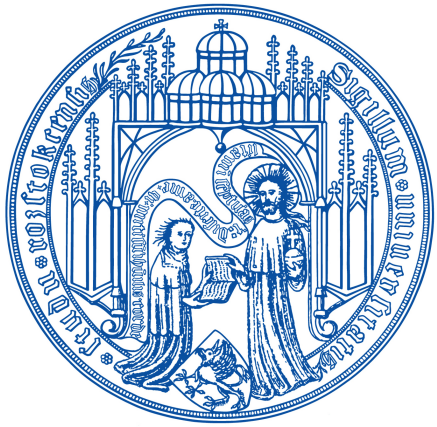
Entwicklung und Implementierung eines ELaX-Interpreters

Studienarbeit

Universität Rostock
Fakultät für Informatik und Elektrotechnik
Institut für Informatik



vorgelegt von: Norman Soetbeer
Matrikelnummer: 6201050
geboren am: 18.01.1985 in Rostock
Erstgutachter: Meike Klettke
Betreuer: Thomas Nösinger
Abgabedatum: 31. März 2014

Abstrakt

In dieser Arbeit wird eine Vorgehensweise zur Umsetzung eines Interpreters für die XML-Schema-Sprache ELaX (Evolution Language for XML-Schema) vorgestellt. Es wird erläutert, wie die einzelnen Komponenten wie Lexer, Parser und Executor funktionieren und implementiert werden. Am Ende werden diese Komponenten in einer CLI-Anwendung vereint, welche die Ausführung von ELaX-Eingaben auf einem XML-Schema ermöglicht. Neben den Vorüberlegungen zur Anwendung und Beispielen zur Implementierung werden auch angewandte Design Pattern vorgestellt.

Inhaltsverzeichnis

1. Einleitung

In der Datenverarbeitung gibt es eine Vielzahl an Datenaustauschformaten. Ein sehr verbreitetes Format stellt XML (Extensible Markup Language) [13] dar, welches vom W3C (World Wide Web Consortium) spezifiziert wurde. Um XML-Dokumente automatisiert auf Korrektheit zu prüfen, gibt es verschiedene XML-Schemasprachen, mit denen die Struktur von XML-Dokumenten definiert werden kann. Dazu zählen unter anderem DTD (Document Type Definition) [4] oder XML-Schema [15, 14], auch XSD (XML Schema Definition) genannt, RELAX NG (REgular LAnguage Description for XML New Generation) [7] oder Schematron [9]. Da - genauso wie die Daten-verarbeitende Software selbst - auch die jeweiligen XML-Schemata im Laufe der Zeit von Veränderungen betroffen sind, ist es wichtig, diese Änderungen festzuhalten (Logging), um sie schrittweise zu wiederholen oder rückgängig zu machen. Zu diesem Zweck wurde die Schema-Manipulationssprache ELaX (Evolution Language for XML-Schema) [6] entwickelt. Mit ihr lassen sich Schema-Eigenschaften, wie z.B. zulässige Elemente, Attribute, simple und komplexe Typen hinzufügen, entfernen und verändern.

Ziel dieser Studienarbeit ist es, einen Prozessor für die Sprache ELaX in Java umzusetzen und die entsprechenden Änderungen am XML-Schema vorzunehmen. Des weiteren soll die Möglichkeit geschaffen werden, diese Veränderungen rückgängig zu machen.

Die folgende Studienarbeit ist wie folgt aufgebaut:

In Kapitel 2 werden verschiedene Update-Sprachen für XML-Schemata, sowie damit verbundene Techniken und Spezifikationen, vorgestellt. Kapitel 3 beschreibt das Konzept, nach welchem die ELaX-Eingabe eingelesen, analysiert und die entsprechenden Änderungen auf ein bestehendes Schema angewendet werden sollen. Anschließend wird in Kapitel 4 auf die konkrete Implementierung mit ihren Zwischenergebnissen eingegangen. Abschließend wird in Kapitel 5 das Ergebnis dieser Studienarbeit zusammengefasst und ausgewertet.

2. State of the Art

2.1. XML-Schema

Die Korrektheit von XML-Dokumenten kann auf zwei Arten validiert werden. Zum einen auf Wohlgeformtheit, also den korrekten syntaktischen Aufbau des Dokuments, und zum zweiten auf die vorliegende Struktur, also die erlaubte Verwendung von Elementen und Attributen. Während die Wohlgeformtheit mit Hilfe des XML-Standards geprüft werden kann, ist für die Gültigkeit eines Dokuments eine Schema-Definition erforderlich. Ein Format, mit dem sich solche Definitionen beschreiben lassen, ist XML-Schema.

Die Spezifikation von XML-Schema besteht aus zwei Teilen: Strukturen und Datentypen.

In der Strukturen-Spezifikation [15] von XML-Schema wird der Aufbau, also die Struktur, von XML-Dokumenten und die Verwendung von XML-Namensräumen beschrieben. Dabei werden auch Definitionen der Datentyp-Spezifikation verwendet. Wichtige Komponenten einer XML-Schema-Struktur sind unter anderem die Deklaration und Verwendung von Elementen und Attributen oder die Definitionen von Typen. Darüber hinaus können Dokument-Strukturen u.a. näher durch die Anzahl von Elementen (minOccurs, maxOccurs) oder die Nullwertfähigkeit (nillable) beschrieben werden.

Im zweiten Teil der Spezifikation geht es um Möglichkeiten, Datentypen [14] zu definieren. Es gibt vordefinierte primitive Datentypen wie Zeichenketten (string), Boolsche Werte (boolean) oder verschiedene Zahlenformate (float, double, decimal). Die Definition neuer Datentypen lässt sich durch Erweiterung bzw. Einschränkung dieser Basistypen erreichen, z.B. durch Einschränkung einer minimalen bzw. maximalen Länge oder Festlegung eines Mindest- bzw. Höchstwerts.

Die Abbildung 2.1 zeigt als Beispiel das Ausgangsschema, welches in den nachfolgenden Kapiteln mit Hilfe von ELaX (Beschreibung in Abschnitt 2.2) erweitert wird.

Listing 2.1: Ausgangsschema

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
       elementFormDefault="qualified" attributeFormDefault="unqualified">
3      <xs:annotation>
4          <xs:documentation xml:lang="de">Beschreibt die Hinfahrts- und
               Rückfahrtszeiten</xs:documentation>
5      </xs:annotation>
6      <xs:element name="hinfahrt" type="pendlerType" />
7      <xs:complexType name="pendlerType">
8          <xs:sequence>
9              <xs:element ref="zeit" minOccurs="1" maxOccurs="unbounded"
                   />
10         </xs:sequence>
11     </xs:complexType>
```

```
12      <xs:element name="zeit" type="zeitType" />
13      <xs:complexType name="zeitType">
14          <xs:choice>
15              <xs:sequence>
16                  <xs:element ref="abfahrt" />
17                  <xs:element ref="ankunft" />
18              </xs:sequence>
19              <xs:sequence>
20                  <xs:element ref="ab" />
21                  <xs:element ref="an" />
22              </xs:sequence>
23          </xs:choice>
24          <xs:attribute ref="rythmus" use="optional" />
25      </xs:complexType>
26      <xs:element name="abfahrt" type="uhrType" />
27      <xs:complexType name="uhrType">
28          <xs:sequence>
29              <xs:element ref="stunde" />
30              <xs:element ref="minute" />
31          </xs:sequence>
32      </xs:complexType>
33      <xs:element name="stunde" type="xs:integer" />
34      <xs:element name="minute" type="xs:integer" />
35      <xs:element name="ankunft" type="uhrType" />
36      <xs:element name="ab" type="xs:time" />
37      <xs:element name="an" type="xs:time" />
38      <xs:attribute default="täglich" name="rythmus" type="rythmusType"
            />
39      <xs:simpleType name="rythmusType">
40          <xs:list itemType="tagType" />
41      </xs:simpleType>
42      <xs:simpleType name="tagType">
43          <xs:restriction base="xs:string">
44              <xs:enumeration value="MO" />
45              <xs:enumeration value="DI" />
46              <xs:enumeration value="MI" />
47              <xs:enumeration value="DO" />
48              <xs:enumeration value="FR" />
49              <xs:enumeration value="SA" />
50              <xs:enumeration value="SO" />
51              <xs:enumeration value="täglich" />
52          </xs:restriction>
53      </xs:simpleType>
54      <xs:element name="rückfahrt" type="pendlerType" />
55  </xs:schema>
```

Der Aufbau dieses Schemas wird in der Tabelle 2.1 zeilenweise beschrieben.

Tabelle 2.1.: Aufbau eines XML-Schemas

Zeile(n)	**Bemerkungen**
1	XML-Version und Kodierung des Dokuments. Dies ist der Standard-XML-Header, der bei XML-Dokumenten angeführt werden kann.
2 - 55	Das Schema. Hier können vordefinierte Werte für Elemente und Attribute festgelegt werden.
3 - 5	Annotationen können das Schema formlos beschreiben.
6, 12, 26, 54	Element-Deklarationen beinhalten den Namen des Elements (z.B. `hinfahrt`) und einen Typ, welcher hier an jeweils anderer Stelle näher beschrieben wird.
7 - 11	Komplexe Typen: können über den Namen referenziert werden und beschreiben, aus welchen anderen Typen sie zusammengestellt werden können. In diesem Beispiel eine Folge von `zeit`-Elementen, welche aus mindestens einem Element bestehen muss.
33 - 34, 36 - 37	Element-Deklarationen mit Basis-Datentypen `integer` (Ganzzahl) und `time` (Zeitangabe)
38	Attributdeklaration, bestehend aus Name, Typ und Standardwert, welcher verwendet wird, falls das Attribut nicht explizit angegeben wird.
42 - 53	Einfache Typen: erweitern einen Basis-Typen oder schränken diesen ein. Hier wird der `string`-Datentyp, also eine Zeichenkette, auf eine Auswahl an möglichen Werten eingeschränkt.

2.2. ELaX

Bei ELaX (Evolution Language for XML) [12] handelt es sich um eine Schema-Update-Sprache. Mit ihr lassen sich neue XML-Schemata von Grund auf erstellen oder existierende Schemata verändern, z.B. durch Hinzufügen neuer Element-Deklarationen, Löschen vorhandener Deklarationen oder dem Update einer Typdefinition. Eine vollständige Liste von ELaX-Operationen befindet sich im Anhang A.1.

2.2.1. Garden of Eden

Um einen gewissen Grad an Flexibilität zu gewährleisten, ist eine wichtige Voraussetzung, dass die XML-Schemata im "Garden of Eden"-Stil [5] vorliegen. Dies bedeutet, dass sämtliche Element- und Typdeklarationen global definiert sind und sich somit auf oberster Ebene im Schema befinden. Das Auffinden bestehender Deklarationen kann somit stark vereinfacht werden. Außerdem ermöglicht dieser Stil eine hohe Wiederverwendbarkeit von Deklarationen.

2.2.2. ELaX-Beispiele

Nachfolgend wird das Beispiel-Schema durch einige ausgewählte ELaX-Operationen erweitert. Es wird die allgemeine Syntax in EBNF (Erweiterte Backus-Naur-Form) dargestellt, gefolgt von einem konkreten ELaX-Befehl, welcher auf das Schema angewandt wird, sowie das Ergebnis nach Ausführung des jeweiligen Befehls abgebildet und beschrieben.

2.2.3. Beispiel 1: Hinzufügen von Attributen

Es sollen die Attribute `start` und `ziel` verfügbar gemacht werden. Die Attributwerte sollen vom Basistyp `string` sein.

EBNF

Um eine Attribut-Definition zum Schema hinzuzufügen, wird die Anweisung `add attribute` verwendet. Die EBNF für diesen Befehl ist in Abbildung 2.2 dargestellt.

Listing 2.2: EBNF für das Hinzufügen eines Attributs

