den vorgesehenen Variablen des Objekts zuweisen oder hinzufügen. Sonderfälle durch den Wert „unbounded“ für das maxOccurs-Attribut werden jeweils abgefangen und durch eine Konstante repräsentiert. Wird diese Konstante einem `Cost`-Objekt hinzugefügt, wird auf der entsprechenden Kostenposition ein vordefinierter Maximalbetrag addiert. Wie die Bestimmung der Beziehung arbeitet auch die Kostenabschätzung auf `CtContent`-Instanzen. Die Parameter $T1$ und $T2$ sind erneut so zu verstehen, dass $T2$ gegen $T1$ getestet wird, d. h. dass $T1$ der zu löschende Typ und $T2$ ein möglicher, kompensierender Typ ist. Weiterhin speichert ein `Cost`-Objekt anfallende Kosten sowohl für die minimale als auch die maximale Häufigkeit der Bestandteile von T_1. Die Gesamtkosten ergeben sich somit aus dem Durchschnitt dieser beiden Betrachtungen. Vorteilhaft ist diese Unterscheidung deshalb, da so die Eignung eines möglichen Kompensations-Typen für das komplette Spektrum des Inhaltsmodells von T_1 und nicht nur einem Teil daraus getestet wird. Damit lassen sich mögliche Unterschiede zwischen zwei Kandidaten T_2 sowohl im minimalen als auch maximalen Fall erkennen. In den meisten Fällen unterscheidet sich die Implementierung bis auf die verschiedene Belegung einer Modus-Variablen nicht, weswegen die Ausführungen hauptsächlich auf den Fall der *maximalen* Elementhäufigkeit in T_1 eingehen.

Kompensation einer Sequenz mit einer Sequenz Die Methode `handleSeqSeq()` ist zuständig für diesen Fall der Bestimmung der Elementkosten. Diese erfolgt in zwei Schritten. Als Erstes betrachtet die Methode `calcIntersectionCost()` die entstehenden Kosten für Elementnamen oder EIDs, die beide Inhaltsmodelle besitzen. Diese Methode ist so konfigurierbar, dass sie Sortierkosten veranschlagt oder nicht und findet auch in anderen Fällen Anwendung.

Es ist unklar, welche Kombinationsmöglichkeiten an minOccurs- und maxOccurs-Werten eine konkrete Instanz von T_1 annimmt. Daher wird einmal die minimale und einmal die maximale Konfiguration mit $|p_i|_{min} = p_{i_{min}} \cdot s_{i_{min}}$ sowie $|p_i|_{max} = p_{i_{max}} \cdot s_{i_{max}}$ für jedes Element p_i in der Schnittmenge beider Typen betrachtet. Die Gesamtkosten für das betroffene Element ergeben sich dann durch die Durchschnittsbildung beider Ergebnisse.

Weiterhin muss eine mögliche Kombination an Kompositor- und Elementhäufigkeiten von T_2 gewählt werden, die möglichst geringe Kosten für alle Elemente in T_1 verspricht, die in der Schnittmenge beider Typen mit einer von n_{Int} Elementen vorliegen (Int steht für die Schnittmenge und n_{Int} für die Anzahl der Elemente in dieser). In einem Zwischenschritt bestimmt die Methode `getBestSeqOcc()` dazu einen Bereich von annehmbaren Werten für die Sequenzwiederholungen von T_2, die ein `GroupRange`-Objekt repräsentiert. Dies geschieht so, dass in einer Schleife jede Häufigkeit $|p_i|_{max}$ einmal mit der Häufigkeit $p_{j_{max}}$ und einmal mit $p_{j_{min}}$ geteilt wird. Diese beiden Werte geben an, wie oft die Sequenz s_2 mindestens und maximal wiederholt werden darf, damit die Menge $|p_i|_{max}$ zahlenmäßig abgedeckt ist. Diese Werte seien daher für $i = j$ wie folgt definiert:

$$s_{2_{MinOpt}}(i,j) = \lceil \frac{|p_i|_{max}}{p_{j_{max}}} \rceil \qquad s_{2_{MaxOpt}}(i,j) = \lceil \frac{|p_i|_{max}}{p_{j_{min}}} \rceil \tag{5.2}$$

Dies wird für alle Häufigkeiten $|p_i|_{max}$ wiederholt, wodurch für die p_i in T_1 verschiedene Intervalle der Form $[s_{2_{MinOpt}}, s_{2_{MaxOpt}}]_{(i,j)}$ entstehen. Dabei können auch Werte entstehen, die außerhalb der minOccurs- und maxOccurs von s_2 liegen, was sich durch das Zurücksetzen auf diese Grenzen korrigieren lässt. Der Algorithmus ermittelt nun den kleinsten gemeinsamen Intervall, was Formel 5.3 ausdrückt.

$$s_{2_{MinOpt}} = \max_{1 \leq i=j \leq n_{Int}} \{s_{2_{MinOpt}}(i,j)\} \qquad s_{2_{MaxOpt}} = \min_{1 \leq i=j \leq n_{Int}} \{s_{2_{MaxOpt}}(i,j)\} \tag{5.3}$$

Es kann passieren, dass bei einem Vergleich der Grenzen zweier Intervalle die obere Grenze unter die untere rutscht und $s_{2_{MaxOpt}}(i,j) < s_{2_{MinOpt}}(i,j)$ gilt. Dies wird so ausgeglichen, dass die eigentlich obere Grenze auf die nun höhere, untere Grenze gesetzt wird, wodurch ein einelementiges Intervall entsteht. Es ließe sich auch andersherum ausgegleichen, allerdings „präferiert“ die erste Methode Einfügekosten vor Löschkosten, im Gegensatz zu zweiten Methode. Jeder Wert in dem nun optimalen Intervall $[s_{2_{MinOpt}}, s_{2_{MaxOpt}}]$ verspricht gleiche und ideale Ergebnisse für möglichst wenig Löschkosten. Im weiteren Verlauf wird daher einfach $s_{2_{MinOpt}}$ genutzt.

Dieser findet direkt im Anschluss in einer Fallunterscheidung Anwendung. In dieser wird überprüft, ob der Wert $|p_i|_{max}$ die möglichen Elementhäufigkeiten von p_j in T_2 unter- oder überschreitet, was Einfüge- oder Löschoperationen zur Folge hat. Diese sind mit den Formeln 5.4 (Einfügekosten EK) und 5.5 (Löschkosten LK) bestimmbar.

$$EK_i = \begin{cases} p_{j_{min}} \cdot s_{2_{MinOpt}} - |p_i|_{max} & , |p_i|_{max} < p_{j_{min}} \cdot s_{2_{MinOpt}} \\ 0 & , \text{sonst} \end{cases} \tag{5.4}$$

$$LK_i = \begin{cases} |p_i|_{max} - p_{j_{max}} \cdot s_{2_{MinOpt}} & , |p_i|_{max} > p_{j_{max}} \cdot s_{2_{MinOpt}} \\ 0 & , \text{sonst} \end{cases} \tag{5.5}$$

Im ersten Fall der Formel 5.4 müssen weitere Elemente eingefügt werden, während im ersten Fall der Formel 5.5 welche zu löschen sind. Der zweite Fall beider Fallunterscheidungen bedeutet, dass die Häufigkeit $|p_i|_{max}$ innerhalb des Bereichs der möglichen Vorkommen des betrachteten Elements in T_2 liegt. Die Untersuchung für die minimale Häufigkeiten $|p_i|_{min}$ in T_1 ist identisch zum erläuterten Verfahren.

Dieser Vorgang wird für jedes Element in der Schnittmenge wiederholt und die Kosten aufsummiert. Diese geben somit Aufschluss darüber, wie teuer die Bereinigung des entsprechenden Teils des Inhaltsmodells von T_1 in seiner minimalen und maximalen Ausprägungen ist. In Anlehnung an den obigen Formeln können die Kosten K_i für ein p_i in der Schnittmenge Int in Formel 5.6

mit $\mu = p_{j_{min}} \cdot s_{2_{MinOpt}}$ zusammengefasst werden.

$$K_i = \begin{cases} EK_i + LK_i & , (|p_i|_{max} < \mu \vee |p_i|_{max} > \mu) \wedge p_i \in T_1 \cap T_2 \\ 0 & , \text{sonst} \end{cases} \tag{5.6}$$

Was bislang ausblieb ist die Betrachtung von Elementen, die exklusiv in $Diff_1$ und/oder $Diff_2$ vorliegen. Kosten für diese berechnet im zweiten Schritt die Methode `calcRemainingCost()`, deren Paremeter so belegt werden können, dass sie entweder nur $Diff_1$, nur $Diff_2$ oder beide beachtet. Damit lässt sich diese wie schon `calcIntersectionCost()` auch für die anderen Fälle der Kompensation nutzen. In diesem Fall sind beide Richtungen zu untersuchen, da Sequenzen jedes ihrer Elemente erfordern. Für alle Elemente die nur in T_2 liegen, wird aus Gründen der Konsistenz erneut $s_{2_{MinOpt}}$ als Kompositor-Häufigkeit verwendet, wodurch für jedes Element p_j notwendige Einfügekosten NEK_j nach Formel 5.7 entstehen.

$$NEK_j = \begin{cases} p_{j_{min}} \cdot s_{2_{MinOpt}} & , p_j \notin T_1 \cap T_2 \\ 0 & , \text{sonst} \end{cases} \tag{5.7}$$

Die lokale Häufigkeit $p_{j_{min}}$ wird genutzt, um nicht unnötig hohe Kosten zu veranschlagen. Die Berechnung ist auch im minimalen Fall von T_1 durchzuführen. Existiert ein Element p_i gar nicht im Inhaltsmodell von T_2, so muss dessen gesamtes Vorkommen gelöscht werden, was in notwendige Löschkosten NLK_i resultiert. Formel 5.8 drückt dies aus.

$$NLK_i = \begin{cases} p_{i_{max}} \cdot s_{1_{max}} & , p_i \notin T_1 \cap T_2 \\ 0 & , \text{sonst} \end{cases} \tag{5.8}$$

Damit sind alle Komponenten abgedeckt, die Einfüge- oder Löschkosten verursachen. Formel 5.9 fasst dies in K_{ges} zusammen.

$$K_{ges} = \sum_{i=1}^{n_{Int}} K_i + \sum_{i=1}^{n_{s_1}} NLK_i + \sum_{j=1}^{n_{s_2}} NEK_j \tag{5.9}$$

Während beider Teilschritte werden parallel die Vorkommen an Elementen addiert, die sich nach der zahlenmäßigen Bereinigung des Inhaltsmodells von T_1 im minimalen und maximalen Fall (bezogen auf die lokalen Elementhäufigkeiten) in diesem befinden. Wurde während der Teilprozesse festgestellt, das bereits die Schnittelemente und/oder die eingefügten Elemente nicht der Ordnung von T_2 entsprechen, fallen zusätzlich (minimale und maximale) Sortierungskosten an, die den oben ermittelten Zahlen entsprechen. Diese sind nicht zu vernachlässigen, denn von zwei ansonsten identischen T_2 erfordert derjenige weitere Operationen, der nicht die gleiche Ordnung besitzt. Eine genaue Unterscheidung in welchen Punkten die Ordnung nicht übereinstimmt, erfolgt allerdings nicht. Dies wird damit begründet, dass ein Sortierungsalgorithmus mit der Laufzeit von $\mathcal{O}(n)$ unabhängig von der Ordnung alle Elemente mindestens einmal durchlaufen müsste. Dementsprechend wurden die Anzahlen der Elemente als geeignetes, vereinfachendes Maß angesehen. Die realen Kosten entstünden dabei nicht durch das Durchlaufen der Elementliste, sondern durch die tatsächlichen Verschiebeoperationen. Weiterhin wird nur auf den s_{min}-Wert eingegangen, da sich die eigentliche Komplexität auch bei höheren Werten nicht ändert (außer für „unbounded“, was jedoch einerseits durch die Konstante für Maximalkosten abgefangen wird und andererseits in realen XML-Dokumenten niemals eintreten kann).

Nachdem alle Teilkosten ermittelt wurden, erfolgt am Ende eine Halbierung aller Lösch-, Einfüge-, und Sortierkosten zur Bildung des Durchschnitts. Damit ist die Bearbeitung des –

`Cost`-Objekt fertiggestellt und wird dem `CtDeleter` zur Anzeige in einem `CtDeleterDialog` zurückgegeben.

Kompensation einer Sequenz mit einer Alternative Für die Kompensation muss nicht unbedingt immer ein Typ mit demselben Kompositor die beste Wahl sein, weswegen auch andere Typen in Betracht gezogen werden. Die Zuständigkeit der Kostenermittlung für diesen Fall liegt bei der Methode `handleSeqChoi()`, die Parameter in der Form T_1 (Sequenz) und T_2 (Alternative) erwartet. Zunächst werden die Kosten für Elemente bzw. EIDs der Schnittmenge von T_1 und T_2 berechnet. Im Anschluss erfolgt die Bestimmung der Löschkosten für Elemente, die ausschließlich in T_1 vorhanden sind.

Die Schnittkosten berechnet die Methode `calcSeqChoiIntersectionCost()`. Das Prinzip basiert ähnlich zur Bestimmung der Beziehung zwischen einer Sequenz und einer Alternative (siehe Paragraph *5.3.2 Vergleich einer Sequenz mit einer Alternative*) auf dem Abziehen bzw. Zählen von Wahlen der Alternative von deren maxOccurs-Wert c_{max}. Im Gegensatz zur Bestimmung der Beziehung, muss hier bis zum Ende der Sequenz geschaut werden, wieviele Elemente von T_1 nicht mehr in T_2 gewählt werden können, um eine klare Aussage über die Kosten zu treffen. Der verwendete Algorithmus zur Bestimmung der Beziehung durchläuft die Schnittmenge in der Sortierung der Sequenz. Für die Bestimmung der Kosten könnte es jedoch von Vorteil sein, eine andere Reihenfolge zu wählen, mit der sich die verfügbaren Wahlen besser ausnutzen lassen und so geringere Kosten entstehen. Eine ungünstig gewählte Reihenfolge könnte so die Kompensation mit T_2 teurer erscheinen lassen, als sie eigentlich ist.

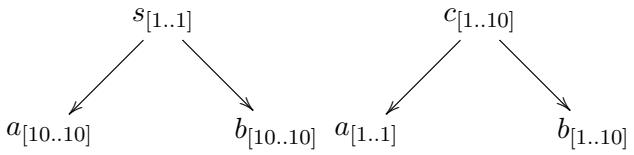


Abbildung 5.3.: Beispielhafte Darstellung des Inhaltsmodells einer Sequenz und einer Alternative.

Würden in dem obigen Beispiel 5.3 die Elemente a und b in der Alternative genau in dieser Reihenfolge gewählt, entstünden 10 Löschkosten für b, da die Alternative alle Wahlen zur Realisierung von a verbraucht hat. Wird hingegen b „zuerst“ gewählt, in dem Sinne, dass Wahlen reserviert werden, käme es lediglich zu einer einzigen Löschoperation für a. Durch die Bevorzugung von b vor a wäre mit einer Wahl der Alternative das komplette Vorkommen von b in T_1 darstellbar und es verblieben immerhin noch neun für a. Der Algorithmus ließe sich daher so verändern, dass jede mögliche Reihenfolge der Bevorzugung von Elementen im Schnitt einmal getestet wird. Dies ergibt jedoch $n!$ zu testende Permutationen, was bei größeren n nicht annehmbar ist. Eine Alternative besteht darin, mit einer Art Lookahead zu arbeiten. Dieser müsste auf die Elementhäufigkeiten in T_1 und T_2 achten. Sollte ein Element p_j, dessen lokale Häufigkeit beispielsweise bereits eher $|p_i|_{max}$ entspricht, in der Sequenz-Ordnung von T_1 erst nach anderen Elementen erscheinen, für die mehrere Wahlen der Alternative nötig wären, müssten die Wahlen der Alternative für das aktuelle p_j wie im obigen Beispiel reserviert werden.

In der Umsetzung wurde dies vereinfacht, indem über die Schnittmenge einmal nach der Ordnung der Häufigkeiten in T_2 und einmal in T_1 absteigend iteriert wird. Für beide Richtungen werden wieder die zu erreichenden minimalen und die maximalen Häufigkeiten $|p_i|_{min}$ und $|p_i|_{max}$ eines Elements betrachtet und etwaige Kosten halbiert. Je nachdem welche Richtung der Elementbevorzugung in T_2 besser ausfällt, stellt deren Ergebnis die Einfüge- und Löschkosten der Elemente im Schnitt dar. Anschließend folgt eine Erläuterung eines Iterationsschritt am Beispiel

der maximalen Häufigkeit $|p_i|_{max}$. Ein Iterationsschritt ermittelt zuerst die zu tätigen Wahlen für das betrachtete Element p und anschließend die Kosten, die von den insgesamt verbleibenden Wahlen abhängen. Die Anzahl der vollständig nötigen Wahlen W_i für ein Element p_i (bzw. p_j, es gilt wieder $i = j$) einer Iteration werden so ermittelt, dass zuerst die nötigen Wahlen der Alternative in Abhängigkeit von der Häufigkeit $p_{i_{max}}$ lokal betrachtet (LW_i) und danach mit s_{max} multipliziert werden, was Formel 5.10 ausdrückt.

$$W_i = LW_i \cdot s_{max} \tag{5.10}$$

Dies ist erforderlich, da eine Alternative mächtig genug sein muss, um auch für weitere Sequenz-Wiederholungen dieselben Wahlen zuzulassen. Damit ergibt sich für die lokalen Wahlen LW_i zunächst die Fallunterscheidung nach folgender Formel 5.11:

$$LW_i = \begin{cases} 1 & , p_{i_{max}} < p_{j_{min}} \\ GW_i + RW_i & , \text{sonst} \end{cases} \tag{5.11}$$

mit GW_i(ganze Wahlen) und RW_i(Restwahl). Der erste Fall bedeutet, dass sich das lokale, maximale (oder minimale) Vorkommen des Elements bereits mit einer einzigen Wahl der Alternative abdecken lässt. Im zweiten Fall müssen i. A. mehrere Wahlen getätigt werden, die sich wie folgt ergeben:

$$GW_i = \begin{cases} \lfloor \frac{p_{i_{max}}}{p_{j_{max}}} \rfloor & , p_{i_{max}} > p_{j_{max}} \\ 1 & , \text{sonst} \end{cases} \qquad RW_i = \begin{cases} 1 & , p_{i_{max}} > p_{j_{max}} \wedge p_{i_{max}} \pmod{p_{j_{max}}} > 0 \\ 0 & , \text{sonst} \end{cases} \tag{5.12}$$

Ist in Formel 5.12 das lokale Elementvorkommen $p_{i_{max}}$ kleiner als das Vorkommen $p_{j_{max}}$ in der Alternative, kann es mit einer einzigen Wahl dargestellt werden. Ansonsten wird so oft $p_{j_{max}}$ gewählt, bis $p_{i_{max}}$ exakt erreicht wird oder eine weitere Wahl diesen Wert überstiege. Im letzten Fall wird dann mit einer weiteren Restwahl eine passende Häufigkeit gewählt, mit der sich die Differenz schließen lässt.

