Fachhochschule Wedel (University of Applied Sciences Wedel)

Thesis im Studiengang Master of Sciences (MSc)

Design und Entwicklung eines Relax NG Schema Validators auf Basis der Haskell XML Toolbox

01.09.2005

Eingereicht von: Torben Kuseler
E-Mail: thesis@kuseler.de

Betreuer: Prof. Dr. Uwe Schmidt
Fachhochschule Wedel
Feldstrasse 143
D-22880 Wedel
E-Mail: si@fh-wedel.de

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Relax NG ist eine XML Schema Sprache, mit der sowohl die strukturelle als auch die inhaltliche Gültigkeit von XML-Instanzdokumenten überprüft werden können. Ein Relax NG Schema ist ebenfalls ein XML-Dokument und definiert über die Kombination von benannten Mustern (Pattern), welchen formalen Aufbau eine zulässige Instanz besitzen kann. Ein XML-Prozessor prüft anhand eines Schemas, ob ein zu validierendes Dokument der geforderten Struktur entspricht und somit gültig ist.

XML wird in immer mehr wirtschaftlichen, technischen und wissenschaftlichen Arbeitsgebieten zum Austausch und Speichern von Informationen aus den verschiedensten Aufgabenbereichen eingesetzt. Die Frage, ob die erhaltenen beziehungsweise gesicherten Daten in einem korrekten und für die weitere Verarbeitung geeigneten Format vorliegen, ist dabei von entscheidender Bedeutung für den erfolgreichen Einsatz von XML.
Mit Hilfe der Validierung von XML-Dokumenten gegen eine XML Schema Sprache, wie zum Beispiel Relax NG, kann diese wichtige Frage bereits zu Beginn der Verarbeitung durch eine zentrale Instanz für alle folgenden Prozesse beantwortet werden. Es ist nach einer erfolgreichen Überprüfung des XML-Dokumentes mit Hilfe des Schemas keine individuelle Kontrolle der Eingangsdaten durch die einzelnen Verarbeitungsprozesse mehr notwendig.
Der Aufwand für die Entwicklung sowie die Komplexität der Verarbeitungssoftware wird im erheblichen Maße reduziert. Zudem kann auf eventuell später notwendige Änderungen innerhalb der festgelegten Datenstruktur flexibler reagiert werden. Es ist für die Datenprüfung lediglich eine Anpassung in der Schema Definition notwendig. Die Modifikationen müssen nicht in sämtlichen beteiligten Komponenten nachgeführt werden.

Das Ziel dieser Arbeit besteht in der Analyse der Relax NG Schema Sprache sowie in der Konzeption und Umsetzung eines Moduls zum Validieren eines Relax NG Schemas gegen ein XML-Instanzdokument in der funktionalen Programmiersprache Haskell.
Technische Grundlage des praktischen Anteils der Arbeit ist die Haskell XML Toolbox [Toolbox 02], die bereits eine Reihe von Funktionen für die Bearbeitung und Transformation von XML zur Verfügung stellt. Sie ist ein Projekt an der Fachhochschule Wedel unter der Leitung von Prof. Dr. Schmidt.
Die Toolbox umfasst in der aktuellen Version einen XML-Parser sowie ein Modul zum Validieren von Instanzen gegen eine Dokument Typ Definition (DTD). Außerdem ist das Verändern, Hinzufügen und Löschen von bestimmten Teilen eines XML-Dokumentes durch die Basisfunktionen und Filter der Toolbox möglich. Über das enthaltene HXPath Modul können zudem beliebige Elemente des Dokumentes über einen XPath-Ausdruck selektiert und ausgegeben werden.

Als normative Referenz und formale Basis für die Umsetzung der Relax NG Schema Sprache in ein Modul für die Haskell XML Toolbox dient die aktuell gültige Relax NG Committee Specification vom 3. Dezember 2001 [Relax 01], die unter dem Dach der Organization for the Advancement of Structured Information Standards (OASIS) veröffentlicht worden ist.

Der schriftliche Teil der Arbeit gliedert sich in die folgenden Abschnitte.

Kapitel zwei legt zu Beginn die Grundlagen in den Bereichen XML und XML Schema Sprachen, die für das Verständnis der weiteren Teile benötigt werden. Zusätzlich erfolgt eine Betrachtung von Datentyp-Bibliotheken und ihrer grundsätzlichen Arbeitsweise.

Die Basis für die Validierung eines XML-Instanzdokumentes gegen ein Relax NG Schema ist eine normalisierte Form der Relax NG Grammatik. Kapitel drei beschreibt die in der Spezifikation festgelegten Umformungen und zeigt dabei auf, wie diese innerhalb des implementierten Moduls umgesetzt worden sind. Am Ende der Transformationen liegt jedes Schema, dass durch den Anwender in der vollständigen Grammatik definiert wurde und bezüglich der Relax NG Spezifikation gültig ist, in der normalisierten Syntax vor.

Die normalisierte Form bildet die Grundlage für den Pattern-Datentyp, dessen Struktur und Generierung in Kapitel vier betrachtet wird. Alle Transformationen des vorherigen Kapitels basieren auf dem Haskell XML Toolbox Datentypen xmlTree. Dieser allgemeine Datentyp für die Darstellung von XML-Dokumenten wird in einen, auf den Validierungsalgorithmus zugeschnittenen, Relax NG Pattern-Datentyp überführt.

Das Behandeln von möglichen Fehlern inklusive dem Erstellen entsprechender Fehlermeldungen, wird im Kapitel fünf thematisiert. Es erfolgt somit ein Trennung in die durchzuführenden Transformationen (Kapitel drei) sowie die Generierung der Datenstrukturen (Kapitel vier) einerseits und der notwendigen Fehlerbehandlung andererseits. Das Ziel der gewählten Aufteilung besteht darin, die zugrunde liegenden Algorithmen einfacher und genauer beschreiben zu können. Um diese klare Abgrenzung zwischen den beiden Teile zu erreichen, wird bis zu diesem Kapitel davon ausgegangen, dass die betrachteten Relax NG Schemata gültig und fehlerfrei sind.

Im Anschluss daran erfolgt innerhalb von Kapitel sechs eine Analyse des Algorithmus zum Validieren des erzeugten Datentyps gegen ein Instanzdokument. Zu Beginn wird auf die von J. Brzozowski entwickelte Technik zum Lösen des „regular expression matching“-Problems eingegangen. Danach wird herausgearbeitet, wie diese Technik zum Validieren eines Relax NG Schemas genutzt werden kann.

Datentyp-Bibliotheken dienen zum Kennzeichnen des möglichen Inhalts von Instanzdokumenten. Ihre Integration in das Relax NG Modul der Haskell XML Toolbox, inklusive der Beschreibung der notwendigen Schnittstellen, erfolgt in Kapitel sieben.

Das vorletzte Kapitel stellt die Organisation sowie die verschiedenen Einstiegspunkte in den entwickelten Relax NG Validator vor. Zudem erfolgt eine Erläuterung der einsetzbaren Aufrufparameter, mit denen das Verhalten des Validators zusätzlich beeinflusst werden kann.

Abschließend werden in Kapitel neun die Schlussbetrachtungen zu dieser Arbeit vorgenommen. Dabei wird auf die generellen, während der Implementation des Moduls aufgetretenen Probleme und die daraus entwickelten Lösungsstrategien eingegangen. Außerdem erfolgt ein Ausblick auf zusätzliche Einsatzmöglichkeiten und denkbare Erweiterungen, die sich aus dieser Arbeit ergeben.

2 Grundlagen

Dieses Kapitel legt die Grundlagen für das Verständnis der folgenden Abschnitte. Im ersten Teil werden die für diese Arbeit relevanten Aspekte der Extensible Markup Language, abgekürzt XML, dargestellt. Zudem werden weitere, innerhalb vom XML verwendete Konzepte, wie beispielsweise Namensräume, betrachtet.
Im Anschluss daran erfolgt eine Analyse von XML Schema Sprachen. Es wird kurz auf die erste entwickelte Schema Sprache, die Dokument Typ Definiton (DTD), eingegangen. Der Hauptteil des Abschnittes befasst sich anschließend mit der W3C Schema Sprache und Relax NG.
Ziel ist es jedoch nicht, eine Referenz für die Schema Sprachen abzubilden. Es sollen vielmehr die konzeptionellen Unterschiede zwischen ihnen dargelegt sowie die wichtigsten Sprachelemente erläutert werden. Für die vollständige Darstellung des jeweiligen Sprachumfanges wird auf die Spezifikationen [W3C Schema 04] und [Relax 01] verwiesen.
Abgeschlossen wird das Kapitel durch die Betrachtung von Datentypen und Datentyp-Bibliotheken, die benötigt werden, um den Inhalt von XML-Dokumenten zu prüfen.

Hinweis:

> Die dargestellten Code-Beispiel sind auf die wesentlichen Aspekte gekürzt und können nicht direkt in einem Programm eingesetzt werden. Die Ziffern auf der linken Seite der Beispiele dienen als Zeilennummern und werden zur Erläuterung der Beispiele eingesetzt. Sie besitzen darüber hinaus keine inhaltliche Bedeutung. In den erklärenden Texten wird die entsprechende wichtige Zeile durch die Angabe der Zeilennummer in Klammern, zum Beispiel (3) für die dritte Zeile, deutlich gemacht. Die Beschreibungen beziehen sich immer auf das nachfolgende Beispiel.

2.1 XML

Die Extensible Markup Language [XML 04] bietet eine einfache, textbasierte und sehr flexibel einsetzbare Möglichkeit der strukturierten Darstellung von Daten. Sie bildet eine Teilmenge von SGML [SGML 95].

```
<?xml version="1.0"?>
<library>
  <book id="b0836217462" available="true">
    <title>Being a Dog Is a Full-Time Job</title>
    <author id="CMS"> <name>C M Schulz</name> </author>
  </book>
</library>
```

Beispiel 2.1: Einfaches XML-Dokument

Die Verbreitung von XML hat seit ihrer Einführung durch das World-Wide-Web-Consortium (W3C) im Februar 1998 deutlich zugenommen. XML stellt heute einen essentiellen Bestandteil vieler Anwendungen aus den unterschiedlichsten Bereichen dar. Hierbei ist das wohl meist genutzte Einsatzgebiet von XML die Darstellung beziehungsweise der Austausch von Informationen, die zum Beispiel aus Datenbanken stammen können.

Die Geschäftspartner im elektronischen Handel (E-Commerce) nutzen XML als standardisiertes Nachrichtenformat für Bestellungen und Abrechnungen einerseits. Es können andererseits natürlich auch Produktbeschreibungen durch XML abgebildet werden.
Aber auch im Multimediabereich und in der Wissenschaft ist eine weite Verbreitung von XML festzustellen. Die SVG-Spezifikation (Scalable Vector Graphics) [SVG 03] setzt XML zur Speicherung der benötigten Vektor-Information ein. Mit SMIL (Synchronized Multimedia Integration Language) [SMIL 98], welche beispielsweise im Real Player integriert ist, werden die eingesetzten Multimediapräsentationen durch XML beschrieben.
Viele (natur-)wissenschaftliche Disziplinen nutzen entweder XML direkt oder aber weitere auf XML basierende Sprachen. Mit der Erweiterung MathML (Mathematical Markup Language) [MathML 03], die ebenfalls durch das W3C entwickelt wurde, können zum Beispiel mathematische Formeln sehr einfach dargestellt werden.

2.1.1 Gültigkeit von XML-Dokumenten

Um XML in den verschiedensten Anwendungsbereichen problemlos einsetzen zu können und dabei eine konfliktfreie Interoperabilität, zum Beispiel während des Datenaustausches, zwischen den konkreten Anwendungsprogrammen zu gewährleisten, sind zwei Anforderungen an XML-Dokumente von besonderer Bedeutung:

1. die Wohlgeformtheit (well-formed)
2. die Gültigkeit (valid)

der Dokumente.

Ein XML-Dokument ist wohlgeformt, wenn sein syntaktischer Aufbau den formalen Ansprüchen von XML genügt. Die wichtigsten Regeln sind hierbei:

- Es muss genau ein Element geben (die Wurzel), welches alle weiteren Elemente enthält.
- Für jedes Element muss ein öffnendes und schließendes Tag vorhanden sein oder das Element muss im Tag explizit als leer gekennzeichnet werden.
- Die Elemente können verschachtelt sein, dürfen sich aber nicht überschneiden.
- Alle Attribute müssen einen Wert haben, der von einfachen oder doppelten Anführungszeichen eingeschlossen ist.
- Elemente dürfen keine zwei Attribute mit demselben Namen besitzen.

Während sich die Wohlgeformtheit nur auf den syntaktischen Aufbau bezieht, betrachtet die Eigenschaft der Gültigkeit zudem die Struktur beziehungsweise den Inhalt eines Dokumentes. Hierbei wird allerdings ein wohlgeformtes Dokument vorausgesetzt, d.h. ein Dokument kann wohlgeformt und nicht gültig sein. Ein gültiges, nicht wohlgeformtes Dokument lässt sich hingegen nicht formulieren.

Ein validierender XML-Prozessor muss diese zwei Anforderungen überprüfen können. Die Wohlgeformtheit eines Dokumentes wird bereits beim Einlesen und Parsen durch die Haskell XML Toolbox kontrolliert. Auch die Gültigkeitsprüfung, für die zusätzlich ein XML Schema notwendig ist (siehe Abschnitt 2.2), kann bereits jetzt durch den Parser erfolgen. Dies geschieht über das von Martin Schmidt entwickelte Modul `Validator` [Schmidt 02], welches allerdings nur die Verarbeitung von DTDs unterstützt.
Diese Arbeit erweitert den XML-Prozessor der Haskell XML Toolbox um die Funktionalität, ein Instanzdokument auch gegen ein Relax NG Schema zu validieren.

2.1.2 Namensräume

Die Einführung von Namensräumen [Namespaces 99] in XML dient in erster Linie der Vermeidung von Namenskonflikten, insbesondere bei der Verwendung zusammengesetzter, meist nicht durch dieselbe Person erstellter, Dokumentdefinitionen. Ein zweites Ziel besteht darin, die Basis für eine leichte Erweiterung von Dokumenten und Schemata zu schaffen. Namensräume waren in der ersten XML Version noch nicht enthalten und wurden der Spezifikation erst später hinzugefügt.

Die Definition eines Default-Namensraumes erfolgt innerhalb des Instanzdokumentes über das `xmlns`-Attribut (2). Dieser Namensraum gilt für alle folgenden Elemente und zwar solange, bis der Default-Namensraum wieder überschrieben wird. Attribute werden durch einen Default-Namensraum nicht betroffen, sie müssen explizit mit einem Namensraum versehen werden. Der Default-Namensraum kann auf zwei Arten überschrieben werden. Zum einen ist die Angabe eines vollständig qualifizierten Namens (lokaler Teil und zugehöriger Namensraum) für ein Element möglich (7). Der redefinierte Namensraum gilt dann entsprechend für alle Kinder des Elementes. Diese Art der Qualifizierung kann ein Dokument aber sehr schnell unhandlich machen, so dass die zweite Variante, der Einsatz von einem Präfix, häufig vorgezogen wird. Zeile 3 bindet den Namensraum `http://example/person` an das Präfix `hr`, welches anschließend verwendet werden kann (8) und die Übersichtlichkeit und Lesbarkeit des Dokumentes deutlich erhöht. Es können dabei beliebig viele Präfixe für unterschiedliche Namensräume definiert werden.

```
1  <?xml version="1.0"?>
2  <library xmlns="http://example.com/library"
3           xmlns:hr="http://example.com/person">
4    <book id="b0836217462" available="true">
5      <title xml:lang="en">Being a Dog Is a Full-Time Job</title>
6      <author id="CMS">
7        <name xmlns="http://example.com/person">C M Schulz</name>
8        <hr:born>1922-11-26</hr:born>
9      </author>
10   </book>
11 </library>
```

Beispiel 2.2: XML-Dokument mit Namensräumen

2.2 XML Schema Sprachen

Für die Prüfung, ob der Inhalt eines Instanzdokumentes gültig ist, bedarf es der zusätzlichen Angabe eines XML Schemas.

Definition[1]**:** Unter einem Schema (griech. von „Gestalt“) versteht man in der Wissenschaft die Darlegung, Zeichnung oder Programmierung eines grundsätzlichen Aufbaues oder Verlaufes.

Im Bereich von XML bedeutet diese Definition, dass mit einem Schema die strukturelle Gliederung eines XML-Dokumentes beschrieben werden kann. Dies schließt einerseits die Beschreibung der erlaubten Elemente und Attribute (Namen und mögliche Wertebereiche) sowie andererseits die Information, wie diese Elemente untereinander verschachtelt werden können, ein.

[1] Definiton nach Wikipedia, die freie Enzyklopädie (http://de.wikipedia.org/wiki/Schema)

Die Angabe des Schemas kann durch eine beliebige Schema Sprache, wie beispielsweise einer Dokument Typ Definition (DTD), einem W3C XML Schema oder Relax NG, erfolgen. Der XML-Prozessor ist mit Hilfe der Schema Informationen in der Lage, ein Instanzdokument zu validieren und dabei zu prüfen, ob das Dokument der geforderten Struktur entspricht und somit Gültigkeit besitzt.
In diesem Zusammenhang muss beachtet werden, dass der gewählte Prozessor die entsprechende Schema Sprache unterstützt und sämtlichen am Austausch des XML-Dokumentes beteiligten Anwendern auch zur Verfügung steht.

2.2.1 Dokument Typ Definition

Die Dokument Typ Definition ist die erste Schema Sprache für XML-Dokumente und wurde als Bestandteil der XML Spezifikation 1.0 vom W3C veröffentlicht.
Das folgende Beispiel zeigt eine DTD, die in der Lage ist, ein Buch mit einem Inhaltsverzeichnis und einer beliebigen Anzahl von Kapiteln darzustellen. Außerdem werden einige Attribute zu verschiedenen Elementen deklariert.

```
<!ELEMENT Online-Buch (Inhaltsverzeichnis, (Kapitel|Verweis)+)>
   <!ATTLIST Online-Buch Titel CDATA #REQUIRED
             Typ (Aufsatz|Nachschlagewerk|Spass) "Spass">
   <!ELEMENT Inhaltsverzeichnis (Kapiteluebersch rift)+>
   <!ELEMENT Kapiteluebersch rift (#PCDATA)>
   <!ELEMENT Kapitel (#PCDATA|Abschnitt|Verweis)*>
   <!ATTLIST Kapitel Nummer ID #IMPLIED>
```

Beispiel 2.3: DTD für den Aufbau eines Buches

Das gewünschte Inhaltsmodell für ein Instanzdokument lässt sich durch eine DTD sehr kurz und kompakt beschreiben, wodurch diese schnell geparst und verarbeitet werden kann. Diese Eigenschaft der DTD wirkt sich vor allem bei der Verarbeitung größerer Dokumente positiv im Verhältnis zu neueren Schema Sprachen aus. Die Definition äquivalenter W3C oder Relax NG Schemata ist etwa drei bis fünf Mal länger. Zudem lässt sich der Einsatz von DTDs durch deren kleinen Sprachumfang schnell erlernen.

Der letztgenannte Vorteil muss aber zugleich auch als großer Nachteil angeführt werden. DTDs fehlen durch den geringen Sprachumfang wichtige Konzepte zum Erweitern und Wiederverwenden erstellter Definitionen. Es ist nicht möglich, DTDs mit Namensräumen zu versehen, da diese erst später in die XML-Spezifikation aufgenommen worden sind. Zudem gibt es kein Vererbungskonzept, so dass bestehende Inhaltsmodelle nur sehr schwierig an erweiterte Anforderungen angepasst werden können.
Mit einer DTD lässt sich die Häufigkeit, mit der ein Element innerhalb eines Dokumentes auftreten darf, nur eingeschränkt beschreiben. Es stehen hierfür lediglich die auch aus den regulären Ausdrücken bekannten Operatoren ?, + und * zur Verfügung. Das Fragezeichen steht für ein optionales Element. Wird das Plus angegeben, muss das Element mindestens einmal vorhanden sein, darf aber beliebig oft wiederholt werden. Elemente mit dem Stern müssen nicht vorhanden sein, können aber ebenfalls beliebig oft angegeben werden. Wird kein Operator spezifiziert, muss das Element genau einmal vorkommen. Soll für ein Instanzdokument festgelegt werden, dass ein Element zwischen drei und fünf Mal enthalten sein darf, ist dieses nur sehr umständlich über die Kombination des Fragezeichens mit einer Wiederholung der Elementdeklaration möglich.
Eine zusätzliche Schwäche ist die nicht XML konforme Syntax von DTDs. Der Anwender ist gezwungen, mit der Erweiterten-Backus-Naur-Form [EBNF] eine zweite Notation zu erlernen. Außerdem sind hierdurch unterschiedliche Werkzeuge für die Verarbeitung von XML-Dokumenten und DTDs notwendig.

2.2.2 W3C XML Schema

Die W3C XML Schema Sprache wurde als Weiterentwicklung der DTDs vom W3C konzipiert und durch eine große Anzahl verschiedener Projekte und Ideen beeinflusst. Hierzu zählen beispielsweise die Document Content Description for XML [DCD 98] oder das Schema for Object-Oriented XML [SOX 99].
Die erste Version der XML Schema Spezifikation wurde vom W3C am 2. Mai 2001 mit dem Status „Recommendation“ versehen; aktuell existiert die Version 2.0 [W3C Schema 04].

Komponenten eines Schemas

Im Gegensatz zu DTDs ist ein W3C XML Schema ein wohlgeformtes XML-Dokument. Es kann somit dieselbe Notation für das Instanzdokument wie für das Schema eingesetzt werden. Zudem werden Namensräume in vollem Umfang durch die Sprache unterstützt.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="book">
    <xs:complexType>
      <xs:element name="title" type="xs:string"/>
      <xs:attribute name="available" type="xs:string"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Beispiel 2.4: Einfaches W3C XML Schema

Ein W3C XML Schema nutzt dieselbe Sichtweise auf die einzelnen Komponenten (`components`) einer XML-Datei (Elemente, Attribute, usw.), die schon innerhalb einer DTD verwendet wurden und setzt diese Technik konsequent fort. Im Fokus der W3C XML Schema Beschreibung befinden sich die Elemente und Attribute zwischen denen eine klare Trennung besteht. Zusätzlich definiert die W3C XML Schema Spezifikation eine Reihe gut voneinander abgegrenzter Bausteine (`distinct components`), die zum Lösen der verschiedenen Aufgaben einer Schema Sprache verwendet werden können.

Elemente und Attribute

Die einfachsten Bausteine innerhalb eines W3C XML Schemas sind `xs:element` und `xs:attribute`, die zum definieren der Elemente beziehungsweise Attribute, die in einem Instanzdokument auftreten dürfen, genutzt werden. Sie besitzen eine Reihe von Attributen, wie beispielsweise `name` oder `type`, durch die exakt festgelegt wird, wie der gültige Aufbau eines Elementes in der Instanz aussieht.
Da eine strikte Trennung zwischen Elementen und Attributen herrscht, ist die Reihenfolge in der die beiden Bausteine verwendet werden, entscheidend. Es müssen erst alle Elemente definiert werden. Anschließend folgen sämtliche Attribute. Eine beliebige Anordnung ist nicht möglich, was zu Einschränkungen bei der Gruppierung von Elementen und Attributen zu größeren Strukturen führt.
Im Beispiel 2.5 entspricht `book1` einer gültigen Definition. Bei der Elementdeklaration `book2` steht das `available`-Attribut vor dem letzten Element. Das Schema ist somit nicht gültig.

```
<xs:element name="book1">
  <xs:element name="title"/>
  <xs:element name="author"/>
  <xs:attribute name="available"/>
</xs:element>
```

```
<xs:element name="book2">
  <xs:element name="title"/>
  <xs:attribute name="available"/>
  <xs:element name="author"/>
</xs:element>
```

Beispiel 2.5: Gültige und ungültige Elementdefinition

Gruppen Kombinatoren

Elemente und Attribute können durch „Gruppen Kombinatoren“, in der W3C Spezifikation `compositors` genannt, zusammengesetzt werden. Das `xs:group`-Element dient hierbei als Container. Das mögliche Auftreten der innerhalb der Gruppe definierten Elemente wird durch einen der drei folgenden Kombinatoren festgelegt.

Kombinator	Aufgabenbeschreibung
xs:sequence	alle Elemente der Gruppe müssen in exakt der vorgegebenen Reihenfolge im Instanzdokument vorkommen
xs:choice	es darf nur eines der deklarierten Kinder in der XML-Instanz auftreten
xs:all	alle Kindelemente können einmal oder gar nicht vorkommen und zwar in beliebiger Reihenfolge

Tabelle 2.1: Gruppen Kombinatoren der W3C XML Schema Sprache

Zu beachten ist allerdings, dass `xs:all` nur auf oberster Ebene eines Inhaltsmodells stehen darf, und als Kinder nur einzelne Elemente und keine Gruppen erlaubt sind. Zudem ist eine Verschachtelung von Elementen und Attributen mit den oben genannten Kombinatoren nicht möglich. Als Kinder von `xs:sequence`, `xs:choice` und `xs:all` können nur Elemente definiert werden. Zum Strukturieren von Attributen ist der Einsatz von `xs:attributeGroup` notwendig, was jedoch nur im Anschluss an die Gruppen Kombinatoren auftreten darf.

Erweiterbarkeit und Wiederverwendbarkeit

Die strikte Trennung zwischen Elementen und Attributen sowie der Position, an der sie definiert werden dürfen, führt bei der Erweiterung und Wiederverwendung von W3C XML Schemata häufig zu erheblichen Problemen.
Über das bereits behandelte Sprachelement `xs:group` lassen sich oft benötigte Strukturen zusammenfassen und an verschiedenen Stellen wiederverwenden. Eine definierte Gruppe erhält dafür über das `name`-Attribut einen Namen (1) und kann anschließend über diesen referenziert (11) werden. Das folgende Beispiel 2.6 ist ohne das in Zeile 5 definierte Attribut `isbn` gültig, da das `age`-Attribute in der `character`-Gruppe (7), genauso wie in dem zusammengesetzten Element `book` (10), hinter allen Elementen steht.
Wird das `isbn`-Attribut zu dem `book-basic`-Element hinzugefügt, ist dieses separat betrachtet immer noch gültig. Allerdings hat die Erweiterung zur Folge, dass innerhalb von `book` (10) das `isbn`-Attribut vor dem `character`-Element steht, so dass das Schema insgesamt ungültig ist.

```
1 <xs:group name="book-basic">
2   <xs:sequence>
3     <xs:element name="title"/> <xs:element name="author"/>
4   </xs:sequence>
5   <xs:attribute name="isbn"/>
```

```
6 </xs:group>
7 <xs:group name="character">
8   <xs:element name="character"/> <xs:attribute name="age"/>
9 </xs:group>
10 <xs:element name="book">
11     <xs:group ref="book-basic"/> <xs:group ref="character"/>
12 </xs:element>
```

Beispiel 2.6: Referenzieren von Schema Definitionen

Zu einem großen Hindernis kann diese Eigenschaft werden, wenn über `<include schemaLocation="http://..."/>` externe Schemata eingebunden werden sollen. Hier ist häufig keine Anpassung des importierten Inhaltsmodells möglich, so dass eine einfache und beliebige Kombination und Erweiterung von bereits definierten Strukturen mit der W3C XML Schema Sprache nicht gewährleistet ist.

Häufigkeitsbeschränkungen

DTDs erlaubten nur eine eingeschränkte Festlegung, wie häufig ein Element in einem Instanzdokument auftreten darf. Diese Häufigkeitseinschränkungen lassen sich mit einem W3C XML Schema sehr viel besser beschreiben. Sie werden mit Hilfe der beiden Attribute `minOccurs` und `maxOccurs` für jedes Element exakt festgelegt. `minOccurs` bildet die untere Grenze, `maxOccurs` entsprechend die obere Grenze der möglichen Anzahl ab. Für beide Attribute darf ein ganzzahliger Wert größer oder gleich Null verwendet werden. Zusätzlich ist für `maxOccurs` die Angabe von `unbounded` (3) möglich. Sie erlaubt, dass das Element beliebig häufig auftreten kann. Zu beachten ist bei der Festlegung, dass der Wert von `minOccurs` nicht größer als `maxOccurs` sein darf. Sind beide Werte identisch, muss das Element exakt so oft im Instanzdokument vorkommen. Fehlt die Angabe eines Attributes, wird der Standardwert „1“ verwendet.

```
1 <xs:complexType name="book">
2   <xs:sequence>
3     <xs:element name="character" minOccurs="0" maxOccurs="unbounded"/>
4     <xs:element name="person" minOccurs="5" maxOccurs="30"/>
5   </xs:sequence>
6 </xs:complexType>
```

Beispiel 2.7: Festlegen von Häufigkeiten in einem W3C Schema

Möchte man die im vorherigen Beispiel 2.7 definierte Anzahl von fünf bis 30 Personen durch eine DTD abbilden, wäre eine umständliche Kombination des Fragezeichens mit einer Wiederholung der `Person`-Definition notwendig.

```
<!ELEMENT book
    (character*,
     person, person, person, person, person, person?,person?,person?,
     person?,person?,person?,person?,person?,person?,person?,person?,
     person?,person?,person?,person?,person?,person?,person?,person?,
     person?,person?,person?,person?,person?,person?) >
```

Beispiel 2.8: Festlegen von Häufigkeiten mit DTDs

2.2.3 Relax NG

Relax NG (Regular Language for XML, New Generation) [Relax 01] entstand aus den beiden Projekten RELAX Core und TREX.
RELAX Core (Regular Language Description for XML) [RelaxCore 00] wurde bis dahin von Murata Makoto entwickelt. Er beschreibt RELAX Core als *„eine einfache Schema Sprache, die auf der Automaten Theorie basiert“*[2].
Parallel zu RELAX Core erstellte James Clark mit TREX (Tree Regular Expressions for XML) [TREX 99] *„eine neue Sprache zum Validieren der Struktur von XML-Dokumenten“*[3].
Diese beiden Ansätze der Entwickler drücken die wichtigsten Eigenschaften von Relax NG aus, ein solides mathematisches Fundament einerseits und das Ziel, die Struktur von XML-Dokumenten zu validieren, andererseits.
Der Zusammenschluss der Ansätze erfolgte 2001 unter dem Dach der OASIS (Organization for the Advancement of Structured Information Standards) mit dem Ziel, eine leistungsfähige Alternative zur W3C XML Schema Sprache zu entwickeln. Relax NG wurde seit seiner Veröffentlichung bereits in einer Vielzahl von Projekten, zum Beispiel dem OASIS DocBook [DocBook 05], erfolgreich eingesetzt. Selbst das W3C nutzt Relax NG in einigen seiner Empfehlungen wie beispielsweise der XHTML 2.0 Spezifikation [XHTML 05].

Benannte Muster

Ein gültiges Relax NG Schema ist ebenfalls ein wohlgeformtes XML-Dokument und wird durch die im Anhang (siehe Abschnitt A.2.1) dargestellte Grammatik beschrieben. Es verwendet im Gegensatz zu der W3C XML Schema Sprache keinen komponentenbasierten Ansatz. Die bei der W3C XML Schema Sprache vorhandene klare Trennung zwischen den einzelnen Bausteinen eines Schemas existiert innerhalb von Relax NG nicht.
Es wird hingegen ein musterbasierter Ansatz verfolgt, der es ermöglicht, ein Relax NG Schema sehr viel flexibler zu entwickeln, zu erweitern und zu kombinieren, als dies mit entsprechenden W3C XML Schemata möglich ist. Alle benötigten Sprachelemente werden über ein identisches Verfahren mit Hilfe von benannten Mustern, so genannten `named pattern`, abgebildet und können anschließend auf dieselbe Art und Weise verarbeitet werden.

Die Pattern `element` und `attribute` bilden die einfachsten Bausteine und repräsentieren ein Element beziehungsweise Attribut im Instanzdokument. Werden mehrere Pattern zu einer größeren Struktur zusammengefasst, ist die Reihenfolge, in der sie im Schema angegeben werden, nicht signifikant. Hierdurch wird eine beliebige Kombination von Relax NG Schemata ermöglicht. Die im Abschnitt 2.2.2 beschriebenen Probleme treten nicht auf.

Gruppen Kombinatoren

Das Gruppieren von einfachen Pattern zu komplexeren Strukturen erfolgt in Relax NG ebenfalls über benannte Muster. Es stehen hierfür, identisch zur W3C Schema Sprache, drei Kombinatoren zur Verfügung, die in der Tabelle 2.2 dargestellt sind.

Eine geordnete Gruppe entspricht dem Standardfall für Relax NG Kombinatoren, so dass das `group`-Muster auch fehlen kann. Das entsprechende Tag `xs:sequence` des W3C XML Schemas muss hingegen vorhanden sein und darf nicht ausgelassen werden.
Die für `xs:all` bestehenden Restriktionen gelten für das `interleave`-Pattern nicht. Es erlaubt nicht nur das Definieren von ungeordneten Gruppen, sondern ermöglicht auch das Verschachteln von Kindern mit Elementen von Untergruppen. Dies ist mit einem W3C XML

[2] Vgl. http://books.xmlschemata.org/relaxng/relax-PREFACE-3.html
[3] Vgl. http://xml.coverpages.org/ni2001-02-16-c.html

Kombinator	Aufgabenbeschreibung
group	dient zur Definition einer sequentiell geordneten Gruppe
choice	es darf nur eines der deklarierten Kinder in der XML-Instanz auftreten
interleave	definiert eine ungeordnete Gruppe, bei der die Reihenfolge der Elemente und Attribute beliebig ist

Tabelle 2.2: Gruppen Kombinatoren von Relax NG

Schema nicht möglich. Außerdem kann das `interleave`-Pattern an einer beliebigen Stelle im Inhaltsmodell eines Elementes stehen und ist nicht auf die oberste Ebene beschränkt.