```
addattribute ::=
    "add" "attribute" "name" NCNAME "type" QNAME
    (("default" | "fixed") STRING)? ("id" ID)?
    ("inheritable" ("true" | "false"))? ;
```

ELaX-Befehl

Der konkrete ELaX-Befehl, um die beiden Attribute hinzuzufügen, besteht aus zwei dieser `add attribute` Anweisungen, dargestellt in Listing 2.3.

Listing 2.3: ELaX-Befehl zum Hinzufügen zweier Attribute

```
add attribute name "start" type "xs:string";
add attribute name "ziel" type "xs:string";
```

Ergebnis

Die Attribut-Deklarationen wurden global zum Schema hinzugefügt (Zeile 55 - 56, Listing 2.4).

Listing 2.4: Ergebnis nach Hinzufügen der Attribute

```
54      <xs:element name="rückfahrt" type="pendlerType" />
55      <xs:attribute name="start" type="xs:string" />
56      <xs:attribute name="ziel" type="xs:string" />
57  </xs:schema>
```

2.2.4. Beispiel 2: Deklaration einer Attributgruppe

Die im ersten Beispiel (Kapitel 2.2.3) deklarierten Attribute sollen nun in einer Attributgruppe referenziert werden. Die Gruppe soll den Namen "tour" tragen.

EBNF

Um eine Attributgruppe zu definieren, wird die Anweisung `add attributegroup` verwendet. Die entsprechende EBNF ist in Abbildung 2.5 dargestellt.

Listing 2.5: EBNF für das Hinzufügen einer Attributgruppe

```
addattributegroupdef ::=
    "add" "attributegroup" "name" NCNAME ("id" ID)?
    "with" (<attributeref>)+ (<attributewildcard>)? ;

attributeref ::=
    "attributeref" QNAME
    (("default" | "fixed") STRING)?
    ("use" ("prohibited" | "optional" | "required"))?
    ("id" ID)? "in" <locator> ;

attributewildcard ::=
    "anyattribute" ("not" (QNAME | "##defined")+)?
    ("namespace" ("##any" | "##other" | ("##local" | ANYURI | "##
       targetnamespace")+)?
    ("not" (ANYURI | ("##targetnamespace" | "##local"))+)?)?
    ("processcontent" ("lax" | "skip" | "strict"))?
    ("id" ID)? "in" <locator> ;

locator ::= <xpathexpr> | emxid ;

xpathexpr ::= ("/" ("." | ("node()" | ("node()[@name='" NCNAME "']"))
   ("[" INT "]")? ) )+ ;
```

ELaX-Befehl

Der konkrete ELaX-Befehl für das Hinzufügen der Attributgruppe, sowie die Referenzierung der beiden vorher definierten Attribute, ist in Listing 2.6 dargestellt.

Listing 2.6: ELaX-Befehl für das Hinzufügen einer Attributgruppe

```
add attributegroup name "tour" with
    attributeref 'start' use "required" in "/node()/node()[@name='tour
       ']/."
    attributeref 'ziel' use "required" in "/node()/node()[@name='tour
       ']/." ;
```

Ergebnis

Es wurde eine neue Attributgruppe mit dem Namen `tour` hinzugefügt und die beiden Attribute darin referenziert (Zeilen 57 - 60, Listing 2.7).

Listing 2.7: Ergebnis nach Hinzufügen der Attributgruppe

```
56      <xs:attribute name="ziel" type="xs:string" />
57      <xs:attributeGroup name="tour">
```

```
58          <xs:attribute ref="start" use="required"/>
59          <xs:attribute ref="ziel" use="required"/>
60      </xs:attributeGroup>
61  </xs:schema>
```

2.3. Interpreter

Um die Eingabe in Form von ELaX-Anweisungen zu interpretieren und die entsprechenden Änderungen auf einem Schema anwenden zu können, ist es erforderlich, die Eingabe lexikografisch zu analysieren, syntaktisch zu validieren und die Operationen am Schema selbst umzusetzen.

2.3.1. Lexikografische Analyse

Bei der lexikografischen Analyse handelt es sich um das Zerlegen der Eingabe in einzelne Token, wobei die Token selbst aus definierten Zeichenfolgen bestehen. Dies können z.B. Schlüsselwörter wie "add" oder "delete" sein oder auch von Anführungszeichen umschlossene Zeichenketten. Für diese Analyse wird ein Lexer (auch Scanner oder Tokenizer genannt) benötigt, welcher diese Zerlegung vornimmt.

2.3.2. Syntaktische Analyse und Syntaxbaum

Für die syntaktische Analyse wird ein Parser benötigt. Dessen Aufgabe ist die Validierung der Tokenfolge des Lexers, sowie das Aufbauen eines Syntaxbaums. Es gibt verschiedene Möglichkeiten, solch einen Parser zu entwickeln. Beispielsweise lassen sich Parser automatisch durch sogenannte Parsergeneratoren generieren. Der dazu passende Lexer kann - je nach verwendetem Werkzeug - mit generiert werden. Das bekannteste, mehrsprachige Werkzeug ist ANTLR (ANother Tool for Language Recognition) [1]. Hierzu muss die Sprache in EBNF (Erweiterte Backus-Naur-Form) angegeben werden. Die entsprechenden Token und Syntax-Regeln werden automatisch aus der EBNF abgeleitet. Als Ergebnis erhält man einen Lexer, der in der Lage ist, alle in der EBNF vorkommenden Token zu erkennen und einen Parser, der Eingaben auf die Syntax der EBNF überprüft und einen abstrakten Syntaxbaum aufbauen kann.

2.3.3. Befehlsausführung

Anstatt - wie beim Compilerbau üblich - aus dem Syntaxbaum den Zielcode zu generieren, werden die Befehle aus dem Syntaxbaum bei einem Interpreter direkt ausgeführt. Im Fall von ELaX erfolgt also direkt die Manipulation des XML-Schemas über die DOM-Schnittstelle.

2.3.4. XML-Manipulation

Es gibt verschiedene Ansätze, um XML-Dokumente in eine Java-Struktur einzulesen und manipulieren zu können. Zu den bekanntesten zählen SAX, StAX, DOM und JDOM, welche nachfolgend kurz beschrieben werden.

SAX - Simple API for XML

Bei SAX (Simple API for XML) [8] handelt es sich um eine schnelle, speicherschonende, Ereignis-orientierte Variante der XML-Verarbeitung. Es werden solange Daten von einer Eingabe gelesen, bis ein öffnender oder schließender XML-Tag vollständig eingelesen wurde. Anschließend wird ein entsprechendes Ereignis mit den Tag-Informationen an registrierte Ereignishandler übergeben, welche z.B. auf das Öffnen des Dokuments (`startDocument()`), das Schließen des Dokuments (`endDocument()`), das Starten eines Elements (`startElement( String namespaceURI, String localName, String qName, Attributes attrs )`) oder das Beenden eines Elements (`endElement( String namespaceURI, String localName, String qName )`) reagieren können.

Dadurch, dass jeweils nur das gerade eingelesene Element im Speicher verbleibt, bis das dazugehörige Ereignis geworfen wird, ist der Speicherverbrauch im Vergleich zu anderen Methoden sehr gering. Somit lassen sich auch sehr große XML-Dokumente speichereffizient verarbeiten. Mit SAX können Dokumente nur eingelesen, jedoch nicht manipuliert oder geschrieben werden.

StAX - Streaming API for XML

Ähnlich wie bei SAX, wird auch mit StAX (Streaming API for XML) [11] das XML-Dokument sequenziell durchlaufen. Jedoch werden hierbei keine Ereignisse zu den registierten Ereignishandlern geschickt ("Push-Parsing"). Stattdessen wird das Iterator-Konzept angewendet. Die Kontrolle des Parsers liegt beim Anwendungsprogramm, welches aktiv das nächste XML-Element anfordern muss ("Pull-Parsing").

Im Gegensatz zu SAX, können mit StAX auch Dokumente erzeugt werden.

DOM - Document Object Model

Das vom W3C entwickelte DOM (Document Object Model) [3] stellt eine sprachunabhängige Schnittstelle für Baumstrukturen dar. Da sich jedes XML-Dokument in eine Baumstruktur umwandeln lässt, ist dies geeignet, um XML-Dokumente zu manipulieren. Dabei werden XML-Elemente und -Attribute durch Knoten im Baum dargestellt, das Dokument an sich ist dabei der Wurzelknoten. Durch das Anfügen oder Löschen von Element-, Attribut- oder Textknoten wird das XML-Dokument verändert.

Da der DOM-Baum das komplette XML-Dokument repräsentiert, wächst der Speicherverbrauch linear zur Größe des XML-Dokuments. Bibliotheken, die eine Form des DOM implementieren, verwenden oftmals SAX oder StAX, um den DOM-Baum aus einem XML-Dokument heraus aufzubauen.

JDOM - Java Document Object Model

Bei JDOM (Java Document Object Model) handelt es sich um eine DOM-Implementierung, die speziell auf Java ausgerichtet ist. Während das DOM selbst eine sprachunabhängige Schnittstelle darstellt, macht JDOM von Java-Datentypen Gebrauch und ist somit effizienter bezüglich der Geschwindigkeit und des Speicherverbrauchs.

2.3.5. Alternative Update-Sprachen

Um XML-Dokumente im Allgemeinen oder XML-Schemata im Speziellen zu manipulieren, gibt es eine Reihe von Update-Sprachen, welche nachfolgend vorgestellt werden.

XQuery

XQuery (XML Query) [16] ist eine Abfragesprache für XML-Dokumente, die sich mit SQL (Structured Query Language) für relationale Datenbanken vergleichen lässt. Ein besonderer Bestandteil von XQuery sind die sogenannten FLWOR-Ausdrücke. FLWOR steht für die Anfangsbuchstaben von `for` (Schleifenkonstrukt), `let` (Erstellen von Variablen bzw. Zuweisung von Werten), `where` (konditionale Anweisung), `order by` (Sortierung) und `return` (Rückgabe von Werten).

Einer der Vorläufer von XQuery ist XPath, womit sich Elemente anhand ihres Pfades von der Wurzel, also dem Dokument selbst, bis zum Zielelement selektieren lassen. Für die Angabe der Pfade ist es jedoch notwendig, die XML-Struktur zu kennen. Auch die Verwendung von Variablen oder das Erzeugen neuer Elemente ist mit XPath nicht möglich.

Nachfolgend wird eine Möglichkeit aufgezeigt, um mit Hilfe von XQuery die zwei Attribute "start" und "ziel" vom Basistyp String global zum Schema hinzuzufügen.

Um zwei Attribute global zu einem XML-Schema hinzuzufügen, kann ein XQuery-Ausdruck verwendet werden, wie in Listing 2.8 dargestellt. Dort werden "direct constructors", also direkte Konstruktoren für die XML-Elemente verwendet. Die Attributewerte für `elementFormDefault` und `attributeFormDefault` werden aus dem Original-Schema übernommen (Zeile 1). Anschließend werden sämtliche Inhalte des ursprünglichen Schemas ausgegeben (Zeile 2). In den Zeilen 3 und 4 werden die neuen XML-Schema-Attribute direkt als XML hinzugefügt.

Listing 2.8: XQuery-Ausdruck für das Hinzufügen von zwei Attributen

```
1 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
     elementFormDefault="{ //xs:schema/@elementFormDefault }"
     attributeFormDefault="{ //xs:schema/@attributeFormDefault }">
2     { //xs:schema/node() }
3     <xs:attribute name="start" type="xs:string" />
4     <xs:attribute name="ziel" type="xs:string" />
5 </xs:schema>
```

Es erfolgt die gleiche Ausgabe, wie auch schon im ELaX-Beispiel zu sehen war. Die Attribut-Deklarationen wurden global zum Schema hinzugefügt (Zeile 55 - 56, Listing 2.9).

Listing 2.9: Ergebnis der XQuery-Abfrage