Anschließend sind die entstehenden Kosten zu berechnen. Dazu werden die aktuelle berechneten Wahlen W_i in jeder Iteration von einer Variable c'_{max}, die am Anfang c_{max} repräsentiert, abgezogen. Bleibt die Differenz $c'_{max_{neu}} = c'_{max_{alt}} - W_i$ über oder gleich 0, so sind für das Element keine Löschoperationen notwendig. Die Methode `addSeqChoiInsertCost()` schaut in diesem Fall nach, ob eventuell Einfügeoperationen durchzuführen sind, die bei $p_{i_{max}} < p_{j_{min}}$ entstehen. Die totale Anzahl EK_i für diese zeigt Formel 5.13:

$$EK_i = \begin{cases} (p_{j_{min}} - p_{i_{max}}) \cdot s_{max} & , p_{i_{max}} < p_{j_{min}} \wedge c'_{max_{neu}} \geq 0 \\ 0 & , \text{sonst} \end{cases} \tag{5.13}$$

Wechselt $c'_{max_{neu}}$ in dem aktuellen Iterationschritt vom positiven Bereich in den negativen, so sind LK_i Löschoperationen notwendig. Diese ergeben sich aus der Differenz der Gesamthäufigkeit $|p_i|_{max}$ des Elements in T_1 und der noch verfügbaren Wahlen für die Realisierung dieser Elementhäufigkeit in T_2, in Kombination mit der dafür benutzten Häufigkeit $p_{j_{use}}$. Die Kosten gestalten sich somit nach Formel 5.14:

$$LK_i = \begin{cases} |p_i|_{max} - (c'_{max_{alt}} \cdot p_{j_{use}}) & , c'_{max_{neu}} < 0 \\ 0 & , \text{sonst} \end{cases} \qquad p_{j_{use}} = \begin{cases} p_{j_{min}} & , p_{i_{max}} < p_{j_{min}} \\ p_{j_{max}} & , p_{i_{max}} > p_{j_{max}} \\ p_{i_{max}} & , \text{sonst} \end{cases} \tag{5.14}$$

Der erste Fall für $p_{j_{use}}$ drückt wieder aus, dass die in T_2 minimal benötigte Häufigkeit erst durch Einfügeoperationen (siehe oben) erreicht werden muss, bevor diese überhaupt wählbar ist. Fall

zwei steht wie oben dafür, dass sogar $p_{j_{max}}$ nicht ausreicht und mehrmals gewählt werden musste, um die Häufigkeit von p_i zu erzielen. Dabei fällt noch mal zusätzlich ein Wert zwischen $p_{j_{min}}$ und $p_{j_{max}}$ an, sollte bei der Ermittlung der Wahlanzahlen $RW_i = 1$ sein. Zu guter Letzt kann während einer Iteration vor der erneuten Berechnung von $c'_{max_{neu}}$ der Wert $c'_{max_{alt}}$ bereits negativ sein. In jenem Fall sind die vollen Vorkommen $p_{i_{max}}$ als Löschkosten zu veranschlagen.

Die Summe aller Wahlen W_i ist ebenfalls interessant. Sollte diese kleiner als c_{min} sein, müssten nochmals notwendige Einfügekosten NEK_j wie in Formel 5.15berechnet werden, da das minimale Inhaltsmodell von T_2 sonst nicht darstellbar ist (wie oben angemerkt, gilt $W_j = W_i$, da mit i und j die gleichen Elementnamen im Schnitt bezeichnet werden).

$$NEK_j = \begin{cases} (c_{min} - \sum_{i=1}^{n_{Int}} W_i) \cdot p_{j_{MinOcc}} & , \sum_{i=1}^{n_{Int}} W_i < c_{min} \\ 0 & , \text{sonst} \end{cases} \tag{5.15}$$

Die Häufigkeit $p_{j_{MinOcc}}$ stammt von demjenigen Element, welches den global kleinsten minOccurs-Wert in T_2 besitzt. In einem glücklichen Umstand, in welchem ein Element mit $p_{j_{min}} = 0$ existiert, wären keine Einfügeoperationen vonnöten.

Damit wurden alle kostenverursachenden Komponenten der Schnittmenge betrachtet. Was noch fehlt ist die Abzählung der notwendigen Löschkosten für Elemente p_i, die nur in T_1 vorliegen, die jeweils $NLK_i = |p_i|_{max}$ betragen. Da exklusive Elemente in einer Alternative nicht verpflichtend sind, brauchen für solche in T_2 keine Einfügekosten bestimmt werden. War die Schnittmenge jedoch leer, muss wenigstens $c_{min} \cdot p_{j_{MinOcc}}$ Mal ein Element eingefügt werden. Damit ergeben sich die Gesamtkosten K_{ges} nach Formel 5.16.

$$K_{ges} = \begin{cases} \sum_{i=1}^{n_{Int}} (LK_i + EK_i) + \sum_{i=1}^{n_s} (NLK_i) + NEK_j & , T_1 \cap T_2 \neq \varnothing \\ \sum_{i=1}^{n_s} (NLK_i) + c_{min} \cdot p_{j_{MinOcc}} & , \text{sonst} \end{cases} \tag{5.16}$$

Auch in dieser Methode wurde darauf geachtet für alle Vergleiche auch die Häufigkeit „unbounded“ zu behandeln. Trifft dies z. B. auf ein $p_{j_{max}}$ zu, würde nur eine Wahl multipliziert mit s_{max} von c'_{max} abgezogen. Ist c_{max} selbst „unbounded“, so wäre mit Löschkosten nicht zu rechnen. Andersherum würden bei s_{max} = „unbounded“ oder einem Element p_i in T_1 mit diesem Wert die maximalen Löschkosten addiert, sollte c_{max} selbst oder das entsprechende Pendant p_j nicht unbeschränkt sein.

Kompensation einer Sequenz mit einer Menge Die Methode `handleSeqAll()` schätzt die Kosten für diese Kompensation. Betrachtet man diesen Fall näher, ähnelt er der Kompensation einer Sequenz mit einer anderen. Dies beruht auf der Tatsache, dass die Nutzung beider Inhaltsmodelle in XML-Dokumenten jeweils alle Elemente (mit ihren lokalen Häufigkeiten) erfordern. Die Menge ermöglicht dabei wesentlich mehr Elementkombinationen. Daher wurde ein Vorgehen gewählt, dass fast identisch zu dem bereits in Paragraph *5.4.2 Kompensation einer Sequenz mit einer Sequenz* vorgestellten ist.

Als Erstes erfolgt auch hier die Betrachtung der Elemente, die in beiden Modellgruppen vorhanden sind. Der Unterschied zu T_2 als Sequenz wirkt sich für den Algorithmus lediglich so aus, dass das Produkt $p_{j_{max}} \cdot a_{max}$ maximal $p_{j_{max}}$ sein kann. Außerdem sind niemals Umsortierungen nötig, da nach einer Anpassung der reinen Elementhäufigkeiten eine Menge dynamisch genug ist, die Ordnung jeder beliebigen Sequenz anzunehmen. Dementsprechend wird die Methode `calcIntersectionCost()` so konfiguriert, dass die Ordnung von T_1 nicht in das Gewicht fällt.

Anschließend werden durch die Methode `calcRemainingCost()` Kosten für die in den jeweiligen Differenzmengen verbleibenden Elementen berechnet.

Kompensation einer Alternative mit einer Sequenz Auch wenn Sequenzen i. A. strikter als Alternativen sind, ist diese Kompensation denkbar, da eventuell nur Sortierungskosten anfallen. Die Methode `handleChoiSeq()` untersucht diesen Fall und berechnet anfallende Operationen für T_1 (Alternative) und T_2 (Sequenz). Weil die von Alternativen beschriebenen Strukturen unterschiedlich ausfallen, wird nicht wie beim Beziehungsvergleich die Belegung der Methode `handleSeqChoi()` einfach vertauscht. Eventuelle Kosten entsprechen mit diesem Vorgehen nicht dem eigentlichen Wesen der Alternative. Formel 5.1 in Paragraph *5.3.2 Vergleich einer Alternative mit einer Alternative* zeigt, dass die Anzahl an möglichen Wahlkombinationen für entsprechende Teile in XML-Dokumenten relativ hoch sein kann. Aus diesem Grund kommt ein Verfahren zum Einsatz, was von einzelnen Extremwerten für die jeweiligen Elementhäufigkeiten ausgeht.

Die Situation, in der die Alternative ihren kompletten minOccurs- oder maxOccurs-Wert für ein einziges $p_{i_{min}}$ oder $p_{i_{max}}$ aufwendet, wird als ein Extremfall angesehen. Es gilt dann wieder z. B. $|p_i|_{max} = p_{i_{max}} \cdot c_{max}$. Da sich nicht mit absoluter Sicherheit sagen lässt, mit welcher Wahl der Häufigkeit der Sequenz das Inhaltsmodell T_1 am besten angenähert wird, kommt auch hier das Verfahren zur Bestimmung der idealen Kompositorhäufigkeit s_{MinOpt} aus *5.4.2 Kompensation einer Sequenz mit einer Sequenz* zum Einsatz. Auch wenn niemals alle Elementvorkommen $|p_i|_{max}$ in einer Alternative gleichzeitig gewählt werden können, gibt s_{MinOpt} auch hier die Häufigkeit an, die am wenigsten Löschkosten verspricht.

Dieses Mal sind die Methodenaufrufe für Elemente in der Schnitt- und/oder Differenzmenge nicht getrennt. Stattdessen ermittelt die Methode `handleChoiSeq()` alle minimalen und maximalen Kosten mit zwei Aufrufen von `getChoiSeqCost()`. Gilt in einem Durchlauf etwa $s_{MinOpt} = 0$, weil die Sequenz eine Kardinalität von [0..0] besitzt, entstehen für jeden maximalen Extremfall $|p_i|_{max}$ Löschoperationen. Sollte dieser Wert durch beispielsweise $c_{min} = 0$ oder $p_{i_{min}} = 0$ auch 0 betragen, sind keine Löschungen für dieses Element vorzunehmen. Im allgemeinen Fall, in dem ein $s_{MinOpt} > 0$ ist, müssen alle anderen Elemente, die sich noch in der Sequenz befinden und nicht von der Alternative gewählt wurden, mindestens mit der minimalen Häufigkeit eingefügt werden, was in Formel 5.17 resultiert.

$$NEK_j = \sum_{j=1}^{n_s} s_{MinOpt} \cdot p_{j_{min}}, (i \neq j) \tag{5.17}$$

Interessant ist dann, wieviele Elemente vom aktuellen $|p_i|_{max}$ mit der Sequenz darstellbar sind oder ob sogar noch weitere hinzuzufügen wären. Hierzu wird die Anzahl mit der Sequenzhäufigkeit geteilt, was der Wert $\zeta = \frac{|p_i|_{max}}{s_{MinOpt}}$ ausdrückt. Die Fallunterscheidungen in den Formeln 5.18 und 5.19 geben Aufschluss darüber, welche Art von Kosten entstehen.

$$EK_i = \begin{cases} p_{j_{min}} \cdot s_{MinOpt} - |p_i|_{max} & , \zeta < p_{j_{min}} \\ 0 & , \text{sonst} \end{cases} \tag{5.18}$$

$$LK_i = \begin{cases} |p_i|_{max} - p_{j_{min}} \cdot s_{MinOpt} & , \zeta > p_{j_{max}} \\ 0 & , \text{sonst} \end{cases} \tag{5.19}$$

Diese Formeln zeigen, warum bei dieser Berechnung die Situation $s_{MinOpt} = 0$ vorher abzufangen ist. Der erste Fall der ersten Formel bedeutet, dass die erschöpfende Wahl von p_i nicht ausreicht um wenigstens den lokalen minOccurs-Wert des gleichen Elements innerhalb von allen Sequenzwiederholungen zu erreichen. Daher muss die Differenz aus der kleinsten lokalen Elementhäu-

figkeit für alle s_{MinOpt} Sequenzwiederholungen und der immerhin erreichten Elementhäufigkeit $|p_i|_{max}$ in der Alternative zur Anpassung an T_2 eingefügt werden. Ist $|p_i|_{max}$ über alle Sequenzwiederholungen jedoch größer als das maximale Elementvorkommen, muss die entsprechende Differenz als Löschkosten hinzugefügt werden, was der erste Fall von Formel 5.19 ausdrückt. Anhand dieser Berechnungen sind die Kosten für ein p_i mit Formel 5.20 bestimmbar.

$$K_i = \begin{cases} EK_i + LK_i + NEK_j & , (\zeta < p_{j_{min}} \vee \zeta > p_{j_max}) \wedge s_{MinOpt} > 0 \\ |p_i|_{max} & , s_{MinOpt} = 0 \\ NEK_j & , \text{sonst} \end{cases} \tag{5.20}$$

Der zweite Fall wurde oben bereits erläutert. Im letzten letzten Fall liegt $|p_i|_{max}$ im Spektrum des Möglichen für das momentan betrachtete s_{MinOpt}, sodass bis auf eventuelle notwendige Einfügungen keinerlei Kosten entstehen.

Befindet sich das Element p_i überhaupt nicht in dem Inhaltsmodell der Sequenz, so sind alle $|p_i|_{max}$ Elemente der Alternative zu löschen, was in Formel 5.20 wieder Fall zwei entspricht. Dieser Vorgang wird für alle n_c Elemente in der Alternative wiederholt. Wie oben angemerkt, betrachtet der Vorgang Extremfälle. Da es nicht möglich ist, alle Wahlen der Alternative gleichzeitig für mehrere dieser Extremfälle auszugeben, ergeben sich die endgültigen Kosten erneut durch die Bildung des Durchschnittswerts nach Formel 5.21.

$$K_{ges} = \frac{\sum_{i=1}^{n_c} K_i}{n_c} \tag{5.21}$$

Dieses Berechnung erfolgt auf identischer Weise für minimale Extremwerte der Alternative. Wie bei den vorherigen Algorithmen findet auch bei diesem der Wert „unbounded“ für das maxOccurs-Attribut eine geeignete Behandlung. Ist z. B. $s_{max} = unounded$, würde bei der Untersuchung von $s_{MinOpt} = s_{max}$ die Konstante für maximale Kosten als Einfügekosten berechnet. Existiert in der Sequenz jedoch ein $p_{j_{min}} = 0$, ließe sich dieses unendlichmal wählen, sodass extra Einfügeoperationen nicht erforderlich sind.

Kompensation einer Alternative mit einer Alternative Diese Kompensation ähnelt stark der aus dem vorherigen Paragraphen *5.4.2 Kompensation einer Alternative mit einer Sequenz*. Hauptverantwortlich ist die Methode `calcChoiChoiCost()` die einen Aufruf von `getChoiChoiCost()` jeweils für die Untersuchung von $|p_i|_{min}$ und $|p_i|_{max}$ durchführt. Der Algorithmus in `getChoiChoiCost()` ist jedoch einfacher. Einerseits ist es bei der Betrachtung eines p_i in Int nicht nötig noch weitere p_j der zweiten Alternative einzufügen. Andererseits braucht z. B. die Häufigkeit $|p_i|_{max}$ nicht auf etwaige Sequenzwiederholungen aufgeteilt werden, was die Berechnung von ζ überflüssig macht. Es zählt also allein die gesamte Häufigkeit des gleichen Elements p_j in T_2. Die Formeln 5.22 und 5.23 geben die Einfüge- oder Löschkosten an.

$$EK_i = \begin{cases} p_{j_{min}} \cdot c_{2_{min}} - |p_i|_{max} & , |p_i|_{max} < p_{j_{min}} \cdot c_{2_{min}} \\ 0 & , \text{sonst} \end{cases} \tag{5.22}$$

$$LK_i = \begin{cases} |p_i|_{max} - p_{j_{max}} \cdot c_{2_{max}} & , |p_i|_{max} > p_{j_{max}} \cdot c_{2_{max}} \\ 0 & , \text{sonst} \end{cases} \tag{5.23}$$

Da diese Formeln direkten Bezug auf die minimalen und maximalen verfügbaren Wahlen für T_2 nehmen, brauchen beispielsweise keine notwendigen Einfügekosten berechnet werden. Anders sieht es aus, wenn sich das betrachtete p_i Element gar nicht im Inhaltsmodell von T_2 befindet

und somit kein Pendant p_j hat. Damit ergeben sich wieder notwendige Löschkosten, die Formel 5.24 zeigt.

$$NLK_i = \begin{cases} p_{i_{max}} \cdot c_{1_{max}} & , p_i \notin T_1 \cap T_2 \\ 0 & , \text{sonst} \end{cases} \tag{5.24}$$

Außerdem müssen wenigstens $c_{2_{min}}$ Wahlen getätigt werden. Mit diesen Werten lassen sich bereits die vorläufigen Gesamtenkosten K'_{ges} wie in Formel 5.25 beschreiben.

$$K'_{ges} = \begin{cases} \sum_{i=1}^{n_{Int}} (LK_i + EK_i) + \sum_{i=1}^{n_{c_1}} (NLK_i) & , T_1 \cap T_2 \neq \varnothing \\ \sum_{i=1}^{n_{c_1}} (NLK_i) + c_{2_{min}} \cdot p_{j_{MinOcc}} & , \text{sonst} \end{cases} \tag{5.25}$$

Wie bei der *Kompensation einer Sequenz mit einer Alternative* soll $p_{j_{MinOcc}}$ die kleinste Häufigkeit eines Elements p_j in T_2 sein. Da hier quasi einzelne Extremfälle betrachtet werden, erfolgt am Ende noch mal die Bildung der durchschnittlichen Gesamtkosten, wie in Formel 5.26 zu sehen.

$$K_{ges} = \frac{K'_{ges}}{n_{c_1}} \tag{5.26}$$

Wie bei den anderen Berechnungen fängt diese Methode den Spezialwert „unbounded“ für das maxOccurs-Attribut in verschiedenen Komponenten ab und fügt gegebenenfalls die definierte Konstante für maximale Kosten als Einfüge- oder Löschkosten hinzu.

Kompensation einer Alternative mit einer Menge Für die Berechnung der Kosten in diesem Fall wird zwar zur klareren Trennung die Methode `handleChoiAll()` mit den Parametern T_1 (Alternative) und T_2 (Sequenz) gerufen, diese bedient sich allerdings wie schon die Methode `handleChoiSeq()` an den Vorgängen in `getChoiSeqCost()`, weswegen an dieser Stelle auf den Paragraphen *5.4.2 Kompensation einer Alternative mit einer Sequenz* verwiesen ist. Wie in Paragraph *5.4.2 Kompensation einer Sequenz mit einer Menge* angesprochen, ist die Menge bezüglich der (lokalen) Elementhäufigkeiten mit einer Sequenz vergleichbar. Da der maxOccurs-Wert für T_2 maximal 1 beträgt, kann a_{MinOpt} maximal 1 betragen und ζ höchstens $|p_i|_{max}$ sein. Der eigentliche Ablauf dieser Methode bleibt dabei der Gleiche.

Kompensation einer Menge mit einer Sequenz Die implementierten Methoden sind auch für diese Kompensation einsetzbar. Aufgrund der Ähnlichkeit einer Menge zu einer Sequenz wurde die Methode `handleAllSeq()` so gestaltet, dass sie ebenfalls nur `calcIntersectionCost()` und `calcRemainingCost()` aufruft, deren Arbeitsweisen in *5.4.2 Kompensation einer Sequenz mit einer Sequenz* ausführlich vorgestellt wurden. Allerdings ist die Ordnung der Menge selbstverständlich nicht mehr an der niedergeschriebenen Elementreihenfolge ablesbar. Auch wenn `calcIntersectionCost()` eine identische Reihenfolge der Schnittelemente von T_1 = Menge und T_2 = Sequenz im Schema bzw. EMX-Modell feststellt, wird im Unterschied zu `handleSeqSeq()` hier immer eine Sortierpauschale erhoben, mit der Ausnahme, dass die Schnittmenge nur ein Element besitzt.

Kompensation einer Menge mit einer Alternative Wie in den anderen Kompensationen, die Mengen-Typen involvieren, wird auch diese auf den grundlegenden Fall der Sequenz zurückgeführt. Dies geschieht hier, indem die Methode `handleAllChoi()` sich an der Methode `calcSeqChoiIntersectionCost()` bedient, die gleichzeitig auch Elemente in $Diff_1$ und $Diff_2$ berücksichtigt. Im Gegensatz zur Kompensation aus *5.4.2 Kompensation einer Sequenz mit einer Alternative* ist die Veranschlagung der Sortierpauschale unabdingbar. Grund hierfür ist, dass die

Alternativen zwar durch ihre Wahlmöglichkeiten verschiedene Elementkombinationen ermöglicht, diese aber z. B. bei der Wahl der Häufigkeit $p_{j_{max}}$ immer hintereinander stehen müssen. Diese Einschränkung hat die Menge nicht, sodass i. A. trotzdem mit Verschiebungen der Elemente zu rechnen ist.