Erweiterung und Wiederverwendung

Deklarierte Strukturen können mit Hilfe der `define`- und `ref`-Pattern wiederverwendet werden. Das `define` (1) besitzt einen Namen, der durch das `ref`-Pattern (9) referenziert wird. Da die Reihenfolge von Element- und Attribut-Deklarationen in Relax NG nicht signifikant ist, kann die `book-basic` Definition problemlos um das Attribut `isbn` (3) erweitert werden. Dies war mit einem W3C XML Schema nicht möglich. Dieser Vorteil gilt natürlich im Besonderen für das Einbinden von externen Quellen, das in Relax NG über die beiden Muster `externalRef` und `include` erfolgt.

```
1 <define name="book-basic">
2   <element name="title"/> <element name="author"/>
3   <attribute name="isbn"/>
4 </define>
5 <define name="character">
6   <element name="character"/> <attribute name="age"/>
7 </define>
8 <define name="book">
9   <ref name="book-basic"/> <ref name="character"/>
10 </define>
```

Beispiel 2.9: Referenzieren von Schema Definitionen

Häufigkeitsbeschränkungen

Bei der Angabe von Häufigkeitsbeschränkungen besitzt Relax NG nur die Ausdruckskraft von DTDs. Es stehen hierfür lediglich die Muster `optinal`, `zeroOrMore` und `oneOrMore` zur Verfügung, die dem Fragezeichen, dem Stern und dem Plus der regulären Ausdrücke entsprechen. Wird kein Kardinalitätsmuster angegeben, muss das Element exakt einmal auftreten. Eine genauere Abstufung bei der Anzahl der Elemente, wie sie mit der W3C XML Schema Sprache sehr einfach möglich ist, kann unter Relax NG nur mit dem für DTD's beschriebenen Mehraufwand (siehe Abschnitt 2.2.2) erreicht werden.

```
<define name="book-basic">
  <optional> <element name="isbn"/> </optional>
  <element name="title"/>
  <oneOrMore> <element name="author"/> </oneOrMore>
  <zeroOrMore> <element name="character"/> </zeroOrMore>
</define>
```

Beispiel 2.10: Festlegen von Häufigkeiten mit Relax NG

2.3 Datentypen

In den vorherigen Abschnitten ist nur die strukturelle Gültigkeit von Instanzdokumenten betrachtet worden. Das heißt, es wurde lediglich festgelegt, welche Elemente und Attribute innerhalb des Dokumentes vorkommen dürfen und wie deren Anordnung zu erfolgen hat. Eine Beschreibung, welche Werte für die einzelnen Elemente gültig sind, erfolgte noch nicht.
Wichtig wird die Validierung von Werten vor allem bei datenorientierten XML-Dokumenten. Man spricht hierbei von *Element-lastigen* Instanzen[4], wie sie beispielsweise bei der Repräsentation einer Datenbanktabelle durch XML vorkommen.
Diese zentrale Aufgabe wird in XML Schema Sprachen durch Datentypen und Datentyp-Bibliotheken übernommen. Ein einzelner Datentyp definiert dabei einen Bereich, in dem sich der Wert des XML-Elementes des Instanzdokumentes befinden muss. Die deklarierten Datentypen können anschließend zu Datentyp-Bibliotheken zusammengefasst werden.
Durch den Einsatz von Datentypen kann schon während des Validierens eines XML-Dokumentes geprüft werden, ob die enthaltenen Daten innerhalb gültiger Wertebereiche liegen. Hierdurch werden Entwickler von Programmen, die auf den Instanzdokumenten arbeiten, von dieser Aufgabe befreit. Es erfolgt bereits eine zentrale Prüfung durch den XML-Prozessor. Zudem kommt die Daten-Typisierung auch der Integrität der gespeicherten Daten zugute.

DTD Datentypen

DTDs sind hauptsächlich für die Auszeichnung von Text konzipiert worden. Aus diesem Grund war ein einziger Datentyp `String` für die Beschreibung des Inhalts von Dokumenten vollkommen ausreichend. Andere Datentypen stehen innerhalb von DTDs nicht zur Verfügung.

W3C Schema Datentypen

Die durch die zunehmende Verbreitung von XML entstandenen neuen Einsatzgebiete forderten allerdings immer häufiger eine detailliertere Beschreibung der enthaltenen Daten, z.B. von Datums- oder Zahlenwerten, die durch Zeichenketten nur sehr schwierig und umständlich abgebildet werden können.
Aus diesem Grund umfasst die W3C XML Schema Sprache in der aktuellen Version knapp 50 verschiedene Datentypen. 19 von ihnen bilden die Klasse der primitiven eingebauten Datentypen (`build-in primitive types`), zu denen beispielsweise `string, date` oder `decimal` zählen. Alle anderen Typen (`normalizedString, byte, usw.`) sind von den primitiven abgeleitet und bilden die Klasse der `build-in derived types` (siehe Abbildung 2.1).
Die primitiven Datentypen stehen dabei in keiner Beziehung zueinander und bilden komplett unterschiedliche Wertebereiche ab. Die abgeleiteten Typen begrenzen hingegen den Wertebereich eines primitiven Typen, stehen also mit ihm in direkter Beziehung und wurden eingeführt, um häufig genutzte Fälle abzudecken und die Anwenderfreundlichkeit zu erhöhen.

Die eingebauten Datentypen können direkt in einer Element oder Attribut Deklaration (1) verwendet werden. Außerdem ist es möglich, eigene problemspezifische Wertebereiche mit Hilfe von Restriktionen, so genannten Fassetten, festzulegen. Dies kann über die Beschränkung des Wertebereiches von bereits deklarierten Datentypen mit Hilfe der Attribute `xs:minInclusive` und `xs:maxInclusive` (4) erfolgen. Sollen nur wenige, nicht aufeinander folgende Werte zugelassen werden, können diese direkt durch die `xs:enumeration`-Fassette (8) aufgezählt werden. Das ausdrucksstärkste Werkzeug für die Definition eigener Datentypen ist jedoch `xs:pattern` (11), welches eine Beschreibung der erlaubten Daten durch reguläre Ausdrücke[5] ermöglicht.

[4] Instanzen mit viel Fließtext und einigen wenigen formatierten Elementen werden als dokumentorientiert bezeichnet (*Dokument-lastig*).

[5] Vgl: W3C Schema - Regular Expressions (http://www.w3.org/TR/xmlschema-0/#regexAppendix)

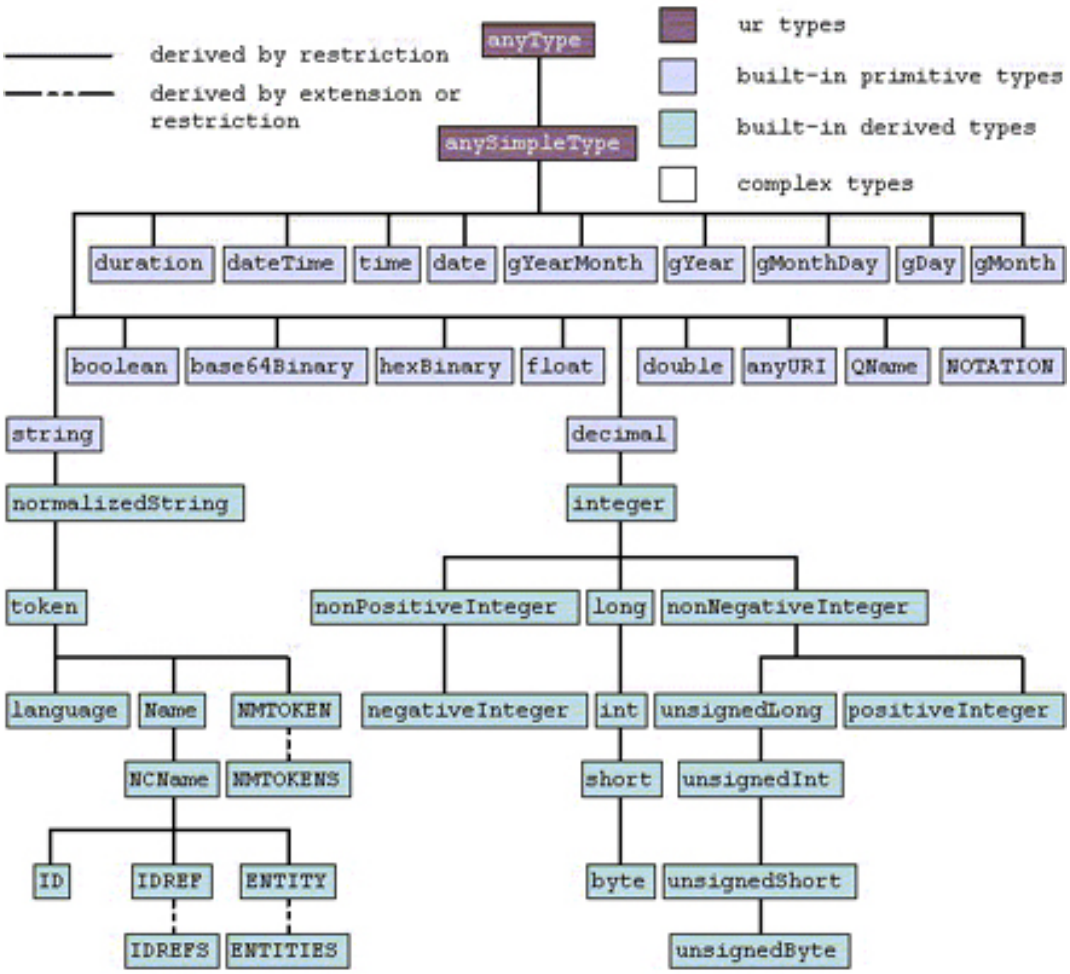

Abbildung 2.1: W3C Schema Datentyp Hierarchie

`xs:pattern` darf allerdings nicht mit dem `pattern`-Begriff, der innerhalb von Relax NG für die verschiedenen Muster verwendet wird, verwechselt werden.

```
1  <xs:element name="isbn" type="xs:unsignedLong"/>
2  <xs:simpleType name="myUnsignedLong">
3    <xs:restriction base="xs:unsignedLong">
4      <xs:minInclusive value="100"/> <xs:maxInclusive value="999"/>
5    </xs:restriction>
6  </xs:simpleType>
7  <xs:restriction base="xs:unsignedLong">
8    <xs:enumeration value="101"/> <xs:enumeration value="264"/>
9  </xs:restriction>
10 <xs:restriction base="xs:string">
11   <xs:pattern value="[0-9]{10}"/>
12 </xs:restriction>
```

Beispiel 2.11: Verwendung von Datentypen in einem W3C Schema

Relax NG Datentypen

Relax NG besitzt ebenfalls ein eingebautes Typsystem. Im Gegensatz zum W3C XML Schema wurde aber beim Design der Sprache entschieden, das System minimal zu halten.

Das eigene Typsystem umfasst lediglich die zwei Datentypen `token` und `string`, die sich nur in der Leerraumzeichenbehandlung voneinander unterscheiden.
Werte vom Typ `token` werden entsprechend der XPath Funktion normalize-space()[6] normalisiert, indem alle Sequenzen von Leerraumzeichen (Leerzeichen, Tab, Zeilenumbruch) durch ein Leerraumzeichen ersetzt und führende sowie abschließende Leerraumzeichen entfernt werden. Für den Datentyp `string` ist diese Normalisierung nicht vorgesehen. Der Zeichenkettenwert bleibt unverändert erhalten.
Die Entscheidung, nur zwei sehr allgemeine Datentypen in die Spezifikation von Relax NG aufzunehmen, kann mit zwei Gründen erklärt werden[7]. Erstens ist Relax NG mit dem Ziel entworfen worden, die Dokumenten-Struktur zu validieren und weniger die Werte innerhalb dieser Struktur. Zweitens waren schon komplette Typ-Bibliotheken vorhanden, so dass es den Entwicklern als nicht sinnvoll erschien, eine weitere Bibliothek für Relax NG zu konzipieren. Der Anwender hat stattdessen die Möglichkeit, verschiedene externe Bibliotheken in seinem Relax NG Schema zu verwenden.

Das Einbinden einer externen Bibliothek erfolgt über das Attribut `datatypeLibrary` (2), welches im Beispiel 2.12 die Datentypen der W3C XML Schema Sprache referenziert. Nach der Angabe können alle Datentypen genauso innerhalb eines Relax NG Schemas verwendet werden, wie es bisher für die eingebauten Typen token und string möglich war. Voraussetzung ist jedoch, dass der eingesetzte Relax NG Validator die Datentyp-Bibliothek unterstützt. Das Attribut `datatypeLibrary` wird dabei innerhalb eines Relax NG Schemas vererbt. Das heißt, der Attributwert gilt auch für alle Kinder und zwar solange, bis dieser wieder überschrieben wird.
Die Angabe eines zu verwendenen Datentyps kann in allen Mustern erfolgen, die den Inhalt eines Dokumentes abbilden. Handelt es sich um einen festen Wert, geschieht dies über das `value`-Muster (4). Ist der Wert variabel, muss das `data`-Pattern verwendet werden. Das `data`-Muster kann somit an Stellen eingesetzt werden, an denen eine Begrenzung des Wertebereiches notwendig ist, die mit dem `text`-Muster nicht erzielt werden kann.

```
1 <element name="book" xmlns="http://relaxng.org/ns/structure/1.0"
2          datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
3   <attribute name="available">
4     <value type="boolean">true</value>
5   </attribute>
6 </element>
```

Beispiel 2.12: Einbinden fremder Datentypen

Die Verwendung der vom W3C Schema bekannten Fassetten kann natürlich auch innerhalb eines Relax NG Schemas erfolgen, wo sie durch das `param`-Muster repräsentiert werden. Das `name`-Attribut bestimmt die gewünschte Fassette. In Zeile 3 wird erst über die `pattern`-Fassette das Format für das Geburtsjahr festgelegt, anschließend verhindert die `minInclusive`-Fassette (4), dass Autoren vor dem 1.1.1900 geboren sein können.

```
1 <element name="author-born">
2   <data type="date">
3     <param name="pattern">[0-9]{4}-[0-9]{2}-[0-9]{2}</param>
4     <param name="minInclusive">1900-01-01</param>
5   </data>
6 </element>
```

Beispiel 2.13: Einschränken von Wertebereichen

[6] Vgl: W3C XPath Spezifikation (http://www.w3.org/TR/xpath#function-normalize-space)
[7] Vgl. http://www.thaiopensource.com/relaxng/design.html#section:10

Die Entscheidung der Relax NG Entwickler, dass das Validieren von Struktur und Inhalt zwei unterschiedliche Probleme darstellen und somit auch durch zwei verschiedene Werkzeuge (in enger Zusammenarbeit) gelöst werden sollten, eröffnet Anwendern eine Vielzahl neuer Möglichkeiten. Sie sind nicht auf die eingebauten Typen beschränkt, sondern können eine beliebige, problemspezifische Typ-Bibliothek in ihre Anwendung integrieren und dafür bereits vorhandene Parser und Validatoren einsetzen.

Wie die Integration einer Typ-Bibliothek in das Relax NG Modul der Haskell XML Toolbox erfolgt, wird detailliert am Beispiel einer Bibliothek für die Abbildung von MySQL-Datentypen [MySQL 05] im Abschnitt Datentyp-Bibliotheken (siehe Kapitel 7) dargestellt.

3 Transformieren eines Relax NG Schemas in die normalisierte Syntax

Ein Schema wird von einem Anwender in der vollständigen Relax NG Schema Syntax angegeben. Bevor eine Validierung gegen ein Instanzdokument erfolgen kann, muss es durch eine Reihe von Transformationen in eine vereinfachte, normalisierte Version (siehe Anhang A.2.2) überführt werden. Die vereinfachte Syntax besteht anschließend nur noch aus einer Teilmenge der vollständigen Syntax.
Ziel der Umformung ist es, eine einheitliche Basis für die weitere Verarbeitung des Schemas zu erreichen. Die im Kapitel 4 beschriebene Umsetzung des Schemas in einen Datentyp für die Validierung gegen ein Instanzdokument wird dadurch deutlich erleichtert.

Dieses Kapitel beschreibt die einzelnen durch die Spezifikation vorgeschriebenen Konvertierungsschritte und zeigt dabei auf, wie sie innerhalb des Haskell Moduls `Simplification` für die Implementation des Relax NG Validators der Haskell XML Toolbox umgesetzt wurden. Zu Beginn jedes Abschnittes wird das jeweilige Ziel der Transformationen kurz dargestellt, gefolgt von einer detaillierteren Betrachtung der einzelnen Komponenten. Abgeschlossen werden die erläuterten Umformungen durch ein konkretes Code-Beispiel.

Hinweis:

> Die Angabe, die sich in Klammern an jede Kurzbeschreibung anschließt, beschreibt den Abschnitt der Relax NG Spezifikation [Relax 01], der gerade bearbeitet wird. die Angabe erfolgt immer in der Form (RS x.y), wobei x.y den Paragraph der Spezifikation angibt.
> Zudem wird in den Haskell Beispielen häufig die Typ-Definition aus Platzgründen als Kommentar hinter der ersten Code-Zeile eingefügt.

Die 21 Transformationsschritte, die in der Relax NG Spezifikation festgelegt sind, wurden zu acht Verarbeitungsfunktionen (`simplificationStep1 ... simplificationStep8`) zusammengefasst, die sequentiell auf das Schema angewendet werden. Die gesamten Umformungen erfolgen dabei auf dem Datenmodell `XmlTree` der Haskell XML Toolbox. Alle, während der Bearbeitung benötigten Zustandsvariablen, werden vorher auf ihren Startwert zurückgesetzt und die Namensraumdeklarationen innerhalb des Schemas propagiert (3). Am Ende erfolgt eine Bereinigung des Schemas (6) von nicht mehr notwendigen Elementen und Informationen.

```
1 createSimpleForm    = seqA $ concat [ simplificationPart1
2                                     , simplificationPart2, finalCleanUp]
3 simplificationPart1 = [ resetStates, propagateNamespaces
4                       , simplificationStep1, ... , simplificationStep4]
5 simplificationPart2 = [ simplificationStep5, ... , simplificationStep8]
6 finalCleanUp        = [ cleanUp ]
```

Beispiel 3.1: Transformationen für die vollständige Syntax

Neben diesen acht Transformationsschritten werden auf das Schema vier weitere Funktionen (`restrictionsStep1 .. restrictionsStep4`) angewendet. Im Gegensatz zu den oben erwähnten Umwandlungen verändern dieses das Schema jedoch nicht. Sie dienen vielmehr dazu, eine Reihe von Einschränkungen, die bei den möglichen Kombinationen der Pattern

bestehen, zu prüfen. Ein Schema ist genau dann gültig, wenn es sämtliche Beschränkungen erfüllt. Da die erste Prüfung nach dem vierten Vereinfachungsschritt durchzuführen ist, sind die Transformationsfunktionen in zwei Blöcke unterteilt worden, in dessen Mitte (1) der Aufruf von `restrictionsPart1` liegt. Die restlichen Kontrollen erfolgen nach dem achten Transformationsschritt (2).

```
1 seqA $ concat [ simplificationPart1, restrictionsPart1, simplificationPart2,
2                 restrictionsPart2, finalCleanUp]
3 restrictionsPart1 = [restrictionsStep1] -- :: [IOSArrow XmlTree XmlTree]
4 restrictionsPart2 = [restrictionsStep2, restrictionsStep3, restrictionsStep4]
```

Beispiel 3.2: Prüfen von Restriktionen

Die Prüfung der Bedingungen ist durch den Aufrufparameter `do-not-check-restrictions` (siehe Abschnitt 8.2) zu deaktivieren. Hierdurch kann ein Performanzgewinn erzielt werden. Dies ist jedoch nur für Schemata zu empfehlen, die bereits als gültig bekannt sind, da anderenfalls die Validierung gegen ein Instanzdokument nicht möglich ist und das Programm mit einer Fehlermeldung abbricht.

Es wird zunächst nur erläutert, welche Kontrollen und Überprüfungen durchgeführt werden müssen. In diesem und dem nächsten Kapitel wird davon ausgegangen, dass die Relax NG Schemata korrekt sind und keine Fehler enthalten. Was im Fehlerfall geschieht und welche Fehlermeldungen zu generieren sind[1], wenn ein Schema nicht korrekt ist, wird im Kapitel Fehlerbehandlung (siehe Abschnitt 5) detailliert betrachtet. An allen relevanten Stellen erfolgt bis dahin ein Querverweis auf die zugehörige Fehlerbehandlung.

Die Gruppierung in die acht Transformations- und vier Prüfabschnitte orientiert sich an der Strukturierung des Buches „RELAX NG“[2] von Eric van der Vlist [Vlist 03]. Aus Performanz-, Übersichts- sowie Implementationsgründen wurde an einigen Stellen von der im Buch dargestellten Gliederung abgewichen. Die exakten Ursachen hierfür werden in den jeweiligen Abschnitten erläutert.

3.1 Anmerkungen, Leerräume und das `datatypeLibrary`-Attribut

Der erste Transformationsschritt umfasst das Entfernen von Anmerkungen (RS 4.1) sowie die Normalisierung von Leerraumzeichen innerhalb von Attributwerten und Textknoten (RS 4.2). Außerdem werden die Attribute `datatypeLibrary` (RS 4.3) und `type` (RS 4.4) in allen `value`- und `data`-Pattern ergänzt, in denen sie bisher noch nicht vorhanden waren. Neben diesen, durch die Relax NG Spezifikation vorgeschriebenen Umformungen, werden weitere vorbereitende Arbeiten für spätere Transformationen durchgeführt.

3.1.1 Entfernen von Anmerkungen

XML-Elemente und Attribute, denen ein Namensraum zugeordnet ist, der nicht dem Relax NG Namensraum[3] entspricht, werden aus dem Dokument entfernt. Sie sind für die weiteren Verarbeitungsschritte des Schemas nicht mehr notwendig. Die `compareURI`-Funktion normalisiert vor dem Vergleich die Namensraum-URIs, indem sie in Kleinbuchstaben umgewandelt werden und ein abschließender Schrägstrich `/`, sofern vorhanden, entfernt wird.

---

[1] in den Code-Beispielen kommt hierfür die Funktion `mkError` als Platzhalter zum Einsatz
[2] Vgl. Kapitel 15: Simplification and Restrictions
[3] http://relaxng.org/ns/structure/1.0

```
none `when`
getNamespaceUri >>> isA (\u -> u /= "" && (not $ compareURI u relaxNS))

compareURI :: String -> String -> Bool
compareURI uri1 uri2 = normalize uri1 == normalize uri2
normalize uri = map toLower (if last uri == '/' then init uri else uri)
```

Beispiel 3.3: Löschen von Anmerkungen

3.1.2 Leerraum Normalisierung

Führende und abschließende Leerraumzeichen[4] werden aus den Werten der Attribute `name`, `type` und `combine` sowie aus dem Inhalt des `name`-Pattern entfernt.

```
(changeAttrValue normalizeWhitespace) `when`
(isAttr >>> hasName "name" `orElse` hasName "type" `orElse` hasName "combine")
(processChildren $ changeText normalizeWhitespace) `when`
(isElem >>> hasName "name")
```

Beispiel 3.4: Leerraum-Normalisierung

Zudem werden Textknoten, die nur Leerraumzeichen enthalten und nicht Kindknoten von `value`- oder `param`-Pattern sind, gelöscht.

```
(processChildren removeWhiteSpace) `whenNot`
(isElem >>> (hasName "param" `orElse` hasName "value"))
```

Beispiel 3.5: Entfernen von Leerraum Knoten

3.1.3 `datatypeLibrary`- und `type`-Attribute

Das `datatypeLibrary`-Attribut[5] wird innerhalb eine Schemas vererbt. Das heißt, jedes `data`- oder `value`-Pattern, das kein eigenes `datatypeLibrary`-Attribut besitzt, erhält den Wert des nächsten Vorfahren mit einem `datatypeLibrary`-Attribut oder einen leeren Attributwert, falls es keinen solchen Vorfahren gibt.
Die Funktion `processdatatypeLib` wird mit der leeren Zeichenkette für den Parameter `lib` initalisiert. Trifft sie beim Durchlaufen des Baumes auf ein Element, welches ein `datatypeLibrary`-Attribute besitzt (3), wird dessen Wert als neuer Parameterwert verwendet. Ein `data`- oder `value`-Pattern erhält ein neues `datatypeLibrary`-Attribut mit dem Wert des `lib`-Parameters, wenn es kein eigenes `datatypeLibrary`-Attribut besitzt (5-7).

```
1 processdatatypeLib lib -- :: (ArrowXml a) => String -> a XmlTree XmlTree
2  = processChildren $ choiceA [
3      (isElem >>> hasAttr "datatypeLibrary")
4           :-> (processdatatypeLib $< getAttrValue "datatypeLibrary"),
5      ( isElem >>> (hasName "data" `orElse` hasName "value") >>>
6        neg (hasAttr "datatypeLibrary"))
7           :-> (addAttr "datatypeLibrary" lib >>> processdatatypeLib lib),
8      this :-> (processdatatypeLib lib)]
```

Beispiel 3.6: Bearbeiten des `datatypeLibrary`-Attributes

[4] Leerzeichen #x20, Tabulator #x9, Zeilenvorschub #xA, Wagenrücklauf #xD
[5] Gleiches gilt für das `ns`-Attribut, welches im Abschnitt 3.3.2 beschrieben wird

Danach werden alle `datatypeLibrary`-Attribute, die nicht zu `data`- oder `value`-Elementen gehören (2), entfernt. Abschließend wird für jedes `value`-Pattern, welches kein `type`-Attribut besitzt (6), ein `type`-Attribut mit dem Wert `token` hinzugefügt und der Wert des `datatypeLibrary`-Attributes auf die leere Zeichenfolge geändert.

```
1 removeAttr "datatypeLibrary" `when`
2 ( isElem >>> neg (hasName "data" `orElse` hasName "value") >>>
3   hasAttr "datatypeLibrary")
4
5 (addAttr "type" "token" >>> addAttr "datatypeLibrary" "") `when`
6 (isElem >>> hasName "value" >>> neg (hasAttr "type"))
```

Beispiel 3.7: Löschen nicht benötigter `datatypeLibrary`-Attribute

3.1.4 Vorbereitende Transformationen für später folgende Schritte

Bevor diese Transformationen jedoch durchgeführt werden können, ist das Sichern einiger Informationen für später folgende Schritte erforderlich.
Nachdem Elemente und Attribute, die nicht dem Relax NG Namensraum angehören, entfernt wurden, stehen keine Angaben zu Namensraumdeklarationen mehr zur Verfügung. Diese sind jedoch für den Kontext von Elementen, für die Verarbeitung von qualifizierten Namen (QNames) und für `href`-Attribute notwendig.

Kontext eines Elementes

Der Kontext eines jeden Elementes besteht aus der Basis-URI des Elementes sowie einer Zuordnung von Präfixen auf Namensräume, die für das betrachtete Element Gültigkeit besitzen. Da der Kontext bei der Validierung eines Schemas gegen ein Instanzdokument jedoch nur für das `value`-Pattern relevant ist, kann er bei allen anderen Elementen ignoriert werden. Er wird notwendig, wenn beispielsweise ein qualifizierter Name im XML-Dokument geprüft werden muss. Dort ist zu berechnen, welcher Namensraum dem Präfix zuzuordnen ist.
Für das `value`-Pattern werden die notwendigen Informationen durch weitere Attribute gesichert. Der Basis-URI wird dem Attribut `RelaxContextBaseURI` (7), der default-Namensraum `RelaxContextDefault` (5) und die Namensraumdeklaration Attributen der Form `RelaxContext:präfix` (2,6) zugeordnet. Der Attributwert enthält jeweils den entsprechenden URI.

```
1 (addAttrl (getBaseURI >>> createAttrL)) `when`(isElem >>> hasName "value")
2 createAttrL = setBaseURI &&& constA (map createAttr env) >>> arr2L (:)
3 createAttr (pre, uri) -- :: (String, String) -> XmlTree
4   = if pre == "" -- default namespace
5     then XN.mkAttr "RelaxContextDefault" [XN.mkText uri]
6     else XN.mkAttr ("RelaxContext:"++pre) [XN.mkText uri]
7 setBaseURI = mkAttr "RelaxContextBaseURI"  (txt $< this)
```

Beispiel 3.8: Sichern des Kontextes eines `value`-Pattern

Das folgende Beispiel 3.9 zeigt oben das Originaldokument sowie darunter das Ergebnis der Transformation. Die Namensraumdeklaration `http://www.w3.org/XML/1998/namespaces` für den Präfix `xml` ist für jedes XML-Element gültig und wird automatisch hinzugefügt[6].

[6] Vgl. http://www.w3.org/TR/REC-xml-names/#ns-using

```
<element name="foo" xmlns="http://relaxng.org/ns/structure/1.0"
                    xmlns:ex="http://example.com" >
  <value type="string" datatypeLibrary=""> bar </value>
</element>

<element name="foo">
  <value type="string" datatypeLibrary=""
         RelaxContextBaseURI="file://.../test.rng"
         RelaxContextDefault="http://relaxng.org/ns/structure/1.0"
         RelaxContext:xml="http://www.w3.org/XML/1998/namespaces"
         RelaxContext:ex="http://example.com"> bar </value>
</element>
```

Beispiel 3.9: Kontext für ein `value`-Pattern

Qualifizierende Namen

Die `element`- und `attribute`-Pattern können ein `name`-Attribut besitzen, dessen Wert festlegt, welche Bezeichnung für einen Elementknoten beziehungsweise für ein Attribut in einem Instanzdokument erlaubt ist. Der Wert beschreibt einen Namen inklusive einem erlaubten Namensraumpräfix. Da es sich jedoch um einen Attribut*wert* und nicht um einen Attribut*namen* handelt, wird der mögliche Präfix nicht durch einen vollständig qualifizierten Namen beim Propagieren der Namensräume ersetzt, sondern bleibt unverändert erhalten.
Im Paragraph RS 4.8 der Relax Spezifikation (siehe Abschnitt 3.3.1) wird das `name`-Attribut durch ein eigenständiges `name`-Pattern ausgetauscht. Der darauf folgende Schritt RS 4.10 ersetzt den eventuell vorhandenen Präfix durch den entsprechenden URI (siehe Abschnitt 3.3.3). Um diese Ersetzung nach dem Löschen der Namensrauminformationen zu gewährleisten, findet die Abbildung des Präfixes auf den Namensraum bereits in dem Attributwert des `element`- und `attribute`-Pattern statt. Ein Wert der Form `pre:localName` mit der Abbildung des Präfixes `pre` auf den Namensraum `http://www.abc.com` wird durch `{http://www.abc.com}localName` ersetzt. Existiert innerhalb des Kontextes des Pattern keine gültige Abbildung des Präfix auf einen Namensraum resultiert dies in einem Fehler (siehe 5.1.2).
Die Grafik 3.1 zeigt den Ablauf der Ersetzungen mit den entstehenden Zwischenergebnissen.

Absolute Form von `href`-Attributen

Externe Schemata (siehe Abschnitt 3.2), die in das aktuelle Schema importiert werden sollen, werden über das `href`-Attribut von `externalRef`- beziehungsweise `include`-Pattern lokalisiert. Hierfür sind im Attributwert nicht erlaubte Zeichen[7] durch ihre Ersatzsequenzen zu substituieren (5). Anschließend wird der URI in seine absolute Form transformiert[8]. Die absolute Form berechnet sich aus dem für das Element gültigen Basis-URI und dem Wert des `href`-Attributes (10). Der Basis-URI kann in jedem Element über das Attribut `xml:base` verändert werden (3) und gilt für alle nachfolgenden Elemente. Befindet sich das `xml:base`-Attribut in einem `externalRef`- oder `include`-Pattern (4) wird erst der neue Basis-URI erreichnet und danach der `href`-Wert verändert (6). Anderenfalls wird nur der neue Wert der Basis-URI bestimmt (8).