```
54     <xs:element name="rückfahrt" type="pendlerType" />
55     <xs:attribute name="start" type="xs:string" />
56     <xs:attribute name="ziel" type="xs:string" />
57 </xs:schema>
```

XSLT

Unter XSLT (Extensible Stylesheet Language Transformations) [17] versteht man eine Sprache zur Transformation von XML-Dokumenten mit Hilfe der XSL (XML Stylesheet Language).

Das Format für solch ein XSL-Stylesheet basiert ebenfalls auf XML. Um ein Dokument zu transformieren, werden sogenannte Templates erstellt, welche auf alle Knoten angewendet werden, die auf den dazugehörigen XPath-Ausdruck zutreffen. Ein Template mit dem XPath-Ausdruck `/` würde also auf den Wurzelknoten angewandt werden. Es können Attribute, Elemente oder Textknoten aus dem Original-Dokument übernommen werden. Neue Elemente oder Attribute lassen sich entweder durch die direkte XML-Schreibweise erstellen oder durch entsprechende XSL-Tags, wie `xsl:element` oder `xsl:attribute` konstruieren.

Um mit XSLT zwei neue Attribute zu einem XML-Schema hinzuzufügen, muss ein neues XSL-Stylesheet angelegt werden, welches wie in Listing 2.10 dargestellt aussehen kann.

Listing 2.10: Beispiel: XSL-Transformation

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/
       Transform">
3
4      <xsl:output method="xml" />
5
6      <xsl:template match="/">
7          <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
8              <xsl:attribute name="elementFormDefault">
9                  <xsl:value-of select="/xs:schema/@elementFormDefault"
                       />
10             </xsl:attribute>
11             <xsl:attribute name="attributeFormDefault">
12                 <xsl:value-of select="/xs:schema/@attributeFormDefault
                       " />
13             </xsl:attribute>
14             <xsl:copy-of select="/xs:schema/*"/>
15             <xs:attribute name="start" type="xs:string" />
16             <xs:attribute name="ziel" type="xs:string" />
17         </xs:schema>
18     </xsl:template>
19
20 </xsl:stylesheet>
```

Darin wird ein Template erstellt, welches auf den Wurzelknoten, also das Schema selbst, angewandt wird. Es wird ein neues Schema-Element erstellt (Zeile 7), und die Attributwerte aus dem originalen Schema übernommen (Zeile 8 - 13). Anschließend werden alle Kindknoten in das neue Schema kopiert (Zeile 14). In den Zeilen 15 und 16 werden dann die neuen Elemente wie gewünscht hinzugefügt.

Listing 2.11: Ergebnis der XSL-Transformation

```
54     <xs:element name="rückfahrt" type="pendlerType" />
55     <xs:attribute name="start" type="xs:string" />
56     <xs:attribute name="ziel" type="xs:string" />
57 </xs:schema>
```

Auch hier wurden die Attribut-Deklarationen global zum Schema hinzugefügt (Listing 2.11, Zeile 55 - 56).

3. Konzept

Um eine Sprache in ausführbaren Code zu übersetzen, ist es notwendig, diese Sprache zu parsen. Solch ein Parser kann mit vorhandenen Tools generiert oder manuell geschrieben werden.

Für das Generieren eines Lexers und des dazugehörigen Parsers ist es notwendig, sich mit der Dokumentation auseinanderzusetzen. Genauer betrachtet wurde hierbei der ANTLR (ANother Tool for Language Recognition) [1] Parser-Generator, welcher neben Java auch für weitere Sprachen verfügbar ist.

ANTLR ermöglicht es, anhand einer EBNF-konformen Beschreibung der Sprache einen Lexer, sowie den dazugehörigen Parser generieren zu lassen. Es wurde untersucht, welche Schritte für solch eine Generierung notwendig sind, doch schon die Beispiele und Dokumentationen ließen die Handhabung recht schwierig erscheinen. Die Vorteile solch eines Parsergenerators liegen in der einfachen Anpassbarkeit der EBNF, wenn die initiale Konfiguration durchgeführt wurde. Hierfür ist jedoch eine gewisse Einarbeitungszeit notwendig, welche als enormer Nachteil anzusehen ist.

Für den Parser-Generator spricht ein stabiles, lange getestetes Projekt. Doch gleichzeitig ist damit auch eine Einarbeitung verbunden. Die Ausführungslogik muss in beiden Fällen geschrieben werden. Als Alternative wurde die manuelle Implementierung eines Lexers und Parsers betrachtet. Die Vorkenntnisse aus der Vorlesung Compilerbau können hier angewandt werden. Einen Nachteil bei der Implementierung stellt der große Umfang der Sprache ELaX dar. Auf der anderen Seite sind die Schritte für Umsetzung klar, weshalb ich mich für diese Variante entschieden habe.

3.1. Zerlegung der EBNF

Die Sprache ELaX liegt in EBNF (Erweiterte Backus-Naur-Form) vor. Da einige Teilausdrücke wiederholend vorzufinden sind, bietet es sich an, diese EBNF so umzuschreiben, dass möglichst wenig Wiederholungen vorhanden sind. Somit muss das Parsen von Teilausdrücken nur jeweils einmal implementiert werden. Ein Beispiel für solch einen Teilausdruck ist `("id" QNAME)?`, welcher in eine eigene Klausel ausgelagert wird: `idclause ::= "id" QNAME`

Somit kann ein komplexer ELaX-Ausdruck, wie in Listing 3.1 dargestellt, durch einen einfacheren Ausdruck ersetzt werden. In Listing 3.2 ist zu erkennen, dass u.a. die Angaben des Namen, Typen oder einer ID durch die jeweilige Klausel ersetzt wurden. Gleichzeitig wurden die neu eingeführten Klauseln in die EBNF aufgenommen.

Listing 3.1: EBNF vor der Vereinfachung

```
1 addattribute ::= "attribute" "name" NCNAME "type" QNAME
2     (("default" | "fixed") STRING)? ("id" ID)?
3     ("inheritable" ("true" | "false"))? ;
```

Listing 3.2: EBNF nach der Vereinfachung

```
addattribute ::= "attribute" NameNcNameClause TypeClause
    DefaultOrFixedClause? IdClause?
    InheritableClause? ;

DefaultOrFixedClause ::= ("default" | "fixed") STRING
IdClause ::= "id" ID
InheritableClause ::= "inheritable" ("true" | "false")
NameNcNameClause ::= "name" NCNAME
TypeClause ::= "type" QNAME
```

Da sich einzelne Sequenzen, wie z.B. `"id" ID`, auch in weiteren Regeln wiederholen, können diese nun auch wiederverwendet werden. Somit entfällt später die mehrfache Implementierung gleicher Sequenzen innerhalb des Parsers, gemäß dem DRY-Prinzip ("Don't Repeat Yourself"). Das Ergebnis ist also eine EBNF, welche deutlich mehr Regeln als zuvor aufweist. Diese Regeln haben aber eine geringere Komplexität als die vorherigen. Eine vollständige Übersicht aller Regeln nach Ersetzung der Klauseln befindet sich im Anhang A.2.

Bei der Unterteilung der einzelnen Regeln fällt auf, dass diese sich in Script (komplette ELaX-Eingabe), Statements (einzelne Anweisungen) und Klauseln / Clauses (zusammenhängende Schlüsselwörter und Werte) einteilen lassen. Dementsprechend wurden die Bezeichner für diese Sequenzen gewählt, bspw. `AddElementStatement` oder `TypeClause`

Bei der Ersetzung der Teilausdrücke muss beachtet werden, dass die Klauseln weiterhin optional bleiben müssen, wenn sie es vorher auch waren.

3.2. Interpreter

Um ein XML-Schema automatisiert mittels ELaX zu verändern sind folgende Schritte notwendig:

1. ELaX-Script lexen
2. Tokenfolge parsen und Syntaxbaum aufbauen
3. XML-Dokument gemäß des Syntaxbaums manipulieren
4. Loggen von Operationen

3.2.1. ELaX-Script lexen

Der Lexer (auch Scanner oder Tokenizer genannt) [2] hat die Aufgabe, die Eingabe, also eine Zeichenkette, in sogenannte Token zu zerlegen. Im Falle von ELaX werden nicht-druckbare Zeichen wie Leerzeichen, Tabulatoren oder Zeilenumbrüche als Trennsymbol angesehen. Ein Token steht für eine Folge von Zeichen aus der Eingabe, welcher ein Token-Typ zugeteilt wird. Token-Typen können u.a. "ID" ("id"-Schlüsselwort) oder "QUOTED_STRING" (in Anführungszeichen gekapselte Zeichenkette) sein. Somit wird den einzelnen Bestandteilen der Eingabe eine Semantik mitgegeben.

TokenGuesser und Matcher

Die Zuordnung eines Tokentyps zu einem Token erfolgt durch den TokenGuesser. Bei diesem werden alle notwendigen TokenMatcher registriert. Es gibt verschiedene TokenMatcher, wie z.B. den `StringTokenMatcher` für konstante Zeichenfolgen (”add”, ”element”, ”id”, usw.) oder den `RegexTokenMatcher` für komplexere Zeichenfolgen, die sich mit Regulären Ausdrücken beschreiben lassen, beispielsweise ”Quoted Strings”, also Zeichenfolgen, die durch Anführungszeichen begrenzt sind, aber auch Leerzeichen oder maskierte Anführungszeichen enthalten dürfen.

Abbildung 3.1.: Token-Sequenz nach dem Lex-Vorgang

Wie in der Abbildung 3.1 dargestellt, wird die Eingabe in Token zerlegt, welche aus dem Wert aus der Eingabe, dem Token-Typ und der Position innerhalb der Eingabe bestehen.

3.2.2. Parsen der Token-Sequenz

Die Aufgabe des Parsers [2] ist es, die Tokenfolge des Lexers auf eine zulässige Reihenfolge zu prüfen und gleichzeitig einen Syntaxbaum aufzubauen. Dazu wird anhand von Schlüsselwörtern entschieden, welche Token-Typen zu erwarten sind. Weicht die tatsächliche Tokenfolge davon ab, liegt ein Syntaxfehler vor, wodurch der Vorgang des Parsens abgebrochen wird. Solange die Tokenfolge stimmt, werden rekursiv Teilbäume des Syntaxbaums erzeugt und somit schrittweise der Syntaxbaum aufgebaut. Dieser Syntaxbaum stellt das Ergebnis des Parsers dar.

3.2.3. Manipulation des XML-Schemas

Das XML-Schema muss zunächst mit Hilfe des JDOM-Parsers geladen werden. Um eine API verwenden zu können, welche auf die Anforderungen der Manipulation von XML-Schemata zugeschnitten ist, ist es erforderlich, eine Fassade (Facade Design Pattern) um die DOM-API herum zu entwickeln. Diese API soll es unter anderem ermöglichen, direkt Attribute (`addAttribute( Attribute attr )` oder Elemente (`addElement( Element elem )`) im Schema hinzuzufügen anstatt DOM-Knoten zu erzeugen und als Kindknoten hinzuzufügen. Das Fassade-Pattern ermöglicht es, eine einfachere, den Anforderungen entsprechende API anzubieten und die hohe Komplexität der ursprünglichen API (hier DOM) zu verstecken.

Anschließend wird der Syntaxbaum durchlaufen und je nach Knotentyp entsprechende Operationen auf dem DOM ausgeführt. Für das Durchlaufen des Baumes kann ein TreeWalker verwendet werden, welcher das Visitor-Pattern [10] umsetzt. Das Visitor-Pattern ermöglicht es, durch verschachtelte Strukturen, wie z.B. Bäume, zu iterieren. Für jeden Knotentyp muss

dabei eine Methode im Visitor implementiert sein, welche mit den jeweiligen Knoten aufgerufen wird.

3.2.4. Loggen von Operationen

Um Operationen nachvollziehen oder gar umkehren zu können, müssen diese geloggt werden. Dies kann, wie auch bei der Manipulation des XML-Schemas, durch einen TreeWalker geschehen. Anstatt die Operationen auf dem Schema auszuführen, werden diese lediglich in einem Logfile gespeichert.

4. Implementierung

Nachfolgend wird auf die Implementierung der CLI-Anwendung sowie die Umsetzung des Lexers und Parsers eingegangen. Ebenfalls werden Zwischenergebnisse der einzelnen Komponenten visualisiert.