Kompensation einer Menge mit einer Menge Als letzte Möglichkeit für eine Kompensation verbleibt dieser Fall. Dieser stimmt jedoch nahezu vollständig mit dem eingangs in Paragraph *5.4.2 Kompensation einer Sequenz mit einer Sequenz* erläuterten überein. Die Aufrufe von `handleAllAll()` beziehen sich ein weiteres Mal auf die Methoden `calcSeqChoiIntersectionCost()` und `calcRemainingCost()`. Deren Vorgehensweisen sind unverändert. Wurden die Elementhäufigkeiten in T_1 durch eventuelle, kostenverursachende Einfüge- und Löschoperationen an denen aus T_2 angepasst, sind beide Typen gleichmächtig. In Abgrenzung zur Kompensation an der zwei Sequenz-Typen teilnehmen, sind danach keine Sortierungen notwendig.

Berechnung von Attributkosten Die Komplextität dieser Aufgabe fällt wesentlich geringer aus. Zuständig für diese sind die Methoden `calcAttributeCost()` und `calcFixedCost()`. Die erste unterscheidet ebenfalls jeweils zwischen „minimaler“ und „maximaler“ Ausprägung bzw. Häufigkeit eines Attributes in T_1. Diese bezieht sich dabei auf den Wert des use-Attributs und ist nur für „optional“ zwischen 0 und 1 variabel. In diesem Fall erzeugt z. B. der Wechsel auf „prohibited“ minimal keine Löschoperation, wenn das Attribut gar nicht genutzt wird, und maximal eine Löschoperation. Die Methode `calcFixedCost()` gibt als einzige der bisher vorgestellten Methoden mögliche Updatekosten UK_i wieder. Da die Kosten statisch sind, wird zur Übersicht möglicher Attributkosten auf die Tabelle 5.2 und Formel 5.27 verwiesen. Die Angabe $att_{i_{use}}$ bzw. $att_{j_{use}}$ meint den Wert der use-Eigenschaft des Attributs att_i in T_1 bzw. att_j in T_2. Wie immer wird T_2 gegen T_1 getestet, sodass die Lesereihenfolge der Tabelle von der ersten Reihe startet. Die Angaben in den Klammern soll andeuten, bei welcher Häufigkeit entsprechende Kosten entstehen, die in den meisten Fällen in der Höhe von 1 berechnet werden. Die Notation (Min, Max) für z. B. „required“ meint dabei, dass Updatekosten im minimalen und maximalen Fall nötig sind, da das Attribut minimal und maximal erscheinen muss (Kardinalität [1..1], es gibt keine höhere obere Grenze für Attribute, die Notation wurde aus Gründen der Konsistenz zu der vorherigen gewählt). Ist ein att_i nicht in T_2 wird dies in der Implementierung einfach mit dem Wert „prohibited“ für die Häufigkeit $att_{j_{use}}$ abgefangen.

$att_{j_{use}}$ \ $att_{i_{use}}$	**prohibited**	**optional**	**required**
prohibited	0	LK (Max)	LK (Min, Max)
optional	0	UK (Max)	UK (Min, Max)
required	EK (Min, Max)	UK (Max)	UK (Min, Max)

Tabelle 5.2.: Kostentabelle für Attribute.

Die Updatekosten UK ergeben sich in Abhängigkeit von den fixed-Attributen $att_{i_{fix}}$ und $att_{j_{use}}$ nach Formel 5.27.

$$UK = \begin{cases} 0 & \Leftrightarrow \exists att_{i_{fix}} \wedge \nexists att_{j_{fix}} \vee att_{i_{fix}} = att_{j_{fix}} \\ 0.5 & \Leftrightarrow \nexists att_{i_{fix}} \wedge \exists att_{j_{fix}} \\ 1 & \Leftrightarrow \text{sonst} \end{cases} \tag{5.27}$$

Im ersten Fall wird der Wertebereich vergrößert, sodass der bisherige, fixierte Wert des Attributs weiter bestehen darf. Ebenso braucht der Wert nicht verändert zu werden, wenn der neue fixed-Wert identisch ist. Wird durch die Kompensation mit T_2 ein neuer Wert eingefügt, kann ohne die Instanzinformationen nicht bestimmt werden, ob der bisherige Attributwert dem neuen fixed-Wert entspricht. Der Wert 0.5 für UK soll ausdrücken, dass entweder keine oder eine Änderungsoperation notwendig sein kann. Existiert in beiden Attributen ein unterschiedlicher fixed-Wert, ist entsprechend Fall drei der bestehende Attributwert von att_i immer anzupassen.

6. Auswertung

Dieses Kapitel befasst sich mit der Auswertung der Implementation. Diese erfolgt einerseits anhand der Betrachtung von ausgewählten Beispielen für die Visualisierung und andererseits durch das Analysieren von Szenarien für die Löschung und Kompensation von Typen. Gerade zum Testen der Visualisierung und der Löschung von einfachen Typen wird erneut eine angepasste Form des laufenden Beispiels für das Testschema über Personen genutzt. Für die Betrachtung von komplexen Typen liegen allerdings nicht genügend unterschiedliche Fälle vor. Aus diesem Grund werden für diese eine Reihe von minimalen Beispielen getestet, die spezielle Aspekte der Bestimmung der Beziehung (und damit der Visualisierung) sowie der Kostenberechnung für die Kompensation hervorheben. Da sich der Großteil der Implementierung nur im laufenden GWT-Programm betrachten lässt, werden die Tests manuell durchgeführt.

6.1. Visualisierung

Das Ziel, eine alternative, Typ-zentrische Ansicht in CodeX bereitzustellen konnte umgesetzt werden. Über zwei Buttons in der Werkzeugleiste lässt sich entweder eine `StViewer`-Instanz oder eine `CtViewer`-Instanz erzeugen, die entsprechend einfache oder komplexe Typen anzeigen. Da beide Komponentenarten doch relativ verschieden sind und bereits untereinander stark verknüpft sein können, wurden die Sichten zum Zweck der Übersicht getrennt.

6.1.1. Einfache Typen

Zur Demonstration werden zunächst die nutzerdefinierten Typen des Testschemas betrachtet. Anschließend werden Beispiele gezeigt, die besondere Aspekte hervorheben sollen. Der Algorithmus für die Platzierung der Typ-Diagramme geht so vor, dass die built-in Datentypen horizontal auf einer bzw. der ersten Ebene angeordnet werden. Ausgehend von diesen wachsen die einzelnen Hierarchiepfade nach unten, wodurch baumähnliche Diagrammstrukturen entstehen, welche die Ableitungsbeziehung widerspiegeln.

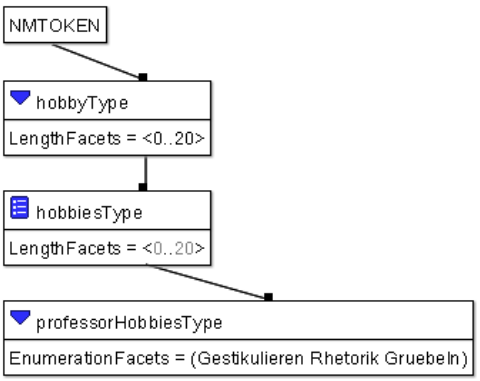


Abbildung 6.1.: Hierarchie unterhalb von „hobbyType“ in der Umsetzung.

Testschema In Abbildung 6.1 ist die Hierarchie unterhalb des Typs „hobbyType“ zu sehen. Wie angestrebt, werden die Facetten eines Typen im Diagramm dargestellt. Damit ein Nutzer einen Einblick in den Wertebereich erhält, der sich hinter einem einfachen Typ verbirgt, braucht dieser somit keine Untermenüs oder Seitenleisten aufsuchen, in denen entsprechende Facetten vermerkt sind. Die Abbildung zeigt zudem, dass die Längen-Facetten in eine Berechsangabe zusammengefasst werden. Ein Hobby ist ein „NMTOKEN“-Typ und kann im Testschema 0- bis 20-mal

wiederholt werden, was den Facetten minLength = „0“ und maxLength = „20“ entspricht. Der nächste Ableitungsschritt beinhaltet die Listenbildung auf Basis dieses Typs zum Typ „hobbies-Type“. Ein entsprechender Icon grenzt diese Ableitungsart ab.

Da in Listentypen keine Facetten definierbar sind, wird stattdessen der zugrundeliegende Wertebereich noch einmal wiederholt. Entsprechend der geplanten Unterscheidung zwischen vererbten und nicht veränderten, veränderten oder neu eingefügten Facetten ist die Angabe der Längen-Facetten für „hobbiesType“ grau eingefärbt, um den ersten Fall der Vererbung zu visualisieren. Die Varietät des Wertebereichs ist nach dieser Ableitung listenwertig, sodass die Facetten des vorangehenden Wertebereichs in der nächsten Einschränkung zum Typ „professorHobbiesType“ nicht mehr wiederholt werden. Wie in der Konzeption angesprochen, ist dies nötig, da die Semantik für jene Facetten sonst eine andere wäre. Korrekterweise stellt „professorHobbiesType“ nur die Facetten seiner eigenen Definition dar. Bei diesen handelt es sich um Aufzählungs-Facetten, die wie die Längen-Facetten zusammengefasst werden. An diesem Beispiel ist erkennbar, dass dieses Vorgehen mitunter zu breiten Diagrammen führen kann, sodass eine klassische Auflistung diskutabel ist. An dieser Stelle sei angemerkt, dass von der ursprünglich geplanten Zusammenfassung der Bereichs-Facetten wie minInclusion oder maxInclusion abgerückt wurde, da die geplante Intervall-Notation sich zu sehr mit der bereits verwendeten Notation für die verschiedenen Kardinalitäten überschneidet. Andererseits sind Veränderungen an dieser Art von Facetten deutlicher, wenn sie auf die herkömmliche Weise notiert werden.

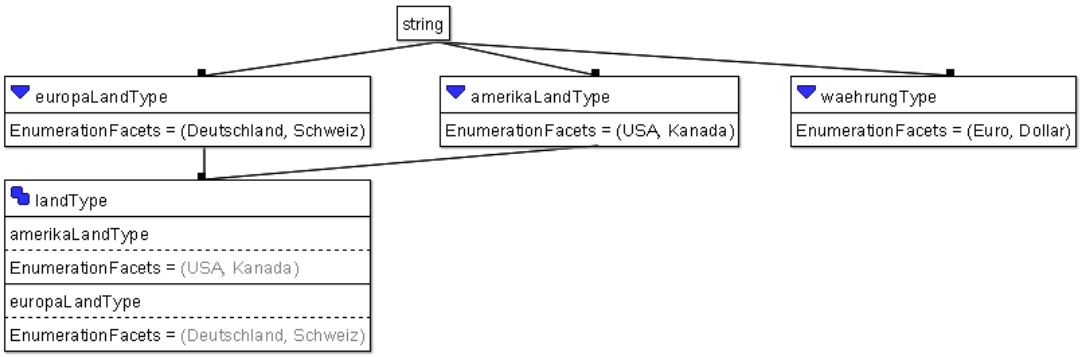


Abbildung 6.2.: Darstellung des Vereinigung „landType“ in der Umsetzung.

Die Abbildung 6.2 zeigt die Vereinigung zum Typen „landType“. Die Typen „amerikaLand-Type“ und „europaLandType“ definieren jeweils einen eigenen Satz an Aufzählungs-Facetten. Jene Facetten sind auch nach der Vereinigung beider Typen zum Typ „landType“ in diesem als Inhalt zu sehen. Um zu signalisieren woher diese stammen, werden diese unterhalb des Namens des entsprechenden Mitgliedstypen notiert. Dies bedeutet einen Gewinn an Informationen über Vereinigungen, besonders dann, wenn sich deren Mitgliedstypen nicht in unmittelbarer Nähe im Diagramm befinden. Außerdem ist auch der Typ „waehrungType“ unterhalb des gleichen built-in Datentypen „string“ zu sehen, von dem keine weiteren Typen abgeleitet sind.

Weitere Beispiele Es kann beobachten werden, dass sich viele der XML-Schemata aus realen Szenarien in vielen Fällen auf eine geringe Anzahl an built-in Datentypen beschränken. Zudem ist die Hierarchie unterhalb dieser Datentypen meistens relativ flach und somit eher ungeeignet für eine Auswertung. Abbildung 6.3 zeigt z. B. einen Ausschnitt aus dem XML-Schema für das Datenformat Machine-Readable Cataloging (MARC). Weitere XML-Schemata die betrachtet wurden, waren z. B. das Metadata Object Description Schema (MODS), eine Version des Schemas für das Datenformat Really Simple Syndication (RSS), das XML-Schema für Scalable Vector Graphics (SVG) oder auch das Schema für die Modellierung mittels Extensible 3D Graphics (X3D).

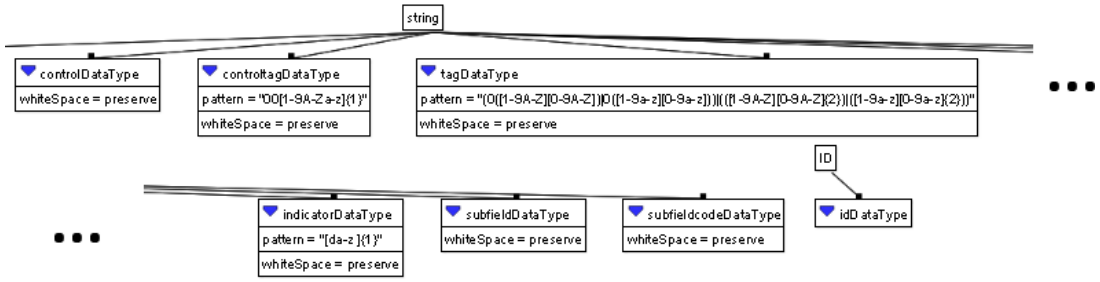


Abbildung 6.3.: Ausschnitt aus der Typhierarchie des MARC XML-Schemas.

Aus diesem Grund wurde ein kleines Schema (nachfolgend mit RLU-Schema bezeichnet) entworfen, welches in drei Ebenen verschiedene Ableitungsschritte kombiniert. Der Einfachheit halber sind die Typen nach dem Anfangsbuchstaben des jeweiligen Ableitungsschritts benannt. Der Typ „RUL“ ist also eine Listen-Typ, der auf einer Vereinigung basiert, die (mindestens) einen einschränkenden Typen beinhaltet. Die Wurzel ist der einschränkende Typ „R“ und ist von „decimal“ abgeleitet. Abbildung 6.4 zeigt die Hierarchie.

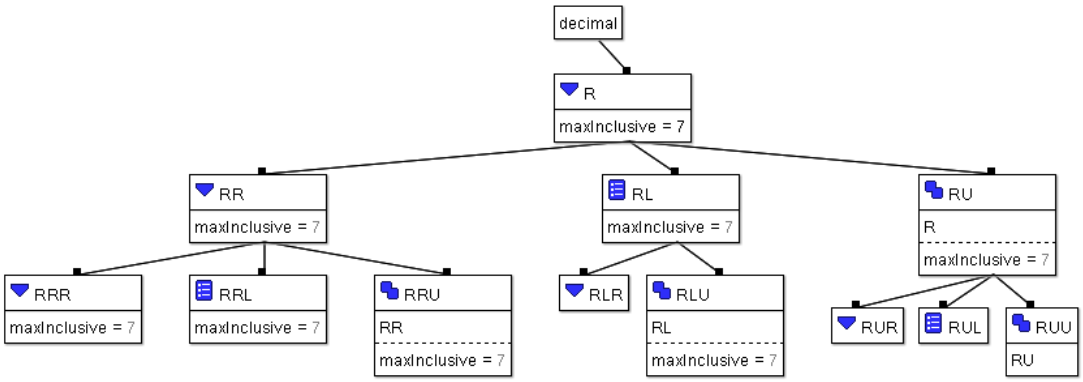


Abbildung 6.4.: Hierarchie einfacher Typen des RLU-Schemas.

Es ist zunächst insgesamt nur eine einzige Facette vorhanden, die vom Typ „R“ definiert und von allen Untertypen als vererbte Facette reflektiert wird. Wie gewünscht, zeigen die Typen „RLR“ und „RUR“ die vererbete Facette nicht mehr an. Nach der Einfügung der Facete maxExclusive = „7“ in den Typ „RR“ wird die vererbte Facette der gleichen Kategorie nicht mehr angezeigt und durch die aktuellere ersetzt. Der Wert 7 als ehemalige, obere Grenze befindet sich nun außerhalb des Wertebereichs, sodass eine Einschränkung vorliegt, was durch eine Einfärbung hervorgehoben wird. Wie in Abbildung 6.5 zu sehen, wird diese Aktualisierung auch in alle Untertypen propagiert.

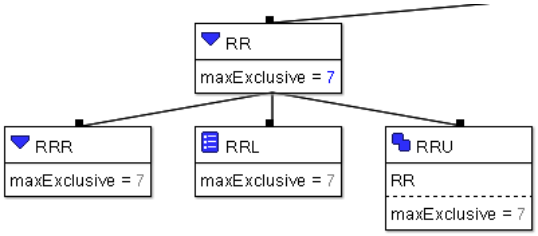


Abbildung 6.5.: Veränderte Facette im RLU-Schema.

In XML-Schema werden verschiedene pattern-Facetten, die in Ober- und Untertypen definiert sind, im Untertyp verundet. Aufgrund der Komplexität, die ein Vergleich der verschiedenen pattern-Facetten zur Bestimmung eines „Durchschnitt-Patterns“ benötigen würde, werden auch vererbte pattern-Facetten in den Diagrammen dargestellt. Werden hingegen mehrere Facetten dieser Art innerhalb der gleichen Typdefinition eingefügt, erfolgt eine intern eine Veroderung der Wertebereiche, die diese Facetten beschreiben. Um Missverständnissen vorzubeugen, soll auch hier die graue Einfärbung neu definierte und vererbte Facetten voneinander trennen. Damit unterscheidbar ist, von welchem Typ die vererbten pattern-Facetten stammen, wird der jeweilige Name hinter dem Wert notiert. Ein Nutzer kann so direkt am Diagramm erkennen, welche der pattern-Facetten zu einer Gruppe gehören (Veroderung) und welche Gruppen den effektiven Wertebereich beschreiben (Durchschnittsbildung). Abbildung 6.6 zeigt dies beispielhaft an einer Manipulation des linken Hierarchiepfades unterhalb des Typen „R“.

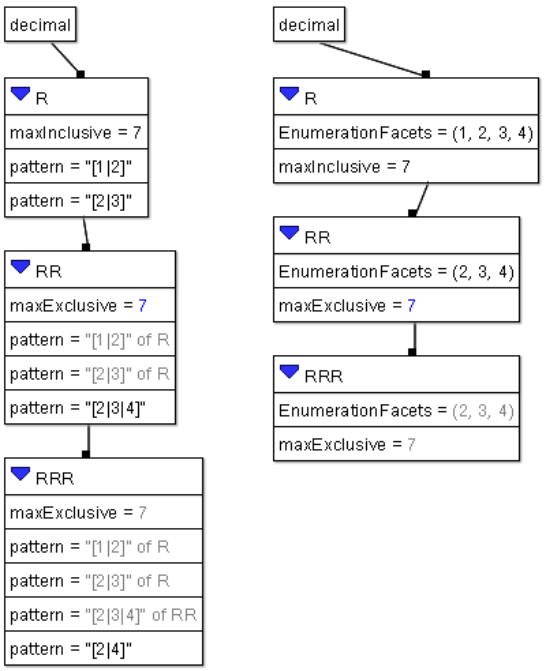


Abbildung 6.6.: Behandlung von pattern- und enumeration-Facetten am Beispiel des RLU-Schema.

Rechts daneben lässt sich der gleiche Hierarchiepfad nach einer anderen Veränderung durch das Einfügen von enumeration-Facetten betrachten. Während der Wertebereich von „R“ die Zahlen 1,2,3 und 4 enthält, schränkt „RR“ den Wertebereich durch eine Auswahl aus dieser Menge ein. Zu Recht wird die ausgeschlossene enumeration-Facette mit dem Wert 1 nicht noch einmal als vererbte Facette dargestellt, was sich auch auf den Typen „RRR“ des weiteren Hierarchiepfades auswirkt.