[7] Vgl. http://www.w3.org/TR/xlink/#link-locators
[8] Vgl. http://www.ietf.org/rfc/rfc2396.txt, Abschnitt 5.2

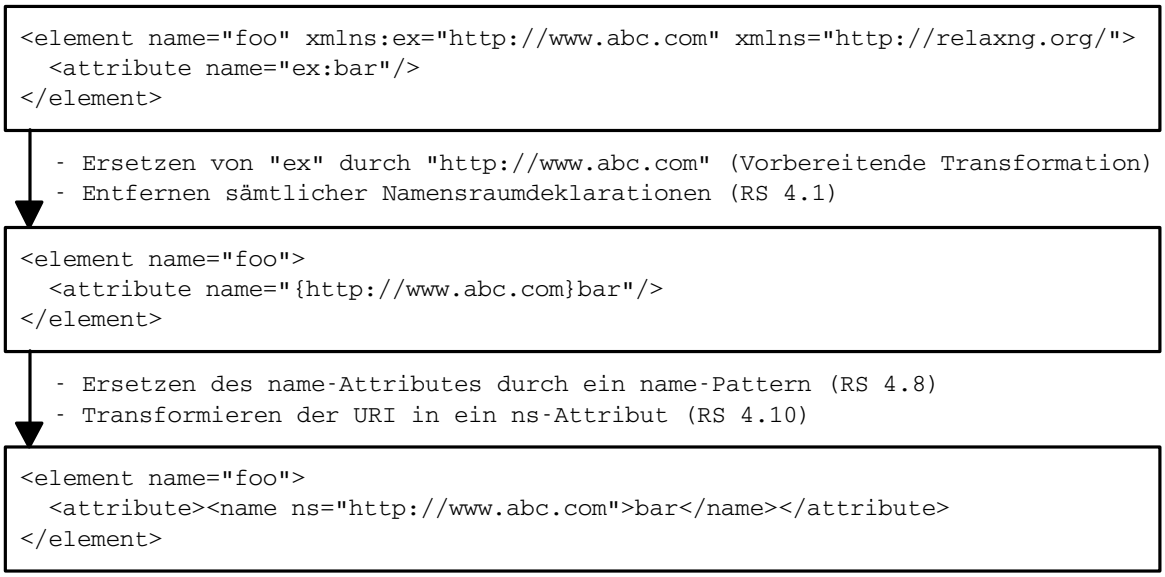

Abbildung 3.1: Verarbeitung von qualifizierten Namen

```
1  processHref uri -- :: String -> IOSArrow XmlTree XmlTree
2   = processChildren $ choiceA [
3      (isElem >>> hasAttr "xml:base") :->
4        (ifA (checkElemName ["externalRef", "include"] >>> hasAttr "href")
5          ( (processAttrl (changeAttrValue escapeURI `when` hasName "href"))
6            >>> (addAttr "href" $< (absURI "href" $< (absURI "xml:base" uri)))
7            >>> (processHref $< absURI "xml:base" uri))
8          (processHref $< absURI "xml:base" uri)),
9      (checkElemName ["externalRef", "include"] >>> hasAttr "href")
10          :-> (addAttr "href" $< absURI "href" uri),
11     this :-> (processHref uri)]
```

Beispiel 3.10: Bestimmen der absoluten Form von `href`-Attributen

3.2 Externe Schemata

Die zweite Transformationsgruppe umfasst das Importieren von externen Schemata. Hierfür stehen die Pattern `externalRef` (RS 4.6) und `include` (RS 4.7) zur Verfügung. Dazu gehört inhaltlich das Verarbeiten der `href`-Attribute (RS 4.5) der beiden Pattern, welches im Buch von [Vlist 03] auch diesem Schritt zugeordnet wurde. Da hierfür jedoch Informationen benötigt werden, die nach den ersten Umformungen nicht mehr vorhanden sind, wurden die `href`-Attribute bereits im vorherigen Abschnitt in ihre absolute Form transformiert.

Einfügen von Pattern durch `externalRef`

Das `externalRef`-Pattern erlaubt sehr einfach, eine Strukturierung und Aufteilung größerer Relax NG Schemata in einzelne Dateien vorzunehmen. Das `externalRef`-Pattern verweisst dabei auf eine Quelldatei, die durch den Relax NG Prozessor eingelesen wird und das Pattern vollständig ersetzt. Im Gegensatz zum `include`-Pattern werden keine weiteren Kombinationen zwischen dem importierten Schema und dem `externalRef` vorgenommen.
Die Funktion `readDocument` liest das, über das `href`-Attribut referenzierte, externe Dokument ein (8). Es kann sich dabei um eine lokale Datei handeln. Ebenso ist aber auch eine Ressource

aus dem Internet möglich, die über das `http`-Protokoll verarbeitet wird. Danach werden auf das eingelesene Dokument alle bisher durchgeführten Transformationen angewendet, so dass das importierte Schema in seiner Normalisierungsstufe identisch zu dem aktuellen Schema ist (8,9). Abschließend wird der Namensraum das aktuellen Schemas auf des neue Schema übertragen, falls dieses kein eigenes `ns`-Attribut besitzt (10).
Die zu importierende Quelle muss der Syntax für ein `pattern` entsprechen (siehe Anhang A.2.1). Diese Prüfung erfolgt durch das Relax NG Modul der Haskell XML Toolbox gegen das Relax Grammatik Schema über die Funktionen `createSpezification` und `validateXMLDoc` (siehe Abschnitt 8.2). Liefern diese einen Fehler, das heißt das zu importierende Schema entspricht nicht der Syntax eines Patterns, wird das Einlesen nicht durchgeführt (7). Stattdessen wird eine entsprechende Fehlermeldung generiert[9]. Das Validieren von externen Quellen kann aus Performanzgründen abgeschaltet werden (5). Zum Deaktivieren stehen die Aufrufparameter `do-not-validate-externalRef` und `do-not-validate-include` zur Verfügung (siehe Abschnitt 8.2).

```
1  (importExternalRef $<< (getAttrValue "ns" &&& getAttrValue "href"))
2  `when` (isElem >>> hasName "externalRef")
3
4  importExternalRef ns href -- :: String -> String -> IOSArrow XmlTree XmlTree
5    = ifA ( validateExternal `guards`
6            (createSpezification >>> validateXMLDoc href))
7        mkError
8        ( readDocument [] href >>> simplificationStep1 >>>
9          simplificationStep2 >>> getChildren >>>
10         ( addAttr "ns" ns `when`
11           (getAttrValue "ns" >>> isA (\a -> a == "" && ns /= ""))))
```

Beispiel 3.11: Einfügen von Schemata durch das `externalRef`-Pattern

Importieren von `grammar`-Elementen durch das `include`-Pattern

Eine Alternative, externe Schemata einzubinden, bietet das `include`-Pattern. Es erlaubt dem Anwender, detaillierter festzulegen, welche Elemente des neuen Schemas eingefügt werden sollen, als dies mit einem `externalRef`-Pattern möglich ist.
Das Einlesen des Dokumentes unterscheidet sich dabei nur unwesentlich von der `externalRef` Verarbeitung. Es werden alle bisherigen Transformationen auf das neue Schema angewendet (6,7). Zudem findet ebenfalls eine Validierung statt, die über die bereits erwähnten Aufrufparameter deaktiviert werden kann (3,4). Im Gegensatz zu `externalRef` muss das neue Dokument aber der Syntax für ein `grammar`-Element entsprechen. Diese Prüfung kann jedoch nicht direkt gegen das Relax NG Grammatik Schema erfolgen, da es als Startelement ein beliebiges Pattern zulässt. Stattdessen erfolgt die Validierung gegen die Schema-Datei `RelaxNG-Schema-grammar.rng` (4).

```
1  (importInclude $< getAttrValue "href") `when` (isElem >>> hasName "include")
2  importInclude href -- :: String -> IOSArrow XmlTree XmlTree
3   = ifA (validateInclude `guards`
4          (readDocument [] "RelaxNG-Schema-grammar.rng" >>> validateXMLDoc href))
5      mkError
6      ( processInclude $< ( readDocument [] href >>> simplificationStep1 >>>
7                            simplificationStep2 >>> getChildren))
```

Beispiel 3.12: Importieren von Schemata durch das `include`-Pattern

[9] Die weitere Fehlerbehandlung bei externen Quellen wird in Abschnitt 5.1.3 betrachtet

Die neu erstellte Schema-Datei `RelaxNG-Schema-grammar.rng` nutzt bereits die Möglichkeiten des `include`-Pattern. Es importiert zuerst die Originalschemadatei (2) und verändert anschließend das Startelement (3) der referenzierten Grammatik. Es zeigt nun nicht mehr auf das `pattern`-Element (7), sondern auf das geforderte `grammar`-Element. Um dies zu ermöglichen, wurde im Originalschema die Deklaration des `grammar`-Elementes (11) in ein eigenes `define`-Pattern ausgelagert (9), da nur diese referenziert werden können (3). Der Rest des Originalschemas kann unverändert übernommen werden.

```
1 <grammar ...> <!-- RelaxNG-Schema-grammar.rng -->
2   <include href="RelaxNG-Schema.rng">
3     <start> <ref name="grammar"/> </start>
4   </include>
5 </grammar>
6 <grammar ..> <!-- original Relax NG Grammatik Schema: RelaxNG-Schema.rng -->
7   <start> <ref name="pattern"/> </start>
8   <define name="pattern">
9     <choice> ... <ref name="grammar"/> </choice>
10  </define>
11  <define name="grammar">
12    <element name="grammar"> ... </element>
13  </define>
14 </grammar>
```

Beispiel 3.13: Aufbau der `RelaxNG-Schema-grammar.rng` Schema-Datei

Nachdem geprüft worden ist, dass das zu importierende Schema der `grammar`-Syntax entspricht, wird eine Kombination des neuen Schemas mit den Definitionen innerhalb des `include`-Pattern vorgenommen.
Das `include`-Element wird in ein `div`-Pattern umbenannt und das nicht mehr benötigte `href`-Attribut entfernt (2). Besitzt das `include`-Pattern eine eigene `start`-Komponente, überschreibt diese das `start`-Element des importierten Schemas. Genauso werden alle `define`-Komponenten aus dem neuen Schema entfernt, die bereits im `include` definiert sind.
Der Arrow `hasStartComponent` berechnet, ob innerhalb der Komponenten des neuen Schemas ein `start`-Pattern existiert (8). Die Komponenten eines Pattern sind alle Kinder des Elementes (7) sowie die Komponenten von `div`-Kindelementen (9). Die Funktion `getDefineComponents` liefert entsprechend eine Liste der Namen (15) sämtlicher `define`-Komponenten (13) zurück. Diese beiden Zwischenergebnisse dienen als Parameter für `insertNewDoc` (3).

```
1 processInclude newDoc -- :: XmlTree -> IOSArrow XmlTree XmlTree
2   = setElemName (QN "" "div" relaxNS) >>> removeAttr "href" >>>
3     (insertNewDoc newDoc $<< hasStartComponent &&& getDefineComponents)
4
5 hasStartComponent :: IOSArrow XmlTree Bool
6 hasStartComponent  = listA hasStartComponent' >>> arr (any id)
7 hasStartComponent' = getChildren >>> choiceA [
8       (isElem >>> hasName "start") :-> (constA True),
9       (isElem >>> hasName "div")   :-> hasStartComponent',
10      this                         :-> (constA False)]
11
12 getDefineComponents -- :: IOSArrow XmlTree [String]
13  = listA getDefineComponents' >>> arr (\xs -> [x | x <- xs, x /= ""])
14 getDefineComponents' = getChildren >>>  choiceA [
15      (isElem >>> hasName "define") :-> (getAttrValue "name"),
```

```
16      (isElem >>> hasName "div")    :-> getDefineComponents',
17      this                          :-> (constA "")]
```

Beispiel 3.14: Berechnen von `start`- und`define`-Komponenten

Die Funktion `insertNewDoc` entfernt auf Basis der Parameter `hasStart` und `defNames` Schema-Elemente aus dem neuen Dokument und fügt den verbleibenden Rest anschließend ein (2). Besitzt `hasStart` den Wert `True`, das heißt dass original `include`-Pattern enthält ein eigenes `start`-Element, werden alle `start`-Komponenten aus dem neuen Schema entfernt (3,6). Ebenso werden alle `define`-Muster gelöscht, deren Namen in der `defNames`-Liste vorkommen (10).

```
1 insertNewDoc :: XmlTree -> Bool -> [String] -> IOSArrow XmlTree XmlTree
2 insertNewDoc newDoc hasStart defNames = insertChildrenAt 0 $
3   constA newDoc >>> (removeStartComponent `when` isA (const hasStart)) >>>
4   ((removeDefineComponent defNames) `when` isA (const $ defNames /= []))
5 removeStartComponent = processChildren $ choiceA [
6     (isElem >>> hasName "start") :-> none,
7     (isElem >>> hasName "div")   :-> removeStartComponent]
8 removeDefineComponent defNames -- :: [String] -> IOSArrow XmlTree XmlTree
9  =processChildren $ choiceA [
10  (hasName "define">>>getAttrVal "name">>>isA \n -> elem n defNames) :->none,
11  (isElem >>> getName>>>isA(== "div")) :-> (removeDefineComponent defNames)]
```

Beispiel 3.15: Einfügen des neuen Schemas

3.3 `name`- und `ns`-Attribute sowie qualifizierende Namen

Nachdem im zweiten Schritt sämtliche externen Referenzen aufgelöst worden sind, beschäftigt sich die dritte Transformationsgruppe mit dem Normalisieren von `name`- (RS 4.8) und `ns`-Attributen (RS 4.9). Zudem werden Elementnamen, die noch einen Namensraumpräfix besitzen, durch einen vollständig qualifizierten Namen ersetzt (RS 4.10).

3.3.1 `name`-Attribute

Bei allen `element`- oder `attribute`-Pattern, bei denen die Festlegung des erlaubten Namens für ein XML Element im Instanzdokument durch ein `name`-Attribut (7) und nicht durch ein `name`-Pattern geschieht, wird das Attribut entfernt (5) und durch ein entsprechendes `name`-Pattern ersetzt (1). Der Inhalt des `name`-Pattern entspricht dabei dem Attributwert.
Besitzt ein `attribute`-Pattern zudem kein `ns`-Attribut, wird dem erzeugtem `name`-Pattern ein `ns`-Attribut mit leerem Attributwert hinzugefügt (2,3).

```
1 ( insertChildrenAt 0 (mkElement "name" none (txt $< getAttrValue "name")) >>>
2   processChildren (addAttr "ns" "" `when` (isElem >>> hasName "name"))
3    `when`
4    (isElem >>> hasName "attribute" >>> hasAttr "name" >>> neg (hasAttr "ns"))
5   >>> removeAttr "name")
6 `when`
7 (isElem >>> hasName "element" `orElse`hasName "attribute" >>> hasAttr "name")
```

Beispiel 3.16: Ersetzen des `name`-Attributes

3.3.2 ns-Attribute

Entsprechend dem vorherigem Abschnitt erhält auch jedes name-, nsName- oder value-Pattern ein ns-Attribut. Da es sich beim ns-Attribut ebenfalls um ein zu vererbendes Attribut handelt, wird der jeweilige Wert entsprechend dem Algorithmus für das datatypeLibrary-Attribut (siehe Abschnitt 3.1.3) bestimmt. Anschließend werden alle ns-Attribute, die nicht einem name-, nsName- oder value-Pattern zugeordnet sind, gelöscht.

3.3.3 Qualifizierende Namen

Die im Abschnitt 3.1.4 durchgeführten vorbereitenden Arbeiten werden erstmalig für die Transformation von qualifizierenden Namen ausgenutzt.
Der Inhalt eines name-Patterns besitzt die Form {uri}localName, wobei der {uri}-Teil optional vorhanden ist. Wenn er vorliegt, wird der URI aus dem Inhalt des name-Pattern entfernt und ein ns-Attribut mit dem entsprechenden Wert hinzugefügt. Das neue Attribut ersetzt dabei jedes eventuell bereits vorhandene ns-Attribut.

```
1 (replaceNameAttr $< (getChildren >>> isText >>> getText))
2 `when` (isElem >>> hasName "name")
3 replaceNameAttr name -- :: (ArrowXml a) => String -> a XmlTree XmlTree
4   = (addAttr "ns" pre >>> processChildren (changeText $ const local))
5     `when` (isA (const $ elem '}' name))
6   where (pre, local) = span (/= '}') name
```

Beispiel 3.17: Ersetzen des name-Attributes

3.4 Normalisieren der Kindelemente und Ersetzen von Pattern

Die vierte Stufe der Vereinfachungen beschäftigt sich einerseits mit der Anzahl der Kindelemente bestimmter Pattern (RS 4.12) und andererseits mit der Ersetzung von div-(RS 4.11), mixed- (RS 4.13), optional- (RS 4.14) und zeroOrMore-Pattern (RS 4.15). Diese lassen sich durch die Kombination anderer Muster beschreiben und dienen lediglich dazu, dem Anwender eine einfachere und schnellere Möglichkeit des Formulierens seines Relax NG Schemas zu erlauben. In einer normalisierten Version sind die oben genannten Pattern nicht mehr existent.

3.4.1 Entfernen von div-Pattern

Die während des Importierens von externen Schemata (siehe Abschnitt 3.2) temporär erzeugten div-Pattern werden gelöscht und durch ihre Kinder ersetzt.

```
getChildren `when` (isElem >>> hasName "div")
```

Beispiel 3.18: Ersetzen von div-Pattern

3.4.2 Anzahl von Kindelementen

Alle define-, oneOrMore-, zeroOrMore-, optional-, list- oder mixed-Pattern dürfen in der vereinfachten Relax NG Syntax genau ein Kindelement haben. Sollten sie mehr als eines besitzen, werden die Kinder innerhalb eines group-Pattern eingeschlossen. Die Funktion checkElemName prüft, ob der Elementname in der als Parameter übergebenen Liste

vorhandenen ist. Für except-Pattern gilt dieselbe Vereinfachung, sie werden jedoch durch ein choice-Pattern[10] umschlossen.

```
(replaceChildren $ mkElement "group" none getChildren) `when`
( checkElemName ["define","oneOrMore","zeroOrMore","optional","list","mixed"]
  >>> listA getChildren >>> isA (\cl -> length cl > 1))
```

Beispiel 3.19: Normalisieren der Anzahl von Kindelementen

Ein element-Pattern wird auf ähnliche Weise verarbeitet. Allerdings soll es exakt zwei Kindelemente besitzen, wobei das erste einen Namen repräsentieren und das zweite ein Pattern sein muss. Besitzt das element-Pattern mehr als zwei Kinder, werden alle bis auf das Namenspattern (2) ebenfalls mit einem group-Pattern umgeben (3,4).

```
1 (replaceChildren $
2  (getChildren >>> checkElemName ["name", "anyName", "nsName"]) <+>
3  ( mkElement "group" none
4     (getChildren >>> neg (checkElemName ["name", "anyName", "nsName"]))))
5 `when`
6 (isElem >>> hasName "element" >>> listA getChildren >>>isA(\l -> length l>2))
```

Beispiel 3.20: Normalisieren von element-Pattern

Die nächste Transformation bezieht sich auf die attribute-Pattern. Besitzen sie jeweils nur ein einziges Kindelement (eine Namensdefinition) dann wird ein text-Pattern als zweites Kind ergänzt.

```
insertChildrenAt 1 (mkElement "text" none none) `when`
(isElem>>> hasName "attribute">>>listA getChildren>>>isA(\l -> length l==1))
```

Beispiel 3.21: Normalisieren von attribute-Pattern

Die choice-, group- und interleave-Pattern besitzen am Ende dieses Vereinfachungsschrittes exakt zwei Kinder. Da eine ähnliche Umwandlung auch noch im Transformationsschritt fünf notwendig ist, wurde die Transformation in die Funktion wrapPattern2Two ausgelagert. Wenn ein Pattern vorher ein Kindelement besessen hat (7), wird es durch dieses ersetzt. Besitzt es hingegen mehr als zwei Kinder (3), werden diese zu einem neuen Element zusammengefasst, wobei das entstehende Element den ursprünglichen Elementnamen erhält (4) und die ersten beiden Kindelemente (4) Kinder des neuen Elementes werden.

Aus	`<choice> p1 p2 p3 </choice>`
wird somit	`<choice> <choice> p1 p2 </choice> p3 </choice>`.

Diese Transformation wird wiederholt (10), bis jedes Pattern exakt zwei Kindelemente enthält.

```
1 wrapPattern2Two n -- :: (ArrowXml a) => String -> a XmlTree XmlTree
2  = choiceA [
3    (listA getChildren >>> isA (\cl -> length cl > 2))
4     :-> (replaceChildren mkElement n none (listA getChildren>>>arrL (take 2))
5                          <+> (listA getChildren >>> arrL (drop 2))
6          >>> wrapPattern2Two name),
7    (listA getChildren >>> isA (\cl -> length cl == 1)) :-> getChildren,
8    (listA getChildren >>> this) :-> this]
```

Beispiel 3.22: Bestimmen von exakt zwei Kindelementen

[10] hierfür ist die when-Bedingung und der Name des erzeugten Elementes zu ändern

3.4.3 Ersetzen des `mixed`-Pattern

Das `mixed`-Pattern kann durch die beiden Muster `interleave` und `text` beschrieben werden. Dabei erhält das neue `interleave`-Element als erste Komponente das Kindelement p des `mixed`-Pattern und als zweites ein `text`-Element. Aus `<mixed> p </mixed>` wird somit `<interleave> p <text/> </interleave>`.

```
(mkElement "interleave" none (getChildren <+> mkElement "text" none none))
`when` (isElem >>> hasName "mixed")
```

Beispiel 3.23: Ersetzen des `mixed`-Pattern

3.4.4 Austauschen des `optional`-Pattern

Ein `optional`-Pattern wird nach dem gleichen Verfahren, allerdings mit Hilfe der Elemente `choice` und `empty`, ersetzt. Aus `<optional> p </optional>` wird demzufolge `<choice> p <empty/> </choice>`.

```
(mkElement "choice" none (getChildren <+> mkElement "empty" none none))
`when` (isElem >>> hasName "optional")
```

Beispiel 3.24: Ersetzen des `mixed`-Pattern

3.4.5 Ersetzen des `zeroOrMore`-Pattern

Als Abschluss dieser Normalisierungssequenz wird noch das `zeroOrMore`-Pattern in eine Auswahl (`choice`) zwischen einem `oneOrMore`- und einem `empty`-Pattern umgewandelt:
`<zeroOrMore> p </zeroOrMore>` wird zu
`<choice> <oneOrMore> p </oneOrMore> <empty/> </choice>`

```
(mkElement "choice" none (mkElement "oneOrMore" none getChildren
                          <+> mkElement "empty" none none))
`when` (isElem >>> hasName "zeroOrMore")
```

Beispiel 3.25: Ersetzen des `zeroOrMore`-Pattern

3.5 Prüfen der ersten Restriktionen

Nachdem die ersten vier Transformationssequenzen durchlaufen worden sind, erfolgt zum ersten Mal eine Kontrolle (RS 4.16), ob die Einschränkungen, die Relax NG im Einsatz von Pattern besitzt, eingehalten werden.
Diesen Schritt ordnet [Vlist 03] noch dem vorherigen Bearbeitungsschritt zu. Da es aber möglich sein soll, die Prüfung von Bedingungen durch einen Aufrufparameter zu steuern beziehungsweise zu deaktivieren, wurden sie in eine eigene Routine ausgelagert und können somit leicht übersprungen werden.
Im Gegensatz zu den drei noch ausstehenden Kontrollen, die erst am Ende des gesamten Vereinfachungsprozesses durchzuführen sind, kann dieser Schritt angewendet werden, ohne dass `ref`- beziehungsweise `parentRef`-Pattern aufgelöst werden müssen. Daraus folgt, dass sich diese Kontrolle auch auf Schema Definitionen bezieht, die in späteren Schritten komplett entfernt werden, da sie entweder nicht erreichbar sind (siehe Abschnitt 3.7.1) oder ein `notAllowed`-Pattern (siehe Abschnitt 3.8.1) enthalten und dadurch gelöscht werden. Fehler,

die in diesem Schritt entdeckt werden, bleiben erhalten und werden nicht durch folgende Transformationen ignoriert oder entfernt.

3.5.1 Kombinationen von `except`-Pattern und Namensdeklarationen

Ein `except`-Pattern, welches als Kindelement eines `anyName`-Pattern auftritt, darf keine weiteren `anyName`-Muster als Nachfahren haben (2,3). Ebenso ist es nicht erlaubt, dass ein `except`-Muster, welches als Kind eines `nsName`-Pattern vorkommt, `nsName` oder `anyName`-Nachfolger besitzt (5,6).

```
1 mkError `when`
2 ( isElem >>> hasName "anyName" >>> getChildren >>> isElem >>>
3   hasName "except" >>> deep (isElem >>> hasName "anyName"))
4 `orElse`
5 ( isElem >>> hasName "nsName" >>> getChildren >>> isElem >>>
6   hasName "except" >>> deep (checkElemName ["anyName", "nsName"]))
```

Beispiel 3.26: Prüfen von `except`-Pattern

3.5.2 `ns`-Attribute und `name`-Pattern

Die zweite Prüfung bezieht sich auf `attribute`-Elemente, die ein `name`- oder `nsName`-Pattern mit einem `ns`-Attribut besitzen.
Hierbei ist es nicht möglich, dass ein `name`-Element, welches als Kind beziehungsweise als Nachfolger des ersten Kindes eines `attribute`-Pattern vorkommt und ein `ns`-Attribut mit leeren Attributwert hat, den Inhalt *xmlns* besitzt. Ebenso darf das `ns`-Attribut nicht den Attributwert *http://www.w3.org/2000/xmlns* aufweisen.

```
mkError `when`
( isElem >>> hasName "attribute" >>> listA getChildren >>> arr head >>>
  ((multi (isElem >>> hasName "name" >>> hasAttr "ns") ) >>>
   (getAttrValue "ns" >>> isA (== "")) `guards`
    (getChildren >>> getText >>> isA (== "xmlns")))
  `orElse`
  ((multi (checkElemName ["name", "nsName"] >>> hasAttr "ns")) >>>
   getAttrValue "ns" >>> isA (compareURI xmlnsNamespace)))
```

Beispiel 3.27: Prüfen von `name`-Pattern

Die zwei folgenden Element Deklarationen sind folglich nicht gültig.

```
<element xmlns="http://relaxng.org/ns/structure/1.0" name="foo">
  <attribute> <name ns="">xmlns</name> </attribute>
</element>
<element xmlns="http://relaxng.org/ns/structure/1.0" name="foo">
  <attribute ns="http://www.w3.org/2000/xmlns" name="bar"/>
</element>
```

Beispiel 3.28: Ungültige Relax NG Schemata

3.5.3 Einsatz von Datentyp-Bibliotheken

Die letzte durchzuführende Prüfung bezieht sich auf den Einsatz von Datentyp-Bibliotheken. Dabei wird kontrolliert, ob die in einem `value`- oder `data`-Pattern eingesetzten Datentypen und angegebenen Parameter innerhalb der deklarierten Bibliothek zur Verfügung stehen und ob diese zudem korrekt kombiniert worden sind. Wie diese Prüfung erfolgt, welche Datentypen innerhalb der Relax NG Implementation der Haskell XML Toolbox zur Verfügung stehen und wie diese um Bibliotheken erweitert werden kann, wird in dem Bereich Datentyp-Bibliotheken (siehe Kapitel 7) näher beschrieben.

3.6 Kombinieren von `define`-, `start`- und `grammar`-Muster

In der fünften Normalisierungsstufe werden alle `define`- und `start`-Pattern, die den gleichen Namen tragen und innerhalb desselben `grammar`-Elementes vorkommen, zusammengefasst (RS 4.17). Außerdem wird das Schema so verändert, dass sein oberstes Element ein `grammar`-Pattern ist und kein weiteres `grammar`-Muster im gesamten Relax NG Schema vorkommt (RS 4.18).

3.6.1 Zusammenfassen der `define`- und `start`-Pattern

Da der Algorithmus zum Zusammenfassen von `define`- und `start`-Pattern identisch ist, beziehen sich die folgenden Aussagen ebenfalls auf die Verarbeitung der `start`-Elemente.
Kommen in einen `grammar`-Pattern mehrere `define`-Elemente mit demselben Namen vor, werden diese entsprechend ihres `combine`-Attributes zusammengefügt. Für jeden Namen darf es dabei nicht mehr als ein `define`-Element geben, welches kein `combine`-Attribut besitzt. Außerdem dürfen sich die `combine`-Attributwerte für denselben Namen nicht unterscheiden. Das heißt, wenn es ein Attribut mit dem Wert `choice` gibt, ist kein Attributwert `interleave` innerhalb dieses `grammar`-Elementes für das `combine`-Attribut erlaubt. Die entsprechende Fehlerbehandlung wird in Abschnitt 5.1.6 betrachtet.

Zwei Definitionen:

```
<define name="n" combine="choice"> p1 </define>
<define name="n" combine="choice"> p2 </define>
```

werden dabei in folgenderweise zu einem `define`-Element zusammengefasst:

```
<define name="n">
  <choice> p1 p2 </choice>
</define>
```

Die Kombination von `define`-Elementen mit demselben Namen wird solange wiederholt, bis zu jedem Namen exakt ein `define`-Pattern existiert.
Die Implementierung des Algorithmus gliedert sich in insgesamt vier Stufen und wird in den folgenden Abschnitten näher beschrieben.

1. Bestimmen sämtlicher `define`-Pattern Namen

Im ersten Schritt werden die Namen sämtlicher Pattern[11] in einer Liste gesammelt (1). Dafür werden nur `define`-Pattern innerhalb desselben `grammar` betrachtet. Wird ein neues `grammar`-Element erreicht (2), ersetzt der Algorithmus dieses durch den `none`-Arrow und bricht

[11] der Parameter pattern hat entweder den Wert `define` oder `start`

ab. Aus der Ergebnisliste werden anschließend doppelt vorhandene Einträge entfernt (4).

```
1 getPatternNamesInGrammar pattern :: String -> IOSArrow XmlTree [String]
2  = processChildren(processTopDown(none `when` (isElem>>>hasName "grammar")))
3    >>> listA ((multi (isElem >>> hasName pattern)) >>> getAttrValue "name")
4 getPatternNamesInGrammar "define" >>> arr nub
```

Beispiel 3.29: Bestimmen sämtlicher `define`-Pattern Namen

2. Kombinieren der Pattern mit demselben Namen

Für jedes Element der Ergebnisliste wird im zweiten Schritt die Funktion `combinePattern` aufgerufen. Sie berechnet einerseits das neue, zusammengefügte `define`-Pattern (2) und löscht andererseits alle alten `define`-Muster aus dem bisherigen Baum (3). Diese beiden Teile werden anschließend zum Endergebnis kombiniert.

```
1 combinePattern p name -- :: String -> String -> IOSArrow XmlTree XmlTree
2  = createPatternElems p name
3    <+> (getChildren >>> deletePatternElems p name)
```

Beispiel 3.30: Kombinieren der Pattern mit demselben Namen

3. Erstellen des neuen Pattern

Der Arrow `getElems` bestimmt alle `define`-Pattern mit dem als Parameter übergebenen Namen (10), die sich innerhalb desselben `grammar`-Elementes befinden (11). Aus dem Ergebnis wird der zum Kombinieren benötigte Wert des `combine`-Attributes bestimmt (3). Gleichzeitig wird das Attribut entfernt (4). Diese beiden Teilergebnisse werden anschließend durch `createPatternElem` zusammengefügt (5). Der Arrow erzeugt ein neues `define`-Pattern (16) und verwendet die bereits vorgestellte Funktion `wrapPattern2Two` (17) um mit Hilfe des `combine`-Wertes alle Pattern mit demselben Namen zu gruppieren.

```
1  createPatternElems :: String -> String -> IOSArrow XmlTree XmlTree
2  createPatternElems pattern name
3    = (getElems pattern name >>> getAttrValue "combine")
4      &&& listA (getElems pattern name >>> removeAttr "combine")
5      >>> createPatternElem pattern name $<< (arr fst &&& arr snd)
6
7  getElems pattern name -- :: (ArrowXml a) => String -> String -> a XmlTree XmlTree
8    = getChildren >>> choiceA [
9        (isElem >>> hasName pattern >>> getAttrValue "name" >>> isA (== name))
10                       :-> (this <+> getElems pattern name),
11       (isElem >>> getName >>> isA (== "grammar") ) :-> none,
12       this            :-> (getElems pattern name)]
13
14 createPatternElem :: (ArrowXml a) => String -> String -> String -> XmlTrees -> a n XmlTree
15 createPatternElem pattern name combine trees
16   = mkElement pattern (mkAttr "name" (txt name))
17         ( (mkElement combine none (arrL (constA trees) >>> getChildren))
18           >>> wrapPattern2Two combine)
```

Beispiel 3.31: Erstellen des neuen Pattern

4. Löschen der alten `define`-Pattern

Abschließend werden die alten, jetzt bereits zusammengefassten `define`-Elemente, aus dem Baum entfernt. Stimmt der Name des aktuellen `define`-Pattern mit dem gesuchten überein, wird es durch den `none`-Arrow gelöscht (3). Bei einem `grammar`-Element bricht die Rekursion ab (4). In allen anderen Fällen wird die Prüfung mit den Kindern fortgesetzt (5).

```
1 deletePatternElems p name
2  = choiceA [
3     (isElem >>> hasName p >>> getAttrValue "name" >>> isA (== name)) :-> none,
4     (isElem >>> getName >>> isA (== "grammar")) :-> this,
5     this :-> (processChildren $ deletePatternElems p name)]
```

Beispiel 3.32: Löschen der alten `define`-Pattern

3.6.2 Entfernen von verschachtelten `grammar`-Elementen

Bevor die verschachtelten `grammar`-Pattern aus dem Schema entfernt werden können, ist sicherzustellen, dass alle `ref`- und `parentRef`-Elemente gültig sind, d.h. das sie erreichbare `define`-Pattern referenzieren. Da es sich hierbei um eine Fehlerprüfung handelt, wird diese erst im Kapitel 5.1.4 betrachtet. Die folgenden vier Schritte werden anschließend bei einem gültigen Schema durchgeführt.