Eine Übersicht der implementierten Klassen und definierten Schnittstellen befindet sich im Anhang A.3.

4.1. CLI-Anwendung

Um die Komponenten im Zusammenspiel ausprobieren zu können, wird eine einfache CLI-Anwendung entwickelt. Es kann ein vorhandenes XML-Schema geladen und mittels auswählbarer ELaX-Befehle manipuliert werden. Das geänderte Schema wird wieder in einer XSD-Datei gespeichert. Für die einfachere Verarbeitung von Konsolen-Argumenten wird "Apache Commons CLI"verwendet. Damit lassen sich optionale Argumente wie der Pfad zur Ein- bzw. Ausgabedatei definieren oder die Hilfe mit allen definierten Optionen auflisten. Weitere verwendete Bibliotheken sind unter anderem "Apache Commons Lang", welche häufig benötigte Hilfsmethoden zur String-Verarbeitung bereitstellt oder JDOM für die Manipulation von XML-Schemata. Eine vollständige Liste der verwendeten Bibliotheken ist im Anhang A.1 zu finden.

Beim Starten der Anwendung wird das angegebene Schema geladen und in einen DOM-Baum konvertiert. Es werden einige ELaX-Beispiele in der CLI-Anwendung vordefiniert, welche in belibieger Reihenfolge auf dem XML-Schema ausgeführt werden können. Die ausgewählten ELaX-Anweisungen werden als Eingabe für den Lexer verwendet. Dieser wird wiederum an den Parser übergeben. Der daraus resultierende Syntaxbaum wird an den ExecutionVisitor übergeben, welcher den DOM-Baum manipuliert. Das Ergebnis der Ausführung wird an den Benutzer ausgegeben. Wenn das Programm erfolgreich beendet wird, wird auch das Schema unter dem Pfad der Ausgabedatei gespeichert.

4.2. Lexer

Der Lexer ist für das Einlesen der Zeichenfolgen, die die ELaX-Befehle darstellen, verantwortlich. Da die Sprache aus eindeutigen Termen, Leerzeichen und markierten Zeichenketten besteht, ist die Umsetzung einfach. Die Zeichenketten werden mit Hilfe von regulären Ausdrücken an den Leerzeichen aufgeteilt mit Ausnahme von Zeichenketten, welche von Anführungszeichen umschlossen sind. Neben der Aufteilung der Zeichenketten besteht die Aufgabe des Lexers weiterhin darin, die einzelnen Terme mit Informationen wie Tokentyp oder Position innerhalb der Eingabezeichenkette anzureichern. Der Tokentyp beschreibt die Semantik des einzelnen Terms. Dies ist notwendig, um später den Aufbau einer ELaX-Anweisung evaluieren zu können.

In dem Beispiel in Abbildung 4.1 wird die "mixed"-Eigenschaft des komplexen Typs mit den Namen "beschreibungTyp" auf den boolschen Wert "true" geändert.

Listing 4.1: Beispiel einer ELaX-Anweisung

```
1 update complextype name "beschreibungType" change mixed true ;
```

Das Ergebnis des Lex-Vorgangs, welcher auf das obige Beispiel (Abbildung 4.1) angewandt wurde, wird in Tablle 4.1 dargestellt.

Tabelle 4.1.: Ergebnis des Lexers

Zeichenfolge	**Tokentyp**	**Position**
update	UPDATE	0
complextype	COMPLEXTYPE	7
name	NAME	22
'beschreibungType'	QUOTED_STRING	27
change	CHANGE	47
mixed	MIXED	54
true	TRUE	60
;	STATEMENT_SEPARATOR	65

4.3. Parser

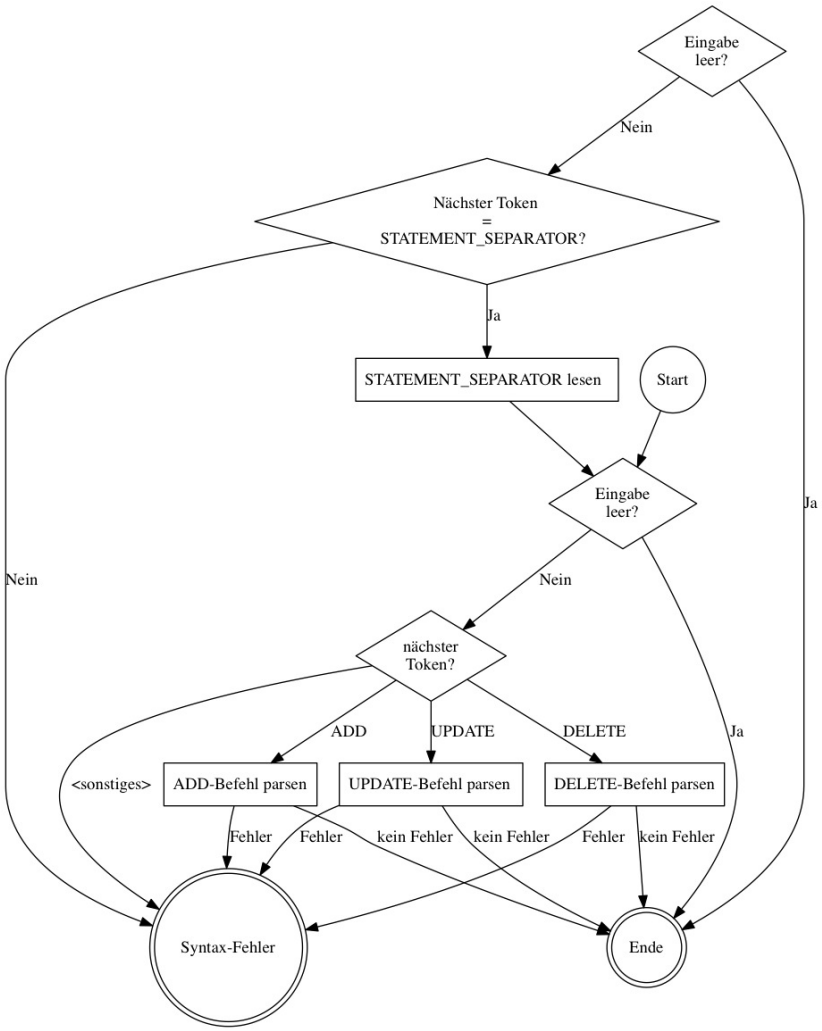
Der Parser ist verantwortlich für die Validierung der Tokenfolge, welche durch den Lexer erzeugt wurde, und für das Aufbauen eines Syntaxbaums daraus. Aufgrund der hohen Komplexität von ELaX war sich die Umsetzung ebenfalls aufwendig. Jede korrekte Abfolge von Token musste geprüft werden und im Anschluss daran entsprechende Knoten-Objekte erstellt und korrekt zusammengesetzt werden.

In der Hauptschleife des Parsers (Abb. 4.1) wird versucht, solange Anweisungen zu parsen, bis die Token-Sequenz vollständig durchlaufen wurde. Innerhalb dieser Hauptschleife werden, abhängig von den vorkommenden Token und entsprechend der ELaX-Syntax die einzelnen Befehle geparst. Ein Beispiel für das Parsen des "`add attributegroupref`"-Befehls wird in der Abbildung 4.2 veranschaulicht.

Nach erfolgreichem Lesen des `ADD`-Tokens stehen mehrere weitere Token zur Auswahl. In diesem Fall handelt es sich um den Token-Typ `ATTRIBUTEGROUPREF`. Die Syntax für diesen Befehl sieht vor, einen QName (XML-Typ: Qualified Name) zu parsen. In ELaX werden QNames durch Strings, also von Anführungszeichen umgebenen Zeichenketten, dargestellt. Der dazugehörige Token-Typ wird als `QUOTED_STRING` bezeichnet. Anschließend erfolgt eine Validierung des String-Wertes auf die korrekte Syntax eines QNames.

Da die nachfolgende ID-Klausel optional ist, muss zunächst geprüft werden, ob es sich bei dem nächsten Token um den Typ `ID` handelt. Wenn dies zutrifft, wird dieser Token gelesen und anschließend der Wert für die ID, ebenfalls in Form einer Zeichenkette, eingelesen und dem entsprechenden Knoten im Syntaxbaum zugewiesen.

Abbildung 4.1.: Hauptschleife des Parsers



Zuletzt erfolgt das Einlesen der `InLocator`-Klausel. Hier muss ein `IN`-Token vorhanden sein und anschließend eine Zeichenkette, die der Syntax eines Locators entspricht.

Sofern die tatsächliche Token-Sequenz von der erwarteten abweicht, resultiert dies in einem Syntax-Fehler, womit der Parse-Vorgang beendet wird. Im Quellcode geschieht dies mit dem Werfen einer `SyntaxErrorException`, welche neben der Fehlermeldung selbst auch Positionsangaben zum Fehler beinhaltet. Somit kann auch außerhalb des Parsers festgestellt werden, an welcher Stelle der Syntax-Fehler aufgetreten ist und welcher Token an dieser Stelle erwartet wurde.

Während des Parse-Vorgangs wird nach erfolgreicher Validierung der einzelnen Token ein abstrakter Syntaxbaum aufgebaut. Dabei handelt es sich um eine Datenstruktur, welche die ELaX-Anweisung repräsentiert. Dabei wird für einzelne Token oder eine Gruppe von Token ein Knoten im Syntaxbaum angelegt, sofern dieser in der ELaX-Anweisung auftaucht.

Wie in Abbildung 4.3 zu sehen ist, entspricht der Wurzelknoten dem gesamten ELaX-Script, also der vollständigen Eingabe. Die einzelnen Anweisungen (Statements) innerhalb der Eingabe werden durch jeweilige Statement-Knoten dargestellt und bilden somit die direkten Kind-Knoten unterhalb der Wurzel. Je nach Aufbau der einzelnen Statements können diese Knoten weitere Teilbäume enthalten. Nicht vorhandene Klauseln oder Werte werden durch einen `null`-Wert als solche dargestellt.

4.4. DOM-Manipulation

Um nun tatsächlich mit Hilfe des aufgebauten Syntaxbaums das XML-Schema manipulieren zu können, muss dieses mit einer entsprechenden XML-Bibliothek eingelesen werden. Die hier betrachteten Bibliotheken zur XML-Schema-Manipulation basieren auf SAX (Simple API for XML), also einem Event-basierten Parser.

Für die Bearbeitung von XML-Schema-Definitionen wurden einige Java-Bibliotheken in der Tabelle 4.2 betrachtet. Dies sind unter anderem XSOM, Apache XML Schema und Xerces-J.

Da ELaX Teile der XML-Schema-Definitionen in der Version 1.1 voraussetzt, sind die meisten dieser Bibliotheken nicht geeignet. Darüber hinaus erfordert ELaX die Ausführung von XPath-Ausdrücken, welche von diesen Bibliotheken nicht unterstützt wird. Standardkonforme Implementierungen sind nur für die DOM-Schnittstelle vorhanden, weshalb beschlossen wurde, eine eigene XML-Schema-API zu implementieren, welche intern auf dem DOM aufbaut und somit XPath-Ausdrücke unterstützt, gleichzeitig jedoch keine beliebige DOM-Manipulation zulässt, sondern nur auf das XML-Schema bezogene. Diese Vorgehensweise ist auch als Fassade (Facade Pattern) [10] bekannt, da es eine komplexe API (die auf die DOM-Struktur) hinter einer einfacheren, auf den Anwendungsfall zugeschnittene, API (XML-Schema) versteckt.

Für diese Implementierung wird JDOM verwendet, welches im Gegensatz zur standardisierten DOM-Schnittstelle Java-spezifische Datentypen verwendet.

Die Umsetzung dieser Komponente für die XML-Schema-Manipulation ist ebenfalls ein zusätzlicher Aufwand, welcher zu Beginn der Arbeit nicht abzusehen war.