Anschließend soll die Visualisierung von Vereinigungs-Typen genauer betrachtet werden. Zunächst wurde der Typ „P“ in die Hierarchie eingefügt. Anschließend wurde der Typ „RLR“ mit den Facetten minLength = „1“ und maxLength = „5“ versehen. Dieser Typ geht daraufhin als neuer Mitgliedstyp in die Vereinigung von „RU“ ein. Weiterhin erfuhr der Typ „RUU“ ebenfalls eine Modifikation, indem der Typ „RRU“ als weiterer Mitgliedstyp hinzugefügt wurde. Somit besteht der Typ „RUU“ aus genesteten Vereinigungen, was in Abbildung 6.7 einsehbar ist.

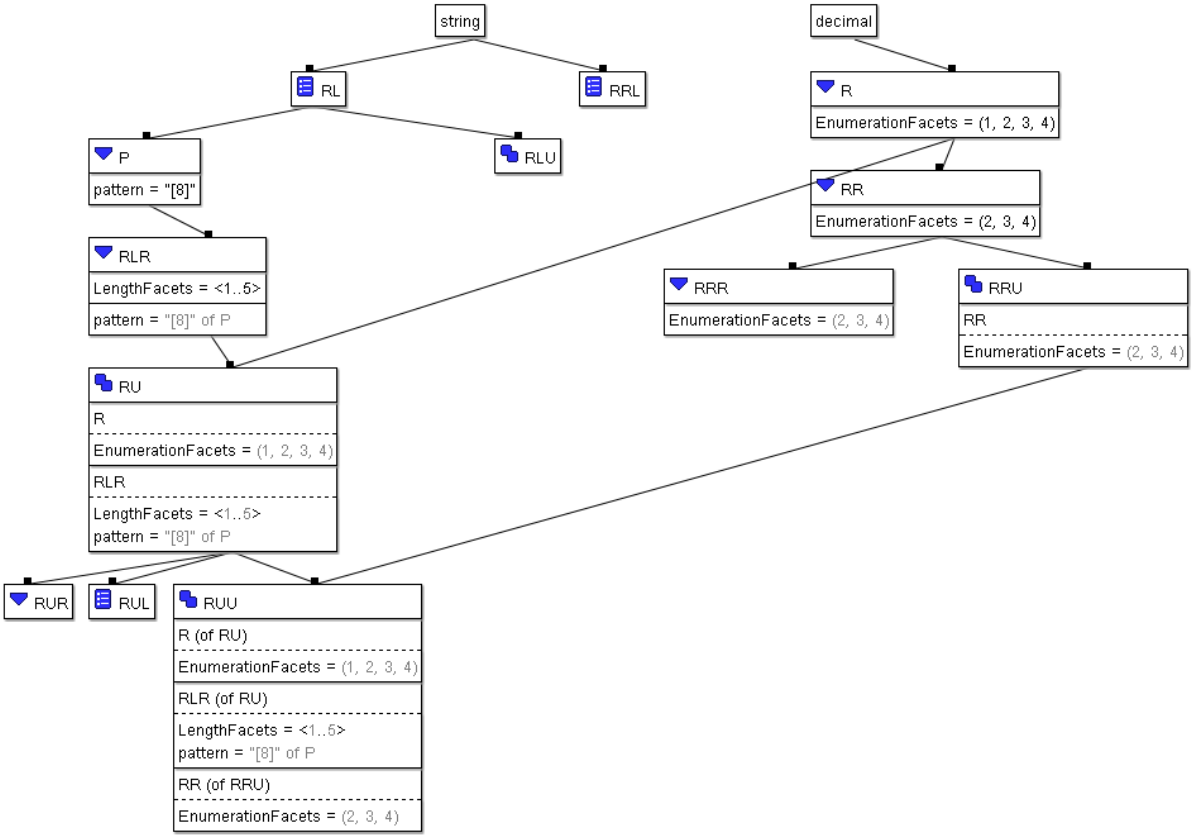


Abbildung 6.7.: Visualisierung von genesteten Vereinigung anhand des RLU-Schemas.

Für eine abwechslungsreichere Demonstration wurde verschiedene built-in Datentypen verwendet. Genauso wie die „normalen“ Vereinigungs-Typen „RU“ und „RRU“ stellt der verschachtelte Typ „RUU“ die beteiligten nutzerdefinierten Typen direkt an. Die einzelnen Ebenen der Vereinigung in einem Schema werden also immer bis zu den ursprünglichen Definitionen aufgeschlüsselt, sodass auch hier ein Nutzer den kompletten, über mehrere Ebenen vereinigten Wertebereich in einem einzigen Diagramm anschauen kann. Damit die eigentliche Bedeutung der Vereinigung nicht abhanden kommt, wird hinter den beteiligten (atomaren oder listenwertigen), ursprünglichen Typen noch der Name derjenigen Vereinigung vermerkt, die jene Typen zur Definition der aktuellen, verschachtelten Vereinigung beigesteuert hat.

6.1.2. Komplexe Typen

Anschließend sollen einige Beispiele für die Visualisierung von komplexen Typen gezeigt werden. Die Beziehungen zwischen den einzelnen Diagrammen werden durch eine Instanz der `CtComparator`-Klasse bestimmt, wobei diese Instanz jeden Typ jedem anderen einmal vergleicht. Diese Vergleiche können dabei ergeben, dass ein und derselbe Typ ein Untertyp bzw. eine Erweiterung von mehr als einem anderen Typ ist. Dies ließe sich durchaus darstellen, doch XML-Schema sieht keine Mehrfachvererbung vor, sodass so eine Darstellung nicht konform ist und für Verwirrung sorgen könnte. Weiterhin wird es dadurch schwierig, die Veränderungen der Inhaltsmodelle im Diagramm anzuzeigen, da ein Untertyp „gleichzeitig“ verschiedene Aspekte seiner konzeptionellen Obertypen einschränken kann, die jedoch nicht alle Obertypen gleichermaßen betreffen, was erneut für unklarheiten sorgen kann.

Daher wurde der Visualisierungs-Algorithmus so eingeschränkt, dass er einem Diagramm bzw. Typ immer nur einem einzigen anderen Eltern-Diagramm zuordnet. Erweiterungsbeziehungen werden dabei bevorzugt. Sollte also eine Erweiterung zur selben Zeit auch eine Einschränkung an irgendeiner Stelle ds EMX-Modells sein, verbleibt das Diagramm dieses Typen als Erweiterung eines anderen Diagramms an Ort und Stelle.

Testschema Für die Demonstration unterlag das Testschema einer Änderung im Aufbau. Im originalen Schema haben lokale Elemente wie „land“ in einer Einschränkung den Wert fixiert (z. B. „USAEliteStudentType“) oder als Typ einen Untertyp von „landType“ (z. B. „europaProfessorType“) verwendet. Dies funktioniert jedoch nur für eben solche lokale Elemente. Da der Garden of Eden Stil und somit CodeX mit globalen Elementen arbeitet, sind diese Formen der Einschränkung nicht möglich, da sie direkt an der Deklaration auszuführen sind. Um den Sinn des originalen Schemas zu erhalten, wäre es nötig, für jedes dieser eingeschränkten Elemente eine eigene globale Deklaration einzuführen, die natürlich auch wieder einen eindeutigen Namen besitzen muss. Dies bedeutet jedoch wiederum, dass die Inhaltsmodelle faktisch unterschiedlich sind, da die Elementnamen nicht übereinstimmen. Eine Obertyp-Untertyp-Beziehung, die auf dieser Einschränkung beruht, könnte so nicht mehr fortbestehen.

Aus diesem Grund wurde das Element „land“ durch ein entsprechendes Attribut ersetzt, welches auch an der Referenz direkt einschränkbar ist. Außerdem ist bedingt durch die Gegebenheiten die Verwendung von komplexen Typen mit einfachen Inhalt nicht mehr möglich. Das Element „gehalt“ welches auf einem solchen Typ „gehaltType“ basiert, kann nicht mehr durch die Verwendung von z. B. „gehaltDollarType“ eingeschränkt werden. Deswegen wird für dieses Element nur noch der neue einfache Typ „gehaltType“ verwendet. Das Attribut „waehrung“, welches ursprünglich an dem „gehalt“-Element gebunden war, wird auf die Typdefinition selbst ausgelagert, was die Einschränkung simuliert.

Insgesamt sind also nur noch Einschränkungen durch Attributfixierungen und Veränderungen der Element- bzw. Attributhäufigkeiten im Schema vorhanden. Abbildung 6.8 zeigt die erste Ebene der Erweitung des Typs „personType“ im modifizierten Schemas.

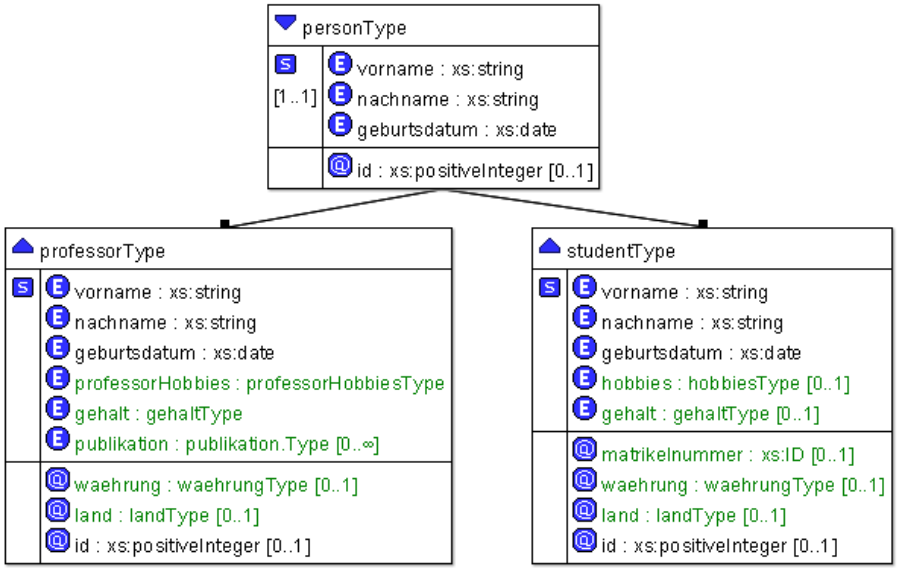


Abbildung 6.8.: Erweiterungen des Testschemas dargestellt in der Umsetzung.

Im Vergleich zum originalen Schema musste das Attribut „id“ explizit angegeben werden, da der vergleichende `CtComparator` sonst für die Typen „studentType“ und „professorType“ sowohl

eine Erweiterung als auch eine Einschränkung von „personType“ ermittelt hätte, was richtigerweise in *PT* resultieren würde. Element- und Gruppenkardinalitäten werden ausgelassen, wenn der Nutzer diese ebenfalls nicht im EMX-Modell angegeben hat. Der Wert „unbounded“ für das maxOccurs-Attribut wird im Diagramm durch ein Symbol repräsentiert. Attributkardinalitäten werden hingegen immer angezeigt. Gemäß der Vorüberlegung, kommen unterschiedliche, einfache Icons zum Einsatz, um einerseits die Ableitungsart und andererseits den Kompositor zu kennzeichnen. Elemente und Attribute, die neu hinzukommen erhalten eine grüne Markierung. So lässt sich beispielsweise die erweiternde Differenz der Elementmengen („professorHobbies“, „gehalt“ und „publikation“) z. B. zwischen „personType“ und „professorType“ schnell überblicken. Im Anschluss lässt sich in Abbildung 6.9 die zweite Ebene der Ableitung durch die Einschränkungen zu „europaProfessorType“ und „USAEliteStudentType“ betrachten.

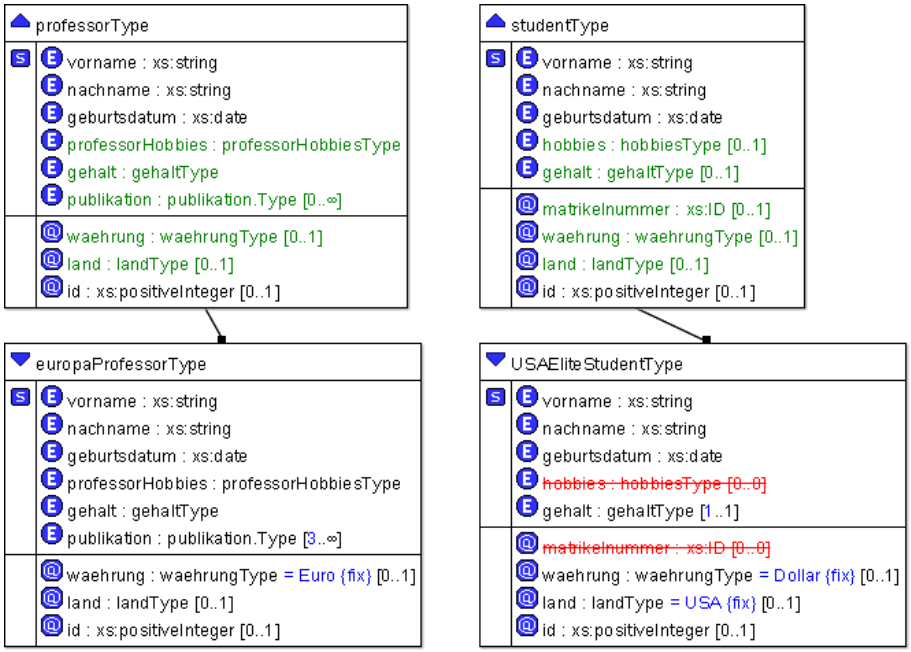


Abbildung 6.9.: Einschränkungen des Testschemas dargestellt in der Umsetzung.

Der `CtComparator` konnte beide Einschränkungen des Wertebereichs bzw. der Objektmengen richtig ermitteln. Im Typ „europaProfessorType“ entsteht diese dadurch, dass der minOccurs-Wert für das Element „publikation“ auf 3 erhöht wird. Außerdem wurde das Attribut „waehrung“ auf den Wert „Euro“ fixiert. Ähnlich sieht es für die Attribute und veränderte Elemente im Typ „USAEliteStudentType“ aus. Im Diagramm selbst sind diese Einschränkungen blau markiert. Löscht eine Einschränkung Elemente durch das Setzen des maxOccurs-Attributs auf 0 oder Attribute durch das Setzen des use-Attributs auf „prohibited“ aus dem Inhaltsmodell ihres Obertypen, wird dies nach der Vorstellung im Konzept durch das Durchstreichen des Komponentennamens markiert. Ein Nutzer weiß in dem Beispiel sofort, dass Instanzen des Typs „USAEliteStudentType“ im Vergleich zu Instanzen von „studentType“ niemals das Element „hobbies“ sowie das Attribut „matrikelnummer“ haben dürfen.

Weitere Beispiele Anschließend sollen weitere Demonstrationen für Ergebnisse der Bestimmung von Beziehungen zwischen zwei Typen gezeigt werden. Dazu kommen überschaubare Minimalbeispiele zum Einsatz. Auf der Konstruktion einer tiefen Hierarchie wird verzichtet, da diese im Wesentlichen keine neuen Einblicke liefert. Stattdessen sollen die Beispiele die Arbeitsweise des `CtComparators` für die einzelnen Belegungen dessen Parameter untermalen. Die Ergebnisse wurden jeweils gegen die Validierung der gleichen Schemafragmete in dem Tool XMLSpy geprüft, wobei keine abweichenden Resultate feststellbar waren.

Beziehungen zwischen Sequenz und Sequenz Initial werden zwei Typen „s1“ und „s2“ definiert, die ein identische Inhaltsmodell besitzen, was Abbildung 6.10 zeigt.

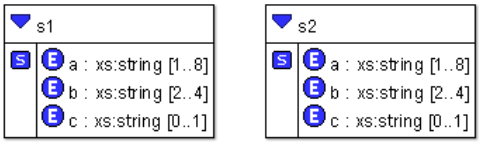


Abbildung 6.10.: Zwei identische Sequenzen.

Da das Urteil „ST“ lautet, kann die Beziehung nicht eindeutig wiedergegeben werden. Der Fall wer Erweiterung wurde bereits durch das Testschema abgedeckt. Da diese zudem nur für eine spezifische Form zweier Sequenzen ermittelt wird, werden vor allem Einschränkungen betrachtet. Diese kann hier auf verschiedene Weisen provoziert werden. Die erste besteht in der Beschränkung der lokalen Elemente. Wie Abbildung 6.11 zeigt, werden Einschränkungen sowohl im minOccurs- als auch maxOccurs-Wert erkannt. Diese zeigt gleichzeitig auch rechts, dass eine alleinige Einschränkung des Kompositors ebenfalls wahrgenommen wird. 6.10 zeigt.

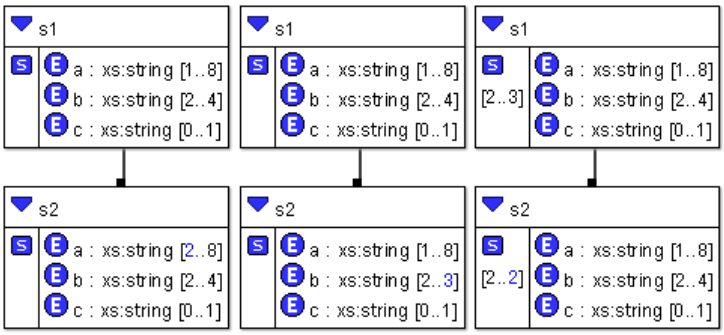


Abbildung 6.11.: Einschränkung der Häufigkeit in einer von zwei Sequenzen.

Eine andere Form der Einschränkung besteht darin, optionale Elemente des Obertyps nicht mehr im Untertypen auftauchen zu lassen. In Abbildung 6.12 ist zu sehen, dass dies auch explizit angezeigt wird. Das manuelle Vergleichen von Diagrammen zur Feststellung der Diferenz ist also nicht nötig. Die Position des nicht mehr wiederholten aber optionalen Elements ist dabei unerheblich. Interessant dabei ist die zweite Konfiguration von rechts. Hier fällt im Vergleich zur Konfiguration links daneben die Einschränkung von Element „b“ weg. Da sich das optionale Element „c“ in „s1“ an letzter Stelle befindet, liegt in der Tat im Vergleich zu „s2“ eine Erweiterung der Objektmenge vor, was auch als solche erkannt wird.

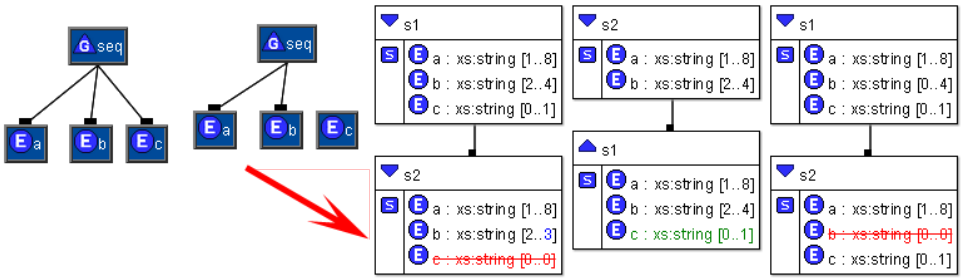


Abbildung 6.12.: Einschränkung durch Entfernen optionaler Elemente einer Sequenz.

Außerdem lassen sich auch „fehlerhafte“ Einschränkungen erkennen. Sobald die Ordnung der Elemente, oder mindestens eine Häufigkeit nicht übereinstimmen, wird auch dies direkt als Paralleltyp erkannt, was Abbildung zeigt.

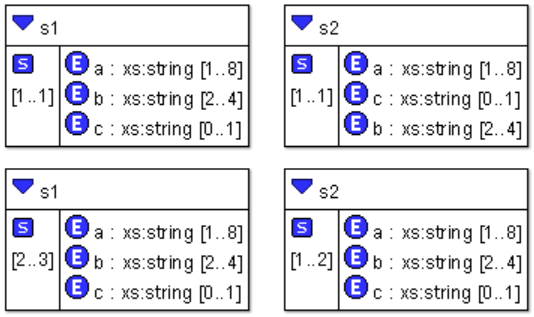


Abbildung 6.13.: Zwei Sequenzen als Paralleltypen, aufgrund der falschen Ordnung (oben) und unterschiedlicher Häufigkeit (unten).