1. Transformieren des obersten Elementes in ein `grammar`-Pattern

Wenn es sich bei dem ersten Pattern `p` des Schemas nicht um ein `grammar`-, sondern beispielsweise um ein `element`-Pattern, handelt, wird das Schema um ein `grammar`- und `start`-Pattern ergänzt: `<grammar> <start> p </start> </grammar>`

```
replaceChildren(mkElement "grammar" none (mkElement "start" none getChildren))
`when` (neg (getChildren >>> hasName "grammar"))
```

Beispiel 3.33: Einfügen eines top-level `grammar`s

2. Umbennen der `define`-Pattern

Im nächsten Schritt werden sämtliche `define`-Pattern so umbenannt, dass keine zwei Pattern denselben Namen besitzen. Dafür ist es notwendig, dass `define`-Pattern sowie alle `ref`- und `parentRef`-Referenzen umzubennen.
Die Funktion `renameDefines` erhält als Parameter zwei Listen. Die erste Liste stellt eine Zuordnung der original Namen sämtlicher `define`-Pattern auf die neuen Namen im aktuellen `grammar`-Muster dar. Die zweite Liste enthält dies entsprechend für alle `define`-Pattern im darüberliegenden `grammar`-Element. Sie wird für die Verarbeitung der `parentRef`-Muster eingesetzt.
Die Initalisierung der ersten Liste geschieht über die bereits aus Kapitel 3.6.1 bekannte Funktion `getPatternNamesInGrammar`, die eine Liste sämtlicher `define`-Namen zurückliefert. Dieses Zwischenergebnis wird durch `createUniqueNames` (6) um die benötigte Abbildung auf die neuen Namen ergänzt. Der zweite Parameter ist die leere Liste (7), da es kein `grammar`-Element als Vorfahren und somit keine `define`-Namen bei der Initialisierung gibt.

```
1 type OldName, NewName = String
2 type RefList = [(OldName, NewName)]
3 renameDefines ref parentRef = ... -- :: RefList -> RefList -> IOSArrow ...
4 renameDefines $<<
5  ( getPatternNamesInGrammar "define" >>>
6    (createUniqueNames $< (getAndSetCounter "define_id" >>> arr read))
7    &&& constA [])
```

Beispiel 3.34: Initialisierung für die Umbenennung

Die neuen `define`-Namen beginnen bei der String-Repräsentation der Zahl Eins und werden fortlaufend für jedes weitere `define`-Element inkrementiert. Hierfür erhält `createUniqueNames` als ersten Parameter die nächste zu verwendene Nummer (1). Sie wird innerhalb des Statuswertes `define_id` gespeichert, um sicherzustellen, dass jeder Aufruf von `createUniqueNames` immer auf dem aktuellen Wert arbeitet. Nachdem die Liste der neuen Namen berechnet worden ist, wird der kommende Wert wieder in `define_id` gesichert (4,8).

```
1 createUniqueNames $< (getAndSetCounter "define_id" >>> arr read)
2 createUniqueNames num --  :: Int -> IOSArrow [String] RefList
3   = arr (\l -> unique l num) >>>
4     perform (setParamInt "define_id" $< arr (max num . getNextValue))
5 unique []     _   = [] -- :: [String] -> Int -> RefList
6 unique (x:xs) num = (x, (show num)):(unique xs (num+1))
7 getNextValue [] = 0 -- :: RefList -> Int
8 getNextValue rl = maximum (map (read . snd) rl) + 1
```

Beispiel 3.35: Erstellen eindeutiger Namen

Im nächsten Schritt wird die Aufgabe der Funktion `renameDefines` näher betrachtet. Trifft sie bei der Bearbeitung der Elemente auf ein `define`-Pattern, wird der neue Name in der ref-Liste nachgeschlagen und das `define`-Muster umbenannt (4). Für ein `ref`-Pattern wird ebenfalls die ref-Liste verwendet (11). Bei einem `parentRef`-Pattern kommt hingegen die parentRef-Liste (13) beim Bestimmen des neuen Namens zum Einsatz.
Erkennt `renameDefines` ein `grammar`-Pattern, berechnet die Funktion eine neue ref-Liste für das aktuelle `grammar`-Muster (8) und setzt die alte ref-Liste ab sofort als parentRef-Liste ein (9). In allen anderen Fällen wird `renameDefines` mit identischen Parametern rekursiv aufgerufen (14).

```
1 renameDefines ref parentRef
2  = processChildren $ choiceA [
3     (isElem >>> hasName "define")
4     :-> (addAttr "name" $<(getAttrVal "name" >>> arr(\n ->lookup n ref))
5          >>> renameDefines ref parentRef),
6     (isElem >>> hasName "grammar")
7     :-> (renameDefines $<< (getPatternNamesInGrammar "define" >>>
8                             (createUniqueNames $< getParamInt 0 "define_id")
9                            ) &&& constA ref),
10    (isElem >>> hasName "ref")
11    :-> (addAttr "name" $<(getAttrVal "name" >>> arr$\n ->lookup n ref)),
12    (isElem >>> hasName "parentRef")
13    :->(addAttr "name" $<(getAttrVal "name" >>> arr$\n ->lookup n parentRef)),
14    this :-> (renameDefines ref parentRef)]
```

Beispiel 3.36: Umbennen der `define`-Pattern

3. Verschieben der `define`-Pattern

Nachdem alle `define`-Pattern einen eindeutigen Namen erhalten haben, können sie innerhalb des Schemas verschoben werden, ohne dass die Zuordnung zu den `ref`-Elementen verloren geht. Der dritte Schritt sorgt dafür, dass alle `define`-Elemente Kinder des obersten `grammar`-Pattern werden.
Der Arrow `deleteAllDefines` löscht dafür sämtliche `define`-Muster aus dem Baum und gibt den Rest unverändert zurück. `getAllDefines` nutzt den multi-Arrow, um auch verschachtelte `define`-Elemente im Baum zu selektieren und als eigenständigen Teilbaum zurückzugeben. Diese verschachtelten `define`-Muster müssen anschließend durch `processChildren deleteAllDefines` wieder aus dem Ergebnis entfernt werden, da sie sonst doppelt vorkommen würden. Die Kombination der beiden Schritte liefert das gewünschte Endergebnis.

```
deleteAllDefines <+> (getAllDefines >>> processChildren deleteAllDefines)
getAllDefines = multi (isElem >>> hasName "define" )
deleteAllDefines=processTopDown (none `when` (isElem >>> hasName "define"))
```

Beispiel 3.37: Verschieben der `define`-Pattern

4. Löschen der `grammar`-Pattern

Als letzter Schritt müssen lediglich die verschachtelten `grammar`-Pattern durch die Kinder ihres `start`-Elementes ersetzt und alle `parentRef-` in `ref`-Pattern umbenannt werden.

```
(getChildren >>> hasName "start" >>> getChildren)
  `when` (isElem >>> hasName "grammar")
(setElemName "ref") `when` (isElem >>> hasName "parentRef")
```

Beispiel 3.38: Löschen der `grammar-` und Umbennen der `parentRef`-Pattern

3.7 Expandieren von `define`-Pattern

Die sechste Stufe verändert das Schema so, dass jedes `element`-Pattern Kind eines `define`-Pattern ist und jedes `define`-Pattern als Kind ein `element`-Pattern besitzt (RS 4.19). Mit anderen Worten wird jedes `element`-Pattern innerhalb eines `define`-Musters gekapselt und alle `ref`-Pattern expandiert, wenn sie nicht auf ein solches `define`-Muster zeigen.

3.7.1 Löschen nicht erreichbarer `define`-Elemente

Zu Beginn werden dafür alle `define`-Elemente, die nicht erreichbar sind, aus dem Schema entfernt. Ein `define`-Element ist genau dann erreichbar, wenn es durch ein erreichbares `ref`-Pattern referenziert wird. Ein `ref`-Pattern ist genau dann erreichbar, wenn es Nachfolger des `start`-Pattern oder eines erreichbaren `define`-Pattern ist.
Die Funktion `getAllDefines` berechnet alle `define`-Pattern im gesamten `grammar`-Element. Da nach der vorherigen Transformation keine verschachtelten `define`-Muster mehr vorhanden sind, reicht an dieser Stelle der `deep`-Arrow (3). Rückgabe der Funktion ist eine Liste mit den Zuordnungen der `define`-Namen auf die `define`-Pattern selbst (1). Die zweite Funktion `getRefsFromStartPattern` liefert eine Liste sämtlicher, vom `start`-Pattern aus erreichbarer, `define`-Elemente.

```
type Env = [(String, XmlTree)]
getAllDefines -- :: IOSArrow XmlTree Env
 = listA $ deep(isElem >>> hasName "define") >>> getAttrValue "name" &&& this
getRefsFromStartPattern -- :: IOSArrow XmlTree [String]
  = listA $ getChildren>>> hasName "grammar" >>>getChildren>>>hasName "start"
            >>> deep (isElem >>> hasName "ref") >>> getAttrValue "name"
```

Beispiel 3.39: Bestimmen sämtlicher `define`-Pattern

Diese beiden Werte bilden zusammen mit dem bisher nicht beschriebenen zweiten Parameter `constA []` die Initalisierung für `removeUnreachDef` (1). Der zu Beginn leere Parameter wird im Ablauf des Algorithmus mit den Namen der verarbeiteten `define`-Pattern gefüllt.
Ist die Differenz zwischen den erreichbaren und den bereits betrachteten `define`-Mustern nicht leer (5,10), es gibt also noch weitere, nicht eingesetzte, aber erreichbare `define`-Elemente, wird der erste Name der Differenz als Basis von `newReachDefs` verwendet (12). Die Funktion berechnet daraus alle neu erreichbaren `define`s, die innerhalb des betrachteten Elementes vorkommen (13) und fügt diese der Liste `reachableDefs` hinzu (14). Zudem wird das jetzt verarbeitete `define`-Pattern in die Liste `processedDefs` aufgenommen (6).
Ist die oben beschriebene Differenz leer, bricht der Suchalgorithmus ab (7). Die Liste `reachDefs` enthält danach die Namen der erreichbaren Definitionen. Abschließend werden alle `define`-Pattern, deren Namen nicht in dieser Liste vorkommen, aus dem Baum entfernt (7,9).

```
removeUnreachDef $<<< getAllDefines &&& constA [] &&& getRefsFromStartPattern

removeUnreachDef :: Env -> [String] -> [String] -> IOSArrow XmlTree XmlTree
removeUnreachDef allDefs processedDefs reachableDefs
 = ifP (const $ unprocessedDefs /= [])
    (removeUnreachDef allDefs (nextTreeName:processedDefs) $< newReachDefs)
    (processChildren $ none `when`
                        ( isElem >>> hasName "define" >>> getAttrValue "name"
                          >>> isA (\n -> not $ elem n reachableDefs)))
unprocessedDefs = reachableDefs \\ processedDefs -- :: [String]
newReachDefs :: IOSArrow n [String]
 = fromJust $ lookup (head unprocessedDefs) allDefs >>>
              listA (deep (isElem >>> hasName "ref") >>> getAttrValue "name")
              >>> arr (noDoubles . (reachableDefs ++))
```

Beispiel 3.40: Entfernen nicht erreichbarer `define`-Pattern

Die Grafik 3.2 soll an einem Beispiel aufzeigen, wie die unterschiedlichen Listen gefüllt werden und wie das Zusammenspiel zwischen ihnen abläuft. Als Basis für das Beispiel dient das folgende Relax NG Schema. Vom `start`-Element ist zu Beginn lediglich das `define`-Pattern `1` erreichbar. Die Funktion `getAllDefines` liefert `1, 2, 3, 4` als Startwerte für die `allDefs`-Liste.

```
<grammar>
  <start> ... <ref name="1"/> ... </start>
  <define name="1"> ... <ref name="2"/> ... <ref name="4"/> ... </define>
  <define name="2"> ... <ref name="1"/> ... <ref name="2"/> ... </define>
  <define name="3"> ... </define>
  <define name="4"> ... </define>
</grammar>
```

Beispiel 3.41: Relax NG Schema

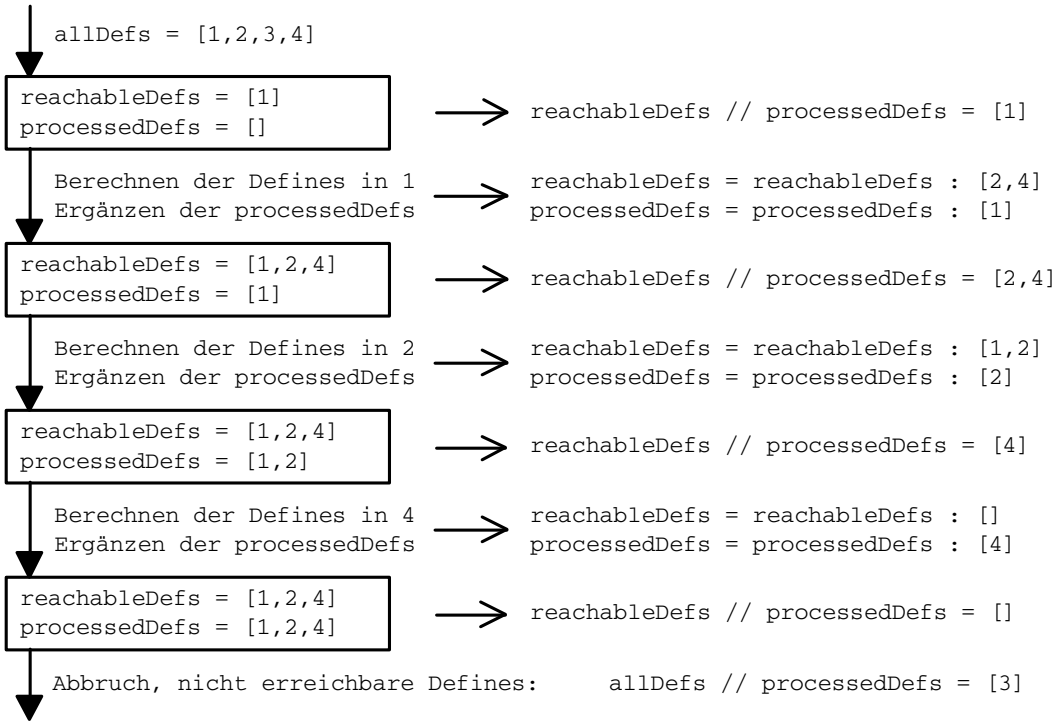

Abbildung 3.2: Aufbau der Listen zum Entfernen von `define`-Pattern

3.7.2 Kapseln der `element`-Pattern

Jedes `element`-Pattern das bisher nicht Kind eines `define`-Musters ist, wird in ein neu anzulegendes `define` ausgelagert. Der Name des neuen `define`-Elementes muss sich dabei von allen schon vorhandenen Namen unterscheiden. An der bisherigen Position des Elementes wird stattdessen ein `ref`-Pattern eingefügt, welches auf das neue `define`-Muster verweisst.
Da die Datenstruktur des XML-Baumes der Haskell XML Toolbox nur Verweise vom Vater zu seinen Kindknoten besitzt, ist es nicht direkt möglich, abzufragen, ob zu einem gefundenen `element`-Pattern ein `define`-Elternteil existiert. Um dieses Problem zu beheben, ist der boolsche Parameter `parentIsDefine` eingeführt worden. Er wird auf den Wert `True` gesetzt, wenn es sich um ein `define`-Pattern handelt (8) und besitzt anderenfalls den Wert `False` (9). Trifft die Funktion `processElements` während ihres Laufes durch den Schema-Baum auf ein `element`-Pattern, wird der Wert des Parameters `parentIsDefine` benötigt (5). Ist er `False`, das Element besitzt kein `define`-Elternteil, muss es ausgegliedert werden (7). Der neue Name wird dabei durch den bereits in Abschnitt 3.6.2 verwendeten und eindeutigen Statuswert `define_id` festgelegt. In allen anderen Fällen ruft sich die Funktion rekursiv wieder auf (9).

```
1 processElements False >>> insertChildrenAt 1 (getParam "elementTable")
2 processElements parentIsDefine -- :: Bool -> IOSArrow XmlTree XmlTree
3  = processChildren $ choiceA [
4      (isElem >>> hasName "element")
5           :-> (ifP (const parentIsDefine)
6                  (processElements False)
7                  (processElems $< getAndSetCounter "define_id")),
8      (isElem >>> hasName "define")  :-> (processElements True),
9      this                           :-> (processElements False)]
```

Beispiel 3.42: Kapseln der `element`-Pattern

Die Funktion `processElems` besitzt zwei Aufgaben. Zum Ersten erzeugt sie mit Hilfe von `storeElement` ein neues `define`-Pattern (3) und sichert dieses innerhalb des Statuswertes

elementTable (4). Die Variable enthält am Ende des Ablaufs alle neu erzeugten define-Pattern und fügt sie als Kinder des grammar-Elementes in den resultierenden Baum (siehe Beispiel 3.42, Zeile 1) ein. Die zweite Aufgabe von processElements besteht darin, eine Referenz auf das neu erzeugte define-Pattern anzulegen (1).

```
1 processElems name = storeElement name >>> mkElement "ref" (mkAttr name) none
2 storeElement name = perform $
3  (mkElement "define" (mkAttr name) (processElements False))
4  &&&(listA $ getParam "elementTable")>>>arr2(:)>>>setParamList "elementTable"
```

Beispiel 3.43: Auslagern eines element-Pattern

3.7.3 Expandieren von ref-Pattern

Die letzte Umformung dieser Reihe behandelt alle expandierbaren ref-Pattern. Ein ref-Pattern ist genau dann expandierbar, wenn es ein define-Pattern referenziert, welches kein element-Pattern als Kind besitzt. Nachdem die Erweiterungen durchgeführt worden sind, werden diese define-Muster gelöscht (3).

```
1 replaceExpandRefs $< getExpandDefs >>> deleteExpandDefs
2 deleteExpandDefs = processTopDown $
3  none `when`
4  (isElem >>> hasName "define" >>> getChildren >>> neg (hasName "element"))
```

Beispiel 3.44: Expandieren von ref-Pattern

Die Funktion getExpandDefs liefert ähnlich dem Arrow getAllDefines (siehe Abschnitt 3.7.1) eine Abbildung der define-Namen auf die define-Muster selbst. Allerdings schränkt sie, im Gegensatz zu getAllDefines, die Auswahl auf expandierbare define-Pattern ein.

```
getExpandDefs -- :: (ArrowXml a) => a XmlTree Env
 = listA $ (multi (( isElem >>> hasName "define" >>> getChildren >>>
                     neg (hasName "element")) `guards` this))
           >>> (getAttrValue "name" &&& this)
```

Beispiel 3.45: Berechnen sämtlicher expandierbarer define-Pattern

Wird nun innerhalb von replaceExpandRefs auf ein ref-Pattern getroffen (3), prüft replaceRef, ob es sich um eine expandierbare Referenz handelt. Ist dies der Fall (7), wird das zu dem Verweis gehörende define-Pattern im env-Parameter gesucht (5) und die Referenz dadurch ersetzt. Anschließend wird der Prozess im ersetzten define-Element fortgeführt (6). Hierbei darf es jedoch nicht zu einer Schleife kommen, das heißt innerhalb einer Expansion ist keine Referenz auf ein bereits ersetztes Pattern erlaubt. Die entsprechenden Fehlerbehandlungen wurden auch an dieser Stelle wieder ausgegliedert (siehe Abschnitt 5.1.4).

```
1 replaceExpandRefs defTab = -- :: Env -> IOSArrow XmlTree XmlTree
2  choiceA [
3   (isElem >>> hasName "ref") :-> (replaceRef $< getAttrValue "name"),
4   this :-> (processChildren $ replaceExpandRefs defTab)]
5 replaceRef n = ( constA(fromJust $ lookup n defTab) >>> getChildren >>>
6                  replaceExpandRefs defTab)
7                `when` (isA (const $ elem n $ map fst defTab))
```

Beispiel 3.46: Ersetzen von Referenzen

3.8 Löschen nicht benötigter Pattern

Der siebte Vereinfachungsschritt führt Bereinigungen innerhalb des Schemas durch, die mit dem nächsten Abschnitt abgeschlossen werden. Zuerst sind hierfür bestimmte Pattern, die ein `notAllowed`-Element enthalten, zu entfernen (RS 4.20). Danach wird das Schema so umgeformt, dass ein `empty`-Muster nicht als Kindelement eines `group`-, `interleave`- oder `oneOrMore`-Pattern (RS 4.21) auftritt.

3.8.1 Entfernen von Pattern mit `notAllowed`-Kindern

Ein `notAllowed`-Pattern darf in der vereinfachten Relax NG Syntax nur als Kindelement eines `start`- oder `element`-Pattern auftreten. Um dieses zu erreichen, werden alle `attribute`-, `list`-, `group`-, `interleave`- oder `oneOrMore`-Muster, die ein `notAllowed`-Element als Kind besitzen, in ein `notAllowed`-Pattern transformiert (2,3). Ein `except`-Element wird hingegen entfernt (4).

```
1 mkElement "notAllowed" none none `when`
2 ( checkElemName ["attribute", "list", "group", "interleave", "oneOrMore"] >>>
3   getChildren >>> hasName "notAllowed")
4 none `when` (isElem>>> hasName "except">>>getChildren>>>hasName "notAllowed")
```

Beispiel 3.47: Entfernen von Pattern mit `notAllowed`-Kindern

Bei einem `choice`-Pattern gilt es zu unterscheiden, ob es ein oder zwei `notAllowed`-Elemente als Kinder enthält. Besitzt es lediglich eines, wird das `choice`-Muster durch das andere Kindelement ersetzt (1). Besteht die Auswahl aus zwei `notAllowed`-Kindern, wird das `choice`-Element ebenfalls durch ein `notAllowed`-Pattern substituiert (4).

```
1 (getChildren >>> neg (hasName "notAllowed")) `when`
2 (isElem >>> hasName "choice" >>> getChildren >>> hasName "notAllowed")
3
4 mkElement "notAllowed" none none `when`
5 (isElem >>> hasName "choice" >>> listA(getChildren >>> hasName "notAllowed")
6  >>> isA (\s -> length s == 2))
```

Beispiel 3.48: Entfernen von `choice`-Pattern mit `notAllowed`-Kindern

3.8.2 Ersetzen von Pattern mit `empty`-Kindern

Ähnlich dem `notAllowed`-Element darf ein `empty`-Pattern nicht als Kindelement eines `group`-, `interleave`- oder `oneOrMore`-Pattern beziehungsweise als zweites Kind eines `choice`-Pattern auftreten.
Von daher wird ein `group`-, `interleave`- oder `choice`-Element, welches zwei `empty`-Kinder besitzt, durch ein `empty`-Muster ersetzt (2,3). Gleiches gilt für das `oneOrMore`-Pattern (4), welches jedoch nur ein Kindelement besitzt.

```
1 mkElement "empty" none none `when`
2 ( checkElemName ["group", "interleave", "choice"] >>>
3   listA (getChildren >>> hasName "empty") >>> isA (\s -> length s == 2))
4 (isElem >>> hasName "oneOrMore" >>> getChildren >>> hasName "empty")
```

Beispiel 3.49: Entfernen von Pattern mit `empty`-Kindern

Die Behandlung von `group`-, `interleave`- und `choice`-Pattern mit nur einem `empty`-Kind ist analog zu dem Vorgehen in Abschnitt 3.8.1. Das Element wird ersetzt beziehungsweise die Kinder vertauscht.

3.8.3 Wiederholen von Transformationen

Die im letzten Abschnitt beschriebenen Transformationen werden so lange wiederholt, bis keine Veränderung in dem Relax NG Schema mehr eintritt. Hierfür werden die oben gezeigten Funktionen um einen Zustand erweitert. Dieser wird zu Beginn eines jeden Durchlaufes mit dem Wert Null initalisiert: `perform (setParamInt "changeTree" 0)`.
Trifft eine der Bedingungen zu, der Baum wird modifiziert, ändert sich der Zustand entsprechend und wird auf Eins gesetzt.

```
(none >>> perform (setParamInt "changeTree" 1)) `when`
(isElem >>> hasName "except" >>> getChildren >>> hasName "notAllowed")
```

Beispiel 3.50: Setzen des Statuswertes bei Veränderung des Baumes

Am Ende des Durchlaufs erfolgt die Prüfung, ob eine Zustandsänderung stattgefunden hat; `changeTree` also den Wert Eins besitzt. Ist dies der Fall, wird die Routine erneut aufgerufen, anderenfalls erfolgt der Übergang zur nächsten Transformationsfunktion.

```
processSimplification7 `when` (getParamInt 0 "changeTree" >>> isA (== 1))
```

Beispiel 3.51: Prüfen, ob eine Veränderung durchgeführt wurde

3.9 Entfernen nicht mehr erreichbarer `define`-Pattern

Die letzte Funktion, die den Baum modifiziert, entfernt nicht mehr erreichbare `define`-Pattern. Hierfür wird der bereits aus Kapitel 3.7.1 bekannte Algorithmus wiederverwendet.
Inhaltlich gehört diese achte Transformation zu den beiden vorhergehenden Abschnitten. Da sie jedoch nicht wiederholt ausgeführt werden muss, ist sie, abweichend von [Vlist 03], in eine eigenständige Funktion ausgegliedert worden.

3.10 Restriktionen auf der vereinfachten Syntax

Mit dem Abschluss der achten Stufe liegt das Relax NG Schema vollständig in vereinfachter Syntax vor. Es folgen weitere Prüfungen, ob sämtliche Beschränkungen, die bei der Kombination von Relax NG Pattern existieren, durch das Schema eingehalten werden. Der Zweck der Einschränkungen liegt dabei einerseits darin, möglichen logischen Fehlern des Anwenders vorzubeugen und andererseits die Implementation eines Validierers für ein Relax NG Schema zu vereinfachen.
Zu den denkbaren logischen Irrtümern, die die Relax NG Grammatik separat betrachtet erlauben würde, zählt beispielsweise die Schachtelung des `attribute`-Pattern. Attribute können in einem XML Dokument keine weiteren Attribute enthalten. Die kontextabhängigen Restriktionen (siehe Abschnitt 3.10.1) verbieten aus diesem Grund, dass ein `attribute`-Pattern ein weiteres `attribute`-Muster als Nachfolger besitzt.
Die nächsten Abschnitte basieren alle auf Kapitel 7 der Relax NG Spezifikation [Relax 01]. Das Generieren aussagekräftiger Fehlermeldungen, wenn eine Restriktion nicht eingehalten wird, ist in Abschnitt 5.1.8 beschrieben.

3.10.1 Kontextabhängige Restriktionen

Zur ersten Klasse der zu überprüfenden Einschränkungen zählen die kontextabhängigen Restriktionen (RS 7.1). Um diese zu beschreiben, wird das Konzept der *verbotenden Pfade* angewendet. Ein Pfad ist dabei eine Sequenz von Elementnamen, die durch einen einfachen `/` oder doppelten `//` Schrägstrich getrennt sind. Die Bedeutung der Schrägstriche entspricht der Idee der abgekürzten Lokalisierungspfade für die `child`- beziehungsweise `descendant-or-self`-Achse aus der XPath Spezifikation [XPath 99]. Der Pfad `foo/bar` trifft auf alle Elemente `foo` zu, die ein Kind `bar` besitzen. Bei `foo//bar` muss das Element `foo` einen beliebig tief angesiedelten Nachfolger `bar` enthalten.

Hinweis:

Für eine kompaktere Angabe der verbotenen Pfade wird die Notation um ein logisches „oder“, dargestellt durch den senkrechten Strich `|`, ergänzt. Der Pfad `a/(b|c)/d` trifft auf den Pfad `a/b/d` oder auf `a/c/d` zu.

`attribute`-Pattern

Die folgenden Pfade sind für Attribute nicht erlaubt: `attribute//(attribute | ref)`

Attribute können keine eigenen Attribute besitzen. Der zweite Pfad ist nicht möglich, da in der vereinfachten Relax NG Syntax jedes `ref`-Pattern ein Element referenziert und ein Attribut kein weiteres Element enthalten darf.

```
mkError ‘when‘
( isElem >>> hasName "attribute" >>> getChildren >>>
  deep (isElem >>> (hasName "attribute" ‘orElse‘ hasName "ref")))
```

Beispiel 3.52: Überprüfen nicht erlaubter `attribute`-Pattern Pfade

`oneOrMore`-Pattern

Die folgenden Pfade sind für `oneOrMore`-Pattern nicht möglich:
`oneOrMore//(group | interleave)//attribute`

Attribute dürfen nicht doppelt vorkommen. Eine Kombination von einer oder mehreren Gruppen beziehungsweise Verschachtelungen von Attributen ist somit in einem gültigen Relax NG Schema nicht erlaubt.

```
mkError ‘when‘
( isElem >>> hasName "oneOrMore" >>> getChildren >>>
  deep (isElem >>> (hasName "group" ‘orElse‘ hasName "interleave")) >>>
  getChildren >>> deep (isElem >>> hasName "attribute"))
```

Beispiel 3.53: Testen von Pattern mit doppelten Attributen

`list`-Pattern

Die folgenden Pfade sind für `list`-Pattern nicht erlaubt:
`list//(ref | attribute | list | text | interleave)`

Listen arbeiten innerhalb von Relax NG auf Textknoten. Daher sind Referenzen auf Elemente, die Angabe von Attributen oder weitere Textelemente als Nachfolger von Listen verboten. Das Schachteln von Listen sowie der Einsatz von `interleave`-Pattern würden die Implementation eines Validators erheblich erschweren. Aus diesem Grund wurden Listen und `interleave`-Pattern als Nachfolger von Listen durch die Relax NG Entwickler ausgeschlossen[12].

```
mkError `when`
( isElem >>> hasName "list" >>> getChildren >>>
  deep (checkElemName ["list","attribute","ref", "text", "interleave"]))
```

Beispiel 3.54: Überprüfen der Nachfolger von Listen

`except`-Pattern

Die folgenden Pfade sind für `except`-Pattern nicht erlaubt:
`data/except//(attribute | ref | text | list | group | interleave | oneOrMore | empty)`

`except`-Pattern, die innerhalb eines `data`-Pattern auftreten, dienen dazu, die möglichen Werte des `data`-Pattern einzuschränken. Von daher dürfen als Nachfolger eines `except`-Elementes nur `value`-, `data`- oder `choice`-Pattern vorhanden sein. Alle anderen Muster sind an dieser Stelle nicht gültig.

```
mkError `when`
( isElem >>> hasName "data" >>> getChildren >>>
  isElem >>> hasName "except" >>>
  deep (checkElemName ["attribute", "ref", "text", "list", "group",
                       "interleave", "oneOrMore", "empty"]))
```

Beispiel 3.55: Testen von Pattern mit doppelten Attributen

`start`-Pattern

Die folgenden Pfade sind für `start`-Pattern nicht erlaubt:
`start//(attribute | data | value | text | list | group | interleave | oneOrMore | empty)`

Nach der Vereinfachung der Relax Syntax beschreibt das `start`-Pattern die möglichen Wurzelelemente des XML Instanzdokumentes. Hier sind nur Elemente, also `ref`-Pattern, und Auswahlen (`choice`) zwischen diesen erlaubt. Die restlichen Muster dürfen nicht vorkommen.

```
mkError `when`
( isElem >>> hasName "start" >>> getChildren >>>
  deep (checkElemName [ "attribute", "data", "value", "text", "list",
                        "group", "interleave", "oneOrMore", "empty"]))
```

Beispiel 3.56: Überprüfen der Nachfolger des `start`-Elementes

[12] Vgl. http://books.xmlschemata.org/relaxng/relax-CHP-15-SECT-2.html#relax-CHP-15-SECT-2.2

3.10.2 Beschränkungen von Inhaltsmodellen

Relax NG erlaubt es nicht, innerhalb eines `element`-Pattern, Elemente zu gruppieren, die einerseits einen Kindknoten abbilden können und andererseits auf eine Zeichenkette passen (RS 7.2). Enthält ein `element`-Muster beide Pattern-Arten, müssen diese als Alternativen zueinander angegeben werden. Das folgende Schema ist somit nicht gültig, da das innere `element`-Pattern (neben `data`, `value`, `list` und `text`) einen Kindknoten darstellt und das `data`-Pattern (neben `value` und `list`) eine Zeichenkette abbilden kann.

```
1 <element name="foo">
2   <group>
3     <element name="bar"> <empty/> </element>
4     <data type="int"/>
5   </group>
6 </element>
```

Beispiel 3.57: Ungültiges Relax NG Schema

Um diese Eigenschaft zu formalisieren, wurde in der Relax NG Spezifikation das Konzept des Inhaltsmodells für ein `element`-Pattern eingeführt. Jedes Pattern, welches als Inhalt eines Elementes möglich ist, besitzt entweder den Inhaltstyp `empty`, `simple` oder `complex`. Das Modell der Spezifikation wurde für die Implementation um den Inhaltstyp `none` ergänzt, der für Pattern steht, die nicht innerhalb eines Elementes vorkommen dürfen. Hieraus resultiert die Definition für den Datentyp `ContentType`.

```
data ContentType = CTEmpty | CTComplex | CTSimple | CTNone
     deriving (Show, Eq, Ord)
```

Beispiel 3.58: Datentyp ContentType

Zur Berechnung, ob ein Pattern als Inhalt eines Elementes möglich ist, werden zwei Hilfsfunktionen benötigt: `max(ct1, ct2)` und `isGroupable(ct1, ct2)`. Die Funktion `max(ct1, ct2)` berechnet das Maximum zweier `ContentTypes`, wobei die Wertigkeit der Typen in aufsteigender Reihenfolge `empty`, `simple`, `complex` und `none` ist. Diese Eigenschaft kann direkt durch die Ableitung des Datentyp `ContentType` von den Haskell Klassen `Eq` und `Ord` erreicht werden. Die Definition einer separaten `max`-Funktion für `ContentTypes` ist nicht notwendig. Die zweite Funktion `isGroupable(ct1, ct2)` lässt sich sehr einfach über das Haskell Pattern-Matching implementieren. Der `empty`-Typ lässt sich mit jedem Typ gruppieren (1,2), ebenfalls können zwei `complex`-Typen miteinander verknüpft werden (3). Alle anderen Kombinationen sind nicht erlaubt (4).

```
1 isGroupable CTEmpty   _         = True -- :: ContentType->ContentType -> Bool
2 isGroupable _         CTEmpty   = True
3 isGroupable CTComplex CTComplex = True
4 isGroupable _         _         = False
```

Beispiel 3.59: isGroupable Funktion für ContentTypes

Ein `element`-Pattern ist nicht gültig, wenn sein Inhalt keinen `ContentType` besitzt, also dem `none`-Typ entspricht.

```
mkError `when`
( isElem >>> hasName "element" >>> listA getChildren >>>
  arr last >>> getContentType >>> isA (== CTNone))
```

Beispiel 3.60: Prüfen eines Pattern auf einen gültigen `ContentType`

Der Ausschnitt aus der Funktion `getContentType` zeigt examplarisch, wie der Typ zu einem Pattern berechnet wird. Das `value`-Muster besitzt immer den Typ `simple` (3), genau wie ein `ref`-Pattern immer vom Typ `complex` ist (4). Der `ContentType` eines `interleave`-Pattern hängt hingegen von seinen Kindelementen ab (6). Hierbei wird erst deren Inhaltstyp bestimmt (10). Wenn diese nicht gruppierbar sind, besitzt das `interleave`-Muster den `none`-Typ. Sind sie jedoch kombinierbar, bestimmt sich der Typ des `interleave`-Pattern aus dem Maximum der Inhaltstypen seiner Kinder (11).

```
1  getContentType = -- :: IOSArrow XmlTree ContentType
2   choiceA [
3     (isElem >>> hasName "value")      :-> (constA CTSimple),
4     (isElem >>> hasName "ref")        :-> (constA CTComplex),
5     (isElem >>> hasName "empty")      :-> (constA CTEmpty),
6     (isElem >>> hasName "interleave") :-> processInterleave,
7     ... ]
8  processInterleave = -- :: IOSArrow XmlTree ContentType
9   listA getChildren >>>
10  ((arr head >>> getContentType) &&& (arr last >>> getContentType)) >>>
11  arr2 (\a b -> if isGroupable a b then max a b else CTNone)
```

Beispiel 3.61: Berechnung des ContentTypes für Pattern

3.10.3 Beschränkungen für `attribute`- und `interleave`-Pattern

Die letzten Prüfungen beziehen sich auf Restriktionen bezüglich der `attribute`- (RS 7.3) und `interleave`-Pattern (RS 7.4). Es wird getestet, ob sich die Namensklassen[13] der definierten Attribute beziehungsweise Elemente beim `interleave`-Pattern nicht überschneiden.