Tabelle 4.2.: Bibliotheken zur Manipulation von XML-Schemata

Bibliothek	**Bemerkungen**
XSOM	• API für XML-Schema vorhanden • Schema kann nur eingelesen und geschrieben werden, aber nicht manipuliert
Apache XML Schema, Apache XML Schema 2	• API für XML-Schema 1.0 vorhanden • keine Unterstützung für XML-Schema 1.1
Xerces-J	• Unterstützung für XML-Schema 1.0 vorhanden • Unterstützung für XML-Schema 1.1 nur in Beta-Version vorhanden • Schema kann nur gelesen, aber nicht manipuliert werden
JBOSS WS XML-Schema	• erweitert Xerces-J API, um XML-Schema auch manipulieren zu können • Aufbau eines eigenen, vom DOM-unabhängigen, Baums • keine eigene XPath-Unterstützung • keine Referenz von DOM-Knoten auf Knoten des JBOSS-Baums
JDOM 2	• erweitert DOM-API mit Java-spezifischen Datentypen • für die Verarbeitung von allgemeinen XML-Dokumenten geeignet • keine spezialisierte API für XML-Schema

Abbildung 4.2.: Entscheidungsbaum für das Parsen einer "add attributegroupref"-Anweisung

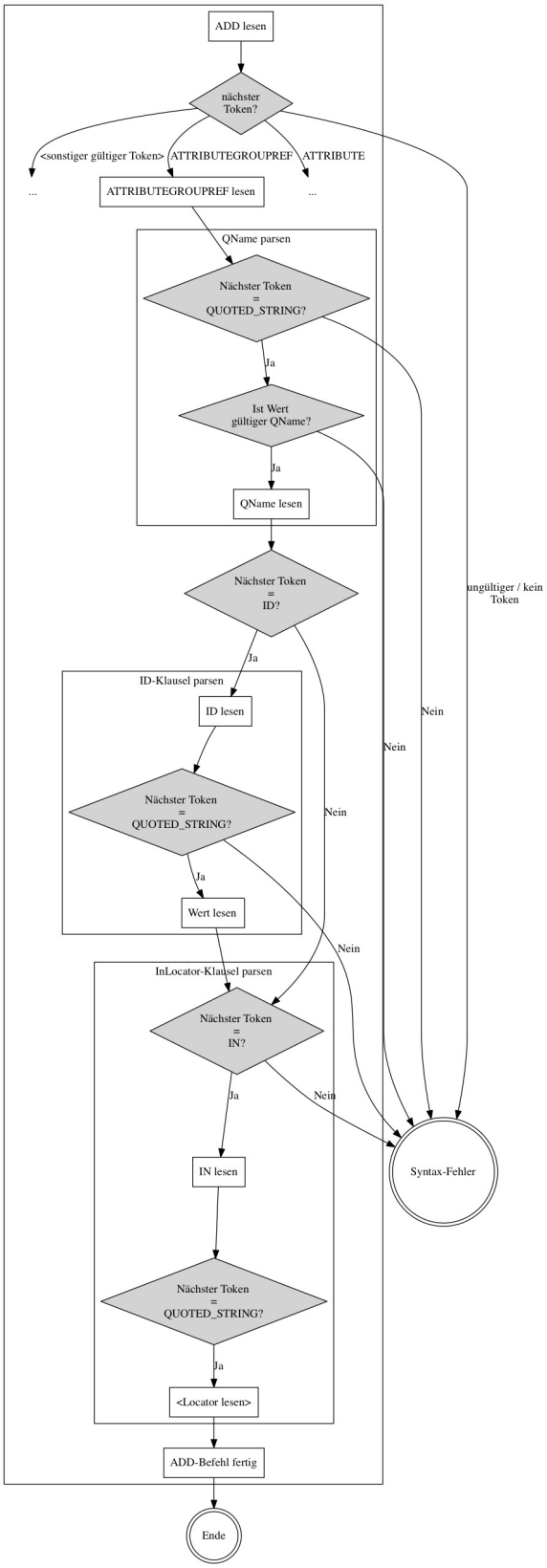
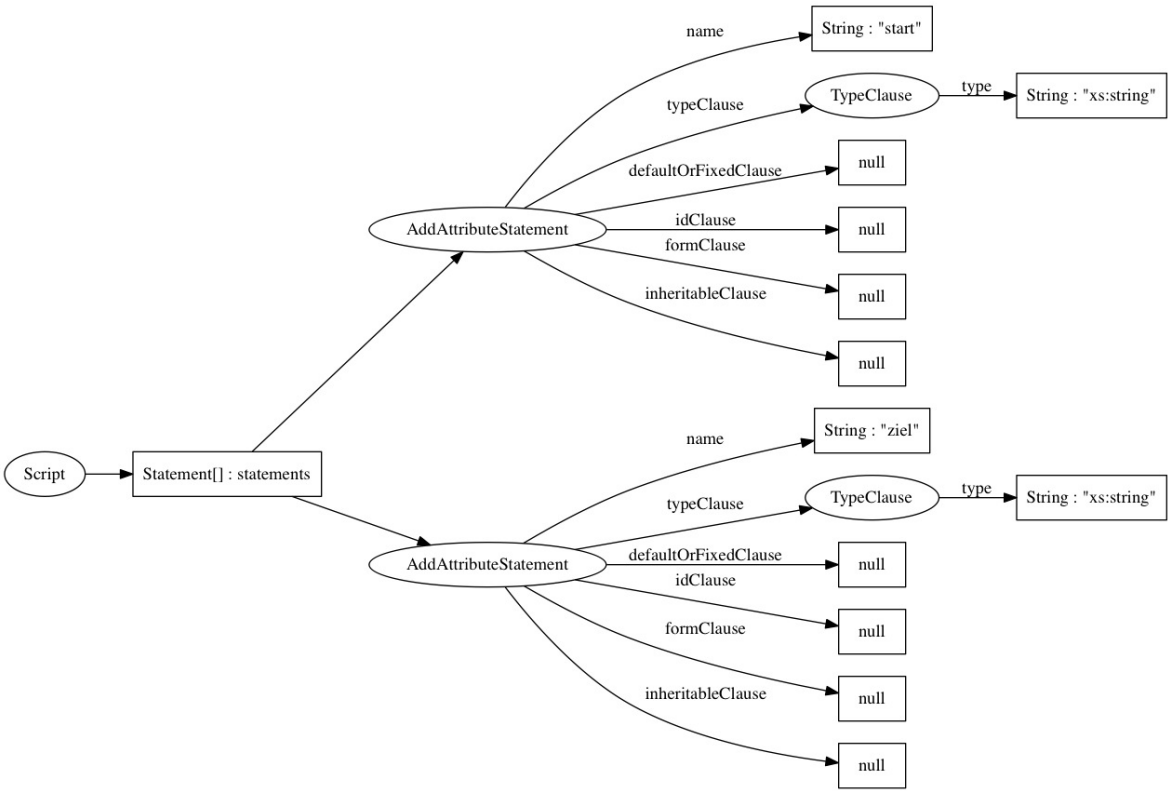


Abbildung 4.3.: Syntax-Baum mit zwei Anweisungen



5. Evaluierung

In diesem Kapitel werden die Ergebnisse dieser Studienarbeit ausgewertet. Es wird darauf eingegangen, welche angestrebten Ziele erreicht wurden, welche nicht erreicht werden konnten und welche Probleme während der Ausarbeitung aufgetreten sind.

5.1. CLI-Anwendung

Es konnte eine lauffähige Anwendung entwickelt werden, welche in der Lage ist, ein XML-Schema zu laden, ELaX-Anweisungen darauf auszuführen und das geänderte Schema zu speichern. Die Pfade für die Ein- und Ausgabedateien für das Schema können über optionale Parameter angegeben werden. Die Auswahl der auszuführenden ELaX-Anweisungen erfolgt über die Standardeingabe. Die Anweisungen werden in Token aufgelöst und geparst und das Schema entsprechend manipuliert.

Um dem Benutzer zusätzliche Rückmeldung über das Ergebnis der Ausführung geben, können Debug-Ausgaben mit dem Parameter `-v` bzw. `--verbose` aktiviert werden. Es werden Informationen über den Lex- und Parse-Vorgang, Fehlermeldungen oder Veränderungen am Schema ausgegeben.

5.1.1. Lexer- und Parserausgaben

Wenn eine ELaX-Anweisung ausgeführt wird, können Zwischenergebnisse des Lexers bzw. des Parsers als zusätzliche Information ausgegeben werden. Dies kann bei der Fehlersuche in ELaX-Anweisungen hilfreich sein. Die Ausgabe für das Hinzufügen von zwei Attributen ist in Listing 5.1 dargestellt.

Listing 5.1: Info-Ausgaben von Parser und Lexer

```
INFO: Execute ELaX: add attribute name "start" type "xs:string";
add attribute name "ziel" type "xs:string";
FINER: Scanned 14 tokens
FINER: Parse AddAttributeStatement
FINER: Parse AddAttributeStatement
FINE: Parsed 2 statement(s)
```

Wie darin zu erkennen ist, wurde die Eingabe in 14 Token aufgelöst, also sieben Token je Anweisung. Der Parser hat die Eingabe zweimal auf AddAttribute-Anweisungen geparst, das Ergebnis sind zwei Anweisungen insgesamt.

5.1.2. Syntax-Fehler

Enthält die ELaX-Eingabe einen Syntax-Fehler, so wirtft der Parser eine entsprechende `SyntaxErrorException`. In der CLI-Anwendung wird diese Exception gefangen und in Form

einer formatierten Fehlermeldung ausgegeben. Es werden Informationen zum erwarteten Token-Typ, dem tatsächlich vorgefundenen Token-Typ und der Fehlerposition ausgegeben. Die Fehlerausgabe zu einem aufgetretenen Syntax-Fehler ist in Abbildung 5.2 dargestellt.

Listing 5.2: Ausgabe eines Syntax-Fehlers

```
Syntax Error in line 2, column 26: Expected TYPE, but found xtype (
   UNKNOWN)
ELaX : add attribute name "ziel" xtype "xs:string";
Error: --------------------------^
```

Bei der Eingabe handelte es sich zwei Anweisungen, getrennt von einem Zeilenumbruch. Der Fehler trat in der zweiten Zeile an Position 26 auf. Es wurde der Token-Typ "TYPE" erwartet, aber ein unbekannter Token-Typ ("UNKNOWN") vorgefunden.

5.1.3. Schema- und Logikfehler

Neben der Korrektheit der Syntax werden auch weitere Prüfungen bei der Ausführung vorgenommen, wie z.B. das Vorhandensein von Definitionen bei einer "add"-Operation. Das Beispiel in Abbildung 5.3 zeigt das Ergebnis einer Ausführung von "add attribute"-Anweisungen, wenn selbige Attribute bereits existieren.

Listing 5.3: Ausgabe eines Logik-Fehlers

```
WARNING: ELAX ERROR: Cannot add attribute 'start', because it already
   exists. Use UPDATE ATTRIBUTE in order to change it.
WARNING: ELAX ERROR: Cannot add attribute 'ziel', because it already
   exists. Use UPDATE ATTRIBUTE in order to change it.
```

Die Attribute "start" und "ziel" sind bereits global im Schema definiert. Deshalb ist eine erneute Definition nicht möglich. Es wird vorgeschlagen, diese Attribute mittels der "update attribute"-Anweisung zu verändern.

5.1.4. Erfolgreiche Ausführung

Bei einer erfolgreichen Ausführung wird neben den Lexer- und Parserergebnissen zusätzlich die Veränderung des Schemas ausgegeben. Es werden alle hinzugefügten bzw. entfernten Zeilen ausgegeben. Abbildung 5.4 zeigt die Ausgabe, nachdem zwei Attribute erfolgreich zum Schema hinzugefügt wurden.

Listing 5.4: Darstellung der Schema-Veränderungen

```
Executed successfully

--- old: line 184, 0 line(s)
+++ new: line 184, 2 line(s)

+   <xs:attribute name="start" type="xs:string" />
+   <xs:attribute name="ziel" type="xs:string" />
```

Wie diese Ausgabe verdeutlicht, wurden zwei neue Zeilen zum Schema ab Zeile 184 hinzugefügt. Die entfernten Zeilen (in diesem Fall nicht vorhanden) werden mit einem - markiert, die neu hinzugefügten Zeilen mit einem vorangestellten +. Zeilen, die geändert wurden,

bspw. bei einer Änderung des Datentyps an einem vorhandenen Attribut, werden durch eine gelöschte und eine hingefügte Zeile dargestellt.