Beziehungen zwischen Sequenz und Alternative Damit eine Sequenz überhaupt ein Obertyp einer Alternative sein kann, muss jedes Element des Inhaltsmodells optional sein. Die Reihenfolge der Sequenz-Elemente spielt hierbei keine Rolle mehr. Im Wesentlichen ist dann nur noch wichtig, wie oft die verschiedenen Elemente maximal auftreten können, wobei sogar die lokale Häufigkeit nicht wie üblich monoton fallen muss. Wird wie in Abbildung 6.14 rechts z. B. die Alternative selbst oder das Element „a“ zu mächtig, ist sie kein Obertyp, was durch parallele Typen ausgedrückt wird.

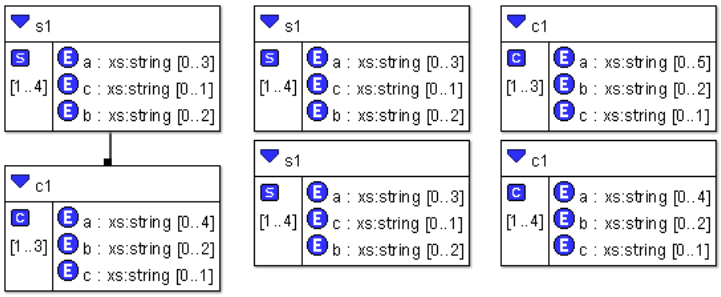


Abbildung 6.14.: Links die Einschränkung einer Sequenz mit einer Alternative, rechts daneben erlaubt die Alternative auf zwei verschiedenen Arten mehr Werte als die Sequenz.

Genauso wird auch die Situation festgestellt, in der die Sequenz selbst bereits ein Element zwingend erfordert, und dadurch weniger Elementkombination als die Alternative zulässt. In

Abbildung 6.15 ist dies für das Element „a“ der Fall, wodurch die Diagramme parallel angeordnet werden.

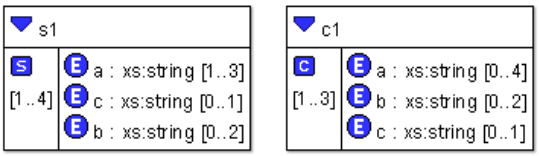


Abbildung 6.15.: Parellele Hierarchie von Sequenz und Alternative, aufgrund eines zwingend erforderlichen Elements.

Andersherum ist eine Alternative ein Obertyp einer Sequenz, wenn die Alternative genügend maximale und nicht zu viele, erforderliche minimale Wahlen im Vergleich zur Sequenz hat. Abbildung 6.16 zeigt diesen Fall der Einschränkung links. Rechts daneben sieht man oben, dass der maxOccurs-Wert der Alternative nicht mehr ausreicht und unten, dass die Alternative minimale zu viele Elemente erfordert. Beide Situationen bemerkt der Algorithmus des `CtComparators`, womit die Diagramme nicht untereinander platziert wurden.

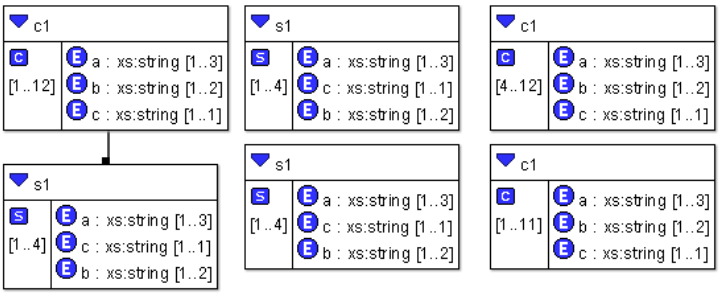


Abbildung 6.16.: Alternative als Obertyp einer Sequenz (links), daneben keine Hierarchie aufgrund der maximalen (rechts unten) und minimalen (rechts oben) Striktheit der Alternative.

Beziehungen zwischen Sequenz und Menge Eine Menge ist grundsätzlich mächtiger als eine Sequenz. Letztere kann nur dann ein Obertyp sein, wenn beide Gruppen nicht mehr als ein Element gemein haben. Das linke Beispiele in Abbildung 6.17 demonstriert, dass diese Form der Einschränkung richtig erkannt wird. Zu sehen ist auch, dass der minOccurs-Wert „2“ der Menge als verändert markiert wird, obwohl im Vergleich zur Sequenz keine Einschränkung der tatsächlichen, minimalen Häufigkeit vorliegt, da der Vergleich der Elementhäufigkeiten momentan nur auf lokaler Basis erfolgt. Rechts daneben ist oben der Fall aufgeführt, indem die Menge „a1“ das Element „b“ ebenfalls enthält. Im Vergleich zur Sequenz „s1“ erlaubt sie mehr Kombinationen. Darunter wurde „s1“ verändert, indem dieser Typ nun das Element „c“ erfordert, was jedoch nicht im Inhaltsmodell von „a1“ ist. Als letztes wurde unten die maximale Häufigkeit von Element „a“ im Typ „a1“ über die des Pendants der Sequenz erhöht. Alle drei Manipulationen wurden entsprechend als Parallelhierarchien festgestellt.

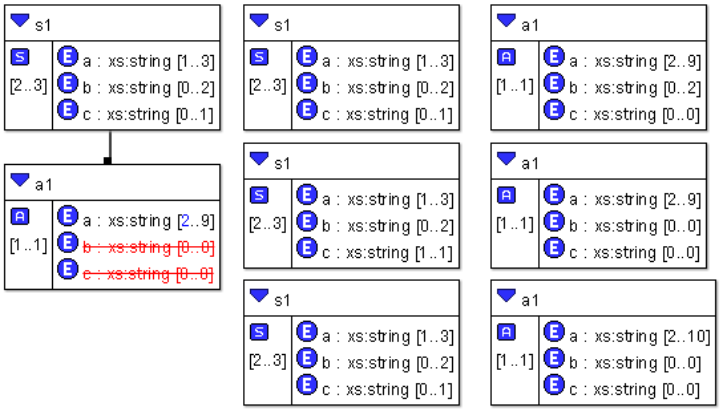


Abbildung 6.17.: Sequenz als Obertyp einer Menge (links), rechts daneben keine Hierarchie aufgrund mehrelementigen Schnittmengen (oben und mittig) sowie nicht übereinstimmender Häufigkeit der Menge (unten).

In umgekehrter Richtung sind lediglich die beteiligten Elementnamen und Gesamthäufigkeiten entscheidend. Eine Menge muss in diesen Aspekten wenigstens identisch sein, um als Obertyp einer Sequenz in Frage zu kommen, was an einem Beispiel links in Abbildung 6.18 erkennbar ist. Rechts daneben sind erneut Möglichkeiten aufgeführt, bei denen richtigerweise keine Obertyp-Untertyp-Beziehung ausgesprochen werden kann. Oben ist das Element „c“ maximal vier Mal in der Sequenz vorhanden, während die Menge nur zwei dieser Elemente vorsieht. Darunter wurde der Kompositor des Typs „s1“ auf die minimale Häufigkeit von 0 erweitert. Da entweder alle Elemente oder Kompositor selbst des Mengen-Typen nicht minimal „0“ sind, kann die Menge kein Obertyp sein. Als letztes wurde das Element „b“ aus der Menge entfernt. Da dessen Auftreten in der Sequenz zwingend ist, ist auch dieses Mal keinerlei Beziehung zwischen beiden Typen vorhanden. Da sich der Vergleich zweier Mengen von den zu untersuchenden Aspekten dieses Falls nicht unterscheidet, werden hierfür keine separaten Beispiele gezeigt.

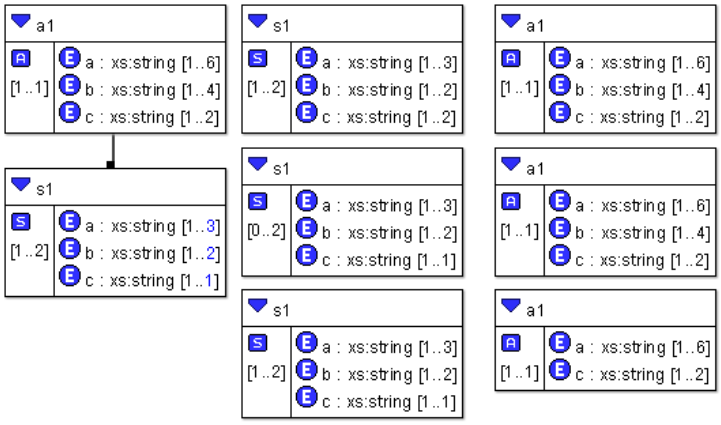


Abbildung 6.18.: Sequenz als Untertyp einer Menge (links), rechts daneben parallele Hierarchie aufgrund nicht übereinstimmender Häufigkeiten (oben und mittig) sowie fehlender Elemente in der Menge (unten).

Beziehungen zwischen Alternative und Alternative Im Gegensatz zu den meisten anderen Fällen, müssen Elemente aus einer Alternative trotz eines minOccurs-Werts größer als „1“ nicht zwingend in einem potentiellen Untertyp wiederholt werden. Außerdem sind die lokalen Elementhäufigkeiten nicht von Bedeutung, stattdessen zählt wieder die jeweilige Gesamthäufigkeit. Wie in Abbildung 6.19 zu sehen, wird dies in der Umsetzung als gültige Einschränkung registriert und mit entsprechend angeordneten Diagrammen ausgedrückt. Element „c“ in dem Typ „c2“ hat dabei einen höheren, lokalen maxOccurs-Wert als das Element „c“ in dem Typ „c1“, liegt insgesamt aber noch im Rahmen der maximalen Gesamthäufigkeit (6 für beide). Im oberen Beispiel rechts daneben wurde der maxOccurs-Wert von Element „c“ im Typ „c2“ auf 4 erhöht, die Gesamthäufigkeit liegt damit bei 8 und übersteigt die des jeweiligen Elements von Typ „c1“. Auperdem ist das Einfügen neuer Elemente nicht erlaubt, wie etwa im unteren Beispiel mit dem Element „d“ für den Typ „c2“. Beide Fälle werden als parallele Hierarchien visualisiert.

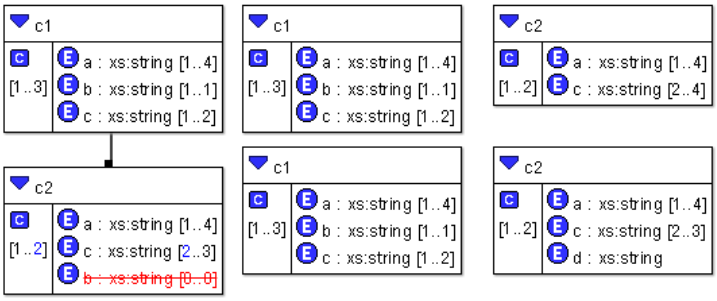


Abbildung 6.19.: Alternative „c2“ als Untertyp von Alternative „c1“ (links), rechts daneben parallele Hierarchie aufgrund nicht übereinstimmender Häufigkeiten (oben) sowie nicht übereinstimmende Elementnamen (unten).

Beziehungen zwischen Alternative und Menge Soll die Menge Obertyp einer Alternative sein, müssen deren Elemente genau wie bei einer Sequenz optional sein. Dadurch können Elemente in der Alternative auch weggelassen werden. Abbildung 6.20 zeigt links eine solche Einschränkung beispielhaft. Rechts daneben ist oben Element „a“ in der Menge verpflichtend, während darunter das maximale Vorkommen von Element „b“ in der Alternative 4 beträgt, in der Menge jedoch nur zwei mal auftreten kann. Auch diese Einschränkung sowie Parallel-Beziehung bemerkt der implementierte Algorithmus.

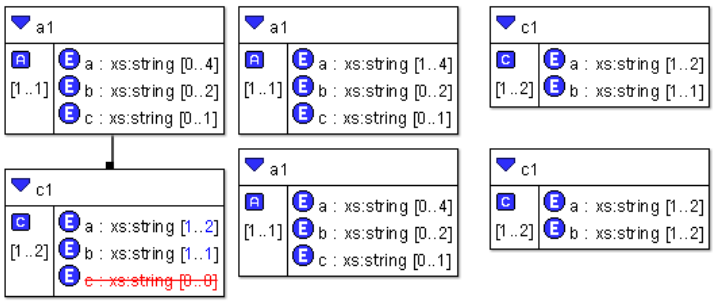


Abbildung 6.20.: Menge als Obertyp einer Alternative (links), rechts daneben keine Beziehung aufgrund des notwendigen Elements „a“ in der Menge (oben) und nicht übereinstimmender maximaler Häufigkeit des Elements „b“ in der Alternative (unten).

Wie ausgeführt, muss jedes Element einer Alternative ein minimales Vorkommen von 1 ermöglichen (d. h. $p_{min} \leq 1$ für alle Elemente p in der Alternative) und diese über genügend Wahlen verfügen, sodass jede Kombination an verschiedenen Elementanzahlen einer Menge möglich sind. Genau dies ist in dem linken Beispiel der Abbildung 6.21 der Fall. Betrachtet man das Beispiel rechts oben daneben, sieht man dass das erforderliche Vorkommen von Elementen in der Alternative mindestens 7 beträgt. Dies übersteigt die Kapazitäten der Menge, sodass diese kein Untertyp sein kann. Gleichermaßen würde die Reduktion des maximalen Vorkommens der Alternative auf 11 die Gesamtanzahl von maximal 12 Elementen in der Menge unterschreiten, was das Beispiel darunter demonstriert. Interessant ist hier vor allem der Fall, in bereits ein einziges Element der Alternative ein $p_{min} > 1$ hat. In letzten Beispiel ist dies für das Element „a“ der Fall. Obwohl ein unbeschränkter maxOccurs-Wert vorliegt, kann die Alternative mit einem minOccurs-Wert von 2 für „a“ niemals ein Obertyp der Menge sein. Auch diese Szenarien werden richtig erfasst und dementsprechend in der Visualisierung reflektiert.

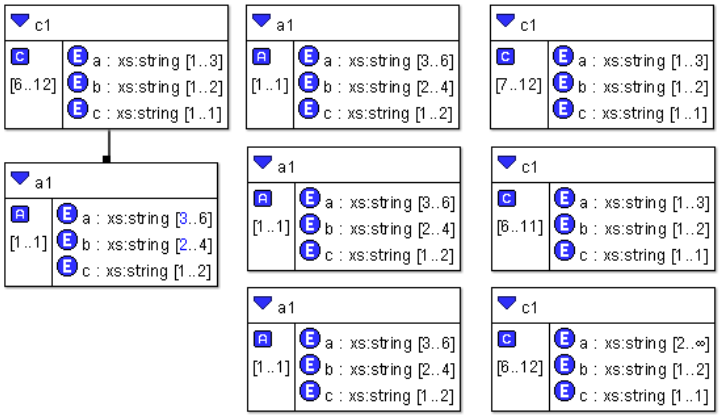


Abbildung 6.21.: Menge als Untertyp einer Alternative (links), rechts daneben parallele Hierarchien aufgrund nicht übereinstimmender minimaler und maximaler Kompositor-Häufigkeit der Alternative (oben und mittig) sowie nicht übereinstimmender lokaler minimaler Häufigkeit des Elements „a“ in der Alternative (unten).

6.2. Löschen von Typen

Anschließend werden einige Beispiele für die Kompensation nach der Löschung eines Typen aus einem EMX-Modell betrachtet. Einfache und komplexe Typen werden erneut getrennt voneinander betrachtet.

6.2.1. Einfache Typen

Die Löschung mitsamt der Kompensation wird zunächst beispielhaft für einen Typen des Testschemas gezeigt. Über einen bereits vorhandenen Dialog in CodeX, dargestellt in 6.22, war es möglich, einen Typen zum Bearbeiten auszuwählen. Die Funktion des Löschens wurde nun an den Knopf „Delete“ gebunden. In dem Beispiel soll der Typ „professorHobbiesType“ gelöscht werden.

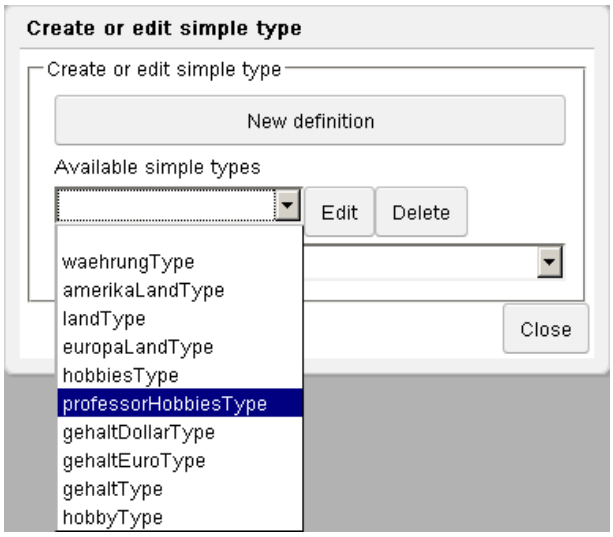


Abbildung 6.22.: Dialog zur Bearbeitung oder Löschung von einfachen Typen in CodeX.

Wird dieser gedrückt, erscheint vor der finalen Entfernung ein weiterer Dialog, der links die abhängigen Deklarationen und weiteren Typen und rechts die einen Typ für die Kompensation vorschlägt. Dieser Typ ergibt sich in den Fällen, in welchen es möglich ist, direkt aus dem Obertyp bzw. einem Mitgliedstyp einer Vereinigung. Es wäre auch denkbar noch weitere Typen etwa oberhalb des gleichen Hierarchiepfades anzuzeigen. Allerdings wurde die implementierte Lösung als sinnvoll erachtet, da der Wertebereich aus der höheren Oberhierarchie nur noch wächst.

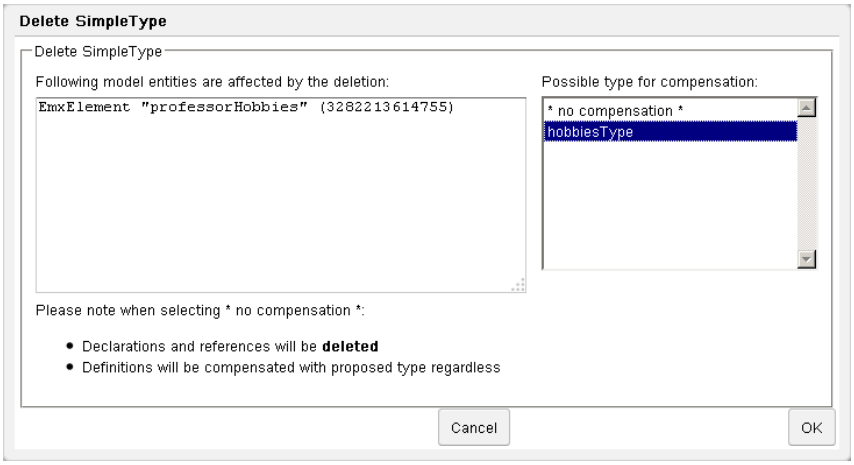


Abbildung 6.23.: Dialog über die Kompensation von „professorHobbiesType“.

Es besteht immer die Wahl, keine Kompensation für den zu löschenden Typen zu wählen, wobei der Nutzer über die Konsequenzen informiert wird. Im Beispiel wurde der Typ „hobbiesType“ vom Algorithmus vorgeschlagen und soll jetzt als Kompensations-Typ dienen. Wird nun der „Ok“-Knopf betätigt, wird die Löschung mitsamt der Kompensation ausgeführt. Der gesamte Vorgang lässt sich auch durch das Drücken des „Cancel“-Knopfs wieder abbrechen.

```
09:39:22 dfg/PersonDefinition4_out.emx: delete simpletype name '3282213614811' ;
09:39:22 dfg/PersonDefinition4_out.emx: update simpletype name '3282213614811' change mode 'restriction' of '3282213614811' remove 'enum
'3282213614812' at '3282213614812' ;
09:39:22 dfg/PersonDefinition4_out.emx: update element name '3282213614755' change type '3282213614810' ;
```

Abbildung 6.24.: ELaX-Statements nach der Löschung von „professorHobbiesType“. Das Update-Statement zeigt, dass der Typ des abhängigen Elements aktualisiert wurde (rot markiert).