Prüfen von `attribute`-Namen

Wie bereits im Kapitel 3.10.1 erwähnt, dürfen Attribute nicht doppelt innerhalb eines XML Elementes vorkommen. Während der genannte Abschnitt testet, ob Attribute durch die Kombination des `oneOrMore`- und `group`-Pattern mehrfach deklariert werden können, beschäftigt sich dieser Absatz mit der Frage, ob sich Attributnamen überschneiden. Dies ist der Fall, wenn für ein Pattern `<group> p1 p2 </group>` oder `<interleave> p1 p2 </interleave>` ein Attribut existiert, dessen Name sowohl in der Namensklasse eines Attributes von `p1` als auch in der Klasse der Attributnamen von `p2` *auftritt*. Das folgende Relax NG Schema ist ungültig, da der Attributname `bar` innerhalb der Namensklasse `anyName` liegt und sich die beiden Attributnamen somit überdecken.

```
<element name="foo">
  <group>
    <attribute name="bar"/>
    <oneOrMore> <attribute> <anyName/> </attribute> </oneOrMore>
  </group>
</element>
```

Beispiel 3.62: Doppelte Attribute sind nicht erlaubt

[13] Die Namensklasse (engl. nameclass) eines Pattern repräsentiert alle Namen, die durch das Pattern beschrieben werden können. Die Namensklasse des `<anyName/>`-Pattern umfasst somit alle syntaktisch korrekten Namen. Die Namensklasse des Pattern `<name>foo:bar</name>` enthält hingegen genau einen Namen, das Element `bar`, welches innerhalb des Namensraum `foo` liegt.

Es wird definiert, dass ein Pattern p1 innerhalb eines Pattern p2 *auftritt*, wenn p1 und p2 identisch sind oder p2 ein `choice`-, `interleave`-, `group`- oder `oneOrMore`-Pattern ist und p1 in einem oder mehreren Kindern von p2 *auftritt*.

Namensklassen Analyse

Um zu prüfen, ob sich die Klassen zweier Attributnamen überschneiden, wird der von James Clark entwickelte Algorithmus „Name class analysis“ [Clark NCA 03] eingesetzt.
Er basiert auf der Idee, für beide Namensklassen jeweils einen allgemeingültige Repräsentanten zu konstruieren und danach zu testen, ob einer der Repräsentanten innerhalb beider Klassen liegt. Ist dies der Fall, überdecken sich die Attributnamen.
Die Stellvertreter werden dabei so gewählt, dass für jeden Namen n, der in einer der Klassen liegen kann, mindestens ein Repräsentant r vorhanden sein muss und r auch nur genau dann Mitglied einer der beiden Klassen ist, wenn auch n darin liegt. Bei der Konstruktion der Menge von Repräsentanten, die auf einem `anyName`- oder `nsName`-Pattern basieren, ist es notwendig, einen Namensraum URI beziehungsweise einen lokalen Namen zu finden, der in beiden Namensklassen nicht vorkommt. Um den Aufwand dafür zu minimieren, wird eine Zeichenkette verwendet, die keinem gültigen URI beziehungsweise keinem lokalem Namen entspricht. Damit ist sichergestellt, dass der Name nicht in einer der beiden Klassen existiert.
Der folgende (nicht vollständig dargestellte) Haskell-Code ist eine Implementierung der wesentlichen Aspekte des beschriebenen Algorithmus. Die zentrale Vergleichsfunktion ist `overlap`. Sie erhält zwei Namensklassen als Parameter und liefert als Ergebnis, ob sie sich überlagern. Die Implementation verwendet den Datentypen `NameClass` (siehe Kapitel 4.1.2), der alle möglichen Klassen abbildet, sowie die Funktion `contains` (siehe Abschnitt 6.2.1), die berechnet, ob ein vollständig qualifzierter Name in einer Klasse enthalten ist. Eine detailliertere Betrachtung dieser Teile findet in den angegebenen Kapiteln statt und ist zum grundsätzlichen Verständnis des Algorithmus an dieser Stelle nicht notwendig.

```
illegalLocalName = ""    -- :: String
illegalUri       = "\x1" -- :: String
overlap nc1 nc2 = -- :: NameClass -> NameClass -> Bool
 any (bothContain nc1 nc2) (representatives nc1 ++ representatives nc2)
bothContain :: NameClass -> NameClass -> QName -> Bool
bothContain nc1 nc2 qn = contains nc1 qn && contains nc2 qn

representatives :: NameClass -> [QName]
representatives (AnyNameExcept nc) =
  (QN illegalLocalName illegalUri) : (representatives nc)
representatives (Name ns ln) = [QN ns ln]
representatives ...
```

Beispiel 3.63: name class analysis Algorithmus

Das folgende Beispiel verdeutlicht den oben beschriebenen Ablauf. Es zeigt dabei auf, warum ein ungültiger, lokaler Name für die Repräsentation des `anyName`-Pattern geeignet ist. Als Ergebnis wird bewiesen, dass sich die beiden nachstehenden Namen überschneiden.

P1:

```
<anyName>
  <except>
    <name>foo</name>
  </except>
</anyName>
```

P2:

```
<anyName>
  <except>
    <name>bar</name>
  </except>
</anyName>
```

Die Pattern werden folgendermaßen auf den `NameClass`-Datentypen abgebildet[14]:

```
P1: AnyNameExcept (Name _ "foo")
P2: AnyNameExcept (Name _ "bar")
```

Die Funktion `representatives` erzeugt daraus die Liste der allgemeingültigen Repräsentanten für P1 und P2:

```
P1: [(QN "\x1" ""), (QN _ "foo")]
P2: [(QN "\x1" ""), (QN _ "bar")]
```

Die Konkatenation der beiden Teilergebnisse[15] ist die Grundlage für die Prüfung durch die Funktionen `overlap` und `bothContain`.

```
[(QN _ "bar"), (QN _ "foo"), (QN "\x1" "")]
```

Die Funktion `bothContain` testet alle Elemente der Liste gegen die Namensklassen und liefert als Ergebnis, ob beide Namen im Listenelement enthalten sind.
Die Prüfung des ersten Listenelementes schlägt fehl (1), da `(QN _ "bar")` nicht in `(AnyNameExcept (Name _ "bar")` enthalten ist, beim zweiten Test ist der Name „foo“ nicht enthalten (5).

```
1 contains (AnyNameExcept (Name _ "bar")) (QN _ "bar")    --> liefert False
2 && contains (AnyNameExcept (Name _ "foo")) (QN _ "bar") --> liefert True
3
4 contains (AnyNameExcept (Name _ "bar")) (QN _ "foo")    --> liefert True
5 && contains (AnyNameExcept (Name _ "foo")) (QN _ "foo") --> liefert False
```

Beispiel 3.64: Prüfen der ersten beiden Elemente der Namensliste

Die letzte Untersuchung erfolgt gegen den `anyName`-Repräsentanten. Hier liefern beide Prüfungen den Wert `True` zurück. Daraus folgt, dass sich die Namen überschneiden. Bei diesem Test wird auch die Wahl des Stellvertreters für den lokalen Namen wichtig. Es dürfte im Beispiel weder `foo` noch `bar` verwendet werden. Durch den Einsatz eines eigentlich ungültigen lokalen Namens wird mit minimalem Aufwand sichergestellt, dass der Vergleich immer das korrekte Ergebnis liefert.

```
contains (AnyNameExcept (Name _ "bar")) (QN "\x1" "") --> liefert True
&&
contains (AnyNameExcept (Name _ "foo")) (QN "\x1" "") --> liefert True
```

Beispiel 3.65: Prüfen des letzten Elementes der Namensliste

Berechnen sämtlicher Namensklassen

Bevor der „name-class-analysis“-Algorithmus angewendet werden kann, ist es notwendig, sämtliche Namensklassen zu berechnen. Dies geschieht für alle `group`- beziehungsweise `interleave`-Pattern, die als Kind eines `element`-Musters auftreten, durch die `occur`-Funktion (3). Das Ergebnis der `occur`-Funktion ist jedoch eine Liste von XML-Bäumen, die durch den Aufruf von `createNameClass` (siehe Abschnitt 4.2.2) in eine Liste von `NameClass`-Werten transferiert werden. Nachdem alle Klassen für das erste und zweite Kind eines `group`- beziehungsweise `interleave`-Pattern bestimmt worden sind, vergleicht `isIn` (9) jedes Element

[14] Der Unterstrich _ steht für einen beliebigen Namensraum
[15] Das zweifach vorkommende Element (QN “\x1“ ““) braucht nur einmal betrachtet zu werden

der ersten mit jedem Element der zweiten Liste. Wird dabei eine Übereinstimmung gefunden, ist das Schema nicht gültig, da es doppelte Attributnamen besitzt. In diesem Fall wird eine Fehlermeldung erzeugt.

```
mkError `when`
( checkElemName ["group", "interleave"] >>> listA getChildren >>>
  (arr head >>> listA occur >>> fromLA createNameClass)
   &&& (arr last >>> listA occur >>> fromLA createNameClass)
  >>> isA (\(a, b) -> isIn a b))

isIn _ []       = False -- :: [NameClass] -> [NameClass] -> Bool
isIn [] _       = False
isIn (x:xs) ys = (any (overlap x) ys) || isIn xs ys
```

Beispiel 3.66: Vergleichen von Namensklassen

Testen von Attributen mit unendlicher Namensklasse

`attribute`-Pattern, die in ihrem möglichen Namen nicht eingeschränkt sind, also der `anyName`- oder `nsName`-Namensklasse angehören, müssen wiederholt werden. Das heißt, das die Konstellation eines `attribute`-Pattern gefolgt von einem `anyName`- oder `nsName`-Pattern nur genau dann gültig ist, wenn das Attribut ein `oneOrMore`-Pattern als Vorfahren besitzen.
Trifft die Funktion `checkInfiniteAttribute` auf ein `oneOrMore`-Pattern, ist die Bedingung erfüllt und die Suche kann beendet werden (4). In dem Fall, dass es sich um ein `attribute`-Pattern handelt, wird auf einen `anyName` oder `nsName` Nachfolger durch den `deep`-Arrow geprüft (6) und wenn notwendig eine Fehlermeldung generiert (1). In allen anderen Fällen wird der rekursive Abstieg durch den Baum fortgesetzt.

```
mkError `when` (isElem >>> hasName "element" >>> checkInfiniteAttribute)
checkInfiniteAttribute
  = getChildren >>> choiceA [
      (isElem >>> hasName "oneOrMore") :-> none,
      (isElem >>> hasName "attribute" >>>
       deep (checkElemName ["anyName", "nsName"])) :-> this,
      this                              :-> checkInfiniteAttribute]
```

Beispiel 3.67: Testen von Attributen mit unendlicher Namensklasse

Beschränkungen des `interleave`-Pattern

Für `interleave`-Pattern gelten ähnliche Beschränkungen wie für die `attribute`-Muster im vorherigen Abschnitt.
Innerhalb eines Pattern `<interleave> p1 p2 </interleave>` darf es keinen Namen geben, der zu einer Namensklasse eines Elementes, welches in `p1` auftritt und der Namensklasse eines zweiten Elementes, welches in `p2` auftritt, gehört. Zudem darf ein `text`-Pattern nicht gleichzeitig in `p1` und `p2` auftreten. Die Erklärung, wann ein Pattern innerhalb eines anderen Pattern auftritt, entspricht der Definition aus Abschnitt 3.10.3.

4 Erzeugen des Pattern-Datentyps

Der gesamte Vereinfachungs- und Normalisierungsprozess für ein Relax NG Schema basiert auf dem Datentyp `XmlTree` der Haskell XML Toolbox. Das Schema wurde bisher als reines XML-Dokument verstanden, dass durch die Anwendung verschiedener Filter, die die Haskell XML Toolbox für die Verarbeitung von XML zur Verfügung stellt, verändert wird.

Dieses Kapitel beschreibt, wie der bis hierher eingesetzte allgemeine Datentyp `XmlTree` für XML-Dokumente in einen spezielleren Datentypen `Pattern` durch das Modul `CreatePattern` transformiert wird. Der Datentyp `Pattern` bildet die Grundlage für den im Kapitel 6 dargestellten Algorithmus zum Validieren eines Relax NG Schemas gegen ein XML-Instanzdokument.

4.1 Notwendige Datentypen

Die definierten Strukturen für die Abbildung der vereinfachten Syntax im Relax NG Modul der Haskell XML Toolbox, basieren auf den von James Clark in [Clark ARV 02] entwickelten Haskell Datentypen.
Sie wurden um weitere Konstruktoren für die Abbildung von Fehlern, inklusive aussagekräftiger Fehlermeldungen, ergänzt und an die bereits vorhandenen Datenstrukturen der Haskell XML Toolbox angepasst. Hierzu zählt beispielsweise die Abbildung eines vollständig qualifizierten Namens, der innerhalb der Haskell XML Toolbox über die Datenstruktur

```
data QName
  = QN { namePrefix :: String, localPart :: String, namespaceUri :: String}
```

erfolgt.
Der von James Clark verwendete Typ `data QName = QName Uri LocalName` enthält jedoch die `namePrefix`-Komponente nicht und musste somit entsprechend der Haskell XML Toolbox Struktur modifiziert werden.

4.1.1 `Pattern`-Datentyp

Der Datentyp `Pattern` bildet alle, in der vereinfachten Syntax von Relax NG, noch vorhandenen Elemente ab. Einige Pattern, wie beispielsweise das `zeroOrMore`- oder das `optional`-Pattern, die Bestandteile der vollständigen Syntax sind, wurden während des Transformationsprozesses durch andere Muster ersetzt. Sie sind somit in der vereinfachten Syntax nicht mehr existent und müssen für die Definition des `Pattern`-Datentyps nicht berücksichtigt werden.

In der XML-Repräsentation eines Relax NG Schemas besitzen die `Empty`- und `Text`-Elemente keine Kinder oder Attribute. Sie werden durch einfache Typkonstruktoren abgebildet.
`Choice`-, `Interleave`-, `Group`-, `After`-, `OneOrMore`- oder `List`-Pattern besitzen Kindelemente. Sie werden daher durch einen `Produkttyp` dargestellt, der entweder einen oder zwei

weitere Werte vom Typ `Pattern` enthält, die die Abbildung der Kinder speichern. Der `After`-Konstruktor (siehe Abschnitt 6.2.4) wird lediglich als interne, temporäre Struktur bei der Validierung eingesetzt und hat keinen entsprechenden Repräsentanten in der vereinfachten Relax NG Syntax. Der `NotAllowed`-Konstruktor wird beim Aufbau des `Pattern`-Datentyp sowie beim Validieren gegen ein XML-Instanzdokument zur Darstellung von Fehlermeldungen genutzt.

```
data Pattern = Empty
             | Text
             | Choice Pattern Pattern
             | Interleave Pattern Pattern
             | Group Pattern Pattern
             | After Pattern Pattern
             | OneOrMore Pattern
             | List Pattern
             | Element NameClass Pattern
             | Attribute NameClass Pattern
             | Data Datatype ParamList
             | DataExcept Datatype ParamList Pattern
             | Value Datatype String Context
             | NotAllowed String
```

Beispiel 4.1: Definition des Datentyp `Pattern`

4.1.2 `NameClass`-Datentyp

Elemente und Attribute besitzen neben ihren Kindern, die ebenfalls durch den `Pattern`-Datentyp abgebildet werden, einen Namen. Um diesen darzustellen, wird der Datentyp `NameClass` eingesetzt.
Er repräsentiert die fünf möglichen Namensklassen für ein Element oder Attribut: `AnyName`, `AnyNameExcept`, `Name`, `NsName` und `NsNameExcept`. Zusätzlich kann über `NameClassChoice` eine Auswahl zwischen zwei verschiedenen Namen realisiert werden. Der letzte Typkonstruktor `NCError` bildet den Fehlerfall ab und wird detaillierter im Abschnitt 5.2.2 betrachtet.
Die Typsynonyme `URI` und `LocalName` stehen für die beiden Bestandteile eines vollständig qualifizierten Namens und sind jeweils als Zeichenkette definiert.

```
type Uri       = String
type LocalName = String
data NameClass = AnyName
               | AnyNameExcept NameClass
               | Name Uri LocalName
               | NsName Uri
               | NsNameExcept Uri NameClass
               | NameClassChoice NameClass NameClass
               | NCError String
```

Beispiel 4.2: Definition des Datentypen `NameClass`

4.1.3 Weitere Datentypen

Neben dem `NameClass`-Typ werden zusätzlich drei weitere Datenstrukturen (`Datatype`, `ParamList` und `Context`) für die Definition der `data`-, `dataExcept`- und `value`-Pattern

benötigt.
Die Datenstruktur `Datatype` identifiziert einen zu verwendenden Datentyp, wobei die erste Komponente `Uri` die Datentyp-Bibliothek und der zweite Teil (`LocalName`) den realen Typ bestimmt. Alle möglichen Parameter, die für einen Datentyp anzugeben sind, repräsentiert `ParamList` als Abbildung des Parameternamens `LocalName` auf dessen Wert. Der Typ `Context` legt die Umgebung fest, in der ein Datentyp zu bearbeiten ist und beschreibt alle Informationen, die für den Kontext (siehe Abschnitt 3.1.4) eines Elementes notwendig sind.
Diese drei Strukturen werden für die Validierung eines im Instanzdokument angegebenen Wertes gegen eine deklarierte Datentyp-Bibliothek genutzt. Eine detaillierte Betrachtung erfolgt daher im Abschnitt „Datentyp-Bibliotheken“ (siehe Kapitel 7).

```
type Datatype  = (Uri, LocalName)
type ParamList = [(LocalName, String)]
type Prefix    = String
type Context   = (Uri, [(Prefix, Uri)])
```

Beispiel 4.3: Definition von weiteren benötigten Datentypen

4.2 Transformation in die Patternstruktur

In der vereinfachten Syntax folgt auf das Wurzelelement des Relax NG Schemas ein `grammar`- und anschließend ein `start`-Element. Diese können beim Generieren des `Pattern`-Datentypen ignoriert werden, da sie für die Validierung gegen ein Instanzdokument nicht von Bedeutung sind. Alle weiteren Elemente werden in das entsprechende Pattern transformiert[1]. Für das `empty`-, `text`- und `notAllowed`-Element kann dies direkt über den `constA`-Arrow erfolgen, da sie keine weiteren Informationen beinhalten.

```
createPatternFromXml env -- :: Env -> LA XmlTree Pattern
 = choiceA [
     isRoot :-> ( getChildren >>> hasName "grammar" >>>
                  getChildren >>> hasName "start" >>>
                  getChildren >>> createPatternFromXml env),
     (isElem >>> hasName "empty")   :-> constA Empty,
     (isElem >>> hasName "choice")  :-> mkChoice env,
     (isElem >>> hasName "data")    :-> mkData env,
     (isElem >>> hasName "element") :-> mkElement env,
     (isElem >>> hasName "ref")     :-> mkRef env,
     ...
     this                           :-> mkError]
```

Beispiel 4.4: Ausschnitt aus der Transformation von `XmlTree` nach `Pattern`

4.2.1 Verarbeiten von Pattern mit Kindelementen

Die Verarbeitung von `choice`-, `group`- und `interleave`-Pattern erfolgt auf identische Weise, da diese jeweils zwei Kinder besitzen. Die Funktion `getTwoChildrenPattern` bestimmt die beiden Kindelemente und transformiert diese in die Patternstruktur. Für Elemente mit nur einem Kind (`oneOrMore` beziehungsweise `list`) wird entsprechend die Funktion `getOneChildPattern` eingesetzt.

[1] Der Fehlerfall (`this`) wird in Abschnitt 5.2.1 betrachtet

```
1 mkChoice env -- :: Env -> LA XmlTree Pattern
2   = listA getChildren >>> getTwoChildrenPattern env >>> arr2 Choice
3
4 getTwoChildrenPattern env -- :: Env -> LA XmlTrees (Pattern, Pattern)
5   = arr head >>> createPatternFromXml env &&&
6     arr last >>> createPatternFromXml env
```

Beispiel 4.5: Umwandlung des `choice`-Pattern

4.2.2 Erzeugen von Namensklassen

Der erste Kindknoten eines `element`- oder `attribute`-Elementes ist stets eine Namensklasse; das zweite Kind entspricht einem Pattern.

```
mkElement env -- :: Env -> LA XmlTree Pattern
 = listA getChildren >>> (arr head >>> createNameClass) &&& secondPattern env
   >>> arr2 Element
```

Beispiel 4.6: Umwandlung des `element`-Pattern

Der Name wird über die Funktion `createNameClass` zu einem Wert des Typs `NameClass` weiterverarbeitet (1). Handelt es sich um ein `anyName`-Element, muss geprüft werden, ob ein `except`-Kind vorhanden ist (9). Ist dies der Fall, wird für alle `except`-Kinder erneut `createNameClass` aufgerufen (10). Anderenfalls kann das `AnyName`-Pattern direkt zurückgegeben werden (11).
Die Transformation eines `nsName`-Elementes erfolgt analog. Es wird lediglich der Wert des `ns`-Attributes als zusätzlicher Parameter genutzt.
Ein einfacher Name wird transformiert, in dem der `ns`-Attributwert und der Textknoten des `name`-Elementes berechnet werden (12).
`processChoice` wandelt beide Kinder in eine Namensklasse um und stellt eine Auswahl zwischen ihnen zur Verfügung. Die Fehlerbehandlung wird im Abschnitt 5.2.2 erläutert.

```
1 createNameClass -- :: LA XmlTree NameClass
2  = choiceA [
3      (isElem >>> hasName "anyName") :-> processAnyName,
4      (isElem >>> hasName "nsName")  :-> processNsName,
5      (isElem >>> hasName "name")    :-> processName,
6      (isElem >>> hasName "choice")  :-> processChoice,
7      this                           :-> mkNameClassError]
8 processAnyName -- :: LA XmlTree NameClass
9  = ifA (getChildren >>> hasName "except")
10     (getChildren >>> getChildren >>> createNameClass >>> arr AnyNameExcept)
11     (constA AnyName)
12 processName = getAttrValue "ns" &&& (getChildren >>> getText) >>> arr2 Name
```

Beispiel 4.7: Transformieren von Namensklassen

4.2.3 Generieren der Pattern für die Darstellung von Daten

Die Elemente `data`, `dataExcept` und `value` werden zum Beschreiben, welche Werte in einem XML-Instanzdokument erlaubt sind, eingesetzt. Das `data`-Pattern besitzt hierfür einen Datentyp, der über die beiden Attribute `datatypeLibrary` und `type` festgelegt wird (9). Zudem

schränken alle `param`-Kinder des `data`-Elementes die möglichen Werte ein. Sie enthalten den Parameternamen als Wert des `name`-Attributes und die Parameterbedingung als Kindelement (11). Die Unterscheidung zwischen `data` und `dataExcept` (2) sowie die Transformation durch `processDataExcept` erfolgen nach dem im vorherigen Abschnitt beschriebenen Verfahren.

```
1 mkData env -- :: Env -> LA XmlTree Pattern
2   = ifA (getChildren >>> hasName "except")
3       (processDataExcept >>> arr3 DataExcept)
4       (processData >>> arr2 Data)
5
6 processData :: LA XmlTree (Datatype, ParamList)
7 processData = getDatatype &&& getParamList
8
9 getDatatype  = getAttrValue "datatypeLibrary" &&& getAttrValue "type"
10 getParamList = listA $ getChildren >>> isElem >>> hasName "param" >>>
11                        (getAttrValue "name" &&& (getChildren >>> getText))
```

Beispiel 4.8: Transformation des `data`- und `dataExcept`-Elementes

Das `value`-Pattern besitzt neben dem bereits erwähnten `Datatype`, die Parameter `Value` und `Context`, um einen einzelnen Wert für ein XML-Element festzulegen und den Kontext, in dem dieses gültig ist, zu definieren.
Es ist in einem Relax NG Schema nicht zwingend notwendig, einen expliziten Wert für das `value`-Pattern anzugeben. Besitzt das `value`-Pattern kein Textelement als Kind (`<value/>`), wird stattdessen die leere Zeichenfolge eingesetzt (3).
Im ersten Normalisierungsschritt wurde der Kontext für das `value`-Pattern durch zusätzliche Attribute gesetzt (siehe Abschnitt 3.1.4). Der Wert von `RelaxContextBaseURI` bestimmt die Basis-URI (4). Alle weiteren Werte enthalten Attribute der Form `RelaxContext:pre`, wobei `pre` für den jeweiligen Präfix steht. Handelt es sich um ein Kontext-Attribut, wird „`RelaxContext:`“ von Namen entfernt (8) und zusammen mit dem Attributwert zurückgegeben (9). Der Wert enthält den von Präfix abgebildeten Namensraum-URI.

```
1 mkValue -- :: LA XmlTree Pattern
2  = getDatatype &&& getValue &&& getContext >>> arr3 Value
3 getValue = (getChildren >>> getText) `orElse` (constA "")
4 getContext = getAttrValue "RelaxContextBaseURI" &&& getMapping
5 getMapping = listA $    -- :: LA XmlTree [(Prefix, Uri)]
6    getAttrl >>>
7    ( (getName >>> isA ("RelaxContext:" `isPrefixOf`)) `guards`
8      (getName >>> arr(drop $ length "RelaxContext:")) &&&
9        (getChildren >>> getText))
```

Beispiel 4.9: Transformation des `value`-Elementes

4.2.4 Transformieren von Referenzen

Es ist innerhalb eines Relax NG Schemas möglich, zirkuläre Strukturen bei der Definition von Pattern aufzubauen. Das heißt, der Nachfolger eines `define`-Elementes kann ein `ref`-Pattern sein, welches als Ziel entweder direkt das deklarierte `define`-Element verwendet oder ein weiteres `define`-Muster referenziert, welches seinerseits das erste `define`-Element referenziert. Der folgende Ausschnitt aus dem Schema für die Relax NG Grammatik (siehe Anhang A.2) nutzt diese Möglichkeit. Das `group`-Element referenziert die Definition `open-patterns` (4). Diese enthält jetzt erneut einen oder mehrere Verweise (10) auf das `define`-Pattern, in dem das `group`-Element deklariert wurde.

```
<define name="pattern">
  <element name="group">
    ...
    <ref name="open-patterns"/>
  </element>
  ...
</define>
<define name="open-patterns">
  ...
  <oneOrMore> <ref name="pattern"/> </oneOrMore>
</define>
```

Beispiel 4.10: Ausschnitt aus dem Schema der Relax NG Grammatik

Die zirkulären Strukturen können nicht direkt in den `pattern`-Datentyp transformiert werden. Bei einem einfachen Aufruf der Funktion `createPatternFromXml` für das Ziel des `ref`-Elementes würde es zu einer Endlosrekursion kommen und die Anwendung nicht mehr terminieren. Um dieses Problem zu lösen, ist die `lazy-evaluation` Eigenschaft von Haskell ausgenutzt worden.
Die Funktion `mkRef` bekommt dafür einen weiteren Parameter von Typ `Env` (1). Diese Umgebungstabelle enthält die Abbildung sämtlicher Namen von `define`-Pattern, die in dem Schema vorhanden sind, auf die deklarierten `define`-Elemente. Diese liegen innerhalb des Parameters jedoch weiterhin als `XmlTree` vor und wurden noch nicht in den `Pattern`-Datentyp transformiert. Da nach der Normalisierung keine geschachtelten `define`-Elemente im Schema vorkommen, reicht der `deep`-Arrow (3) zum Bestimmen der Einträge; der `multi`-Arrow ist nicht notwendig.

```
type Env = [(String, XmlTree)]
createEnv :: LA XmlTree Env
createEnv = listA $ deep (isElem >>> hasName "define")
                    >>> (getAttrValue "name" &&& getChildren)
```

Beispiel 4.11: Aufbau der Umgebung für die `mkRef`-Funktion

Muss während der Transformation in den `Pattern`-Datentyp ein `ref`-Element verarbeitet werden, sucht die Funktion `mkRef` in der Umgebungstabelle nach dem passenden Eintrag zu dem Ziel des `ref`-Elementes (3).
Vorher wird allerdings auf die Elemente der bisherigen Umgebungstabelle die Funktion `transformEnv` angewendet (5). Diese transformiert alle `XmlTree`s über die Funktion `createPatternFromXml` in den neuen Datentyp `Pattern` (7).

```
mkRef env -- :: Env -> LA XmlTree Pattern
 = getAttrValue "name" >>>
   arr (\n -> fromMaybe (NotAllowed errorMsg) . lookup n $ transformEnv env)
transformEnv :: [(String, XmlTree)] -> [(String, Pattern)]
transformEnv e = [ (treeN, (transformEnvElem tree e)) | (treeN, tree) <- e]
transformEnvElem :: XmlTree -> [(String, XmlTree)] -> Pattern
transformEnvElem tree env = head $ runLA (createPatternFromXml env) tree
```

Beispiel 4.12: Transformation des `ref`-Elementes

Die `lazy-evaluation` Eigenschaft von Haskell bewirkt dabei, dass die Berechnung der Elemente für die neue `Pattern`-Liste nicht sofort ausgeführt wird, was erneut eine Endlosrekursion zur Folge hätte. Sie wird stattdessen so lange verzögert, bis der Eintrag in der Liste tatsächlich benötigt wird und die Bestimmung des realen `Pattern`-Wertes notwendig ist.

Dieser Fall tritt jedoch erst bei der Validierung des Schemas gegen ein Instanzdokument ein. Muss dabei kontrolliert werden, ob ein XML-Element auf ein Pattern passt, wird genau das Element der Umgebungsliste berechnet, welches für die Prüfung notwendig ist. Alle anderen Einträge werden weiterhin nicht betrachtet. Hierdurch kann eine unendliche Berechnung der zirkulären Strukturen umgangen werden.

Die Transformationen des vorherigen Kapitels werden alle innerhalb eines `IOStateListArrow` ausgeführt. Dieser ist notwendig, da Relax NG Schemata eingelesen (`Input-Output`-Aktion) und Informationen in einem globalen Zustand gesichert werden müssen. Für die Transformation in den `Pattern`-Datentyp kann der `IOStateListArrow` hingegen nicht eingesetzt werden. Eine mögliche IO-Aktion, die in einem Arrow durchgeführt werden könnte, würde Haskell dazu veranlassen, alle Funktionen vollständig zu berechnen. Die `lazy-evaluation` Eigenschaft kann dadurch nicht mehr eingesetzt werden. Aus diesem Grund basieren alle Funktionen des CreatePattern-Moduls auf dem `ListArrow`, für den keine IO-Aktionen oder globalen Zuständen definiert sind.

4.3 Anzeigen der Patternstruktur

Die resultierende Patternstruktur kann über die drei Funktionen `xmlTreeToPatternString`, `xmlTreeToPatternFormatedString` und `xmlTreeToPatternStringTree` aus dem Modul `PatternToString` ausgegeben werden. Alle drei Arrows bekommen einen `XmlTree` der normalisierten Version eines Relax NG Schemas als Eingabe, berechnen daraus den Pattern-Datentyp und geben eine Zeichenkettenrepräsentation der Struktur zurück. Das folgende Schema soll als Beispiel für die unterschiedlichen Ausgaben durch die drei Funktionen dienen.

```
<element name="foo" xmlns:foo="www.bar.baz">
  <choice>
    <value>lorem ipsum dolor</value>
    <data type="VARCHAR" datatypeLibrary="http://www.mysql.com">
      <param name="minLength">2</param> <param name="maxLength">5</param>
    </data>
    <element name="bar">
      <group> <ref name="baz"/> <text/> </group>
    </element>
    <define name="baz">
      <element name="baz">
        <oneOrMore> <attribute><anyName/></attribute> </oneOrMore>
      </element>
    </define>
  </choice>
</element>
```

Beispiel 4.13: Relax NG Schema für die Ausgabefunktionen

4.3.1 Ausgabe durch xmlTreeToPatternString

Die Funktion `xmlTreeToPatternString` liefert eine nicht weiter formatierte Darstellung der Haskell Datentypen. Sie sollte von daher nur zur internen Fehlersuche eingesetzt werden, bei der es notwendig ist, die exakten Datentypen zu erhalten. Das Schema-Beispiel 4.13 wird folgendermaßen ausgegeben[2]:

[2]die Zeilenumbrüche werden durch `xmlTreeToPatternString` nicht generiert

```
Element {}foo (Choice (Choice (Value ("","token") "abc"
("file://test.rng",[("xml","http://www.w3.org/XML/1998/namespaces"),
("foo","www.bar.baz")]))(Data ("http://www.mysql.com","VARCHAR")
[("length","2"),("maxLength","5")])) (Element {}bar (Group (Element {}baz
(OneOrMore (Attribute AnyName Text))) Text)))
```

Beispiel 4.14: Ausgabe durch `xmlTreeToPatternString`

Zu beachten ist bei der Funktion `xmlTreeToPatternString` allerdings, dass sie nicht für die Ausgabe zirkulärer Strukturen geeignet sind. Da eine direkte Repräsentation der Haskell Datentypen erfolgt, wird die Umgebungsliste für die `ref`-Elemente beliebig tief berechnet, so dass eine unendliche Wiederholung der Ausgabe stattfindet.