5.2. Unit-Tests

Um die funktionale Korrektheit der Anwendung zu gewährleisten, werden möglichst viele Teile der Anwendungen mit Hilfe von Unit-Tests getestet. Dabei werden kleine Komponenten und einzelne Methoden mit Tests abgedeckt, um Fehler eingrenzen und schneller beheben zu können. Dadurch lässt sich ebenfalls die Korrektheit einzelner Komponenten sicherstellen, wenn andere Komponenten einer Änderung (Code Refactoring) unterzogen wurden. Unterstützend werden Integration-Tests verwendet, um die Gesamtfunktionalität der Anwendung stichprobenhaft zu testen. Beispielsweise kann so ein XML-Schema durch einen ELaX-Ausdruck manipuliert und das Ergebnis automatisiert mit dem erwarteten verglichen werden.

6. Zusammenfassung

In dieser Studienarbeit konnte eine CLI-Anwendung entwickelt werden, die in der Lage ist, die Schema-Update-Sprache ELaX (Evolution Language for XML) [12] auf bestehende XML-Schemata anzuwenden. Für die Umsetzung dieser Anwendung konnte auf Methoden aus dem Bereich Compilerbau zurückgegriffen werden, um die Sprachanalyse durchzuführen. Ebenfalls konnten Implementierungen anderer Spezifikationen wie SAX (Simple API for XML oder DOM (Document Object Model) dazu verwendet werden, die eigentlichen Veränderungen am XML-Schema vorzunehmen.

6.1. Ausblick

Die in dieser Studienarbeit entwickelte Architektur erlaubt es, den Sprachumfang von ELaX (Evolution Language for XML) [12] zu erweitern oder weitere Features im Bereich der Schema-Manipulation umzusetzen. Beispielsweise wäre das Loggen von ausgeführten Schema-Änderungen möglich.

Durch die Implementierung eines weiteren Visitors [10] wäre auch die Umkehrung einer ELaX-Operation denkbar.

Literatur

[1] *ANTLR*. URL: http://www.antlr.org/.

[2] Andrew W. Appel. *Modern Compiler Implementation in Java*. Second Edition. Cambridge University Press, 2004.

[3] *DOM*. URL: http://www.w3.org/TR/2014/WD-dom-20140204/ (besucht am 16. 03. 2014).

[4] Guy Lapalme. *DTD*. URL: http://www.iro.umontreal.ca/~lapalme/ForestInsteadOfTheTrees/HTML/ch03s01.html (besucht am 24. 03. 2014).

[5] Eve Maler. „Schema Design Rules for UBL...and Maybe for You“. In: *XML 2002 Proceedings by deepX*. 2002. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.199.5993&rep=rep1&type=pdf.

[6] Thomas Nösinger. *ELaX Syntax*. URL: http://www.ls-dbis.de/elax (besucht am 18. 03. 2014).

[7] *RELAX NG*. URL: http://relaxng.org/ (besucht am 24. 03. 2014).

[8] *SAX Project*. URL: http://www.saxproject.org/ (besucht am 04. 03. 2014).

[9] *Schematron*. URL: http://xml.ascc.net/resource/schematron/ (besucht am 24. 03. 2014).

[10] Stephan Schmidt. *PHP Design Patterns*. 2. Auflage. O'Reilly Germany, 2009.

[11] *StAX*. URL: http://de.wikipedia.org/wiki/Streaming_API_for_XML (besucht am 16. 03. 2014).

[12] Andreas Heuer Thomas Nösinger Meike Klettke. *XML Schema Transformations - The ELaX Approach*. Techn. Ber. CS-02-13. Institut für Informatik, Universität Rostock, 2013. URL: http://dbis.informatik.uni-rostock.de/fileadmin/dbis/files/Thomas/xml-schema-tansformations.pdf.

[13] *XML 1.0*. URL: http://www.w3.org/TR/2008/REC-xml-20081126/ (besucht am 02. 03. 2014).

[14] *XML-Schema 1.1 Data Types*. URL: http://www.w3.org/TR/2012/REC-xmlschema11-2-20120405/ (besucht am 04. 03. 2014).

[15] *XML-Schema 1.1 Structures*. URL: http://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/ (besucht am 04. 03. 2014).

[16] *XQuery 1.0*. URL: http://www.w3.org/TR/2010/REC-xquery/20101214/ (besucht am 16. 03. 2014).

[17] *XSLT 2.0*. URL: http://www.w3.org/TR/2007/REC-xslt20-20070123/ (besucht am 04. 03. 2014).

Abbildungsverzeichnis

Tabellenverzeichnis

A. Anhang

A.1. ELaX-Syntax

```
1  elax ::= ((<add> | <delete> | <update>) ";")+ ;
2
3  Operations for identification and/or localisation of nodes in XML-
       Schema:
4
5  position ::= ("after" | "before" | ("as" ("first" | "last") "into") |
       "in") <locator> ;
6  refposition ::= (("first" | "last" | "all" | ("at" "position" INT)) "
       in" <xpathexpr>) | emxid ;
7  locator ::= <xpathexpr> | emxid ;
8  xpathexpr ::= ("/" ("." | ("node()" | ("node()[@name='" NCNAME "']"))
       ("[" INT "]")? ) )+ ;
9
10 ADD operation in detail:
11
12 add ::= "add" (<addannotation> | <addattributegroup> | <addgroup> | <
       addst> | <addct> | <addelement> | <addmodule> | <addconstraint>) ;
13 addannotation ::= "annotation" ("appinfo" STRING)? ("documentation"
       STRING)? ("id" ID)? "in" <locator> ;
14 addattributegroup ::= <addattributegroupdef> | <addattribute> | <
       addattributeref> | <addattributegroupref> | <addattributewildcard>
        ;
15 addattributegroupdef ::= "attributegroup" "name" NCNAME ("id" ID)? "
       with" (<addattributeref>)+ (<addattributewildcard>)? ;
16 addattribute ::= "attribute" "name" NCNAME "type" QNAME (("default" |
       "fixed") STRING)? ("id" ID)? ("inheritable" ("true" | "false"))? ;
17 addattributeref ::= "attributeref" QNAME (("default" | "fixed") STRING
       )? ("use" ("prohibited" | "optional" | "required"))? ("id" ID)? "
       in" <locator> ;
18 addattributegroupref ::= "attributegroupref" QNAME ("id" ID)? "in" <
       locator> ;
19 addattributewildcard ::= "anyattribute" ("not" (QNAME | "##defined")+)
       ? ("namespace" ("##any" | "##other" | ("##local" | ANYURI | "##
       targetnamespace")+)? ("not" (ANYURI | ("##targetnamespace" | "##
       local"))+)?)? ("processcontent" ("lax" | "skip" | "strict"))? ("id
       " ID)? "in" <locator> ;
20 addgroup ::= "group" "mode" ("sequence" | ("choice" ("with" <
       groupdefault>)?)  | "all")  ("minoccurs" INT)? ("maxoccurs" STRING
       )? ("id" ID)? "in" <locator> ;
21 groupdefault ::= "first" | "last" | INT ;
22 addst ::= "simpletype" "name" NCNAME ("id" ID)? "mode" (("built-in" |
```

```
    "list") QNAME | "union" (QNAME)+ | "restriction" "of" QNAME "with"
     <facet>+ ) ("final" ("#all" | ("union" | "list" | "restriction" |
     "extension")+ ))? ;
23 facet ::= ( ((("minexclusive" | "mininclusive" | "maxexclusive" | "
    maxinclusive" | "totaldigits" | "fractiondigits" | "length" | "
    minlength" | "maxlength" | "whitespace" | "explicittimezone")
    STRING ("fixed" STRING)?) | (("enumeration"| "pattern") STRING) )
    ("id" ID)?) | <assert> ;
24 assert :: = "assert" STRING (<xpathdefaultnamespace>)? ("id" ID)? ;
25 addct ::= "complextype" "name" NCNAME ("mixed" ("true" | "false"))? ("
    final" ("#all" | "restriction" | "extension"))? ("mode" ("
    extension_cc" | "extension_sc" | "restriction_cc" | ("
    restriction_sc" <facet>*) ) "with" "base" QNAME)? ("id" ID)? ("
    defaultattributesapply" ("true" | "false"))? (<assert>*)? ;
26 addelement ::= <addelementdef> | <addelementref> | <addelementwildcard
    > ;
27 addelementdef ::= "element" "name" NCNAME "type" QNAME (("default" | "
    fixed") STRING)? ("final" ("#all" | "restriction" | "extension"))?
     ("nillable" ("true" | "false"))? ("id" ID)?  ;
28 addelementref ::= "elementref" QNAME ("minoccurs" INT)? ("maxoccurs"
    STRING)? ("id" ID)? <position> ;
29 addelementwildcard ::= "any" ("not" (QNAME | ("##defined" | "##
    definedsibling"))+)? ("namespace" ("##any" | "##other" | ("##local
    " | ANYURI | "##targetnamespace")+)? ("not" (ANYURI | ("##
    targetnamespace" | "##local"))+)?)? ("processcontent" ("lax" | "
    skip" | "strict"))? ("minoccurs" INT)? ("maxoccurs" STRING)? ("id"
     ID)? "in" <locator> ;
30 addmodule ::= "module" "from" ANYURI "mode" (("import" "with" "
    namespace" ANYURI "prefix" STRING) | ("redefine" (<addst> | <addct
    > | <addattributegroupdef>)*) | "include" | ("override" (<addst> |
     <addct> | <addattributegroupdef> | <addelementdef> | <
    addattribute>)*)) ("id" ID)? ;
31 addconstraint ::= "constraint" "name" NCNAME ("id" ID)? "type" ("key"
    | "unique" | ("keyref" "refer" QNAME)) "with" <addconstraintpath>
    "in" <locator> ;
32 addconstraintpath ::= ("selector" <selectorpath> ("|" <selectorpath>)*
     ("id" ID)? (<xpathdefaultnamespace>)?) ("field" <fieldpath> ("id"
     ID)? (<xpathdefaultnamespace>)?)+ ;
33 xpathdefaultnamespace ::= "xpathdefaultnamespace" (ANYURI | ("##
    defaultnamespace" | "##targetnamespace" | "##local")) ;
34 selectorpath ::= (".//")? <constraintstep> ("/" <constraintstep>)* ;
35 fieldpath ::= (".//")? (<constraintstep> "/")* (<constraintstep> | ("@
    " (QNAME | "*" | (NCNAME ":" "*")))) ;
36 constraintstep ::= "." | (QNAME | "*" | (NCNAME ":" "*")) ;
37
38 Delete operation in detail:
39
40 delete ::= "delete" (<delannotation> | <delattributegroup> | <delgroup
    > | <delst> | <delct> | <delelement> | <delmodule> | <
    delconstraint>) ;
41 delannotation ::= "annotation" "at" <locator> ;
```

```
42 delattributegroup ::= <delattributegroupdef> | <delattribute> | <
      delattributeref> | <delattributegroupref> | <delattributewildcard>
       ;
43 delattributegroupdef ::= "attributegroup" "name" QNAME ;
44 delattribute ::= "attribute" "name" QNAME ;
45 delattributeref ::= "attributeref" "at" <locator> ;
46 delattributegroupref ::= "attributegroupref" "at" <locator> ;
47 delattributewildcard ::= "anyattribute" "at" <locator> ;
48 delgroup ::= "group" "at" <locator> ;
49 delst ::= "simpletype" "name" QNAME ;
50 delct ::= "complextype" "name" QNAME ;
51 delelement ::= <delelementdef> | <delelementref> | <delelementwildcard
      > ;
52 delelementdef ::= "element" "name" QNAME ;
53 delelementref ::= "elementref" "at" (<locator> | <refposition>) ;
54 delelementwildcard ::= "any" "at" <locator> ;
55 delmodule ::= "module" "at" <locator> ;
56 delconstraint ::= "constraint" "name" QNAME ;
57
58 Update operation in detail:
59
60 update ::= "update" (<updannotation> | <updattributegroup> | <updgroup
      > | <updst> | <updct> | <updelement> | <updmodule> | <
      updconstraint> | <updschema>) ;
61 updschema ::= "schema" "change" ("targetnamespace" ANYURI)? ("
      targetnamespaceprefix" NCNAME)? ("language" NCNAME)? ("version"
      STRING)? ("elementform" ("qualified" | "unqualified"))? ("
      attributeform" ("qualified" | "unqualified"))? "finaldefault" ("#
      all" | ("extension" | "list" | "restriction" | "union")+))? ("id"
      ID)? ("defaultattribute" QNAME)? (<xpathdefaultnamespace>)? ;
62 updannotation ::= "annotation" "at" <locator> "change" ("appinfo"
      STRING)? ("documentation" STRING)? ("id" ID)? ("move" "into" <
      locator>)? ;
63 updattributegroup ::= <updattributegroupdef> | <updattribute> | <
      updattributeref> | <updattributegroupref> | <updattributewildcard>
       ;
64 updattributegroupdef ::= "attributegroup" "name" QNAME "change" ("name
      " NCNAME)? ("id" ID)? ("add" <addattributeref>)* ("delete" <
      delattributeref>)* (("add" <addattributewildcard>) | ("delete" <
      delattributewildcard>))? ;
65 updattribute ::= "attribute" "name" QNAME "change" ("name" NCNAME)? ("
      type" QNAME)? (("default" | "fixed") STRING)? ("id" ID)? ("
      inheritable" ("true" | "false"))? ;
66 updattributeref ::= "attributeref" QNAME "at" <locator> "change" ("ref
      " QNAME)? (("default" | "fixed") STRING)? ("use" ("prohibited" | "
      optional" | "required"))? ("id" ID)? ("move" "into" <locator>)? ;
67 updattributegroupref ::= "attributegroupref" QNAME "at" <locator> "
      change" ("ref" QNAME)? ("id" ID)? ("move" "into" <locator>)? ;
68 updattributewildcard ::= "anyattribute" "at" <locator> "change" ("not"
       (QNAME | "##defined")+)? ("namespace" ("##any" | "##other" | ("##
      local" | ANYURI | "##targetnamespace")+)? ("not" (ANYURI | ("##
```

```
    targetnamespace" | "##local"))+)?)? ("processcontent" ("lax" | "
    skip" | "strict"))? ("id" ID)? ;
69  updgroup ::= "group" "at" <locator> "change" ("mode" ("sequence" | ("
    choice" ("with" <groupdefault>)?)  | "all"))? ("minoccurs" INT)?
    ("maxoccurs" STRING)? ("id" ID)? ;
70  updst ::= "simpletype" "name" QNAME "change" ("name" NCNAME)?  ("id"
    ID)?  ("mode" (("built-in" | "list") QNAME | ("union" ("insert"
    QNAME+)* ("remove" QNAME+)*) | ("restriction" "of" QNAME (("insert
    " | "remove" | "modify") <facet> "at" <locator>)*)))? ("final" ("#
    all" | ("union" | "list" | "restriction" | "extension")+ ))? ;
71  updct ::= "complextype" "name" QNAME "change" ("name" NCNAME)? ("mixed
    " ("true" | "false"))? ("final" ("#all" | "restriction" | "
    extension"))? ("mode" ("extension_cc" | "extension_sc" | "
    restriction_cc" | ("restriction_sc" (("insert" | "remove" | "
    modify") <facet> "at" <locator>)* ) ) "with" "base" QNAME)? ("id"
    ID)? ("defaultattributesapply" ("true" | "false"))? ((("insert" |
    "remove" | "modify") <assert> "at" <locator>)*)? ;
72  updelement ::= <updelementdef> | <updelementref> | <updelementwildcard
    > ;
73  updelementdef ::= "element" "name" QNAME "change" ("name" NCNAME)? ("
    type" QNAME)? (("default" | "fixed") STRING)? ("final" ("#all" | "
    restriction" | "extension"))? ("nillable" ("true" | "false"))? ("
    id" ID)? ;
74  updelementref ::= "elementref" QNAME (("at" <locator>) | <refposition
    >) "change" ("ref" QNAME)? ("minoccurs" INT)? ("maxoccurs" STRING)
    ? ("id" ID)? ("move" "to" <position>)? ;
75  updelementwildcard ::= "any" "at" <locator> "change" ("not" (QNAME |
    ("##defined" | "##definedsibling"))+)? ("namespace" ("##any" | "##
    other" | ("##local" | ANYURI | "##targetnamespace")+)? ("not" (
    ANYURI | ("##targetnamespace" | "##local"))+)?)? ("processcontent"
     ("lax" | "skip" | "strict"))? ("minoccurs" INT)? ("maxoccurs"
    STRING)? ("id" ID)? ;
76  updmodule ::= "module" "at" <locator> "change" ("from" ANYURI)? ("mode
    " (("import" "with" "namespace" ANYURI "prefix" STRING) | "
    redefine" | "include" | "override"))? ("id" ID)? ;
77  updconstraint ::= "constraint" "name" QNAME "at" <locator> "change" ("
    name" NCNAME)?  ("id" ID)?  ("type" ("key" | "unique" | ("keyref"
    "refer" QNAME)))? (("insert" | "remove" | "modify") <
    updconstraintpath>)* ("move" "into" <locator>)? ;
78  updconstraintpath ::= ("selector" <selectorpath> ("id" ID)?  (<
    xpathdefaultnamespace>)?)? ("field" <fieldpath> ("id" ID)?  (<
    xpathdefaultnamespace>)? "at" <locator>)* ;
```