Wie die ELaX-Ausdrücke aus Abbildung 6.24 zeigen, wird der betroffene Typ aus der Entitäten-Liste des EMX-Modells gelöscht (statt der XML-Namen werden für diese jedoch eindeutige EIDs genutzt). Dem referenzierende Element „professorHobbies“ wurde der Typ „hobbiesType“ erfolgreich zugewiesen.

Weitere Beispiele Um nachfolgend andere Situationen für die Kompensation zu betrachten, werden Typen des RLU-Schemas genutzt. Als erstes wird versucht, den Typ „RL“ zu löschen. Das EMX-Modell wurde zu Demonstrationszwecken so erweitert, dass die Typen „RLR“ und „RLRR“ (ein neuer, einschränkender Typ) jeweils einen eigenen Satz an Längen-Facetten bereitstellen. (siehe Abbildung 6.25 links). Da der Basistyp „R“ allgemein genug ist die Whitespace-getrennten Werte anzuzeigen, wird dieser als Kompensation vorgeschlagen (in der Abbildung unten). Die Änderung wird daraufhin vom Algorithmus angenommen. In der Abbildung sieht man nun rechts, dass dieser während seiner Abarbeitung alle Längen-Facetten von „RLR“ und „RLRR“ gelöscht hat. Dies zeigt die Umsetzung der besprochenen und nötigen Entfernung von Facetten, die sonst eine andere Semantik erhielten und eventuell zu Ungültigkeiten von Dokumenten führen könnten.

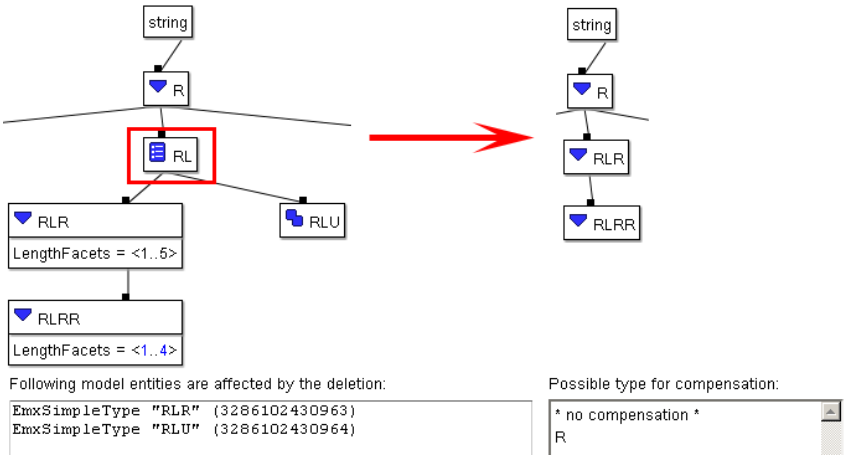


Abbildung 6.25.: Kompensation der Löschung eines Listentypen (Knoten), Längen-Facetten der tieferen Hierarchie werden gelöscht.

Bei genauerer Betrachtung fällt auf, dass der Typ „RLU“ nach der Kompensation fehlt. Dies ist ein Programmfehler, der leider nicht mehr rechtzeitig behoben werden konnte.

In einer zweiten Operation soll der Listentyp „RUL“ gelöscht werden, welcher auf dem atomar-vereinigten Typ „RU“ basiert. Für dieses Beispiel wurde diesem zuerst der built-in Datentyp „NMTOKEN“ als weiterer Mitgliedstyp zugewiesen. Keiner der beiden Mitgliedstypen (der andere ist weiterhin „R“, der von „decimal“ abgeleitet ist) kann den Wertebereich des jeweils anderen darstellen, sodass „RU“ für die Kompensation der Löschung von „RUL“ nicht in Frage kommt (in Abbildung 6.26 links zu sehen). Ein weiteres Problem wäre dabei, dass keiner der Typen Whitespaces zulässt. Als Alternative wird vorgeschlagen, den Typen „string“ für die Kompensation

zu verwenden und gegebenenfalls neu zu erstellen, sollte dieser Typ noch nicht im EMX-Modell vorhanden sein.

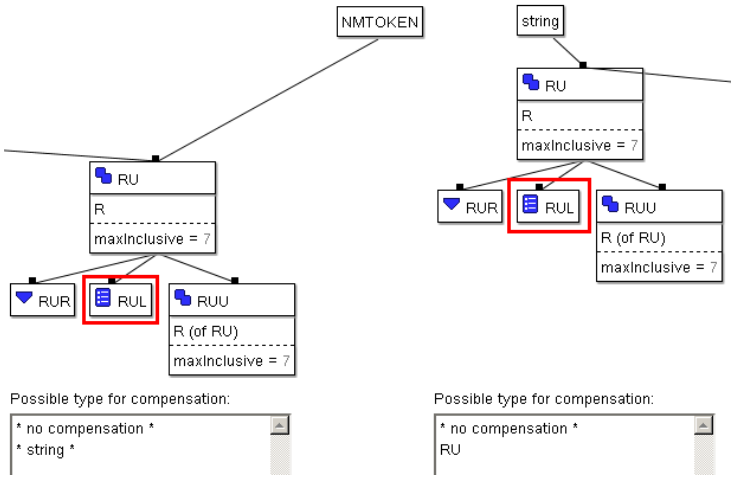


Abbildung 6.26.: Kompensation der Löschung eines Listentypen (Blatt), links stimmen die Wertebereiche des Basistypen nicht überein, rechts hingegen schon.

Die Situation sieht anders aus, wenn der Mitgliedstyp „NMTOKEN“ direkt durch z. B. „string“ ersetzt wird. Dieser erlaubt weiterhin Whitespaces und gleichzeitig auch die Werte des Typs „decimal“. Der vorgeschlagene Kompensationstyp lautet daher „RU“, was Abbildung 6.26 rechts zeigt. Zuletzt wird die Kompensation für die Löschung des Vereinigungs-Typen „RU“ betrachtet. Das Beispiel wurde ein weiteres Mal verändert. Mitgliedstypen für diesen sind zuerst weiterhin „R“ und der built-in Datentyp „integer“. Abbildung 6.27 zeigt, dass der Typ „R“ vorgeschlagen als Kompensation wird. Aus Sicht der Varietäten und den beteiligten Wertebereichen der built-in Datentypen (integer ist Untertyp von decimal) ist dies nachvollziehbar. Da dieser Typ den Wertebereich auf maximal 7 einschränkt, wäre stattdessen die Antwort „decimal“ richtig. Aufgrund der Priorisierung anderer Aufgaben konnte allerdings ein umfangreicher Vergleich der Facetten, nicht mehr fertiggestellt werden.

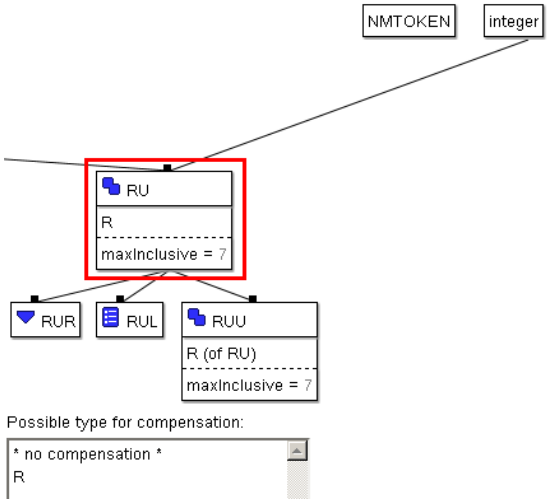


Abbildung 6.27.: Kompensation der Löschung eines atomar-vereinigten Typen (Knoten).

Dennoch ist dieses Beispiel interessant, wie Abbildung 6.28 zeigt. Hier findet anstelle des Typs „integer“ der Listentyp „IntegerList“ Verwendung als Mitgliedstyp. Obwohl „R“ (bzw. eigentlich

„decimal“) in der Lage wäre, die Werte von „integer“ darzustellen, bleibt die Liste der konkreten Vorschläge leer. Grund hierfür ist der listenwertig-vereinigte Charakter von „RU“ durch den Mitgliedstyp „IntegerList“ in diesem Szenario. Die Implementierung stellt also richtig fest, dass die in den Mitgliedstypen der Vereinigung beteiligten built-in Datentypen nicht allgemein genug sind, um auch alleine Listenwerte darzustellen.

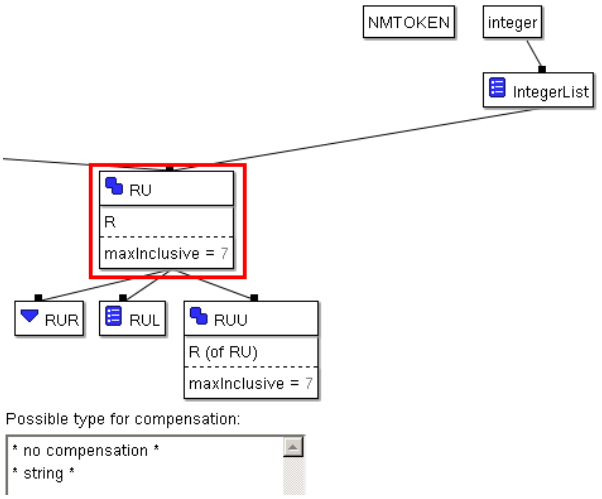


Abbildung 6.28.: Kompensation der Löschung eines listenwertig-vereinigten Typen (Knoten).

6.2.2. Komplexe Typen

Anhand des Testschemas soll kurz die Kernfunktionalität der Kompensation von komplexen Typen gezeigt werden. Möchte man versuchen, den Typ „personType“ zu löschen, erscheint ein Dialog (in Abbildung 6.29 zu sehen), in welchem rechts die Typen des Schemas sortiert nach ihren abgeschätzten, durchschnittlichen Kosten für die Kompensation aufgelistet werden. Links daneben sind alle abhängigen Elemente aufgeführt. Ein Mouseover-Tooltip schlüsselt die Durchschnittskosten in durchschnittliche Einfüge- und Löschkosten für Attribute und Elemente, sowie in durchschnittliche Sortierkosten für Elemente und Updatekosten für Attribute auf.

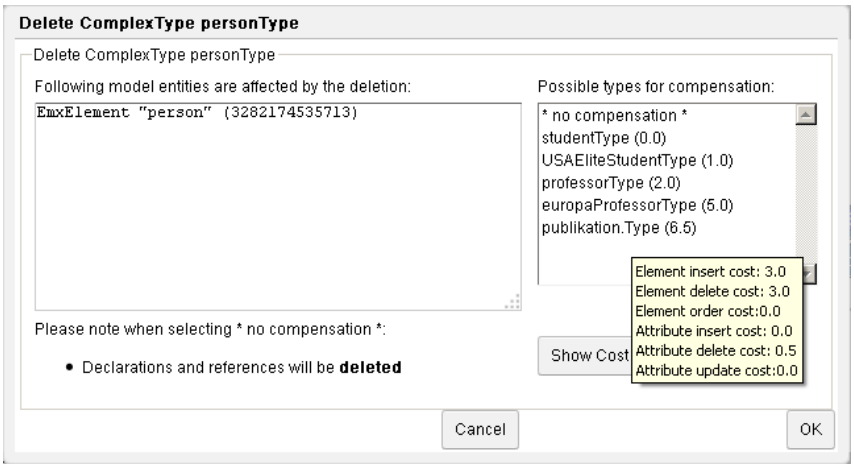


Abbildung 6.29.: Dialog über die Kompensation von „personType“.

In diesem Beispiel zeigt der Tooltip die Kosten für die Kompensation mit dem Typ „publikation.Type“. Dieser schneidet mit durchschnittlich 3 Lösch- und Einfügeoperationen für Elemente

am schlechtesten ab. Außerdem kann es zu einer Löschung eines Attributs kommen. Durch das Klicken des Knopfs „Show Cost“ lassen sich in einer Kostentabelle weitere Informationen einholen, wobei für eine Analyse mehrere parallel geöffnet werden können. Abbildung 6.30 zeigt die Kostentablle für die ausgewählte Kompensation im Detail.

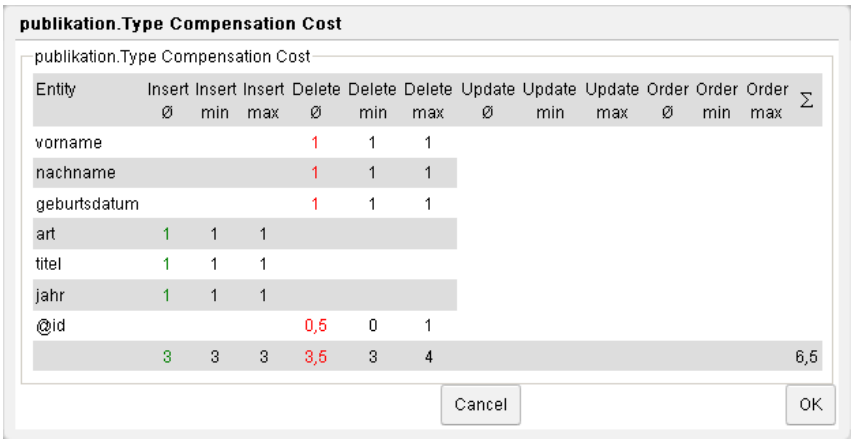


Abbildung 6.30.: Kostentablle für die Kompensation mit dem Typ „publikation.Type“.

Nach einem Blick in diese Tabelle ist es nicht verwunderlich, warum dieser Typ als schlechtestes abschneidet. Einerseits wären die Elemente von „personType“ komplett zu entfernen werden und andererseits die neuen Elemente einzufügen. Das eventuell zu löschende Attribut ist „id“. Da dieses optional ist, werden Kosten auch nur dann nötig sein, wenn dieses tatsächlich in einem Dokument auftaucht. In der rechten äußersten Spalte sind die Durchschnittskosten noch einmal zusammengefasst. Wird dieser Typ tatsächlich ausgewählt und die Löschung durch das Drücken des „Ok“-Knopfs (des Kompensations-Dialogs) bestätigt, werden alle nötigen ELaX-Statements generiert und der Typ endgültig gelöscht. Wie der Log von CodeX verrät, wurden alle abhängigen Komponenten in einem Zug gelöscht und das Element „person“ dem Typ Kompensations-Typ zugeordnet (siehe Abbildung 6.31).

```
23:13:48 dfg/PersonDefinition3_out.emx: delete elementref at '3282174535736' ;
23:13:48 dfg/PersonDefinition3_out.emx: delete elementref at '3282174535737' ;
23:13:48 dfg/PersonDefinition3_out.emx: delete elementref at '3282174535738' ;
23:13:48 dfg/PersonDefinition3_out.emx: delete group at '3282174535734' ;
23:13:48 dfg/PersonDefinition3_out.emx: delete attributegroupref at '3282174535735' ;
23:13:48 dfg/PersonDefinition3_out.emx: delete complextype name '3282174535733' ;
23:13:48 dfg/PersonDefinition3_out.emx: update element name '3282174535713' change type '3282174535790' ;
```

Abbildung 6.31.: ELaX-Statements für die Kompensation. Das Update-Statement (rot markiert) drückt die Zuweisung des neuen Typen aus.

Interessant an diesem Beispiel ist, dass der Typ „studentType“ mit 0 Kosten bewertet wurde (siehe Abbildung 6.29). Dies liegt daran, dass im Inhaltsmodell ausschließlich optionale neue Elemente definiert sind. Im Vergleich dazu entstünden bei der Wahl von „professorType“ als neuer Typ 2 Einfügekosten, da dessen neue Elemente „professorHobbies“ und „gehalt“ nicht optional sind. Da eine Instanz von „europaProfessorType“ beispielsweise mindestens 3 Elemente von „publikation“ enthalten muss, fällt die Kompensation hier noch einmal teuer aus. Abbildung 6.32 zeigt die Ausschnitte der jeweiligen Kostentabellen.

professorHobbies	1	1	1
gehalt	1	1	1

professorHobbies	1	1	1
gehalt	1	1	1
publikation	3	3	3

Abbildung 6.32.: Kosten für die Kompensation mit „professorType“ (links) und „europaProfessorType“ (rechts).

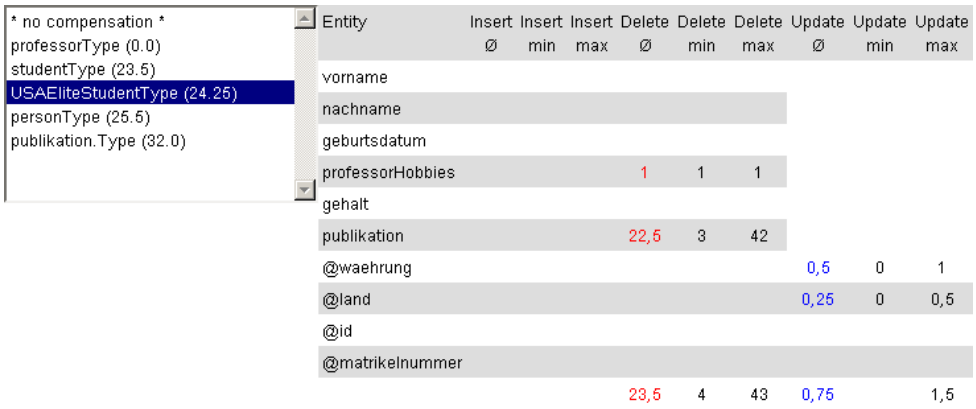
Möchte man den Typen „USAEliteStudentType“ löschen, entstehen durchschnittliche Kosten, die Abbildung 6.33 links zeigt. Auch hier entstehen für die Kompensation mit dem Typ „studentType“ keine Kosten, was die Obertyp-Untertyp-Beziehung beider Typen widerspiegelt. Die Kompensations-Kosten mit „europaProfessorType“ sind in der Abbildung rechts zu sehen. Da das Attribut „waehrung“ in beiden Typen mit jeweils anderen Werten fixiert ist, wäre die Veränderung des Attributwerts unumgänglich, wenn es verwendet worden ist.

* no compensation *
studentType (0.0)
professorType (1.0)
personType (2.0)
europaProfessorType (4.5)
publikation.Type (8.5)

Entity	Insert Ø	Insert min	Insert max	Delete Ø	Delete min	Delete max	Update Ø	Update min	Update max
professorHobbies	1	1	1						
publikation	3	3	3						
@matrikelnummer									
@waehrung							0,5	0	1
@land									
@id									

Abbildung 6.33.: Kosten für die Löschung von „USAEliteStudentType“ (links), Ausschnitt der Kostentabelle für die Kompensation mit „europaProfessorType“ (rechts).

Wesentlich anders sieht es für die Löschung von „europaProfessorType“ aus, da die Kosten hier für fast alle Typen hoch sind (siehe Abbildung 6.34). Die Ausnahme ist der Obertyp „professorType“, für den keine Änderungsoperationen nach einer Kompensation nötig wären. Die Ursache für die hohen Kosten liegt in der Komponente „publikation“, welche ein unbeschränktes maximales Vorkommen besitzt. Aus diesem Grund wird die Maximal-Konstante 42 als Kostenzahl genutzt. Diese kommt auch dann zum Einsatz, wenn die Differenz in endlichen Werten zu groß wird. Damit soll ausgedrückt werden, dass die Kosten für ein einziges Element ab diesem arbiträren Schwellenwert untragbar sind. Zur Abgrenzung zu anderen „Härtefällen“ werden die Gesamtkosten trotzdem summiert. Die Kostentabelle rechts im Bild gehört zum Typ „USAEliteStudentType“ und zeigt eine Auffälligkeit in den Updatekosten für das Attribut „land“. Dieses ist in „europaProfessorType“ im Gegensatz zu „USAEliteStudentType“ nicht fixiert. Wie im Kapitel über die Implementierung angesprochen, ist unklar ob der Wert „USA“ nicht vielleicht schon genutzt wurde. Daraus entstehen 0.5 Kosten, wenn das Attribut in Instanzen vorhanden ist.