4.3.2 Darstellung von Pattern über xmlTreeToPatternFormatedString

Die zweite Möglichkeit der Ausgabe bietet `xmlTreeToPatternFormatedString`. Sie orientiert sich ebenfalls an der Haskell Datenstruktur. Allerdings werden die einzelnen Pattern unterschiedlich für die Ausgabe formatiert, so dass eine bessere Lesbarkeit durch `xmlTreeToPatternFormatedString` gewährleistet wird. Hierdurch können größere Relax NG Schemata, die auch Referenz-Kreisläufe enthalten dürfen, kompakt ausgegeben werden.

```
Element {}foo (Choice (Choice ( Value = abc,
datatypelibrary = http://relaxng.org/ns/structure/1.0, type = token,
context (base-uri =file://test.rng,
parameter: xml = http://www.w3.org/XML/1998/namespaces, foo = www.bar.baz),
Data datatypelibrary = http://www.mysql.com, type = VARCHAR,
parameter: length = 2, maxLength = 5),
Element {}bar (Group (Element {}baz (OneOrMore (
Attribute AnyName (Text))), Text))))
```

Beispiel 4.15: Ausgabe durch `xmlTreeToPatternFormatedString`

4.3.3 Ausgabe durch xmlTreeToPatternStringTree

Für die Darstellung von zirkulären Strukturen eignet sich auch die letzte Funktion `xmlTreeToPatternStringTree`. Sie nutzt die von der Haskell XML Toolbox zur Verfügung gestellten Routinen zur Formatierung eines XML-Dokumentes als Baum und stellt den resultierenden Pattern-Datentyp ebenfalls in einer baumartigen Struktur dar.
Dabei wird jedes Element, nur auf dieses kann ein `ref`-Pattern nach der Normalisierung noch verweisen, genau einmal ausgegeben. Ist eine weitere Darstellung notwendig, wird anstelle des Elementes der Text „reference to element NAME“ eingesetzt.
Die Funktionen `element2PatternTree` arbeitet hierfür auf einem `StateListArrow`, der in einer Statusvariablen vom Typ `[NameClass]` alle Namen der Elemente sichert, die bereits verarbeitet worden sind. Trifft der Algorithmus auf ein `element`-Pattern, wird im Status nachgesehen (3), ob der Elementname in der Liste vorhanden ist. Wenn ja, wird lediglich der oben genannte Text ausgegeben (5). Ist dieser hingegen noch nicht im Status eingetragen, wird der Name ergänzt (6) und das `element`-Pattern inklusive seiner gesamten Nachfolger in eine Zeichenkettenrepräsentation umgewandelt (7).

```
1 element2PatternTree -- :: SLA [NameClass] Pattern PatternTree
2   = ifA ( (arr getNameClassFromPattern &&& getState) >>>
3           isA(\ (nc, liste) -> elem nc liste))
4       ( arr getNameClassFromPattern >>>
```

```
5         arr (\nc -> NTree ("reference to element " ++ show nc) []))
6       ( changeState (\ s p -> (getNameClassFromPattern p) : s) >>>
7         createPatternTreeFromElement "element")
```

Beispiel 4.16: Ausgabe eines `element`-Pattern als Baum

Das Relax NG Schema 4.13 wird für das Beispiel zu `xmlTreeToPatternStringTree` um eine zirkuläre Struktur erweitert. Das `define`-Pattern erhält jetzt neben der Attributdeklaration auch eine Referenz auf sich selbst.

```
<define name="baz">
  <element name="baz">
    <oneOrMore> <attribute> <anyName/> </attribute> </oneOrMore>
    <ref name="baz"/>
  </element>
</define>
```

Beispiel 4.17: Zirkuläre Deklaration eines `define`-Pattern

```
---element {}foo
   |
   +---choice
       |
       +---choice
       |   |
       |   +---value = abc
       |   |   |
       |   |   +---datatypelibrary = http://relaxng.org/ns/structure/1.0,
       |   |   |   type = token
       |   |   |
       |   |   +---context
       |   |       |
       |   |       +---base-uri = file://test.rng
       |   |       |
       |   |       +---namespace environment
       |   |           |
       |   |           +---xml = http://www.w3.org/XML/1998/namespaces
       |   |           |
       |   |           +---foo = www.bar.baz
       |   |
       |   +---data
       |       |
       |       +---datatypelibrary = http://www.mysql.com, type = VARCHAR
       |       |
       |       +---parameter
       |           |
       |           +---length = 2
       |           |
       |           +---maxLength = 5
       |
       +---element {}bar
           |
           +---group
               |
```

```
+---element {}baz
|   |
|   +---group
|       |
|       +---oneOrMore
|       |   |
|       |   +---attribute AnyName
|       |       |
|       |       +---text
|       |
|       +---reference to element {}baz
|
+---text
```

Beispiel 4.18: Ausgabe durch `xmlTreeToPatternStringTree`

Zusätzlich stehen `patternToStringTree` und `patternToFormatedString` für die Darstellung von Pattern zur Verfügung. Die Ausgaben entsprechen den oben genannten `xmlTreeTo...`-Funktionen, jedoch erzeugen die `patternTo...`-Funktionen vorher keine Pattern-Datenstruktur aus dem XML-Baum, sondern erwarten sie bereits als Arrow-Eingabe.

5 Fehlerbehandlung

Um die Erläuterung der entwickelten Algorithmen möglichst einfach zu halten, wurde in den vorherigen Kapiteln davon ausgegangen, dass es sich um gültige Relax NG Schemata und Instanzdokumente handelt. Eine detaillierte Fehleranalyse erfolgte bisher nicht. Es wurde lediglich die Funktion `mkError` als Platzhalter für die vollständige Fehlerbetrachtung und das Generieren aussagekräftiger Meldungen eingesetzt.
Dieses Kapitel greift die möglichen Fehlersituationen, die während des Normalisierens und beim Erstellen des Pattern-Datentyps entstehen können, auf und ergänzt die vorgestellten Algorithmen um eine Fehlerbehandlung.

5.1 Fehler im Normalisierungsprozess

Innerhalb des Normalisierungsprozesses können zwei grundsätzliche Fehlerarten unterschieden werden. Erstens sind dies Fehler, die bei der Transformation des Schemas in die vereinfachte Syntax auftreten. Hierzu zählen beispielsweise unerreichbare externe Schemata oder `ref`-Pattern, die `define`-Elemente referenzieren, welche im Schema nicht vorhanden sind. Die zweite Fehlergruppe resultiert aus dem Überprüfen der Restriktionen. Hierbei sind Fehlermeldungen zu erstellen, wenn die Bedingungen, die die Relax NG Spezifikation an ein Schema stellt, nicht erfüllt werden.
Für die erste Fehlerklasse ist die Angabe einer kurzen textuellen Meldung ausreichend. Die Position des Fehlers kann danach im Schema einfach lokalisiert werden. Für die zweite Gruppe ist hingegen eine aufwändigere Fehlerbehandlung notwendig, da einige Pattern des Schemas während der Transformation durch andere Elemente ersetzt werden. Eine Fehlerangabe für das Originaldokument ist anschließend nicht mehr möglich.

5.1.1 Wert des `datatypeLibrary`-Attributes

Die erste Fehlerüberprüfung erfolgt vor dem Vererben des `datatypeLibrary`-Attributes (siehe Abschnitt 3.1.3). Es muss sichergestellt werden, dass der Wert des Attributes einem gültigen Relax NG URI für Datentyp-Bibliotheken entspricht. Basis dafür ist der `anyURI`-Datentyp (6) der W3C Schema Sprache[1]. Dieser darf innerhalb von Relax NG jedoch weder in der relativen Form angegeben sein (6), noch einen Fragment-Bezeichner (`fragment identifier`) besitzen (7). Die Angabe einer leeren Zeichenkette wird für den URI aber zusätzlich erlaubt (6).

```
1 (mkRelaxError "" $< ( getAttrValue "datatypeLibrary" >>>
2     arr (\a -> "datatypeLibrary attribute: " ++ a ++ " is not a valid URI")))
3 `when` (isElem >>> hasAttr "datatypeLibrary" >>>
4         getAttrValue "datatypeLibrary" >>> isA (not . isRelaxAnyURI))
5 isRelaxAnyURI s  -- :: String -> Bool
6  = s == "" || ( isURI s && not (isRelativeReference s) &&
7    let (URI _ _ path _ frag)= fromMaybe(URI "" Nothing "" "" "") $ parseURI s
8    in (frag == "" && path /= ""))
```

Beispiel 5.1: Überprüfung des `datatypeLibrary`-Attributes

[1] Vgl. http://www.w3.org/TR/xmlschema-2/#anyURI

5.1.2 Fehlende Namensraumdeklaration

Ist für einen Element- oder Attributnamen ein Präfix angegeben, muss innerhalb des Kontextes eine zugehörige Namensraumdeklaration existieren. Alle für das Element gültigen Abbildungen von Präfixen auf URIs werden dafür in einer Liste gesichert. Existiert eine entsprechende Zuordnung für den aktuellen Präfix nicht (2,7), wird ein Fehler ausgelöst (3).

```
1 replaceQNames env name -- :: [(String, String)] -> String -> IOSArrow ...
2   = ifP (const $ uri == Nothing)
3       (mkRelaxError "" $ "No Namespace-Mapping for the prefix " ++ pre ++
4                          " in the Context of Element: " ++ name)
5       (addAttr "name" ('{' : (fromJust uri) ++ "}" ++ local))
6   where
7   uri = lookup pre env -- :: Maybe String
```

Beispiel 5.2: Test auf fehlende Namensraumdeklaration

5.1.3 Import externer Schemata

Vor dem Importieren von externen Schemata durch das `externalRef`-Pattern (siehe Kapitel 3.2) sind drei Kontrollen durchzuführen. Zuerst wird geprüft, ob die im `href`-Attribut angegebene Ressource vorhanden ist und gelesen werden kann (4). Danach ist sicherzustellen, dass es beim Importieren von externen Quellen nicht zu einer Endlosschleife kommt (6,7). Dies ist der Fall, wenn durch ein Schema eine externe Quelle referenziert wird, die erneut das Originalschema einbindet. Zwischen dem ersten und letzten Schema des Kreislaufes können dabei beliebig viele weitere Dokumente liegen. Die Schemata im Beispiel 5.4 zeigen eine entsprechende Konstellation. Zusätzlich ist die generierte Fehlermeldung angegeben.
Um einen Importkreislauf aufdecken zu können, ist die Funktion `simplificationStep2` um zwei Listen erweitert worden, die jeweils URIs enthalten (1). Der erste Parameter wird für das `externalRef`-, der zweite für das `include`-Pattern verwendet (2). Ist ein externes Schema erfolgreich eingelesen, wird die entsprechende Liste um seine URI ergänzt (14). Im zweiten Kontrollschritt wird geprüft, ob die einzulesende Quelle bereits in der Liste vorhanden ist (6). Ist dies der Fall, besteht ein Kreis und der Import bricht mit einer entsprechenden Fehlermeldung ab (7). Abschließend erfolgt die bereits beschriebene Validierung des neuen Schemas gegen die Relax NG Spezifikation (10).

```
1 simplificationStep2 :: [Uri] -> [Uri] -> IOSArrow XmlTree XmlTree
2 simplificationStep2 extHRefs includeHRefs = ...
3 importExternalRef ns href -- :: String -> String -> IOSArrow XmlTree XmlTree
4  = ifA (neg $ constA href >>> getPathFromURI >>> isIOA doesFileExist)
5     (mkRelaxError $ "Can't read " ++ href ++ ", referenced in externalRef")
6     (ifP (const $ elem href extHRefs)
7       (mkRelaxError $ "loop in externalRef-Pattern, " ++
8                       formatStringList " -> " (reverse $ href:extHRefs))
9       (ifA ( validateExternal `guards`
10             (createSpezification >>> validateXMLDoc href))
11         (mkRelaxError $ "The content of the schema " ++ href ++
12           ", referenced in externalRef does not match the syntax for pattern")
13         (readDocument [] href >>> ... >>>
14          simplificationStep2 (href:extHRefs) includeHRefs >>> ... )))
```

Beispiel 5.3: Prüfen der Referenzen von `externalRef`-Pattern

```
<?xml version="1.0"?> <!-- foo.rng -->
  <element name="foo"> ... <externalRef href="bar.rng"/> ... </element>
<?xml version="1.0"?> <!-- bar.rng -->
  ... <element name="bar"> <externalRef href="baz.rng"/> </element> ...
<?xml version="1.0"?> <!-- baz.rng -->
  <element name="baz"> <externalRef href="foo.rng"/> ... </element> ...

Fehlermeldung: loop in externalRef-Pattern,
  file://bar.rng -> file://baz.rng -> file://foo.rng -> file://bar.rng
```

Beispiel 5.4: Importkreislauf durch `externalRef`-Pattern

Komponenten eines `include`-Pattern

Die im vorherigen Abschnitt beschriebenen Prüfungen eines `externalRef`-Pattern werden analog auf ein `include`-Pattern angewendet. Es erfolgen allerdings im Anschluss daran noch zwei weitere inhaltliche Tests.
Die in Abschnitt 3.2 erläuterte Funktion `processInclude` wird um eine Funktion `checkInclude` erweitert[2]. Dort wird zuerst geprüft, ob die erforderliche `start`-Komponente im neuen Schema vorhanden ist, wenn das `include`-Muster eine solche Komponente besitzt (4,5). Der zweite Test kontrolliert die Deklarationen von `define`-Pattern. Für jedes innerhalb des `include`-Pattern angegebene `define`-Element, muss ein äquivalentes `define`-Element im referenzierten Schema existieren, welches anschließend ersetzt wird. der Arrow `getDefineComponents` liefert alle `define`-Namen als Liste zurück (6). Nach dem Entfernen doppelter Einträge wird die Differenz zwischen den beiden Listen gebildet (7). Ist sie leer, wurden alle `define`-Elemente korrekt deklariert; das Schema ist gültig und kann eingelesen werden.

```
1  processInclude href doc -- :: String -> XmlTree -> IOSArrow XmlTree XmlTree
2    = setElemName "div" >>> removeAttr "href" >>> checkInclude href doc
3  checkInclude href doc -- :: String -> XmlTree -> IOSArrow XmlTree XmlTree
4    = ifA ( hasStartComponent &&& (constA doc >>> hasStartComponent) >>>
5            isA (\(a, b) -> if a then b else True))
6        (ifA (getDefineComponents &&& (constA doc >>> getDefineComponents) >>>
7              isA (\(a, b) -> ((noDoubles a) \\ (noDoubles b)) == []))
8            (insertNewDoc doc $<< hasStartComponent &&& getDefineComponents)
9            (mkRelaxError $ "Define-pattern missing in schema " ++ href ++
10                           ", referenced in include-pattern"))
11       (mkRelaxError $ "Grammar-element without a start-pattern in schema " ++
12                       href ++ ", referenced in include-pattern")
```

Beispiel 5.5: Inhaltliche Überprüfung eines externen Schemas

5.1.4 Test von internen Referenzen

Nicht nur beim Importieren von externen Quellen kann es zu Fehlern kommen. Dies ist auch bei internen Referenzen über die `ref`- beziehungsweise `parentRef`-Pattern möglich.
Bei diesen Elementen sind zwei Kontrollen wichtig. Die erste Prüfung testet, ob ein referenziertes `define`-Pattern auch tatsächlich im Schema existiert. Im Anschluss daran muss zudem

[2] Die Implementationen von `hasStartComponent` und `getDefineComponents` sowie die Definition einer Komponente entsprechend Abschnitt 3.2

sichergestellt werden, dass es auch bei internen Referenzen nicht zu Kreisen kommt.
Bevor ein `ref`-Pattern während der Normalisierung einen eindeutigen Namen zugewiesen bekommt (siehe Abschnitt 3.6.2), wird innerhalb der Funktion `renameDefines` überprüft, ob der referenzierte `define`-Name innerhalb des Parameters `ref` vorhanden ist (4). Dieser enthält alle `define`-Elemente des aktuellen `grammar`-Pattern. Ist der referenzierte Name in der Liste enthalten, wird dem `ref`-Muster der neue und eindeutige Name zugewiesen (6). Zusätzlich wird der ursprüngliche Name in einem weiteren Attribut `RelaxDefineOriginalName` gesichert (5). Dies ist notwendig, um dem Anwender bei eventuell später folgenden Fehlern eine aussagekräftige Meldung mit dem ursprünglichen Namen zur Verfügung zu stellen. Am Ende der Normalisierung wird das hinzugefügte Attribut wieder aus dem Schema entfernt.
Ist hingegen kein `define`-Element mit dem gesuchten Namen im Schema vorhanden, wird eine Fehlermeldung generiert (7).

```
1  renameDefines ref parentRef -- :: RefList -> RefList -> IOSArrow XmlTree ...
2   = processChildren $ choiceA [
3     (isElem >>> hasName "ref")
4      :-> (ifA (getAttrValue "name" >>> isA (\n -> (elem n (map fst ref))))
5            (addAttr "RelaxDefineOriginalName" $< (getAttrValue "name") >>>
6             addAttr "name" $< (getAttrValue "name">>>arr(\n -> lookup n ref)))
7            (mkRelaxError "" $< (getAttrValue "name" >>>
8                                 arr(\n -> "Define-Pattern with name " ++ n ++
9                                     " referenced in ref not found in schema"))
10      )), ... ]
```

Beispiel 5.6: Test von internen Referenzen

Die Bestimmung von Endlosschleifen in `ref`-Pattern ist identisch mit dem im vorherigen Abschnitt beschriebenen Algorithmus.
Es wird auch hier eine zusätzliche Liste, sämtlicher bereits expandierter Namen mitgeführt. Kommt der aktuelle Name in der Liste vor, ist ein Kreis gefunden worden und die weitere Verarbeitung wird mit einer Fehlermeldung abgebrochen.

5.1.5 Kinder der `grammar`-Elemente

Jedes `grammar`-Pattern muss als Kind ein `start`-Element besitzen, welches den Einstiegspunkt in den `grammar`-Inhalt festlegt.

```
(mkRelaxError "" "A grammar must have a start child element") `when`
(isElem >>> hasName "grammar" >>> neg (getChildren >>> hasName "start"))
```

Beispiel 5.7: Test auf `grammar`- mit folgendem `start`-Element

5.1.6 Kombinieren von `define`- und `start`-Pattern

Beim Zusammenfassen von `define`-Pattern mit demselben Namen oder `start`-Pattern, die innerhalb eines `grammar`-Elementes vorkommen, dürfen sich die Werte des `combine`-Attributes nicht unterscheiden. Das in Abschnitt 3.6.1 beschriebene Kombinationsverfahren wird dafür um den Arrow `checkPatternCombine` ergänzt (3).
Der Arrow erhält als Eingabe eine Liste sämtlicher `combine`-Attributwerte, die für das betrachtete `define`-Pattern im Schema vorkommen (2, 5). Die Rückgabe des Arrows besteht in einem Tupel, wobei der erste Teil einen Fehlercode repräsentiert. Die Null entspricht einem Fehler, bei einer Eins ist die Prüfung erfolgreich verlaufen. Das zweite Element des Tupels stellt entweder den `combine`-Wert oder die erzeugte Fehlermeldung dar.

Enthält die Eingabeliste nur einen Eintrag (7), ist das Schema gültig und es muss keine Kombination durchgeführt werden. Ist mehr als eine leere Zeichenkette in der Liste vorhanden (8), existieren zu viele `define`-Pattern ohne `combine`-Attribut. Im dritten Fall werden erst alle leeren Werte und danach doppelte Einträge entfernt (11). Besitzt die Liste im Anschluss daran noch mehr als ein Element, sind unterschiedliche Werte für das `combine`-Attribut angegeben worden, was in einem gültigen Relax NG Schema nicht erlaubt ist. In allen anderen Konstellationen (14) ist das Schema korrekt und es wird der Attributwert für das Zusammenfassen zurückgegeben.

```
1  createPatternElems pattern name
2    = ( (listA (getElems pattern name >>> getAttrValue "combine")) >>>
3        checkPatternCombine pattern name)
4      &&& (listA (getElems pattern name >>> removeAttr "combine")) >>> ...
5  checkPatternCombine :: String -> String -> IOSArrow [String] (Int, String)
6  checkPatternCombine pattern name = choiceA [
7        (isA (\cl -> length cl == 1)) :-> constA (1, ""),
8        (isA (\cl -> (length $ elemIndices "" cl) > 1))
9          :-> constA (0, "More than one " ++ pattern ++ "-Pattern " ++ name ++
10                        " without an combine-attribute in the same grammar"),
11       (isA (\cl -> (length $ nub $ deleteBy (==) "" cl) > 1))
12         :-> arr(\cl -> (0, "Different combine-Attributes " ++ noDoubles cl ++
13                           " in the same grammar")),
14       this :-> arr (\cl -> (1, fromJust $ find (/= "") cl))]
```

Beispiel 5.8: Überprüfen der `combine`-Attributwerte

5.1.7 Einfache Textmeldungen für die Restriktionsprüfungen

Bei der Darstellung von Fehlermeldungen, die während der ersten Kontrollen von Relax NG Beschränkungen auftreten können (siehe 3.5), sind keine weiteren Informationen notwendig. Wird eine der Restriktionen nicht eingehalten, reicht eine einfache textuelle Meldung aus. Der Aufbau der weiteren Fehlerprüfungen des ersten Restriktionsabschnittes ist analog zum dargestellten Beispiel.

```
(mkRelaxError "" $ "An except element that is a child of an anyName " ++
                   "element must not have any anyName descendant elements")
`when`
(isElem >>> hasName "anyName" >>> getChildren >>> isElem >>> hasName "except"
 >>> deep (isElem >>> hasName "anyName"))
```

Beispiel 5.9: Fehlermeldung für eine ungültige Kombination von `except` und `anyName`

5.1.8 Generieren von zusätzlichen Transformationstexten

Für die letzten, nach der vollständigen Normalisierung des Schemas, durchzuführenden Restriktionsprüfungen reicht die Angabe eines einfachen Fehlertextes hingegen nicht mehr aus. Im Verlauf der Transformationen sind eine Reihe von Pattern durch andere Elemente ersetzt worden. Enthält das zu prüfende Schema einen ungültigen Pfad (siehe Abschnitt 3.10.1), wie beispielsweise `oneOrMore//interleave//attribute`, wäre die alleinige Ausgabe der Patternnamen in der Fehlermeldung für den Anwender irreführend, da diese im Original überhaupt nicht vorkommen. Das folgende Beispiel zeigt auf der linken Seite das Ausgangsschema. Die rechte Seite entspricht dem erzeugten Schema nach den Umformungen. Eine Fehlermeldung

der Form *„interleave-Pattern not allowed as a descendent of a oneOrMore-Pattern followed by an attribute descendent“* würde nicht genügen, da weder ein `oneOrMore`- noch ein `interleave`-Pattern im ursprünglichen Schema vorhanden waren.

```
<element name="foo">          |  <element> <name>foo</name>
   <zeroOrMore>               |    <choice>
                              |      <empty/>
                              |      <oneOrMore>
    <mixed>                   |        <interleave>
      <attribute name="b"/>   |          <attribute><name>b</name></attribute>
      <attribute name="c"/>   |          <attribute><name>c</name></attribute>
                              |          <text/>
    </mixed>                  |        </interleave>
  </zeroOrMore>               |      <oneOrMore>
                              |    </choice>
</element>                    |  </element>
```

Beispiel 5.10: Ungültiges Schema vor (links) und nach (rechts) der Transformation

Um dieses Problem zu lösen, werden bei den Transformationsschritten, die Veränderungen an Pattern durchführen, zusätzlich beschreibende Informationen an die bearbeiteten (2) sowie neu entstandenen Elemente angehängt (5).

```
1 mkElement "interleave"
2   (setChangesAttr "mixed is transformed into an interleave")
3   ( getChildren <+>
4     mkElement "text"
5       (setChangesAttr "new text-pattern: mixed is transformed " ++
6                       "into an interleave with text") none)
7 `when` (isElem >>> hasName "mixed")
```

Beispiel 5.11: Ergänzen zusätzlicher Informationen bei Veränderungen des Schemas

Dies erfolgt über den Arrow `setChangesAttr`, der ein Attribut mit dem Namen `relaxSimplificationChanges` erzeugt. Der Wert das Attributes enthält die Transformationsbeschreibung (6). Ist das Element bereits bearbeitet worden und besitzt folglich ein entsprechendes Attribut (3), wird der Wert um die aktuell durchgeführte Änderung ergänzt (4,5). Es entsteht dadurch eine chronologische Reihenfolge der Umwandlungen, die für die Fehlerlokalisierung genutzt werden kann.

```
1 a_relaxSimplificationChanges = "relaxSimplificationChanges" -- :: String
2 setChangesAttr str -- :: String -> IOSArrow XmlTree XmlTree
3   = ifA (hasAttr a_relaxSimplificationChanges)
4       ( processAttrl (changeAttrValue (++ (", " ++ str))
5         `when` (hasName a_relaxSimplificationChanges)))
6       (mkAttr (QN "" a_relaxSimplificationChanges "") (txt str))
```

Beispiel 5.12: Hinzufügen des Attributes für die Beschreibung von Umwandlungen

Wird eine Restriktion nicht eingehalten, werden die erzeugten Transformationsbeschreibungen zum Generieren der Fehlermeldung eingesetzt. Die Funktion `mkRelaxError` besitzt dafür zwei Parameter (2). Der erste beschreibt die Veränderungen des aktuellen Elementes (3). Der zweite enthält die grundlegende Fehlermeldung (6,7) inklusive der Umwandlungstexte weiterer beteiligter Pattern (5). Da für die bisher betrachteten Situationen eine einfache Textausgabe ausreichte, war der erste Parameter von `mkRelaxError` immer leer und es wurde nur der

zweite verwendet.
Zusätzlich wird sichergestellt, dass bereits vorhandene Fehler unterhalb des aktuellen Elementes nicht durch den neu zu erzeugenden Fehler überschrieben werden (1).

```
deep (isElem >>> hasName "relaxError") <+>
(mkRelaxError $<<
  (getChangesAttr &&&
   (listA (getChildren >>> deep (checkElemName ["group", "interleave"]) >>>
    (getName &&& getChangesAttr >>> arr2 (++))) &&& getChangesAttr
   >>> arr2 (\n c -> formatStringList "-, " n ++ "-pattern not allowed as "++
           "descendent of a oneOrMore" ++ c ++" followed by an attribute"))))
`when` invalidPath
```

Beispiel 5.13: Generieren der Fehlermeldungen

Für das Beispiel 5.10 wird anschließend die Fehlermeldung *„interleave (mixed is transformed into an interleave)-pattern not allowed as a descendent of a oneOrMore-pattern (zeroOrMore is transformed into a choice between oneOrMore and empty) followed by an attribute descendent“* erzeugt. Die einzelnen Veränderungsschritte werden dabei in Klammern hinter sämtlichen wichtigen Elementen angegeben.
Da durch die zusätzlichen Informationen die Fehlermeldung deutlich länger wird und erfahrene Relax NG Anwender sie häufig nicht benötigen, ist die Ausgabe standardmäßig abgeschaltet. Sie kann aber jederzeit über den Aufrufparameter `output-pattern-transformations` wieder aktiviert werden (2).

```
getChangesAttr -- :: IOSArrow XmlTree String
  = getAttrValue a_relaxSimplifChanges &&& getParamString a_output_changes
    >>> ifP (\(changes, param) -> changes /= "" && param == "1")
          (arr2 (\l _ -> " (" ++ l ++ ")")) (constA "")
```

Beispiel 5.14: Ausgabe der Transformationsbeschreibungen

Vor dem Aufbau der Pattern-Datenstruktur wird das `relaxSimplificationChanges`-Attribut wieder aus dem Baum entfernt. Es ist für die folgenden Schritte nicht mehr von Bedeutung.

5.1.9 Sammeln und Ausgeben von Fehlern

Für den Normalisierungsprozess kann über den Aufrufparameter `do-not-collect-errors` festgelegt werden, dass die Transformationen des Schemas bereits nach dem ersten Fehler stoppen. Standardmäßig wird die Umwandlung fortgesetzt und sämtliche noch folgenden Fehler gesammelt. Ein sofortiger Abbruch kann aber vor allem bei größeren Schemata sinnvoll sein.
Bei jedem Aufruf von `mkRelaxError` wird der globale Zustand `numberOfErrors` um den Wert Eins erhöht: `perform (getAndSetCounter a_numberOfErrors)`. Vor jedem Transformationsschritt wird über den Arrow `collectErrors` geprüft (1), ob Fehler gesammelt werden sollen. Ist dies nicht der Fall (4) und wurde bereits ein Fehler entdeckt (3), wird die Transformation übersprungen (1).

```
simplificationStep8 = ( ... ) `when` collectErrors
collectErrors -- :: IOSArrow XmlTree XmlTree
 = none `when` (stopAfterFstE >>> getParamInt 0 a_numberOfErrors >>> isA (>0))
stopAfterFstE = getParamString a_do_not_collect_errors >>> isA (== "1")
```

Beispiel 5.15: Sammeln von Fehlern

Durch das Sichern der erkannten Fehleranzahl in einem globalen Zustand braucht der Baum nicht über den `deep`-Arrow nach vorhandenen Fehlerknoten durchsucht werden, was einen erheblichen Performanzgewinn bringt.

Abschließend können alle entstandenen Fehler über die Funktion `getErrors` ausgegeben werden. Der Arrow erzeugt einen neuen XML-Baum, der sämtliche im Schema vorhandenen Fehlermeldungen enthält. Sind keine Probleme aufgetreten, wird der `none`-Arrow als Ergebnis geliefert.

```
getErrors = (getParamInt 0 a_numberOfErrors >>> isA (>0))
            `guards` (root [] [multi (isElem >>> hasName "relaxError")])
```

Beispiel 5.16: Ausgabe der Fehler

5.2 Benötigte Fehlerstrukturen im `Pattern`-Datentyp

Beim Aufbau des `Pattern`-Datentyps werden zwei Typkonstruktoren für die Darstellung von Fehlermeldungen eingesetzt. Das ist zum einen der `NotAllowed`-Konstruktor, über den alle ungültigen Relax NG Pattern Kombinationen und Fehler abgebildet werden, die während der Normalisierung auftreten. Als Zweites kommt `NCError String` als Bestandteil des `NameClass`-Datentyp zum Einsatz.
Wird die Normalisierung mit den Restriktionsprüfungen und dem Validieren von externen Schemata durchgeführt, die Aufrufparameter `do-not-check-restrictions`, `do-not-validate-externalRef`, `do-not-validate-include` sind also nicht gesetzt, und findet am Anschluss daran ein Test und eventueller Abbruch bei vorhandenen Fehler statt, kann es beim Erzeugen des `Pattern`-Datentyps nicht zu Fehlern kommen. Alle möglichen Probleme werden bereits vorher erkannt.
Die Kontrollen innerhalb des Moduls `CreatePattern` sind von daher nicht zwingend erforderlich. Sie wurden jedoch trotzdem implementiert, um einen sinnvollen Programmausstieg zu gewährleisten, wenn die oben genannten Tests nicht korrekt durchgeführt worden sind. Des Weiteren sind sie für die Typ-Vollständigkeit der Haskell-Funktionen beim Kompilieren mit dem GHC Parameter `Wall` notwendig. Ohne die Fehlerbehandlung würden einige Konstruktoren der eingesetzten Datentypen nicht abgebildet werden.