A.2. ELaX-Syntax nach Ersetzung häufig genutzter Klauseln

```
elax ::= ((<add> | <delete> | <update>) ";")+ ;

Operations for identification and/or localisation of nodes in XML-
   Schema:

position ::= ("after" | "before" | ("as" ("first" | "last") "into") |
   "in") <locator> ;
refposition ::= (("first" | "last" | "all" | ("at" "position" INT)) "
   in" <xpathexpr>) | emxid ;
locator ::= <xpathexpr> | emxid ;
xpathexpr ::= ("/" ("." | ("node()" | ("node()[@name='" NCNAME "']"))
   ("[" INT "]")? ) )+ ;

New Clauses:

AppInfoClause ::= "appinfo" STRING
AtLocatorClause ::= "at" <locator>
AttributeFormClause ::= "attributeform" ("qualified" | "unqualified")
DefaultOrFixedClause ::= ("default" | "fixed") STRING
DocumentationClause ::= "documentation" STRING
IdClause ::= "id" ID
InheritableClause ::= "inheritable" ("true" | "false")
InLocatorClause ::= "in" <locator>
MaxOccursClause ::= "maxoccurs" STRING
MinOccursClause ::= "minoccurs" INT
MixedClause ::= "mixed" ("true" | "false")
MoveIntoLocatorClause ::= "move" "into" <locator>
NameNcNameClause ::= "name" NCNAME
NameQNameClause ::= "name" QNAME
ProcessContentClause ::= "processcontent" ("lax" | "skip" | "strict")
RefClause ::= "ref" QNAME
TargetNamespaceClause ::= "targetnamespace" ANYURI
TargetNamespacePrefixClause ::= "targetnamespaceprefix" NCNAME
TypeClause ::= "type" QNAME
UseClause ::= "use" ("prohibited" | "optional" | "required")
VersionClause ::= "version" STRING

ADD operation in detail:

add ::= "add" (<addannotation> | <addattributegroupdef> | <
   addattribute> | <addattributeref> | <addattributegroupref> | <
   addattributewildcard> | <addgroup> | <addst> | <addct> | <
   addelement> | <addmodule> | <addconstraint>) ;
addannotation ::= "annotation" AppInfoClause? DocumentationClause?
   IdClause? InLocatorClause ;
addattributegroupdef ::= "attributegroup" NameNcNameClause IdClause? "
   with" (<addattributeref>)+ (<addattributewildcard>)? ;
addattribute ::= "attribute" NameNcNameClause TypeClause
   DefaultOrFixedClause? IdClause? InheritableClause? ;
```

```
40 addattributeref ::= "attributeref" QNAME DefaultOrFixedClause?
      UseClause? IdClause? InLocatorClause ;
41 addattributegroupref ::= "attributegroupref" QNAME IdClause?
      InLocatorClause ;
42 addattributewildcard ::= "anyattribute" ("not" (QNAME | "##defined")+)
      ? ("namespace" ("##any" | "##other" | ("##local" | ANYURI | "##
      targetnamespace")+)? ("not" (ANYURI | ("##targetnamespace" | "##
      local"))+)?)? ProcessContentClause? IdClause? InLocatorClause ;
43 addgroup ::= "group" "mode" ("sequence" | ("choice" ("with" <
      groupdefault>)?)  | "all")  MinOccursClause? MaxOccursClause?
      IdClause? InLocatorClause ;
44 groupdefault ::= "first" | "last" | INT ;
45 addst ::= "simpletype" NameNcNameClause IdClause? "mode" (("built-in"
      | "list") QNAME | "union" (QNAME)+ | "restriction" "of" QNAME "
      with" <facet>+ ) ("final" ("#all" | ("union" | "list" | "
      restriction" | "extension")+ ))? ;
46 facet ::= ( ((("minexclusive" | "mininclusive" | "maxexclusive" | "
      maxinclusive" | "totaldigits" | "fractiondigits" | "length" | "
      minlength" | "maxlength" | "whitespace" | "explicittimezone")
      STRING ("fixed" STRING)?) | (("enumeration"| "pattern") STRING) )
      IdClause?) | <assert> ;
47 assert :: = "assert" STRING (<xpathdefaultnamespace>)? IdClause? ;
48 addct ::= "complextype" NameNcNameClause MixedClause? ("final" ("#all"
       | "restriction" | "extension"))? ("mode" ("extension_cc" | "
      extension_sc" | "restriction_cc" | ("restriction_sc" <facet>*) ) "
      with" "base" QNAME)? IdClause? ("defaultattributesapply" ("true" |
       "false"))? (<assert>*)? ;
49 addelement ::= <addelementdef> | <addelementref> | <addelementwildcard
      > ;
50 addelementdef ::= "element" NameNcNameClause TypeClause
      DefaultOrFixedClause? ("final" ("#all" | "restriction" | "
      extension"))? ("nillable" ("true" | "false"))? IdClause?  ;
51 addelementref ::= "elementref" QNAME MinOccursClause? MaxOccursClause?
       IdClause? <position> ;
52 addelementwildcard ::= "any" ("not" (QNAME | ("##defined" | "##
      definedsibling"))+)? ("namespace" ("##any" | "##other" | ("##local
      " | ANYURI | "##targetnamespace")+)? ("not" (ANYURI | ("##
      targetnamespace" | "##local"))+)?)? ProcessContentClause?
      MinOccursClause? MaxOccursClause? IdClause? InLocatorClause ;
53 addmodule ::= "module" "from" ANYURI "mode" (("import" "with" "
      namespace" ANYURI "prefix" STRING) | ("redefine" (<addst> | <addct
      > | <addattributegroupdef>)*) | "include" | ("override" (<addst> |
       <addct> | <addattributegroupdef> | <addelementdef> | <
      addattribute>)*)) IdClause? ;
54 addconstraint ::= "constraint" NameNcNameClause IdClause? "type" ("key
      " | "unique" | ("keyref" "refer" QNAME)) "with" <addconstraintpath
      > InLocatorClause ;
55 addconstraintpath ::= ("selector" <selectorpath> ("|" <selectorpath>)*
       IdClause? (<xpathdefaultnamespace>)?) ("field" <fieldpath>
      IdClause? (<xpathdefaultnamespace>)?)+ ;
56 xpathdefaultnamespace ::= "xpathdefaultnamespace" (ANYURI | ("##
```

```
   defaultnamespace" | "##targetnamespace" | "##local")) ;
selectorpath ::= (".//")? <constraintstep> ("/" <constraintstep>)* ;
fieldpath ::= (".//")? (<constraintstep> "/")* (<constraintstep> | ("@
   " (QNAME | "*" | (NCNAME ":" "*")))) ;
constraintstep ::= "." | (QNAME | "*" | (NCNAME ":" "*")) ;

Delete operation in detail:

delete ::= "delete" (<delannotation> | <delattributegroupdef> | <
   delattribute> | <delattributeref> | <delattributegroupref> | <
   delattributewildcard> | <delgroup> | <delst> | <delct> | <
   delelementdef> | <delelementref> | <delelementwildcard> | <
   delmodule> | <delconstraint>) ;
delannotation ::= "annotation" AtLocatorClause ;
delattributegroupdef ::= "attributegroup" NameQNameClause ;
delattribute ::= "attribute" NameQNameClause ;
delattributeref ::= "attributeref" AtLocatorClause ;
delattributegroupref ::= "attributegroupref" AtLocatorClause ;
delattributewildcard ::= "anyattribute" AtLocatorClause ;
delgroup ::= "group" AtLocatorClause ;
delst ::= "simpletype" NameQNameClause  ;
delct ::= "complextype" NameQNameClause ;
delelementdef ::= "element" NameQNameClause ;
delelementref ::= "elementref" "at" (<locator> | <refposition>) ;
delelementwildcard ::= "any" AtLocatorClause ;
delmodule ::= "module" AtLocatorClause ;
delconstraint ::= "constraint" NameQNameClause ;

Update operation in detail:

update ::= "update" (<updannotation> | <updattributegroupdef> | <
   updattribute> | <updattributeref> | <updattributegroupref> | <
   updattributewildcard> | <updgroup> | <updst> | <updct> | <
   updelementdef> | <updelementref> | <updelementwildcard> | <
   updmodule> | <updconstraint> | <updschema>) ;
updschema ::= "schema" "change" TargetNamespaceClause?
   TargetNamespacePrefixClause? ("language" NCNAME)? VersionClause?
   ("elementform" ("qualified" | "unqualified"))? AttributeFormClause
   ? "finaldefault" ("#all" | ("extension" | "list" | "restriction" |
    "union")+))? IdClause? ("defaultattribute" QNAME)? (<
   xpathdefaultnamespace>)? ;
updannotation ::= "annotation" AtLocatorClause "change" AppInfoClause?
    DocumentationClause? IdClause? MoveIntoLocatorClause? ;
updattributegroupdef ::= "attributegroup" NameQNameClause "change"
   NameNcNameClause? IdClause? ("add" <addattributeref>)* ("delete" <
   delattributeref>)* (("add" <addattributewildcard>) | ("delete" <
   delattributewildcard>))? ;
updattribute ::= "attribute" NameQNameClause "change" NameNcNameClause
   ? TypeClause? DefaultOrFixedClause? IdClause? InheritableClause? ;
updattributeref ::= "attributeref" QNAME AtLocatorClause "change"
   RefClause? DefaultOrFixedClause? UseClause? IdClause?
```

```
    MoveIntoLocatorClause? ;
87  updattributegroupref ::= "attributegroupref" QNAME AtLocatorClause "
    change" RefClause? IdClause? MoveIntoLocatorClause? ;
88  updattributewildcard ::= "anyattribute" AtLocatorClause "change" ("not
    " (QNAME | "##defined")+)? ("namespace" ("##any" | "##other" |
    ("##local" | ANYURI | "##targetnamespace")+)? ("not" (ANYURI |
    ("##targetnamespace" | "##local"))+)?)? ProcessContentClause?
    IdClause? ;
89  updgroup ::= "group" AtLocatorClause "change" ("mode" ("sequence" | ("
    choice" ("with" <groupdefault>)?)  | "all"))? MinOccursClause?
    MaxOccursClause? IdClause? ;
90  updst ::= "simpletype" NameQNameClause "change" NameNcNameClause?
    IdClause?  ("mode" (("built-in" | "list") QNAME | ("union" ("
    insert" QNAME+)* ("remove" QNAME+)*) | ("restriction" "of" QNAME
    (("insert" | "remove" | "modify") <facet> AtLocatorClause)*)))? ("
    final" ("#all" | ("union" | "list" | "restriction" | "extension")+
     ))? ;
91  updct ::= "complextype" NameQNameClause "change" NameNcNameClause?
    MixedClause? ("final" ("#all" | "restriction" | "extension"))? ("
    mode" ("extension_cc" | "extension_sc" | "restriction_cc" | ("
    restriction_sc" (("insert" | "remove" | "modify") <facet>
    AtLocatorClause)* ) ) "with" "base" QNAME)? IdClause? ("
    defaultattributesapply" ("true" | "false"))? ((("insert" | "remove
    " | "modify") <assert> AtLocatorClause)*)? ;
92  updelementdef ::= "element" NameQNameClause "change" NameNcNameClause?
     TypeClause? DefaultOrFixedClause? ("final" ("#all" | "restriction
    " | "extension"))? ("nillable" ("true" | "false"))? IdClause? ;
93  updelementref ::= "elementref" QNAME (AtLocatorClause | <refposition>)
     "change" RefClause? MinOccursClause? MaxOccursClause? IdClause?
    ("move" "to" <position>)? ;
94  updelementwildcard ::= "any" AtLocatorClause "change" ("not" (QNAME |
    ("##defined" | "##definedsibling"))+)? ("namespace" ("##any" | "##
    other" | ("##local" | ANYURI | "##targetnamespace")+)? ("not" (
    ANYURI | ("##targetnamespace" | "##local"))+)?)?
    ProcessContentClause? MinOccursClause? MaxOccursClause? IdClause?
    ;
95  updmodule ::= "module" AtLocatorClause "change" ("from" ANYURI)? ("
    mode" (("import" "with" "namespace" ANYURI "prefix" STRING) | "
    redefine" | "include" | "override"))? IdClause? ;
96  updconstraint ::= "constraint" NameQNameClause AtLocatorClause "change
    " NameNcNameClause?  IdClause?  ("type" ("key" | "unique" | ("
    keyref" "refer" QNAME)))? (("insert" | "remove" | "modify") <
    updconstraintpath>)* MoveIntoLocatorClause? ;
97  updconstraintpath ::= ("selector" <selectorpath> IdClause?  (<
    xpathdefaultnamespace>)?)? ("field" <fieldpath> IdClause?  (<
    xpathdefaultnamespace>)? AtLocatorClause)* ;
```