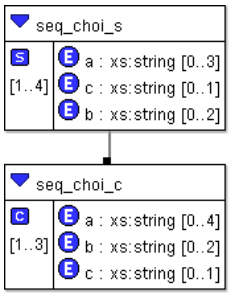


Entity	Insert Ø	Insert min	Insert max	Delete Ø	Delete min	Delete max	Update Ø	Update min	Update max
vorname									
nachname									
geburtsdatum									
professorHobbies				1	1	1			
gehalt									
publikation				22,5	3	42			
@waehrung							0,5	0	1
@land							0,25	0	0,5
@id									
@matrikelnummer									
				23,5	4	43	0,75		1,5

Abbildung 6.34.: Kosten für die Löschung von „europaProfessorType“ (links), Ausschnitt der Kostentabelle für die Kompensation mit „USAEliteStudentType“ (rechts).

Weitere Beispiele Da bisher ausschließlich zwei Sequenzen verglichen wurden, soll anschließend die Kostenberechnung auf Basis verschiedener Kompositoren gezeigt werden. Außerdem lassen sich keine neuen Erkenntnisse aus Kosten für Attribute ziehen, sodass nur noch die Kosten für Elemente betrachtet werden sollen. Als Grundlage dienen Beispiele aus Abschnitt 6.1.2. Da viele Fälle die gleichen Methodenaufrufe beinhalten, werden nur markante Fälle genutzt, die jeweils unterschiedliche Aufrufe involvieren. Nachfolgend bezeichnet zur Abkürzung T_1 wieder den gelöschten und T_2 den kompensierenden Typ.

Kompensation einer Sequenz mit einer Alternative Grundlage für dieses Beispiel ist der in Abbildung 6.14 links dargestellte Fall. Abbildung 6.35 zeigt diesen hier noch einmal, wobei lediglich die Typbezeichnungen ausgetauscht wurden. Als erstes soll die Sequenz gelöscht und mit der Alternative ersetzt werden. Da die Sequenz ein (echter) Obertyp ist, ist die Integration der zugeordneten Komponenten in die Objektmenge des Untertypen verlustbehaftet. Der Algorithmus des `CtCostCalculator` ermittelt für diese Kompensation insgesamt 7,5 Löschkosten. Die minimalen Kosten sind jeweils leicht zu erklären. Die Elemente beider Typen haben jeweils ein minimales Vorkommen von 0, sodass weder Einfüge- noch Löschkosten entstehen. Beim maximalen Fall hingegen kommt es zu Löschkosten, die sich folgendermaßen erklären.



Entity	Insert Ø	Insert min	Insert max	Delete Ø	Delete min	Delete max	Update Ø	Update min	Update max	Order Ø	Order min	Order max	Σ
a				1,5	0	3							
c				2	0	4							
b				4	0	8							
				7,5		15							7,5

Abbildung 6.35.: Kosten für die Kompensation einer Sequenz mit einer Alternative.

Der Algorithmus bemerkt, dass a_{max} in T_1 3 beträgt und damit kleiner als a_{max} in T_2 ist. Da die Sequenz mit $s_{max} = 4$ mehrfach wiederholt wird, reicht es aus, wenn die Alternative ihr Element „a“ mit jeweils der lokalen Häufigkeit 3 wählt. Dies müsste nun viermal getan werden, jedoch beträgt c_{max} lediglich 3. Alle Wahlen sind nach bereits einer Iteration für das Element „a“ ausgegeben. Dabei fallen sogar noch wie in der Abbildung zu erkennen Löschkosten in der Höhe von 3 für „a“ selbst an, da es mit der gewählten lokalen Häufigkeit insgesamt nur neunmal in der

Alternative vorkommt, aber zwölfmal in der Sequenz. Im Prinzip entstehen hier sogar zu hohe Kosten, da das Element in der Alternative lokal jeweils viermal hätte gewählt werden können, wodurch 0 Kosten entstünden. Allerdings wären dann Sortierkosten fällig, da die Elemente im Vergleich zu den Sequenzwiederholungen „falsch“ verteilt wäre, was jedoch dem Löschen klar zu bevorzugen wäre.

Der Algorithmus ließe sich in diesem Aspekt verbessern. Dazu wäre erneut eine Form des Lookaheads nötig, um zu erkennen in welchen Situationen der Wechsel von „positiven“ verbleibenden Wahlen zu „negativen“ (und damit entstehende Kosten) durch die Wahl eines höheren, lokalen Vorkommens des aktuellen Elements negiert werden könnte. Da in weiteren Iterationen die Wahlen der Alternative erschöpft sind, erklären sich die Kosten für die Elemente „b“ und „c“ somit schnell. Diese ergeben sich jeweils aus dem maximalen Vorkommen über alle Sequenzwiederholungen hinweg und sind damit korrekterweise 8 für das Element „b“ und 4 für das Element „c“. Diese ließen sich jedoch trotz der erläuterten, besseren Wahl von „a“ in T_2 nicht verhindern, da die Wahlen auch dann nach einem Iterationsschritt verbraucht wären.

Entity	Insert Ø	Insert min	Insert max	Delete Ø	Delete min	Delete max
a						
c				2	0	4
b				4	0	8
				6		12

Entity	Insert Ø	Insert min	Insert max	Delete Ø	Delete min	Delete max
a						
c						
b						
				4		8

Entity	Insert Ø	Insert min	Insert max	Delete Ø	Delete min	Delete max
a						
c						
b						
				4	0	8

Entity	Insert Ø	Insert min	Insert max	Delete Ø	Delete min	Delete max
a	0,5	1	0			
c	2	4	0			
b	0,5	1	0			
	3	6				

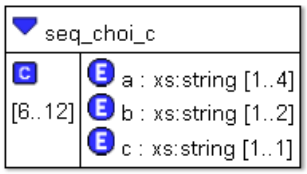


Abbildung 6.36.: Kosten für $c_{max} = 4$ (links oben), $c_{max} = 8$ (rechts oben), $c_{max} = 12$ (links unten) und $c_{min} = 6$ (rechts unten).

Würde man nun beispielsweise c_{max} erhöhen, würden kontinuierlich weniger Kosten entstehen. Abbildun 6.36 zeigt links oben die Kostentabelle für $c_{max} = 4$, die den ebend erwähnten Sachverhalt wiedergibt. Nach der Erhöhung auf $c_{max} = 8$ wie rechts oben im Bild, reichen die Wahlen aus, um auch das Element „c“ vollständig abzudecken. Erst nachdem c_{max} den Wert 12 überschritten hat, entstehen keine Kosten mehr, was links unten dargestellt ist. Interessant sind auch die minimalen Kosten. Rechts unten wurde c_{min} auf 6 gesetzt und außerdem die minimale Häufigkeit jedes Elements auf 1. Bei der Kompensation müssten nun im minimalen Fall der Sequenz zunächst jedes Element mindestens ein Mal eingefügt werden. Damit wären jedoch die minimalen Wahlen der Alternative nicht erreicht, sodass das Inhaltsmodell mit einem Element aufgefüllt werden muss. Da alle lokalen minimalen Häufigkeiten identisch sind, hat der Algorithmus im Beispiel zufällig das Element „c“ gewählt, was nun insgesamt viermal einzufügen ist.

Kompensation einer Menge mit einer Alternative Das obige Beispiel wurde ein wenig abgeändert, und sieht nun so aus, dass eine Menge gelöscht und mit einer Alternative kompensiert werden soll, deren maximale Häufigkeit 2 ist. Das Ergebnis der berechneten Kosten mitsamt den neuen Inhaltsmodellen sind in Abbildung 6.37 einsehbar.

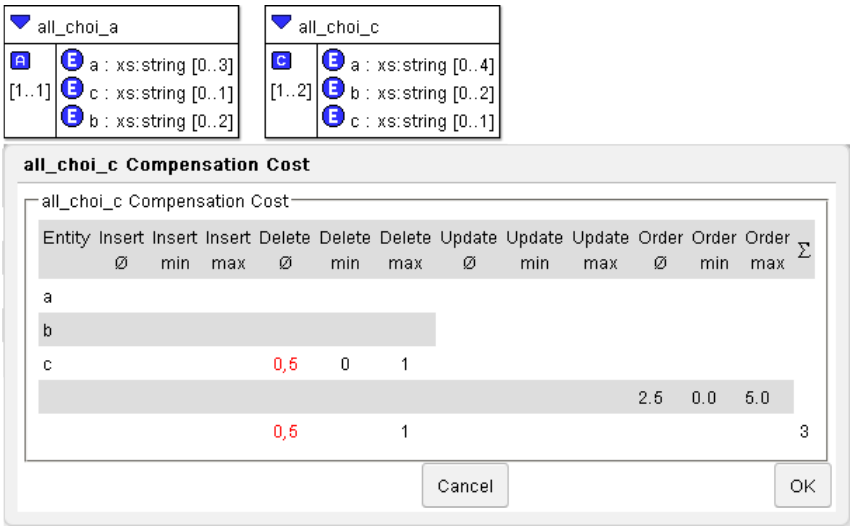


Abbildung 6.37.: Kosten für die Kompensation einer Menge mit einer Alternative.

Die Kosten bestehen dieses Mal aus 1 Löschkosten und 5 Sortierkosten im maximalen Fall. Diese resultieren daraus, dass die die Elemente „a“ und „b“ mit den verfügbaren Wahlen zahlenmäßg vollständig in der Alternative abgebildet werden können. Einzig Element „c“ kann nicht mehr dargestellt werden. Weiterhin kann man nicht sicher sein, ob die Instanzen die Mengen-Elemente genau so anordnen, wie sie von der Alternative benötigt werden. Aus diesem Grund wird die Sortierpauschale (immer) erhoben, die hier nur im maximalen Fall nötig ist. Wie erläutert, basiert diese auf den Elementhäufigkeiten des zu löschenden Typen. Anhand der Abbildung ist erkennbar, dass nicht die vollen 6 Elemente angerechnet werden. Dies ist so gewünscht, da die Sortierkosten erst nach Abzug der im alten Inhaltsmodell zu löschenden Elemente anfallen.

Kompensation einer Alternative mit einer Sequenz Zum Abschluss wird die Funktionsweise dieser Kompensation ebenfalls noch an einem Beispiel erläutert. Als Grundlage dient jetzt die Situation, die in Abbildung 6.36 links unten dargestellt ist. c_{max} der Alternative ist also 12 und zusätzlich werden hier die minimalen Vorkommen der Elemente in Sequenz und Alternative auf jeweils 1 gesetzt. Nach dem Auslösen des Kompensations-Dialogs ergeben sich insgesamt 15 Kosten entsprechend Abbildung 6.38.

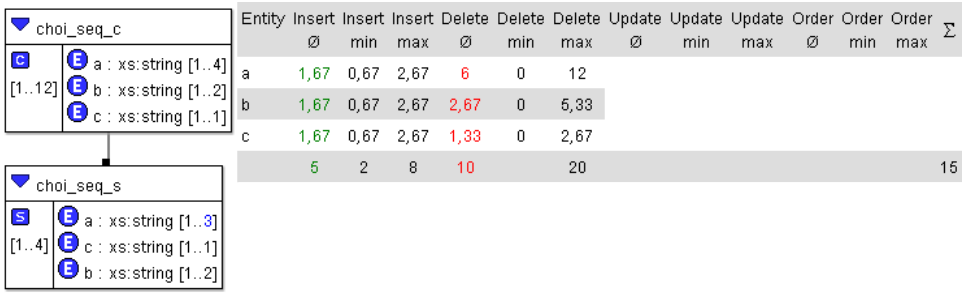


Abbildung 6.38.: Kosten für die Kompensation einer Alternative mit einer Sequenz.

Der Algorithmus berechnet die Kosten der Extremfälle für jede maximale Belegung eines Elements in der Alternative und bildet daraus den Durchschnitt. Dadurch passiert es, dass die berechneten Zahlen in der Kostentablle vermengt werden und so gleichzeitig Einfüge und Löschkosten erscheinen. Dies ist zwar nicht ideal, gibt aber immer noch Aufschluss über die Konse-

quenzen der Kompensation. Die Tabelle ist lesbar, wenn Spalten für minimale und maximale Einfüge- bzw. Löschkosten paarweise betrachtet werden.

Die Iteration zur Bestimmung der Maximalkosten startet bei Element „a“. Als Extremfall ist dieses in der Alternative 48-mal vorhanden. Der Algorithmus stellt fest, dass diese Anzahl relativ hoch ist, sodass die ideale Kompositorhäufigkeit der Sequenz s_{MinOpt} 4 beträgt, um nicht zu viele Löschkosten zu erzeugen. Mit diesem Wert fährt die Iteration fort, sodass maximal 12 „a“-Elemente in der Sequenz möglich sind. Es entstehen 36 Löschkosten, die durch die Elementnamen-Anzahl der Alternative zur Durchschnittsbildung geteilt werden, was wie in der Abbildung den 12 Kosten Löschkosten entspricht. Außerdem entstehen $s_{MinOpt} \cdot p_{j_{min}}$ Einfügekosten für die übrigen Elemente der Sequenz, die jeweils 4 betragen. Dieses Vorgehen wird nun auch für „b“ und „c“ in ihren maximalen Ausprägungen wiederholt. In diesen Durchläufen entstehen jeweils $24 - 8 = 16$ Löschkosten für Element „b“ und $12 - 4 = 8$ Löschkosten für Element „c“. Beide Werte werden ihrerseits mit der Zahl der Elementnamen geteilt, sodass durchschnittlich jeweils 5,33 und 2,67 Löschkosten entstehen. Außerdem fallen wieder 4 Einfügekosten für die jeweils zwei nicht gewählten Elemente in der Sequenz an. Dadurch entstehen im Durchschnitt wieder $\frac{4 \cdot 2}{3} = 2,67$ Einfügekosten. Alle Werte sind so auch in der Kostentabelle notiert.

Bei der Untersuchung des minimalen Falls beträgt s_{MinOpt} 1, mit dem die geringsten Einfügekosten entstehen. In jeder Iteration wird das aktuelle Element einmal gewählt und deckt direkt das erforderliche Vorkommen in der Sequenz ab. Somit fehlen in einem Schritt nur noch wie oben die zwei übrigen Elemente. Diese werden einfach mit ihren kleinsten Häufigkeiten als Einfügekosten festgehalten. Da alle Werte identisch verteilt sind, ergeben sich durchschnittlich jeweils für ein Element $\frac{1 \cdot 2}{3} = 0,67$ Einfügekosten, womit auch diese Werte der Kostentabelle erklärbar sind. Da die minimalen Elementvorkommen der Alternative die jeweiligen Vorkommen in der Sequenz mit s_{MinOpt} nicht übersteigen, sind keine Löschoperationen für den minimalen Fall nötig.

7. Fazit und Ausblick

In dieser Arbeit wurde untersucht, welche Möglichkeiten für das Management von Typhierarchien eines XML-Schemas in der XML-Schemaevolution existieren. Einleitend erfolgte dazu die Auseinandersetzung mit den wichtigen und grundlegenden Komponenten von XML-Schema wie die Deklarationen und Definitionen. Dabei wurde gezeigt, dass die Art ihrer Verwendung (auch als Modellierungsstile bezeichnet) großen Einfluss auf die XML-Schemamodellierung im Allgemeinen und die XML-Schemaevolution im Speziellen hat. In diesem Zuge wurden die verschiedenen Ableitungsformen von einfachen und komplexen Typen vorgestellt und gezeigt, wie sich diese zur Bildung von Typhierarchien verwenden lassen. Mit diesen Grundlagen war es möglich, ein XML-Schema zu entwerfen, welches alle Formen abdeckt und somit als laufendes Beispiel für die Arbeit diente.

Daraufhin erfolgte die Sichtung aktueller Werkzeuge und Prototypen, die für die Editierung von Dokumentstrukturen und für die Modellierung von Domänenkonzepten jeweils unterschiedlich gut geeignet sind. Mithilfe des entworfenen Beispielschemas wurden die einzelnen Fähigkeiten der Werkzeuge getestet und in einer zusammenfassenden Gegenüberstellung verglichen. Das hauptsächliche Ziel dabei war es, einen Einblick in den aktuellen Stand für den Umgang mit Typhierarchien eines Schemas zu erhalten. Von Interesse war dabei sowohl die Visualisierung, als auch die Behandlung von Konsequenzen nach einer Manipulation solcher Hierachien.

Die verschiedenen Erkenntnisse dieser Untersuchung flossen in die Konzeption einer eigenen Möglichkeit der Visualisierung von einfachen und komplexen Typen sowie deren Hierarchien ein. Diese Ideen wurden genutzt, um den Prototypen CodeX mit Typ-zentrischen Sichten für jeweils einfache und komplexe Typen zu erweitern. Weiterhin wurde in einer umfassenden Analyse ermittelt, unter welchen Bedingungen Operationen an verschiedenen Aspekten von Typen erlaubt sind und welche Auswirkungen diese auf eine mögliche Hierarchie haben. Die Bedingungen ergeben sich dabei aus den Regeln der Spezifikation von XML-Schema. Auf Basis dieser Regeln wurde geklärt, welche Folgeänderungen an einem manipulierten XML-Schema selbst und an den zugeordneten XML-Instanzen nach derartigen Operationen nötig sind, um die Gültigkeit von Schema und Dokumenten bei möglichst geringem Informationsverlust zu bewahren. Im Vordergrund stand dabei vor allem die Löschung bereits existierender Typen, für die ein geeigneter Ersatz zu ermitteln ist, damit abhängige Deklarationen und Untertypen weiterbestehen können.

Neben der konzeptionellen Analyse dieses Kompensations-Mechanismus erfolgte auch die programmiertechnische Umsetzung einiger Aspekte in CodeX. Dies umfasst sowohl die Kompensation für einfache als auch komplexe Typen. Als Teil der Implementierung für komplexe Typen wurde zunächst eine Methode entwickelt, mit der die Beziehung für eine Spezialform unabhängig von Hierarchieinformationen bestimmbar und visualisierbar ist. Aufbauend auf den Ergebnissen wurde anschließend eine Funktionalität implementiert, welche die Kosten für eine Ersetzung eines komplexen Typen mit einem anderen, kompensierenden von vornherein abschätzt. Als Abschluss erfolgte eine Auswertung der Implementierung anhand von selbstgewählten Beispielen.

Die umgesetzte Visualisierung bietet die Möglichkeit die Hierarchien von Typen zu erkennen. Dabei wurden Versuche unternommen, das Verständnis eines Nutzers zu soweit es geht zu fördern. Momentan sind beide Sichten voneinander getrennt. In einer Erweiterung wäre es denkbar, beide Typarten in einer einzigen Sicht darzustellen. Außerdem wären die Abhängigkeiten von Deklarationen besser erkennbar, wenn auch die Referenzbeziehungen mit angezeigt werden. Es wäre zudem eine Überlegung wert, einige Aspekte der Diagramme wie das Ein- und Ausblenden von Inhalten interaktiv zu gestalten, wobei die eigentliche Aufgabe der Modellierung im Hauptfenster von CodeX verbleiben sollte.

Außerdem lässt sich die Kompensation für einfache und komplexe Typen in einigen Aspekten weiter verbessern. Die Kompensation für gelöschte Vereinigungs-Typen arbeitet z. B. aktuell nur auf den lokalen Mitgliedstypen. In Fällen, in denen weitere Vereinigungs-Typen im Schema

vorhanden sind, wäre ein Vergleich mit diesen lohnenswert, da hier weitere Kandidaten vorhanden sein könnten. Gleiches gilt für andere atomare oder listenwertige Typen. Dazu wäre ein Algorithmus nötig, der die Facetten stärker als bisher beachtet.

Außerdem bietet die Kompensation von und Bestimmung der Beziehung zwischen komplexen Typen weitere Ansatzpunkte. Es wäre z. B. sinnvoll auch Wildcards einzubeziehen. Damit könnten Typen eher als Kandidaten während der Kompensation in Frage kommen, die solche Wildcards besitzen. Andersherum ließen sich die Löschkosten für Typen mit Wildcards genauer bestimmen. Ein Vergleich der Namensraumangaben könnte sich jedoch als aufwendig erweisen. Weiterhin werden Zusicherungen nicht beachtet. Diese in einen Vergleich exakt einzubeziehen ist technisch sehr anspruchsvoll. Für die Kompensation wirken sich Zusicherungen im Prinzip als weitere Einschränkung aus. Um diese dennoch zu beachten, könnte eine Möglichkeit darin bestehen, „Strafpunkte“ zu verteilen, je mehr Zusicherungen ein Typ besitzt.