5.2.1 Darstellen ungültiger Pattern

Das Pattern `NotAllowed` schlägt während der Validierung gegen ein Instanzdokument immer fehl und wurde der Relax NG Spezifikation hinzugefügt, um Bereiche zu kennzeichnen, die durch Referenzen oder externe Schemata überschrieben werden sollen. Es ermöglicht die abstrakte Deklaration von Elementen, die später durch eine konkrete Schema Definition ersetzt werden.

```
<grammar> <!-- Schema mit abstrakter Definition -->
  ... <include name="bar.rng"/> ...
  <define name="foo" combine="choice"> <notAllowed/> </define>
</grammar>

<grammar> <!-- externes Schema mit konkreter Definition: bar.rng -->
  ... <define name="foo"> <element name="foo"/> </define> ...
</grammar>
```

Beispiel 5.17: Abstrakte und konkrete Schema Definition

Am Ende der Normalisierungsphase sollte in einem für den realen Einsatz konzipierten Relax NG Schema kein `NotAllowed`-Pattern mehr vorkommen, da anderenfalls jede Instanz Validierung gegen das Schema fehlschlägt.
Innerhalb des `Pattern`-Datentyps ist der Konstruktor für `NotAllowed` um einen zusätzlichen Wert vom Typ `String` ergänzt worden (1). Dieser existiert im definierten Datentyp von James Clark [Clark ARV 02] nicht und wird im Relax NG Modul der Haskell XML Toolbox zur Repräsentation von Fehlermeldungen eingesetzt. Hierdurch ist kein weiterer eigenständiger Konstruktor für den Fehlerfall notwendig.
Trifft der Algorithmus beim Erzeugen des `Pattern`-Datentyps aus dem XML-Baum auf ein `notAllowed`-Element, wird der Text *„notAllowed-pattern in schema definition“* an den `NotAllowed`-Konstruktor übergeben. Dadurch wird kenntlich gemacht, dass nach der Normalisierung noch ein `notAllowed`-Element im Schema vorhanden ist (6). Falls das Schema noch Elemente enthält, die in der vereinfachten Grammatik nicht mehr vorkommen, wird `mkRelaxError` zur weiteren Verarbeitung aufgerufen (8).

```
1 data Pattern = NotAllowed String | ...
2 createPatternFromXml env -- :: Env -> LA XmlTree Pattern
3  = choiceA [
4      (isElem >>> hasName "empty")      :-> constA Empty,
5      (isElem >>> hasName "notAllowed")
6           :-> constA (NotAllowed "notAllowed-pattern in schema definition"),
7      ...
8      this                              :-> mkRelaxError]
```

Beispiel 5.18: Erzeugen von unterschiedlichen Pattern aus XML-Knoten

Handelt es sich bei dem nicht erkannten XML-Knoten um ein `relaxError`-Element, wird die während des Normalisierungsprozesses erzeugte Fehlermeldung einfach weitergegeben (2). Für jedes andere Element oder Attribut, erfolgt der Hinweis, dass dieses in einem gültigen Relax NG Schema nicht erlaubt ist (3,5). Der Typ `isError` wird von den Funktionen der Haskell XML Toolbox verwendet und der enthaltene Fehlertext ebenfalls nur weitergereicht (7). In allen anderen Fällen kann das Problem nicht näher beschrieben werden (8).

```
1 mkRelaxError = choiceA [ -- :: LA XmlTree Pattern
2  (isElem>>> hasName "relaxError") :-> (getAttrValue "desc">>>arr NotAllowed),
3  isElem  :-> (getName >>> arr (\n -> NotAllowed $ "Pattern " ++ n ++
4                                      " is not allowed in Relax NG Schema")),
5  isAttr  :-> (getName >>> arr (\n -> NotAllowed $ "Attribute " ++ n ++
6                                      " is not allowed in Relax NG Schema")),
7  isError :-> (getErrorMsg >>> arr NotAllowed),
8  this    :-> arr (\e -> NotAllowed $ "Can't create Pattern from " ++ show e)]
```

Beispiel 5.19: Fehlermeldung von `mkRelaxError`

Beim Transformieren von `ref`-Pattern kann es vorkommen, dass das referenzierte Element in der aufgebauten Umgebungstabelle (siehe 4.2.4) nicht vorkommt. Liefert die Funktion `lookup` den Wert `Nothing` zurück, wird ein `NotAllowed`-Konstruktor erzeugt.

```
1 mkRelaxRef e -- :: Env -> LA XmlTree Pattern
2  = getAttrValue "name" >>>
3    arr (\name -> fromMaybe (NotAllowed $ "Define-Pattern with name " ++
4                  name ++ " not found") . lookup name $ transformEnv e)
```

Beispiel 5.20: Transformation von `ref`-Pattern

5.2.2 Abbilden fehlerhafter Namensklassen

Die Fehlerbehandlung beim Erzeugen des `NameClass`-Datentyps verläuft analog zum beschriebenen Verfahren für den `Pattern`-Datentyp. Handelt es sich bei dem betrachteten XML-Knoten um ein Element, welches keine Namensklasse abbildet (3), übernimmt der `mkNameClassError`-Arrow das Erstellen der Meldung. Die implementierten Unterscheidungen entsprechen dabei vollständig `mkRelaxError`. Es wird jedoch der `NCError`-Konstruktor verwendet, der dem `NameClass`-Datentyp hinzugefügt wurde.

```
createNameClass = choiceA [ -- :: LA XmlTree NameClass
  (isElem >>> hasName "anyName") :-> processAnyName,
  ...
  this                           :-> mkNameClassError]
mkNameClassError = choiceA [ -- :: LA XmlTree NameClass
 (isElem >>> hasName "relaxError") :-> (getAttrValue "desc" >>> arr NCError),
 ...
 this    :-> arr (\e ->  NCError $ "Can't create NameClass from " ++ show e)]
```

Beispiel 5.21: Fehlermeldung von `mkNameClassError`

6 Validieren eines Relax NG Schemas gegen ein Instanzdokument

Die im vorherigen Kapitel aufgebaute `Pattern`-Datenstruktur bildet die Basis für den Validierungsprozess eines Relax NG Schemas gegen ein XML-Instanzdokument, welcher Inhalt der folgenden Abschnitte ist.
Die Grundidee des eingesetzten Algorithmus basiert auf der von Janusz A. Brzozowski bereits im Jahr 1964 entwickelten Technik für die *Ableitung von regulären Ausdrücken* [Brzozowski 64].

6.1 Ableitung regulärer Ausdrücke

Das Validieren des Inhalts eines XML-Dokumentes gegen ein Schema kann zu der Problemklasse des *„regular expression matching“* gezählt werden. Hierbei ist die Frage zu beantworten, ob zu einem gegebenen regulären Ausdruck `e` und einer Zeichenkette `s` die Zeichenkette zu der vom regulären Ausdruck akzeptierten Sprache gehört, also $s \in L(e)$ gilt. Die Arbeitsweise des Algorithmus lässt sich auf die folgende Art beschreiben[1].

Sei `A` ein Alphabet[2], `L` eine Sprache über A ($L \subseteq A*$) und `s` ein Wort aus `A*`. Die Ableitung von `L` in Bezug auf `s` ist definiert als:

$$s \setminus L = \{ w : sw \in L \}$$

Die resultierende Sprache $s \setminus L$ enthält die Suffixe sämtlicher Wörter aus `L`, die mit dem Präfix `s` beginnen. Informell ausgedrückt bedeutet dies, dass s von allen Worten der Sprache `L` die mit `s` anfangen entfernt wird und der Rest zu $s \setminus L$ gehört.

Für einen regulären Ausdruck `e` und ein Symbol `x` aus dem Alphabet wird ein neuer regulärer Ausdruck $\Delta\ e\ x$ berechnet, so dass gilt:

$$L(\Delta\ e\ x) = x \setminus L(e)$$

Eine Zeichenkette s gehört genau dann zu L(e), wenn das leere Wort in $s \setminus L(e)$ enthalten ist

$$\epsilon \in s \setminus L(e)$$

beziehungsweise der Ausdruck in beliebig vielen Schritten auf das leere Wort abgeleitet werden kann:

$$\Delta* e\ x \vdash \epsilon$$

Beispiele

Der reguläre Ausdruck `foobar` soll in Bezug auf die Zeichenkette „`foo`“ abgeleitet werden. Nachdem der Ausdruck auf das Wort angewendet wurde, bleibt `bar` als neu entstandener

[1] Vgl. http://www.flightlab.com/~joe/sgml/validate.html
[2] eine endliche nicht leere Menge von Zeichen

regulärer Ausdruck übrig. Dieser lässt sich nicht auf das leere Wort ableiten. Daraus folgt, dass die Zeichenkette „`foo`“ nicht durch den regulären Ausdruck `foobar` beschrieben wird.
Im zweiten Beispiel soll `aab*` in Bezug auf „`aabbb`“ abgeleitet werden. Als Ergebnis der Δ-Produktion entsteht `b*`. Das Resultat `b*` kann anschließend auf die leere Zeichenkette abgeleitet werden, da der Stern für Null oder mehr `'b'`-Zeichen steht. Der reguläre Ausdruck passt somit auf das Wort.

6.2 Algorithmus zum Validieren von Relax NG

James Clark nutzt die von J. Brzozowski beschriebene Technik als Basis für seinen in Java implementierten Relax NG Validator Jing [Jing 03]. Zum Erläutern des dort eingesetzten Algorithmus hat er eine Reihe von Haskell Funktionen und Datentypen entwickelt, die das Validieren eines Relax NG Schemas gegen ein XML-Dokument vollständig abbilden [Clark ARV 02]. Das Schema sowie das XML-Dokument werden dabei schrittweise gegeneinander verglichen und reduziert. Am Ende wird überprüft, ob einerseits das Instanzdokument vollständig durch des Schema beschrieben wurde und andererseits der noch vorhandene Rest der Schemas auf das leere Pattern abgeleitet werden kann.
Die innerhalb dieser Arbeit durchgeführte Implementierung des Moduls `Validation` für die Haskell XML Toolbox erweitert den Algorithmus von J. Clark um eine Fehlerbehandlung. Zudem ist eine Anpassung der Datentypen an die vorhandenen Strukturen der Toolbox durchgeführt worden. Die Fehlermeldungen werden analog zum Verfahren beim Aufbau des `pattern`-Datentyps (siehe Abschnitt 5.2.1) über das `notAllowed`-Pattern abgebildet.

6.2.1 Benötigte Hilfsfunktionen

Die Entscheidung, ob ein regulärer Ausdruck das leere Wort enthält, übernimmt für das Validieren von Schemata die Funktion `nullable`. Sie bestimmt, ob ein Pattern auf die leere Sequenz abgeleitet werden kann. Für das `Group`- und `Interleave`-Pattern ist dies der Fall, wenn jeweils beide Kinder auf das leere Pattern abgeleitet werden können. Für `Choice` muss dies nur für eines der Kinder gelten. Die `Empty`- und `Text`-Muster sind die beiden einzigen Elemente für die `nullable` den Wert `True` zurückliefert. Ein `Text`-Pattern kann auf das leere Element zurückgeführt werden, da es null oder mehr Textknoten im Instanzdokument repräsentiert.
Alle im Beispiel fehlenden Pattern (`Attribute`, `Data`, usw.) sind hingegen nicht auf die leere Sequenz ableitbar.

```
nullable:: Pattern -> Bool
nullable (Group p1 p2) = nullable p1 && nullable p2
nullable (Interleave p1 p2) = nullable p1 && nullable p2
nullable (Choice p1 p2) = nullable p1 || nullable p2
nullable (OneOrMore p) = nullable p
nullable Empty = True
nullable Text = True
nullable (Element _ _) = False
...
```

Beispiel 6.1: Prüfen, ob ein Pattern auf die leere Sequenz abgeleitet werden kann

Für das Validieren von Elementen und Attributen muss geprüft werden, ob sich deren Name innerhalb der im Schema definierten Namensklasse befindet. Für `AnyName` ist dies immer der Fall; bei `AnyNameExcept` darf der Instanzname nicht Teil der einschränkenden Klasse sein,

anderenfalls ist sein Wert beliebig. Ist im Schema ein `NsName`-Pattern vorhanden, muss der Namensraum des Elementes beziehungsweise des Attributes mit dem angegebenen Wert identisch sein. Der lokale Anteil ist nicht relevant. `NsNameExcept` verhält sich analog zu `AnyNameExcept`. Für die `Name`-Namensklasse ist es notwendig, dass der lokale Teil und die Namensraum-URI übereinstimmen. Sind mehrere mögliche Klassen über ein `NameClassChoice` deklariert, muss sich der Instanzname in einem von ihnen befinden, um gültig zu sein.

```
contains :: NameClass -> QName -> Bool
contains AnyName _ = True
contains (AnyNameExcept nc) n = not (contains nc n)
contains (NsName ns1) (QN _ _ ns2) = ns1 == ns2
contains (NsNameExcept ns1 nc) qn@(QN _ _ ns2)
   = ns1 == ns2 && not (contains nc qn)
contains (Name ns1 ln1) (QN _ ln2 ns2) = (ns1 == ns2) && (ln1 == ln2)
contains (NameClassChoice nc1 nc2) n = (contains nc1 n) || (contains nc2 n)
contains (NCError _) _ = False
```

Beispiel 6.2: Test, ob ein Name in einer Namensklasse liegt

Während der Berechnung von Ableitungen kann es notwendig werden, neue `Choice`-, `Group`-, `Interleave`-, `OneOrMore`- oder `After`-Pattern zu erzeugen. Hierfür sind fünf Konstruktorfunktionen implementiert worden. Abhängig vom Pattern sind jedoch nicht alle Kombinationen der Parameter sinnvoll. Ist eine Auswahl zwischen einem beliebigen Pattern p und `NotAllowed` zu erstellen, muss das `NotAllowed`-Pattern nicht weiter betrachtet werden, da es bei der Auswahl ohnehin ignoriert würde. Die Rückgabe von p reicht in diesem Fall aus.
Enthält ein `Group`-, `Interleave`- oder `OneOrMore`-Pattern hingegen einen `NotAllowed`-Operanden, ist das Ergebnis der Konstruktorfunktion ebenfalls `NotAllowed`. Das `Empty`-Element verhält sich für die eben genannten Pattern entsprechend dem `NotAllowed`-Element bei einem `Choice`-Pattern und muss nicht weiter betrachtet werden.

```
choice, group, interleave, oneOrMore :: Pattern -> Pattern -> Pattern
choice p (NotAllowed _) = p
choice (NotAllowed _) p = p
choice p1 p2 = Choice p1 p2

group _ n@(NotAllowed _) = n
group n@(NotAllowed _) _ = n
group p Empty = p
group Empty p = p
group p1 p2 = Group p1 p2
-- interleave wird identisch zu group behandelt

oneOrMore n@(NotAllowed _) = n
oneOrMore p = OneOrMore p
```

Beispiel 6.3: Konstruktorfunktionen

6.2.2 Zentrale Validierungsfunktion

Aus Performanzgründen ist es von Vorteil, wenn die Validierung mit einem einzigen Durchlauf über die aufgebaute Pattern-Datenstruktur möglich ist. Um dies zu erreichen, wird ein XML-Element in fünf Komponenten aufgeteilt.

1. ein öffnendes Start-Tag inklusive des vollständig qualifizierten Namens

2. null oder mehr Attribute
3. ein schließendes Start-Tag
4. null oder mehr Kindelemente
5. ein Ende-Tag

Die vollständige Ableitung eines Relax NG Pattern in Bezug auf ein Element entspricht der Ableitung des Pattern gegen jede der fünf Komponenten in der angegebenen Reihenfolge. Die zentrale Funktion, welche diese Aufgabe übernimmt, ist `childDeriv`. Handelt es sich bei dem aktuell zu verarbeitendem Element um einen Textknoten, ist nur die enthaltene Zeichenkette von Bedeutung. Weitere Attribute und Kindknoten können bei einem Textknoten nicht vorhanden sein. Für ein XML-Element werden schrittweise die fünf Komponenten berechnet.

```
childDeriv :: Context -> Pattern -> XmlTree -> Pattern
childDeriv cx p (NTree (XText s) _) = textDeriv cx p s
childDeriv _ p (NTree (XTag qn atts) children) =
  let p1 = startTagOpenDeriv p qn
      p2 = attsDeriv cx p1 atts
      p3 = startTagCloseDeriv p2
      p4 = childrenDeriv cx p3 children
  in endTagDeriv p4
```

Beispiel 6.4: Zentrale Validierungsfunktion

6.2.3 Ableitung von Textknoten

Die Ableitung eines `Choice`-Pattern gegen einen Text wird durchgeführt, indem die Funktion `textDeriv` auf beide Kinder angewendet wird (2). Bei `Interleave` besteht ebenfalls eine Auswahl zwischen den beiden Kinder, allerdings wird für jedes Kindelement erneut ein `Interleave`-Pattern, mit wechselnder Position der `textDeriv`-Funktion, erzeugt. Für das `Group`-Pattern hängt das Ergebnis davon ab, ob das erste Kindelement auf die leere Sequenz abgeleitet werden kann. Ist dies möglich, muss die Zeichenkette zusätzlich gegen das zweite Kind getestet werden (7). Obwohl der erste Operand auf das leere Pattern abzuleiten ist, darf er jedoch nicht entfernt werden, da er noch weitere relevante Informationen beinhalten kann. Ist p hingegen nicht auf die leere Folge abzubilden, dann ist die Zeichenkette nur auf das erste Kind des `Group`-Pattern anzuwenden (6).
`OneOrMore` wird zu einer Gruppe expandiert. Der zweite Parameter der Gruppe ist eine Auswahl zwischen dem original `OneOrMore`-Pattern und `Empty`. Hierdurch wird sichergestellt, dass das Element mehr als einmal auftreten kann, aber nicht muss (9).

```
1 textDeriv :: Context -> Pattern -> String -> Pattern
2 textDeriv cx (Choice p p2) s = choice (textDeriv cx p s) (textDeriv cx p2 s)
3 textDeriv cx (Interleave p p2) s =
4  choice (interleave (textDeriv cx p s) p2) (interleave p (textDeriv cx p2 s))
5 textDeriv cx (Group p1 p2) s =
6   let p = group (textDeriv cx p1 s) p2
7   in if nullable p1 then choice p (textDeriv cx p2 s) else p
8 textDeriv cx (OneOrMore p) s =
9   group (textDeriv cx p s) (choice (OneOrMore p) Empty)
```

Beispiel 6.5: Ableitung von Text (Teil 1)

Da das `Text`-Element null oder mehr Textknoten repräsentiert, ist die Ableitung erneut ein `Text`- und kein `Empty`-Pattern (1). Um zu testen, ob ein `List`-Pattern auf einen Textknoten

passt, wird der Wert des Textknotens in einzelne Abschnitte aufgeteilt (3). Die Trennung erfolgt anhand der Leerraumzeichen. Die Einträge in der resultierenden Liste werden anschließend jeweils gegen das `List`-Pattern geprüft (11). Sind alle Elemente gültig, wird `Empty` zurückgegeben, anderenfalls `NotAllowed` mit einer entsprechenden Fehlermeldung (4).
Für die Verarbeitung von `Value`-, `Data`- und `DataExcept`-Pattern sind Datentyp-Bibliotheken notwendig. Diese werden detailliert im Kapitel 7 betrachtet. Ein `NotAllowed`-Pattern wird unverändert zurückgeliefert (6). In allen anderen Situationen kann das Pattern nicht erfolgreich gegen den Text abgeleitet werden (7).

```
1  textDeriv _ Text _ = Text
2  textDeriv cx (List p) s
3    = if nullable (listDeriv cx p (words s)) then Empty
4      else NotAllowed $ "List with value(s) " ++ show p ++ " expected, but " ++
5                        "value(s) " ++ (words s) ++ " found"
6  textDeriv _ n@(NotAllowed _) _ = n
7  textDeriv _ p s = NotAllowed $ "Element " ++ getPatternName p ++
8                                  "expected, but text " ++ s ++ " found"
9
10 listDeriv _ p [] = p -- :: Context -> Pattern -> [String] -> Pattern
11 listDeriv cx p (x:xs) = listDeriv cx (textDeriv cx p x) xs
```

Beispiel 6.6: Ableitung von Text (Teil 2)

6.2.4 Verarbeiten des öffnenden Start-Tags

Das Ableiten eines Pattern gegen ein XML-Element beginnt mit der Verarbeitung des öffnenden Start-Tags und dem darin enthaltenen Elementnamen. Um eine Validierung der Struktur mit nur einem Durchlauf zu ermöglichen, ist der Hilfskonstruktor `After Pattern Pattern` notwendig. Dieser ist nicht in der Relax NG Spezifikation definiert und wird nur für die Validierung benötigt. Ein Pattern `After x y` trifft auf ein Element x, gefolgt von einem Ende-Tag, gefolgt von y, zu.
Soll im folgenden Beispiel das `bar`-Element des Instanzdokumentes (rechte Seite) gegen `<element> <anyName/>` des Schemas (linke Seite) getestet werden, ist es notwendig, die vorher definierten Pattern `oneOrMore` und `group` zu sichern, da sie für die Verarbeitung der restlichen Elemente (`baz`, `xyz`) des XML-Dokumentes benötigt werden. Das `After`-Pattern verlagert diese Validierung nach hinten, so dass mit dem Elementnamen begonnen werden kann, aber keine Schema Informationen verloren gehen.

```
...                                                 |  ...
  <oneOrMore>                                       |    <bar/>
    <group>                                         |    <baz>Lorem ipsum</baz>
      <element> <anyName/> <text/> </element>       |    dolor sit amet
      <optional> <text/> </optional>                |    <xzy/>
    </group>                                        |  ...
  </oneOrMore>                                      |
...                                                 |
```

Beispiel 6.7: Relax NG Schema und Instanzdokument

Die Ableitung des öffnenden Start-Tags und des Namens eines XML-Elementes gegen ein `Choice`-Pattern erfolgt auf dieselbe Weise wie für Textknoten. Der Name wird gegen beide Operanden getestet (3). Stimmt für ein `Element`-Pattern die angegebene Namensklasse mit dem Instanznamen überein, wird ein `After`-Pattern erzeugt (5). Der zweite Parameter ist

dabei `Empty`, da keine weiteren Informationen bei einem `Element`-Pattern gesichert werden müssen. Befindet sich der Name nicht in der Namensklasse, wird eine Fehlermeldung erzeugt (6). Ein `NotAllowed`-Element wird erneut nur weitergereicht (8).

```
1 startTagOpenDeriv :: Pattern -> QName -> Pattern
2 startTagOpenDeriv (Choice p1 p2) qn =
3   choice (startTagOpenDeriv p1 qn) (startTagOpenDeriv p2 qn)
4 startTagOpenDeriv (Element nc p) qn@(QN _ local uri) =
5   if contains nc qn then after p Empty
6   else NotAllowed $ "Element with name " ++ nc ++ " expected, "
7                     "but {" ++ uri ++ "}" ++ local ++ " found"
8 startTagOpenDeriv n@(NotAllowed _) _ = n
```

Beispiel 6.8: Verarbeiten des öffnenden Start-Tags (Teil 1)

Die Validierung eines XML-Elementes gegen `Interleave`, `OneOrMore` und `Group` entspricht der Ableitung gegen einen Textknoten, jedoch mit dem entscheidenden Unterschied, dass das Ergebnis des rekursiven Aufrufs von `startTagOpenDeriv` nicht direkt an die Konstruktorfunktionen weitergegeben wird. Stattdessen werden die Konstruktoren über die Funktion `applyAfter` in den zweiten Operanden des `After`-Pattern verlagert und ihre Berechnung dadurch verzögert.
Alle weiteren Pattern sind an dieser Stelle nicht erlaubt und resultieren in einem Fehler.

```
startTagOpenDeriv (Interleave p1 p2) qn =
 choice (applyAfter (flip interleave p2) (startTagOpenDeriv p1 qn))
        (applyAfter (interleave p1) (startTagOpenDeriv p2 qn))
startTagOpenDeriv (OneOrMore p) qn =
 applyAfter (flip group (choice (OneOrMore p) Empty))(startTagOpenDeriv p qn)
startTagOpenDeriv (Group p1 p2) qn =
 let x = applyAfter (flip group p2) (startTagOpenDeriv p1 qn)
 in if nullable p1 then choice x (startTagOpenDeriv p2 qn) else x
startTagOpenDeriv (After p1 p2) qn =
 applyAfter (flip after p2) (startTagOpenDeriv p1 qn)
startTagOpenDeriv _ (QN _ local uri) =  NotAllowed $ show p ++ " expected," ++
                            "but Element {" ++ uri ++ "}" ++ local ++ " found"
```

Beispiel 6.9: Verarbeiten des öffnenden Start-Tags (Teil 2)

Die funktion `applyAfter` erhält als ersten Parameter eine Funktion und wendet diese auf den zweiten Operanden des `After`-Pattern an (2).
Die Konstruktion des `After`-Pattern bedingt, dass es nur an bestimmten Stellen innerhalb der Pattern-Datenstruktur auftreten kann. Es ist lediglich als Nachfolger eines `Choice`- oder eines anderen `After`-Pattern möglich. Zudem kann der erste Operand beziehungsweise ein Nachfolger des ersten Operanden niemals ein `After`-Pattern sein. Ist der zweite Parameter von `applyAfter` somit kein `Choice`- oder `After`-Pattern, muss eine Fehlermeldung generiert werden (5).

```
1 applyAfter :: (Pattern -> Pattern) -> Pattern -> Pattern
2 applyAfter f (After p1 p2) = after p1 (f p2)
3 applyAfter f (Choice p1 p2) = choice (applyAfter f p1) (applyAfter f p2)
4 applyAfter _ n@(NotAllowed _) = n
5 applyAfter _ _ = NotAllowed "Call to applyAfter with wrong arguments"
```

Beispiel 6.10: Verzögern der Berechnung von Pattern

6.2.5 Validieren der Attribute

Im nächsten Schritt sind die vorhandenen Attribute zu validieren. Die Ableitung eines Pattern gegen eine Liste von Attributen entspricht der sequenziellen Ableitung gegen jedes Element der Liste (2). Die Implementierung für `After`-, `Choice`-, `OneOrMore`- und `Interleave`-Pattern in der `attDeriv`-Funktion ist identisch zu `textDeriv`. Für das `Group`-Pattern (9) unterscheiden sich die beiden Funktion allerdings, da bei Attributen die Reihenfolge, in der sie im Instanzdokument angegeben wurden, nicht von Bedeutung ist. Die Kombinationsmöglichkeiten der beiden `Group`-Operanden entsprechen somit denen des `Interleave`-Pattern (6).

```
1 attsDeriv _ p [] = p -- :: Context -> Pattern -> XmlTrees -> Pattern
2 attsDeriv cx p (a@(NTree (XAttr _) _):xs) = attsDeriv cx (attDeriv cx p a) xs
3
4 attDeriv :: Context -> Pattern -> XmlTree -> Pattern
5 attDeriv cx (Interleave p1 p2) att =
6   choice (interleave (attDeriv cx p1 att) p2)
7          (interleave p1 (attDeriv cx p2 att))
8 attDeriv cx (Group p1 p2) att =
9   choice (group (attDeriv cx p1 att) p2) (group p1 (attDeriv cx p2 att))
```

Beispiel 6.11: Validieren der Attribute (Teil 1)

Für den Vergleich gegen ein `Attribute`-Pattern wird der Wert des Attributes im Instanzdokument berechnet (2). Passen entweder der Attributname oder der Wert nicht auf das angegebene Pattern (4), wird abhängig davon, welcher Teil nicht korrekt war, eine Fehlermeldung generiert (6,8). Ein Attributwert ist gültig, wenn er nur aus Leerraumzeichen besteht und das verbleibende Pattern auf die leere Sequenz abgeleitet werden kann. Bei einem nicht leeren Wert erfolgt eine textuelle Ableitung (11).

```
1 attDeriv cx (Attribute nc p) (NTree (XAttr qn) attrValue)
2  = attDeriv' (xshow attrValue)
3 attDeriv' val
4   = if contains nc qn && valueMatch cx p val then Empty
5     else ( if (not $ contains nc qn)
6            then ( NotAllowed $ "Attribut with name " ++
7                   show nc ++ " expected, but " ++ qn2String qn ++ " found")
8            else ( NotAllowed $ "Attributvalue " ++ val ++
9                   " expected, but " ++ show p ++ " found"))
10 valueMatch cx p s = -- :: Context -> Pattern -> String -> Bool
11   (nullable p && whitespace s) || nullable (textDeriv cx p s)
```

Schließendes Start-Tag

Die Bearbeitung des schließenden Start-Tags erfolgt ebenfalls nach dem bereits bekannten `textDeriv`-Schema. Lediglich für das `Attribute`-Pattern ist zusätzlich eine Fehlermeldung einzubauen, da keine weiteren Attribute im Instanzdokument mehr vorkommen können, wenn das schließende Start-Tag zu verarbeiten ist.

```
startTagCloseDeriv :: Pattern -> Pattern
startTagCloseDeriv (Choice p1 p2) =
  choice (startTagCloseDeriv p1) (startTagCloseDeriv p2)
...
startTagCloseDeriv (Attribute nc _)
```

```
  = NotAllowed $ "Attribut with name, " ++ show nc ++
                 " expected, but no more attributes found"
```

Beispiel 6.12: Validieren des schließenden Start-Tags

6.2.6 Reduzieren der Kindelemente und des Ende-Tags

Das Reduzieren der Patternstruktur gegen sämtliche Kinder eines Elementes wird erneut auf das Validieren gegen jedes einzelne Kind in der vorgegebenen Reihenfolge zurückgeführt. Besitzt das Element keine Kinder (2), wird stattdessen eine Ableitung gegen einen Textknoten mit leerer Zeichenkette durchgeführt[3]. Existiert genau ein Textknoten als Kind und besteht dieser nur aus Leerraumzeichen, ist eine textuelle Ableitung gegen den Kindknoten (4) oder eine Ableitung gegen das Instanzelement selbst möglich (5). In allen anderen Fällen werden Textknoten, die nur aus Leerraumzeichen bestehen (11), ignoriert (10). Es erfolgt lediglich eine Reduzierung der Patternstruktur über die Funktion `childDeriv` gegen Instanzknoten, die einen Wert enthalten.

```
1  childrenDeriv :: Context -> Pattern -> XmlTrees -> Pattern
2  childrenDeriv cx p [] = childrenDeriv cx p [(NTree (XText "") [])]
3  childrenDeriv cx p [(NTree (XText s) children)] =
4    let p1 = childDeriv cx p (NTree (XText s) children)
5    in if (all isspace s) then choice p p1 else p1
6  childrenDeriv cx p children = stripChildrenDeriv cx p children
7
8  stripChildrenDeriv _ p [] = p -- :: Context -> Pattern -> XmlTrees -> Pattern
9  stripChildrenDeriv cx p (h:t) =
10   stripChildrenDeriv cx (if strip h then p else (childDeriv cx p h)) t
11 strip (NTree (XText s) _) = all isSpace s
12 strip _                   = False
```

Beispiel 6.13: Reduzieren der Kinder eines Elementes

Ende-Tag

Für das Validieren des Ende-Tags wird erneut ausgenutzt, dass `After`-Pattern nur an bestimmten Positionen innerhalb der Struktur auftreten können. Kann der erste Operand des `After`-Pattern nicht auf die leere Sequenz abgeleitet werden, fehlt ein durch das Schema gefordertes Element im Instanzdokument. Ein `NotAllowed`-Pattern wird unverändert zurückgegeben, alle anderen Pattern dürfen an dieser Stelle nicht auftreten.

```
endTagDeriv :: Pattern -> Pattern
endTagDeriv (Choice p1 p2) = choice (endTagDeriv p1) (endTagDeriv p2)
endTagDeriv (After p1 p2) =
  if nullable p1 then p2 else NotAllowed $ show p1 ++ " expected"
endTagDeriv n@(NotAllowed _) = n
endTagDeriv _ = NotAllowed "Call to endTagDeriv with wrong arguments"
```

Beispiel 6.14: Validieren des Ende-Tags

[3] Vgl. Absatz 6.2.7 Relax NG Spezifikation (weak match 3)

7 Datentyp-Bibliotheken

Relax NG besitzt im Gegensatz zur W3C XML Schema Sprache keine eigene vollständige Datentyp-Bibliothek. Stattdessen besteht die Möglichkeit, beliebige externe Bibliotheken in ein Relax NG Schema einzubinden (siehe Abschnitt 2.3).
Dieses Kapitel beschreibt, wie eine Datentyp-Bibliothek in das Relax NG Modul der Haskell XML Toolbox integriert werden kann und welche Schnittstellen dabei zu unterstützen sind. Die Erläuterungen erfolgen am Beispiel einer Bibliothek, die in der Lage ist, die Tabellentypen einer MySQL Datenbank [MySQL 05] abzubilden. Die Implementierung der Typen entspricht dabei den Vorgaben der zurzeit aktuellen MySQL-Version 5.0.3[1].

Datentypen kommen nur in `value`- oder `data`-Pattern zum Einsatz. Sie sind die einzigen Relax NG Elemente, die zum Validieren des textuellen Inhalts in einem XML-Instanzdokument eingesetzt werden. Die Angabe einer Datentyp-Bibliothek ist über das `datatypeLibrary`-Attribut allerdings in jedem Relax NG Pattern möglich und wird durch das gesamte Schema vererbt.
Während der Normalisierung erfolgt eine Prüfung, ob die deklarierten Datentypen und die dazugehörigen Parameter innerhalb der verwendeten Bibliothek vorhanden sind und ob ihre Kombination entsprechend den Vorgaben der Bibliothek korrekt ist (siehe Abschnitt 3.5.3).