A.3. Projektbaum

```
de.unirostock.dbis.elax - konkrete ELaX-Implementierungen
    ElaxCli - Hauptanwendung, um ELaX-Anweisungen auszuführen und auf
       XML-Schemata anzuwenden
    Lexer - Lexer, dem Regeln zur Zerlegung von ELaX-Token bekannt
       sind
    Parser - Parser, der ELaX-Tokenfolgen parsen kann
    SchemaManipulator - führt Komponenten (Lexer, Parser, Executor)
       zusammen
    TokenGuesser - TokenGuesser, dem ELaX-Token-Definitionen bekannt
       sind
    TokenType - Aufzählung (Enum) aller bekannten ELaX-Token-Typen

de.unirostock.dbis.elax.common - allgemeine, abstrahierte Klassen mit
   Basisfunktionalitäten
    Lexer - Allgemeiner Lexer mit Basisfunktionalität, um Eingaben in
       Token zu zerlegen oder Tokentypen zu prüfen
    Parser - Allgemeiner Parser mit Basisfunktionialität, um z.B.
       Syntaxfehler zu erzeugen
    SyntaxErrorException - Exception für Syntax-Fehler
    Token - Token-Klasse
    TokenGuesser - Allgemeiner TokenGuesser mit Basisfunktionalität,
       um Token-Typen zu bestimmen

de.unirostock.dbis.elax.dom - enthält Fassaden für XML-Schema-
   Manipulation
    Annotation
    AttributeDeclaration
    AttributeGroupDefinition
    AttributeGroupReference
    AttributeReference
    ComplexType
    Constraint
    Document
    ElementDefinition
    ElementReference
    Group
    Module
    Node
    NodeFactory
    Schema
    SchemaException - Exception für Logikfehler auf Schema-Ebene
    SimpleType

de.unirostock.dbis.elax.matcher - Matcher-Klassen, um Token zu
   erkennen
    DontCareMatcher - Matcher, welcher unabhängig vom Token-Wert immer
        den selben Wert zurückgibt (zu Testzwecken)
    RegexTokenMatcher - Matcher, der TRUE zurückgibt, wenn Token-Wert
       mit Regulärem Ausdruck übereinstimmt
```

```
    StringTokenMatcher - Matcher, der TRUE zurückgibt, wenn Token-Wert
        mit String übereinstimmt
    TokenMatcher - TokenMatcher-Interface

de.unirostock.dbis.elax.parser.AST - enthält Knoten, wie Statements
   oder Klauseln, für den Syntaxbaum
    ASTException - Exception für Fehler im Syntaxbaum
    AddAnnotationStatement
    AddAttributeGroupDefinitionStatement
    AddAttributeGroupReferenceStatement
    AddAttributeReferenceStatement
    AddAttributeStatement
    AddAttributeWildcardStatement
    AddComplexTypeStatement
    AddConstraintStatement
    ...
    IdClause
    InheritableClause
    KeyTypeClause
    LocatorClause
    MaxOccursClause
    MinOccursClause
    MixedClause
    ...

de.unirostock.dbis.elax.parser.AST.visitor - Visitor-Implementierungen
   , um Syntaxbäume zu verarbeiten
    AbstractVisitor
    ExecutionException - Exception für Fehler während der Ausführung
       von ELaX-Anweisungen
    ExecutionVisitor - Visitor für die Ausführung von ELaX-Anweisungen
    NodeVisitor - Visitor-Interface
    VisitorException - allgemeine Visitor-Exception
```

A.4. Verwendete Java-Bibliotheken

Tabelle A.1.: Verwendete Java-Bibliotheken

Bibliothek	**Beschreibung und Verwendung**
Apache Commons Lang 3.1	Erweiterung der Java-Standard-Bibliothek, die häufig benötigte Hilfsmethoden bereitstellt, wie z.B. `StringUtils.join(Object[] array, String separator)` für die Zusammenführung von Array-Elementen oder `StringUtils.repeat(String str, int repeat)` für die Wiederholung von Zeichenketten.
Apache Commons CLI 1.2	Bei dieser Bibliothek handelt es sich um einen CommandLine-Parser, also ein Werkzeug zur einfacheren Verarbeitung von Argumenten und Optionen einer CLI-Anwendung. Dieses wird in der Hauptanwendung verwendet, um die Pfade für die Ein- und Ausgabedatei einzulesen oder den Debug-Modus (`-v`) zu aktivieren.
DiffUtils 1.2.1	Diese Bibliothek findet ihren Einsatz in der Hauptanwendung, um nach erfolgreicher Ausführung von ELaX-Anweisungen die resultierende Veränderung (Diff) des XML-Schemas darzustellen.
Jaxen	Jaxen wird für die XPath-Komponente von JDOM benötigt.
JDOM 2.0.5	JDOM ist eine Java-optimierte Implementierung der standardisierten DOM-Schnittstelle. Dies wird für die Manipulation des XML-Schemas verwendet.
JUnit 4.10	Ein Java Unit-Testing-Framework für das Testen einzelner Komponenten (Unit Tests) bzw. dem Zusammenspiel mehrerer Komponenten (Integration Tests).
XML-Unit 1.4	Framework zum Testen und Vergleichen von XML-Dokumenten. Anstatt die serialisierte Textform von XML-Dokumenten zu vergleichen, kann hiermit die eigentliche Struktur verglichen werden. Abweichende Zeilenumbrüche oder Einrückungen durch nichtdruckbare Zeichen werden somit vernachlässigt.

Selbständigkeitserklärung

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

Rostock, den 31. März 2014