Ein weiterer Punkt bezieht sich auf den Vergleich der Namen von Deklarationen. In Vielen Fällen wären Lösch- und Einfügeoperationen überflüssig, wenn nicht übereinstimmende Deklarationen stattdessen umbenannt werden. Voraussetzung dafür ist allerdings, dass die Typen des zu löschenden und des einzufügenden Elements oder Attributs identisch sind oder sich in einer Untertyp-Obertyp-Beziehung befinden. Wenn der zugewiesene Typ komplex ist, ergibt sich daraus die Notwendigkeit, die Typdefinitionen in allen Bestandteilen rekursiv zu vergleichen. Für einfache Typen wäre erneut ein Vergleich der Varietät, der built-in Datentypen und der Facetten vonnöten. Besitzen gleich mehrere, nicht gleich benannte Element- bzw. Attributpaare des gelöschten Typs und der Kompensation, ähnliche Typen, muss außerdem entschieden werden, welche Kombination der Umbenennung und der damit indirekt verbundenen Typzuweisung alle Paare optimal berücksichtigt. Mit der finalen Ausführung einer Umbenennung könnten daher sogar noch mehr „organisatorische“ Kosten verbunden sein, als mit einer simplen Löschung und Wiedereinfügung.

Abbildungsverzeichnis

Beispielverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Alt14] Altova GmbH: *Altova XMLSpy*. http://www.altova.com/xmlspy.html, 2014. [letzter Zugriff am 24.06.2014].

[BMV05] Barbosa, Denilson, Laurent Mignet und Pierangelo Veltri: *Studying the XML Web: Gathering Statistics from an XML Sample*. World Wide Web, 8(4):413–438, Dezember 2005.

[BNVdB04] 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.

[BPSM+08] Bray, Tim, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler und François Yergeau: *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. World Wide Web Consortium, http://www.w3.org/TR/REC-xml/, November 2008. [letzter Zugriff am 24.06.2014].

[Car06] Carlson, David A: *Semantic Models for XML Schema with UML Tooling*. In: Kendall, Elisa F., Daniel Oberle, Jeff Z. Pan, Phil Tetlow, Marwan Sabbouh und Holger Knublauch (Herausgeber): *Workshop on Semantic Web Enabled Software Engineering (SWESE 2006)*, Athens, GA, USA, 2006.

[Car12] Carlson, David A: *Semantic Models for XML Schema with UML Tooling*. http://xmlmodeling.com, 2012. [letzter Zugriff am 24.06.2014].

[Cos] Costellor, Roger: *Global versus Local*. http://www.xfront.com/GlobalVersusLocal.html. [letzter Zugriff am 24.06.2014].

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

[Ecl13] Eclipse Foundation: *Eclipse*. http://www.eclipse.org, 2013. [letzter Zugriff am 24.06.2014].

[FW04] Fallside, David C. und Priscilla Walmsley: *XML Schema Part 0: Primer Second Edition*. World Wide Web Consortium, http://www.w3.org/TR/xmlschema-0/, Oktober 2004. [letzter Zugriff am 24.06.2014].

[GIMN10] Gelade, Wouter, Tomasz Idziaszek, Wim Martens und Frank Neven: *Simplifying XML Schema: Single-type Approximations of Regular Tree Languages*. In: *Proceedings of the Twenty-ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '10, Seiten 251–260, New York, NY, USA, 2010. ACM.

[Gru11] Grunert, Hannes: *XML-Schema Evolution: Kategorisierung und Bewertung*. Bachelorarbeit, Universität Rostock, September 2011.

[Gru13] Grunert, Hannes: *Integration von Integritätsbedingungen bei der XML-Schemaevolution*. Masterarbeit, Universität Rostock, April 2013.

[GSMT12] Gao, Shudi, C. M. Sperberg-McQueen und Henry S. Thompson: *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. World Wide Web Consortium, http://www.w3.org/TR/xmlschema11-1/, April 2012. [letzter Zugriff am 24.06.2014].

[Kap13] Kaping, Chris: *Transformation von Modellierungsstilen*, Mai 2013.

[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.

[Kle] Klettke, Meike: *Modellierung, Bewertung und Evolution von XML-Dokumentkollektionen.* Habilitation.

[Mal02] Maler, Eve: *Schema Design Rules for UBL...and Maybe for You.* XML conference and exposition, 2002.

[MBV03] Mignet, Laurent, Denilson Barbosa und Pierangelo Veltri: *The XML Web: A First Study.* In: *Proceedings of the 12th International Conference on World Wide Web*, WWW '03, Seiten 500–510, New York, NY, USA, 2003. ACM.

[Mic13] Microsoft Corporation: *Microsoft Visual Studio.* `www.visualstudio.com`, 2013. [letzter Zugriff am 24.06.2014].

[MNSB06] Martens, Wim, Frank Neven, Thomas Schwentick und Geert Jan Bex: *Expressiveness and Complexity of XML Schema.* ACM Trans. Database Syst., 31(3):770–813, September 2006.

[MW06] Michel, Felix und Erik Wilde: *XML Schema Editors — A Comparison of Real-World XML Schema Visualizations.* Technischer Bericht TIK Report 265, Dezember 2006.

[NKH12] Nösinger, Thomas, Meike Klettke und Andreas Heuer: *Evolution von XML-Schemata auf konzeptioneller Ebene - Übersicht: Der CodeX-Ansatz zur Lösung des Gültigkeitsproblems.* In: Schmitt, Ingo, Sascha Saretz und Marcel Zierenberg (Herausgeber): *Grundlagen von Datenbanken*, Band 850 der Reihe *CEUR Workshop Proceedings*, Seiten 29–34. CEUR-WS.org, 2012.

[NKH13] 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, 2013. ISSN 0944-5900.

[NKMH13] Nečaský, Martin, Jakub Klímek, Jakub Malý und Irena Holubová: *Methodology for Design and Evolution of XML Schemas Using Conceptual Modeling.* In: *Proceedings of International Conference on Information Integration and Web-based Applications & Services*, IIWAS '13, Seiten 508:508–508:517, New York, NY, USA, 2013. ACM.

[Ora13a] Oracle Corporation: *Eclipse Web Tools Platform Project.* `http://projects.eclipse.org/projects/webtools`, 2013. [letzter Zugriff am 24.06.2014].

[Ora13b] Oracle Corporation: *JDeveloper.* `http://www.oracle.com/technetwork/developer-tools/jdev/overview/index.html`, 2013. [letzter Zugriff am 24.06.2014].

[PGM+12] Peterson, David, Shudi Gao, Ashok Malhotra, C. M. Sperberg-McQueen und Henry S. Thompson: *W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes.* World Wide Web Consortium, `http://www.w3.org/TR/xmlschema11-2/`, April 2012. [letzter Zugriff am 24.06.2014].

[RBG02] Routledge, Nicholas, Linda Bird und Andrew Goodchild: *UML and XML Schema*. Aust. Comput. Sci. Commun., 24(2):157–166, Januar 2002.

[Sch04] Schwentick, Thomas: *XPath Query Containment*. SIGMOD Rec., 33(1):101–109, März 2004.

A. Abbildungen

A.1. Typhierarchie von XML-Schema

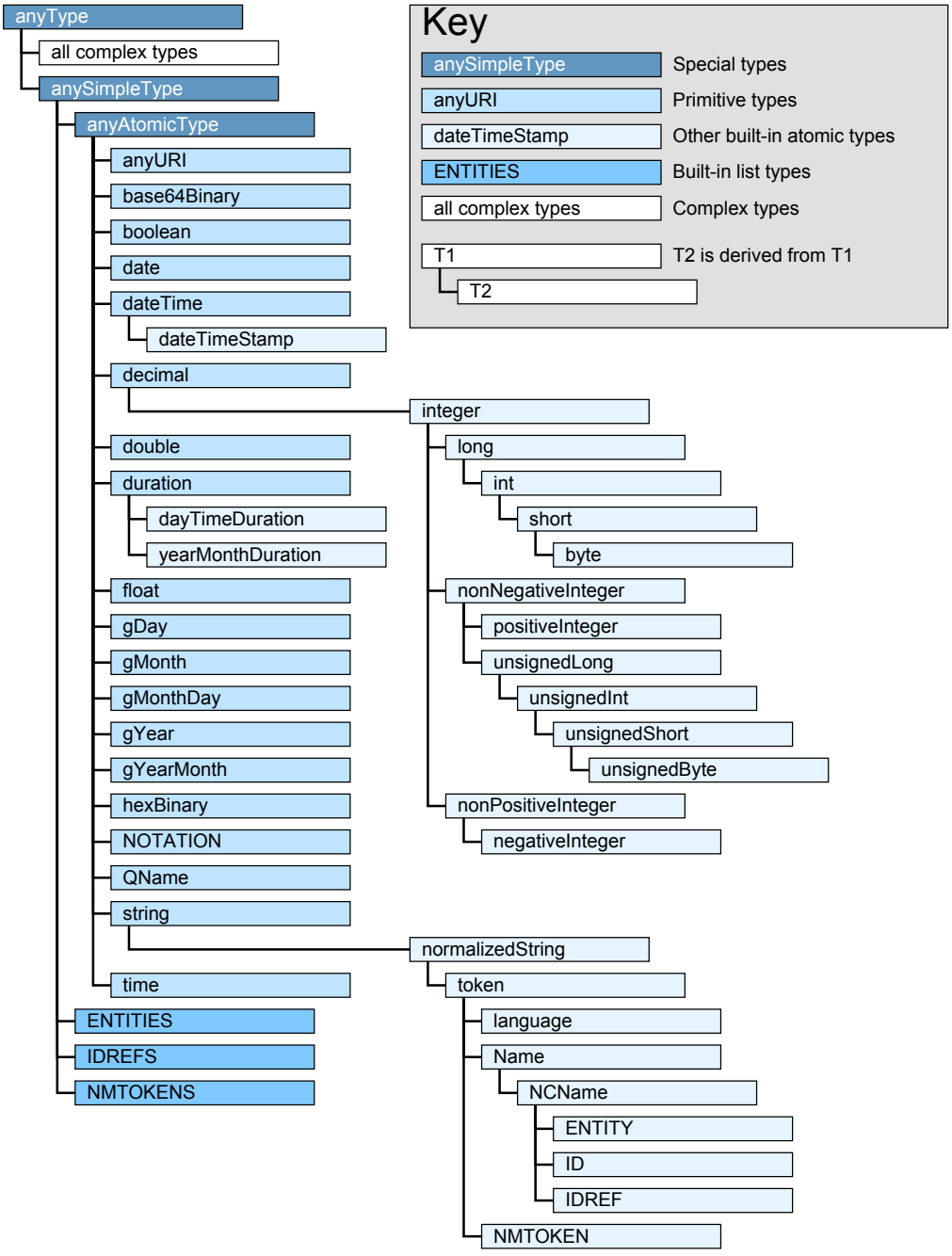


Abbildung A.1.: Typhierarchie von XML-Schema (entnommen aus [PGM+12])

A.2. Vollständige Visualisierung des Testschemas in hyperModel

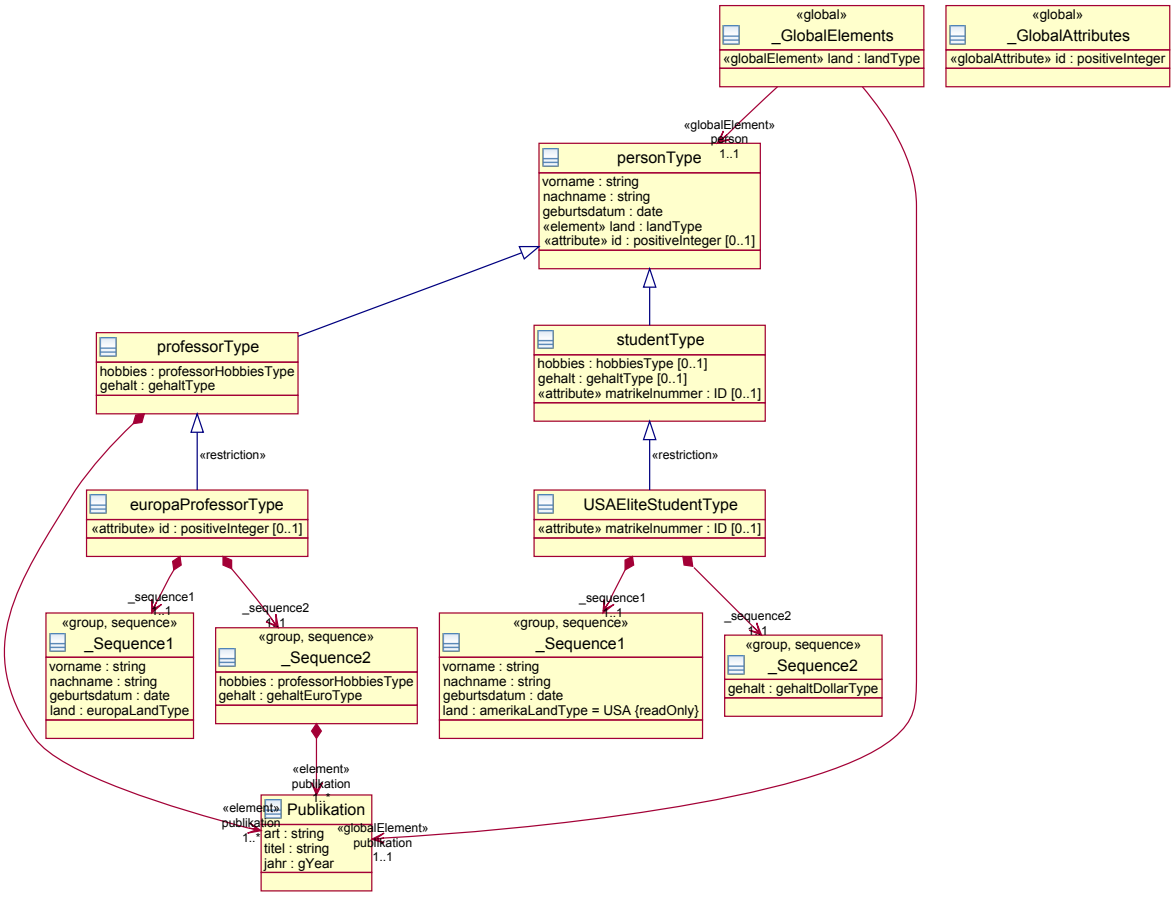


Abbildung A.2.: Vollständige Visualisierung des Testschemas in hyperModel

B. Beispiele für XML-Schemata

B.1. Person-Ableitungs-Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

 <xs:element name="person" type="personType"/>
 <xs:element name="land" type="landType"/>
 <xs:element name="publikation">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="art" type="xs:string"/>
    <xs:element name="titel" type="xs:string"/>
    <xs:element name="jahr" type="xs:gYear"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>

 <xs:attribute name="id" type="xs:positiveInteger"/>

 <xs:complexType name="personType">
  <xs:sequence>
   <xs:element name="vorname" type="xs:string"/>
   <xs:element name="nachname" type="xs:string"/>
   <xs:element name="geburtsdatum" type="xs:date"/>
   <xs:element ref="land"/>
  </xs:sequence>
  <xs:attribute ref="id"/>
 </xs:complexType>

 <xs:complexType name="professorType">
  <xs:complexContent>
   <xs:extension base="personType">
    <xs:sequence>
     <xs:element name="hobbies" type="professorHobbiesType"/>
     <xs:element name="gehalt" type="gehaltType"/>
    </xs:sequence>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>

 <xs:complexType name="europaProfessorType">
  <xs:complexContent>
   <xs:restriction base="professorType">
    <xs:sequence>
     <xs:sequence>
      <xs:element name="vorname" type="xs:string"/>
      <xs:element name="nachname" type="xs:string"/>
      <xs:element name="geburtsdatum" type="xs:date"/>
      <xs:element name="land" type="europaLandType"/>
     </xs:sequence>
     <xs:sequence>
      <xs:element name="hobbies" type="professorHobbiesType"/>
```

```
      <xs:element name="gehalt" type="gehaltEuroType"/>
     </xs:sequence>
    </xs:sequence>
   </xs:restriction>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="studentType">
 <xs:complexContent>
  <xs:extension base="personType">
   <xs:sequence>
    <xs:element name="hobbies" type="hobbiesType" minOccurs="0" maxOccurs=
       "1"/>
    <xs:element name="gehalt" type="gehaltType" minOccurs="0" maxOccurs="1
       "/>
   </xs:sequence>
   <xs:attribute name="matrikelnummer" type="xs:ID"/>
  </xs:extension>
 </xs:complexContent>
</xs:complexType>

<xs:complexType name="USAEliteStudentType">
 <xs:complexContent>
  <xs:restriction base="studentType">
   <xs:sequence>
    <xs:sequence>
     <xs:element name="vorname" type="xs:string"/>
     <xs:element name="nachname" type="xs:string"/>
     <xs:element name="geburtsdatum" type="xs:date"/>
     <xs:element name="land" type="amerikaLandType" fixed="USA"/>
    </xs:sequence>
    <xs:sequence>
     <xs:element name="hobbies" type="hobbiesType" minOccurs="0" maxOccurs
        ="0"/>
     <xs:element name="gehalt" type="gehaltDollarType" minOccurs="1"
        maxOccurs="1"/>
    </xs:sequence>
   </xs:sequence>
   <xs:attribute name="matrikelnummer" type="xs:ID" use="prohibited"/>
  </xs:restriction>
 </xs:complexContent>
</xs:complexType>

<xs:simpleType name="europaLandType">
 <xs:restriction base="xs:string">
  <xs:enumeration value="Deutschland"/>
  <xs:enumeration value="Schweiz"/>
 </xs:restriction>
</xs:simpleType>

<xs:simpleType name="amerikaLandType">
 <xs:restriction base="xs:string">
  <xs:enumeration value="USA"/>
  <xs:enumeration value="Kanada"/>
 </xs:restriction>
</xs:simpleType>
```

```
 <xs:simpleType name="landType">
  <xs:union memberTypes="europaLandType amerikaLandType"/>
 </xs:simpleType>

 <xs:simpleType name="waehrungType">
  <xs:restriction base="xs:string">
   <xs:enumeration value="Dollar"/>
   <xs:enumeration value="Euro"/>
  </xs:restriction>
 </xs:simpleType>

 <xs:complexType name="gehaltType">
  <xs:simpleContent>
   <xs:extension base="xs:nonNegativeInteger">
    <xs:attribute name="waehrung" type="waehrungType"/>
   </xs:extension>
  </xs:simpleContent>
 </xs:complexType>

 <xs:complexType name="gehaltDollarType">
  <xs:simpleContent>
   <xs:restriction base="gehaltType">
    <xs:attribute name="waehrung" type="waehrungType" fixed="Dollar"/>
   </xs:restriction>
  </xs:simpleContent>
 </xs:complexType>

 <xs:complexType name="gehaltEuroType">
  <xs:simpleContent>
   <xs:restriction base="gehaltType">
    <xs:attribute name="waehrung" type="waehrungType" fixed="Euro"/>
   </xs:restriction>
  </xs:simpleContent>
 </xs:complexType>

 <xs:simpleType name="hobbyType">
  <xs:restriction base="xs:NMTOKEN">
   <xs:maxLength value="20"/>
  </xs:restriction>
 </xs:simpleType>

 <xs:simpleType name="hobbiesType">
  <xs:list itemType="hobbyType"/>
 </xs:simpleType>

 <xs:simpleType name="professorHobbiesType">
  <xs:restriction base="hobbiesType">
   <xs:enumeration value="Gestikulieren Rhetorik Gruebeln"/>
  </xs:restriction>
 </xs:simpleType>

</xs:schema>
```

Beispiel B.1: Vollständiges Beispielschema mit verschiedenen Typableitungsformen