7.1 Notwendige Datenstrukturen und Funktionsschnittstellen

Die Liste `DatatypeLibraries` (1) bildet alle Bibliotheken ab, die im Modul der Haskell XML Toolbox zur Verfügung stehen und dort eingesetzt werden können. Dies sind zurzeit die eingebaute Relax NG Typ-Bibliothek mit den beiden Typen `string` und `token`, die MySQL-Bibliothek sowie ein kleiner Teil der Datentypen der W3C XML Schema Sprache. Aus der zuletzt genannten Bibliothek können allerdings nur die Datentypen `NCName`, `anyURI`, `QName` und `string` verwendet werden, da diese innerhalb der Relax NG Spezifikation eingesetzt werden und ihre Implementierung dadurch erforderlich war. Alle weiteren Datentypen der W3C XML Schema Sprache stehen in der aktuellen Version des Relax NG Validators noch nicht zur Verfügung.
Eine einzelne Bibliothek wird durch den Typ `DatatypeLibrary` abgebildet und über ihren `URI`[2] identifiziert (2). `DatatypeCheck` enthält alle wichtigen Angaben, die zum Prüfen der Typen und Parameter sowie zum Validieren eines Instanzdokumentes benötigt werden. Der `URI` und die `DatatypeLibrary`-Deklaration inklusive des `DTC`-Konstruktors (4) sind die einzigen Informationen, die durch ein Bibliothek-Modul exportiert werden müssen.

```
1 type DatatypeLibraries = [DatatypeLibrary]
2 type DatatypeLibrary   = (Uri, DatatypeCheck)
3
4 data DatatypeCheck = DTC { dtAllowsFct    :: DatatypeAllows
5                          , dtEqualFct     :: DatatypeEqual
6                          , dtAllowedTypes :: AllowedDatatypes}
```

Beispiel 7.1: Haskell Datentypen zum Beschreiben von Datentyp-Bibliotheken

[1] Vgl. http://dev.mysql.com/doc/mysql/en/column-types.html
[2] für die MySQL Datentyp-Bibliothek wurde der URI auf http://www.mysql.com festgelegt

Alle weiteren Funktionen und Datenstrukturen, die innerhalb des Moduls zusätzlich benötigt werden, müssen von außen nicht erreichbar sein und können vor dem Anwender verborgen bleiben.

```
1 module DataTypeLibMysql
2   ( mysqlNS, mysqlDatatypeLib )
3 where
4 mysqlNS = "http://www.mysql.com" -- :: String
5 mysqlDatatypeLib -- :: DatatypeLibrary
6   = (mysqlNS, DTC datatypeAllowsMysql datatypeEqualMysql mysqlDatatypes)
```

Beispiel 7.2: Export des MySQL Datentyp-Bibliothek Moduls

Der Konstruktor `DTC` enthält neben den beiden Funktionen, die zum Validieren von `value`- (`dtEqualFct`) beziehungsweise `data`-Pattern (`dtEqualFct`) eingesetzt werden, eine Liste der verfügbaren Datentypen (4) der Bibliothek. Jeder Datentyp wird durch seinen Namen (1) abgebildet und besitzt zudem ein Liste von zugehörigen Parametern (3).

```
1 type DatatypeName      = String
2 type ParamName         = String
3 type AllowedParams     = [ParamName]
4 type AllowedDatatypes  = [(DatatypeName, AllowedParams)]
```

Beispiel 7.3: Abbildung der Datentypen einer Bibliothek

Die Datentypen von MySQL lassen sich in verschiedene Klassen einteilen. Die erste Gruppe sind die numerischen Datentypen (2), zu ihnen zählen beispielsweise `SIGNED-TINYINT` oder `UNSIGNED-SMALLINT`. Weitere Klassen bilden Typen für die Darstellung von Zeit- und Datums-Werten (`DATETIME`, `TIMESTAMP`, usw.) oder Zeichenketten (`CHAR`, `BLOB`, usw.).
Da sich die einsetzbaren Parameter für Typen, die innerhalb derselben Klasse liegen, nicht unterscheiden, werden sie zu den `numericParams` (2,8) beziehungsweise `stringParams` (5,9) für Zeichenketten zusammengefasst. Die Listen enthalten jeweils eine Aufzählung sämtlicher Parameter, die im `name`-Attribut eines `param`-Patterns angegeben werden dürfen. Sie können beliebig verändert und dem zugrunde liegenden Datentypen angepasst werden.

```
1 mysqlDatatypes -- :: AllowedDatatypes
2   = [ ("SIGNED-TINYINT", numericParams)     -- Numerische Typen
3     , ("UNSIGNED-SMALLINT", numericParams)
4     , ...
5     , ("CHAR", stringParams)                -- Zeichenketten
6     , ("BLOB", stringParams)
7     , ... ]
8 numericParams = ["maxExclusive", "minExclusive", ...]     -- :: [ParamName]
9 stringParams  = ["length", "maxLength", "minLength", ...] -- :: [ParamName]
```

Beispiel 7.4: Datentypen von MySQL und ihre Parameter

7.2 Überprüfen des korrekten Einsatzes von Datentypen

Durch die ersten Kontrollen von Restriktionen beim Aufbau der vereinfachten Syntax wird auch überprüft, ob der Einsatz von Datentypen durch ein `value`- oder `data`-Pattern korrekt erfolgt. Die Werte der Attribute `datatypeLibrary` und `type` (1) bilden dabei die Grundlage für die Funktion `checkDatatype` (4). Sie testet, ob die angegebene Datentyp-Bibliothek

unterstützt wird, der URI also in der Liste `datatypeLibraries` vorkommt (6). Ist dies der Fall, werden die möglichen Datentypen der Bibliothek bestimmt (10) und die Prüfung mit der Funktion `checkType` fortgesetzt. Ist hingegen eine ungültige Bibliothek angegeben worden, wird eine entsprechende Fehlermeldung generiert (8).

```
1 (checkDatatype $<< getAttrValue "datatypeLibrary" &&& getAttrValue "type")
2 `when` (isElem >>> (hasName "data" `orElse` hasName "value"))
3
4 checkDatatype :: Uri -> DatatypeName -> IOSArrow XmlTree XmlTree
5 checkDatatype libName typeName
6   = ifP (const $ elem libName $ map fst datatypeLibraries)
7       (checkType libName typeName allowedDataTypes)
8       (mkRelaxError $ "DatatypeLibrary " ++ libName ++ " not found")
9   where
10  DTC _ _ allowedDataTypes = fromJust $ lookup libName datatypeLibraries
```

Beispiel 7.5: Überprüfen der angegebenen Datentyp-Bibliothek

Die Überprüfung des eingesetzten Datentyps läuft analog zum Test der Bibliothek ab. Ist der Typ nicht in der Liste der erlaubten Datentypen enthalten (3), wird eine Fehlerbehandlung ausgelöst. Anderenfalls werden die möglichen (7) sowie angegebenen Parameter (5) bestimmt und die Kontrolle mit diesen beiden Werten fortgesetzt (4). Parameter können allerdings nicht für `value`-Pattern angegeben werden. Sie dürfen nur bei `data`-Elementen auftreten (12).
Da bei `checkParams` nicht nur ein einzelner Wert gegen eine Liste zu testen ist, sondern geprüft werden muss, ob alle Werte gültig sind, weicht die Bearbeitung von den bisherigen Schritten ab. Es wird stattdessen die Differenz zwischen den angegebenen und erlaubten Parametern berechnet (13). Ist diese leer (12), sind alle Parameter für den Datentyp zugelassen und das `value`- beziehungsweise `data`-Pattern wurde korrekt im Schema angewendet. Anderenfalls werden in der Fehlermeldung alle ungültigen Parameterwerte ausgegeben (11).

```
1 checkTyp :: Uri -> DatatypeName -> AllowedDatatypes -> IOSArrow ...
2 checkTyp libName typeName allowedTypes
3  = ifP (const $ elem typeName $ map fst allowedTypes)
4     (checkParams typeName libName getParams $<
5       (listA (getChildren >>> hasName "param" >>> getAttrValue "name")))
6     (mkRelaxError $ "Datatype " ++ typeName ++ " not declared for ...")
7 getParams = fromJust $ lookup typeName allowedTypes
8
9 checkParams :: DatatypeName -> String -> AllowedParams -> [String] -> IOS...
10 checkParams typeName libName allowedParams paramNames
11  = (mkRelaxError $ "Param(s): " ++ diff ++ " not allowed for Datatype ...")
12    `when` (isElem >>> hasName "data" >>> isA (const $ diff /= []))
13 diff = filter (\param -> not $ elem param allowedParams) paramNames
```

Beispiel 7.6: Überprüfen des angegebenen Datentyps sowie der Parameter

7.3 Prüfen von Instanzwerten gegen einen Datentyp

Wird durch die eben beschriebenen Überprüfungen während des Normalisierungsprozesses bereits festgestellt, dass alle deklarierten Bibliotheken, Datentypen und Parameter korrekt eingesetzt sind, könnte auf weitere Kontrollen des Schemas beim Validieren verzichtet werden. Es wäre nur noch der Test gegen die Werte, die im XML-Instanzdokument angegeben sind,

erforderlich. Da sämtliche Überprüfungen des Relax NG Schemas und somit auch die Datentyp Überwachung aus Performanzgründen durch einen Aufrufparameter deaktiviert werden können, ist als zusätzliche Sicherheitsvorkehrung ein erneuter Test integriert. Hierdurch wird gewährleistet, dass das Programm immer in einem gültigen Zustand terminiert.
Für das Validieren von Daten gegen das `value`-Pattern ist für jede Bibliothek eine Funktion vom Typ `DatatypeEqual`, für das Testen eines `data`-Pattern hingegen eine Funktion des Typs `DatatypeAllows` zu entwickeln. Sie werden als Bestandteile des `DTC`-Konstruktors durch das Bibliothek-Modul exportiert und von der Funktion `textDeriv` des `Validation`-Moduls eingesetzt (siehe Abschnitt 6.2.3).
Der Rückgabetyp der Funktionen ist `Maybe String`, wobei der Wert `Nothing` für eine erfolgreiche Validierung steht. Wird `Just` als Ergebnis geliefert, beschreibt der zusätzliche String den aufgetretenen Fehler.

```
type DatatypeEqual  = DatatypeName -> String -> Context -> String ->
                      Context -> Maybe String
type DatatypeAllows = DatatypeName -> ParamList -> String ->
                      Context -> Maybe String
```

Beispiel 7.7: Typsignaturen für die Validierungsfunktionen

7.3.1 Validieren der Daten von `value`-Pattern

Die Einteilung der MySQL Datentypen in Klassen lässt sich auch für die Implementierung der `datatypeEqualMysql`-Funktion einsetzen. Ein Wert im Instanzdokument ist gültig bezüglich eines beliebigen Zeichenketten-Typs, wenn der Vergleich über die Standard Haskell Funktion `==` für Strings den Wert `True` liefert (2). Handelt es sich um einen numerischen Datentyp, müssen die zu vergleichenden Werte vorher normalisiert werden (3). Dabei werden führende und abschließende Leerraumzeichen sowie führende Nullen entfernt (7,8). Dies ist notwendig, da sich Zahlen auf syntaktisch unterschiedliche Weise, aber mit derselben semantischen Bedeutung, darstellen lassen[3].

```
1 datatypeEqualMysql d s1 _ s2 _ --  :: DatatypeEqual
2  | elem d stringTypes = if (s1 == s2) then Nothing else Just "ErrorMsg ..."
3  | elem d numericTypes = if (normalizeNumber s1 == normalizeNumber s2)
4                          then Nothing else Just "ErrorMsg ..."
5  | ...
6  | otherwise = Just "Datatype ... not allowed for DatatypeLibrary ..."
7 normalizeNumber = reverse . dropWhile (== ' ') . reverse .
8                   dropWhile (\x -> x == '0' || x == ' ')
```

Beispiel 7.8: `datatypeEqual`-Funktion für die MySQL Datentypen

7.3.2 Prüfen der Werte von `data`-Pattern

Die Gruppierung in Klassen kann auf ähnliche Weise auch für die `datatypeAllows`-Funktion verwendet werden. Für numerische Datentypen erfolgt die Validierung über die Funktion `checkNumeric`; für Zeichenketten durch `checkString`.
Jeder Datentyp besitzt einen eigenen Definitionsbereich, in dem die Werte des Instanzdokumentes liegen müssen, um gültig zu sein. Der Bereich des MySQL `SIGNED-TINYINT`-Datentyps reicht beispielsweise von -128 bis 127 (3), wird der vorzeichenlose Datentyp eingesetzt, sind Werte von 0 bis 255 möglich (5). Für die String-Datentypen entspricht die obere Grenze der

[3]Der numerische Wert „42“ kann beispielsweise als „ 42 “oder „00042“ repräsentiert werden

maximalen Länge der Zeichenkette; für `VARCHAR` sind dies zum Beispiel 65535 Zeichen (7). Die untere Grenze liegt für diese Datentypen immer bei Null.
Neben der unteren und oberen Begrenzung erhalten die Funktion zusätzlich den zu testenden Wert und die angegebenen Parameter. Der Name des Datentyps wird nur für eine eventuell notwendige Fehlermeldung übergeben.

```
1 datatypeAllowsMysql :: DatatypeAllows
2 datatypeAllowsMysql d@"SIGNED-TINYINT" params value _
3  = checkNumeric d value (-128) 127 params
4 datatypeAllowsMysql d@"UNSIGNED-TINYINT" params value _
5  = checkNumeric d value 0 255 params
6 datatypeAllowsMysql d@"VARCHAR" params value _
7   = checkString d value 0 65535 params
8 ...
9 datatypeAllowsMysql t p v _
10 = Just "Datatype ... not allowed for DatatypeLibrary ..."
```

Beispiel 7.9: `datatypeAllows`-Funktion für die MySQL Datentypen

Prüfen von String-Datentypen

Die Überprüfung eines Wertes beginnt mit dem Test, ob die Länge der Zeichenkette innerhalb des Definitionsbereiches des String-Datentypen liegt (3). Für die obere Grenze ist neben einem Wert größer 0 auch die Angabe -1 möglich (3). Sie bedeutet, dass die Länge des Instanzwertes nach oben hin nicht beschränkt ist, was beispielsweise für den `string`-Datentyp der W3C Schema Sprache gilt. Tritt bei der Bereichsprüfung kein Fehler auf, werden im nächsten Schritt die Parameter kontrolliert (4).

```
1 checkString :: String -> String -> Int -> Int -> ParamList -> Maybe String
2 checkString datatype v lowerB upperB params
3   = if (length v >= lowerB) && ((upperB == (-1)) || (length v <= upperB))
4     then checkParamsString value params
5     else Just $ "Length of " ++ value ++ " out of Range ... "
```

Beispiel 7.10: Test der oberen und unteren Datentyp-Grenzen

Die Liste der Parameter besteht aus Tupeln, in der die erste Komponente den Namen und die zweite den Wert des Parameters enthält (1). `[("maxLength","5"), ("minLength","2")]` bedeutet zum Beispiel, dass der Instanzwert zwischen zwei und fünf Zeichen umfassen muss, um zulässig zu sein. Die Verarbeitung der Liste erfolgt sequentiell bis keine weiteren Parameter mehr zu prüfen sind (2) oder der Test eines Elementes fehlschlägt und eine Fehlermeldung generiert wird (5).
Die Abbildung eines Parameternamens auf eine ausführbare Funktion erfolgt über die, für den Vergleich von Zeichenketten entwickelte, Funktionstabelle `fctTableString` (4,7).

```
1 type ParamList = [(LocalName, String)]
2 checkParamsString _ [] = Nothing -- :: String -> ParamList -> Maybe String
3 checkParamsString value ((pName, pValue):xs)
4   = if (getFct pName) value pValue then checkParamsString value xs
5     else Just $ "Error in Restriction ... "
6 getFct :: String -> (String -> String -> Bool)
7 getFct paramName = fromJust $ lookup paramName fctTableString
```

Beispiel 7.11: Testen der Parameter

Die Tabelle enthält für jeden Namen eine zweistellige Haskellfunktion, welche die Berechnung durchführt (2,3). Die Werte eines Tupels `("length","5")` sind beide vom Typ `String`. Da der Vergleich jedoch auf dem Haskell Datentyp `Int` (5) ausgeführt wird[4], ist eine Konvertierung des Parameterwertes über die Prelude Funktion `read` in einen `Int` notwendig. Um sicherzustellen, dass dabei kein Fehler auftritt, wird der String vorher geprüft (5). Die Funktion `parseNumber` nutzt dafür das von Daan Leijen entwickelte Modul `parsec` [Parsec 00] und kontrolliert, ob es sich bei der angegebenen Zeichenkette um eine syntaktisch korrekte Zahlendarstellung handelt. Ist dies der Fall, kann der `String` problemlos in einen `Int`-Wert konvertiert werden.

```
1 fctTableString :: [(String, String -> String -> Bool)]
2 fctTableString = [ ("length", (checkStrWithNumParam (==)))
3                  , ("maxLength", (checkStrWithNumParam (<=))), ... ]
4 checkStrWithNumParam fct a b --:: (Int -> Int -> Bool) ->String->String->Bool
5   = if parseNumber b then ((length a) `fct` (read b)) else False
```

Beispiel 7.12: Funktionstabelle für Zeichenketten

Prüfen numerischer Datentypen

Die Prüfung von numerischen Datentypen erfolgt analog zu der eben beschriebenen Vorgehensweise über die Funktion `checkNumeric`. Hierbei ist allerdings eine sofortige Konvertierung des Instanzwertes in eine Zahl notwendig (4,7), die erneut über `parseNumber` abgesichert wird (3). Zum Testen der Parameter wird `checkParamsNumeric` eingesetzt (5). Die Arbeitsweise dieser Funktion ist zu der von `checkParamsString` identisch, verwendet jedoch eine andere Funktionstabelle (8), da keine weitere Konvertierung der Werte vor dem Vergleich notwendig ist.

```
1 checkNumeric :: String -> String -> Int -> Int -> ParamList -> Maybe String
2 checkNumeric datatype value lowerBound upperBound params
3   = if parseNumber value
4     then ( if (x >= lowerBound) && (x <= upperBound)
5            then checkParamsNumeric x params else Just $ "Value out of ... ")
6     else Just ("Value = " ++ value ++ " is not a number")
7     where x = read value
8 fctTableNum :: (Ord a, Num a) => [(String, a -> a -> Bool)]
9 fctTableNum = [ ("maxExclusive", (<)), ("minExclusive", (>)), ... ]
```

Beispiel 7.13: Test der Grenzen für numerische Datentypen

Integrieren einer neuen Bibliothek

Da numerische und string-Datentypen in vielen Bibliotheken vorkommen und sich die Implementierungen der Prüfroutinen ihrer Werte und Parameter sehr häufig gleichen, sind die Funktionen `checkNumeric` und `checkString` inklusive der beschriebenen Hilfsfunktionen und Strukturen, wie beispielsweise der Funktionstabellen, in ein eigenes Modul `DataTypeLibUtils` ausgelagert worden. Soll eine weitere Bibliothek in die Haskell XML Toolbox integriert werden, die ebenfalls Datentypen für Zahlen oder Zeichenketten enthält, lassen sich diese Funktionen auch dort einsetzen und müssen durch den Anwender nicht neu entwickelt werden.

[4] Die length-Funktion besitzt die Typsignatur `length :: [a] -> Int`

Das Modul `DataTypeLibraries` verwaltet alle notwendigen Information für den Einsatz von Datentyp-Bibliotheken. Es exportiert die Liste der vorhandenen Bibliotheken sowie die beiden für die Validierung notwendigen Funktionen `datatypeEqual` und `datatypeAllows`. Zum Hinzufügen einer neuen Bibliothek durch den Anwender sind die oben beschriebenen Schnittstellen zu implementieren und die Liste `datatypeLibraries` um einen weiteren Eintrag zu ergänzen.

```
1 module DataTypeLibraries
2   ( datatypeLibraries, datatypeEqual, datatypeAllows )
3 where
4 import DataTypeLibMysql (mysqlDatatypeLib)
5 import DataTypeLibW3C   (w3cDatatypeLib)
6 datatypeLibraries :: DatatypeLibraries
7 datatypeLibraries = [ relaxDatatypeLib, mysqlDatatypeLib, w3cDatatypeLib ]
```

Beispiel 7.14: Export des Moduls `DataTypeLibraries`

8 Programmorganisation und Testfälle

8.1 Modul Struktur

Abbildung 8.1 zeigt die Abhängigkeiten zwischen den einzelnen Teilen des Relax NG Validators. Das Modul `RelaxNG` exportiert alle notwendigen Funktionen, die zum Einsatz des Validators in einem Programm benötigt werden und dient als Einstiegspunkt für den Haskell XML Toolbox Anwender.
Das Modul `PatternToString` liefert Möglichkeiten zum Darstellen der innerhalb des Moduls `CreatePattern` aufgebauten Pattern-Struktur (siehe Abschnitt 4.3). Das Modul `PatternFunctions` beinhaltet Basisfunktion zum Verarbeiten von Relax NG Pattern. Diese wurden ausgegliedert, um eine Trennung zwischen den Arrows in `CreatePattern` und den in „Standard-Haskell“ implementierten Funktionen innerhalb von `PatternFunctions`, zu erzielen.
Alle grundlegenden Datentypen und Hilfsfunktionen, die von den verschiedenen Relax NG Modulen benötigt werden, sind in `DataTypes` beziehungsweise `Utils` definiert.
Das `Validator`-Modul fasst die Arrows für das Normalisieren (`Simplification`) eines Schemas sowie das Validieren gegen ein Instanzdokument (`Validation`) zusammen. Es ergänzt den Export der beiden Module zudem um weitere Funktionen (siehe Abschnitt 8.2), die dem Anwender den Einsatz des Validators in einem Programm erleichtern.

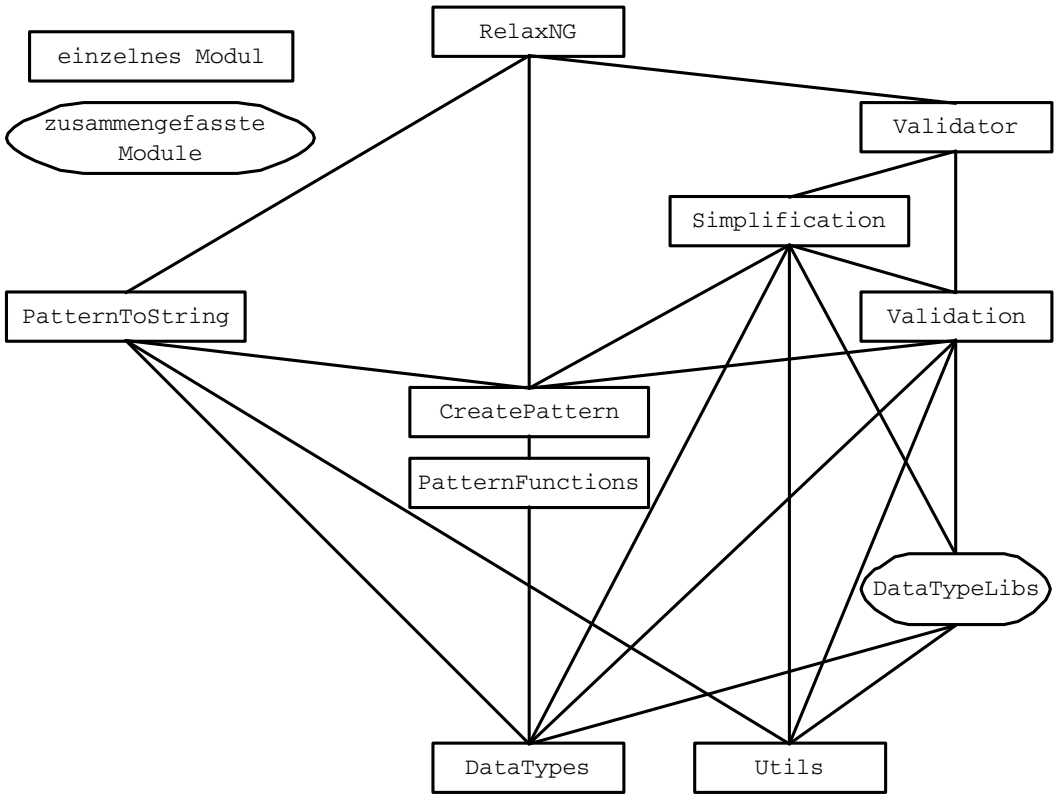

Abbildung 8.1: Modul Hierarchie des Relax NG Validators

Modul Hierarchie der Datentyp-Bibliotheken

Alle für die Integration von Datentyp-Bibliotheken (siehe Kapitel 7) benötigten Module sind in der Grafik 8.2 abgebildet. Das Modul `DataTypeLibraries` bildet die Schnittstelle zu den Modulen `Simplification` sowie `Validation` und exportiert für jede unterstützte Bibliothek die verfügbaren Datentypen, Parameter und Validierungsfunktionen.
`DataTypeLibW3C` und `DataTypeLibMysql` sind die beiden zurzeit implementierten Datentyp-Bibliotheken. Über die geschaffene Schnittstelle können aber beliebig viele weitere Bibliotheken dem Relax NG Validator hinzugefügt werden. Dafür sind in `DataTypeLibUtils` eine Reihe von Hilfsfunktionen vorhanden, die das Ergänzen neuer Bibliotheken erleichtern.

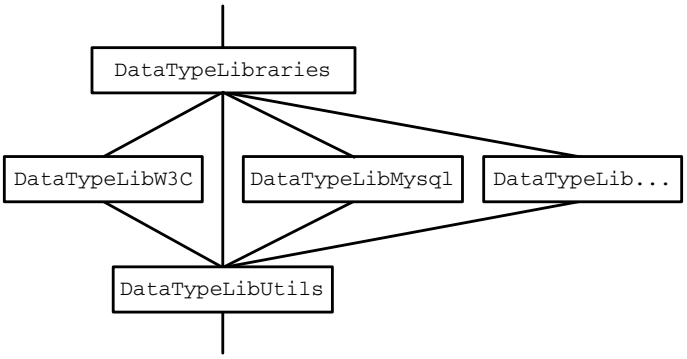

Abbildung 8.2: Struktur der Datentyp-Bibliotheken

8.2 Validierungsfunktionen und Aufrufparameter

Der entwickelte Relax NG Validator kann über die fünf Funktionen `validate`, `validateSchema`, `validateWithSpezification`, `validateSchemaWithSpezification` und `validateWithoutSpezification` aus dem Modul `Validator` in eine Anwendung eingebunden werden.
Der Arrow `validate` bekommt als ersten Parameter eine Liste von Attributen, die beim Einlesen des Schemas über die Funktion `readDocument` Anwendung finden. Der zweite und dritte Parameter beschreiben, wo sich das XML-Instanzdokument beziehungsweise das Relax NG Schemas, gegen das die Instanz validiert werden soll, befinden. Bei der Positionsangabe kann es sich um einen lokalen Dateipfad oder auch um eine URL handeln. Identisch zu `readDocument` wird der Arrow-Input von `validate` ignoriert. Die Funktion `validate` erzeugt das Relax NG Schema für Relax NG (siehe A.1) und übergibt die weitere Verarbeitung an `validateWithSpezification`.
Der Arrow `validateSchema` entspricht weitestgehend `validate`, erhält jedoch als Eingabe nur ein Relax NG Schema und kein Instanzdokument. Die Funktion kann somit zur Überprüfung der syntaktischen und semantischen Gültigkeit eines Relax NG Schemas genutzt werden.

```
validate :: Attributes -> FilePath -> FilePath -> IOSArrow n XmlTree
validate al xmlDoc relaxSchema
  = createSpezification >>> validateWithSpezification al xmlDoc relaxSchema
validateSchema :: Attributes -> FilePath -> IOSArrow n XmlTree
validateSchema al relaxSchema = validate al "" relaxSchema
```

Beispiel 8.1: Funktionen zum Validieren (Teil 1)

Der Arrow `validateWithSpezification` bekommt als Eingabe das Schema für Relax NG (2) und testet zu Beginn, ob das im Parameter angegebene Schema gültig bezüglich

der Spezifikation ist (4). Liefert diese Kontrolle einen Fehler, wird keine Validierung des Instanzdokumentes durchgeführt, sondern die Fehler als Ergebnis zurückgegeben. Ist das Schema korrekt, erfolgt der Test des XML-Dokumentes (5).
Der Arrow `validateWithoutSpezification` liest das angegebene Relax NG Schema ein (10), normalisiert es (11) und führt die Validierung aus, wenn beim Vereinfachen des Schemas keine Probleme aufgetreten sind (12,13). Die Funktion `validateWithoutSpezification` sollte immer dann genutzt werden, wenn das Schema bereits als gültig bekannt ist und keine Kontrolle gegen die Spezifikation mehr notwendig ist.
Der letzte Einstiegspunkt in das Validator Modul ist die Funktion `validateSchemaWithSpezification`. Diese hat die Aufgabe, das im zweiten Parameter angegebene Schema gegen die Arrow-Eingabe zu testen (16). Im Gegensatz zu `validateSchema` ist hierbei nicht festgelegt, gegen welche Spezifikation das Schema validiert wird.

```
1  validateWithSpezification :: Attributes -> FilePath -> FilePath
2                                                  -> IOSArrow XmlTree XmlTree
3  validateWithSpezification al xmlDoc relaxSchema
4   = (validateRelax $< readDocument ((a_check_namespaces, "1"):al) relaxSchema)
5     `orElse` (validateWithoutSpezification al xmlDoc relaxSchema)
6
7  validateWithoutSpezification :: Attributes -> FilePath -> FilePath
8                                                       -> IOSArrow n XmlTree
9  validateWithoutSpezification al xmlDoc relaxSchema
10   = readDocument ((a_check_namespaces, "1"):al) relaxSchema >>>
11     createSimpleForm >>>
12       getErrors `orElse`
13       ((const $ xmlDoc /= "") `guardsP` (validateXMLDoc [] xmlDocument))
14
15 validateSchemaWithSpezification :: Attributes -> FilePath
16                                                -> IOSArrow XmlTree XmlTree
17 validateSchemaWithSpezification al relaxSchema
18   = validateWithSpezification al "" relaxSchema
```

Beispiel 8.2: Funktionen zum Validieren (Teil 2)

Aufrufparameter

Das Verhalten des implementierten Relax NG Validators lässt sich über die in der folgenden Tabelle 8.1 dargestellten Parameter beeinflussen. Ein Parameter ist gesetzt, wenn er den Wert „1“ enthält, in allen anderen Fällen gilt er als nicht relevant.

Parameter	Beschreibung
do-not-check-restrictions	Es werden während der Normalisierung keine Restriktionen überprüft. Der Parameter beinhaltet zudem die Funktionalität von do-not-validate-externalRef und do-not-validate-include.
do-not-validate-externalRef	Über ein `externalRef`-Pattern eingebundene externe Schemata werden nicht gegen die Relax NG Spezifikation validiert und ohne weitere Prüfung in das Schema integriert.
Fortsetzung auf der nächsten Seite	

Parameter	Beschreibung
validate-externalRef	Vor dem Einfügen des referenzierten Schemas erfolgt eine Validierung gegen die Relax NG Spezifikation (Standardeinstellung).
do-not-validate-include	Über ein `include`-Pattern eingebundene externe Schemata werden nicht gegen die Relax NG Spezifikation validiert und ohne weitere Prüfung in das Schema integriert.
validate-include	Vor dem Einfügen des referenzierten Schemas erfolgt eine Validierung gegen die Relax NG Spezifikation (Standardeinstellung).
output-pattern-transformations	Tritt während der Normalisierung ein Fehler auf, werden weitere Transformationsinformationen der Fehlermeldung hinzugefügt.
do-not-collect-errors	Die Normalisierung eines Schemas wird nach dem ersten gefunden Fehler beendet.

Tabelle 8.1: Aufrufparameter des Relax NG Moduls

Ist der Parameter `do-not-check-restrictions` gesetzt, werden keine Restriktionen geprüft und externe Schemata nicht validiert. Über die Kombination mit `validate-externalRef` beziehungsweise `validate-include` kann erreicht werden, dass das Schema nicht geprüft, externe Inhalte jedoch kontrolliert werden.

8.3 Relax NG Testsuite

Die korrekte Funktionsweise des implementierten Relax NG Validators wurde mit Hilfe einer Reihe von James Clark entwickelter Testfälle geprüft.
Die Relax NG Testsuite [TestSuite 03] umfasst in der derzeitigen Version 213 inkorrekte Schema Definitionen, die alle erfolgreich als solche durch den Parser erkannt werden. Zudem sind 160 korrekte Relax NG Schemata enthalten, die gegen 272 zulässige und 257 fehlerhafte XML-Instanzdokumente getestet werden. Auch diese Prüfungen werden alle erfolgreich durch den entwickelten Validator durchgeführt.

Die Testsuite enthält Prüfungen für sämtlichen Bereiche der Spezifikation. Zu Beginn werden syntaktische Fehler kontrolliert. Danach folgen eine Reihe von Tests auf eine korrekt durchgeführte Normalisierung. Am Ende finden schließlich Überprüfungen zu semantischen Fehlern und nicht eingehaltenen Restriktionen statt.
Alle Testfälle sind in der Datei `RelaxTestCases.tgz` enthalten und werde als Bestandteil der Haskell XML Toolbox zum Download zur Verfügung gestellt. Das Entpacken erzeugt ein Verzeichnis `testCases`, in dem sich 373 Unterverzeichnisse befinden. Jedes Unterverzeichnis entspricht dabei einer Testreihe zu einem bestimmten Bereich der Spezifikation und besteht aus einem gültigen oder ungültigen Relax NG Schema sowie beliebig vielen XML-Instanzen, die gegen das Schema geprüft werden müssen.

8.3.1 Aufbau der Testumgebung

Das Ausführen der Tests basiert auf dem System HUnit [HUnit 02], welches ein schnelles und automatisiertes Prüfen einer Vielzahl von Testfällen ermöglicht. Bevor die Tests gestartet werden können, ist die aktuelle Verzeichnisstruktur einzulesen und es sind daraus die einzelnen Testfälle zu konstruieren. Der entwickelte Datentyp `Entry` bildet die notwendige Struktur ab. Der `DirContent`-Konstruktor stellt den Inhalt eines Unterverzeichnisses und

damit einer Testreihe dar. Der erste Parameter vom Typ `FilePath` enthält das Relax NG Schema, der zweite Parameter die Liste der XML-Instanzdokumente. Der Konstruktor `Dir` fasst die einzelnen Testreihen zu der eigentlichen Testsuite zusammen.

```
data Entry = Dir [Entry] | DirContent FilePath [FilePath]
```

Beispiel 8.3: Datentyp für das Abbilden einer Verzeichnisstruktur

Über die Funktion `getDirectoryContents` wird der Inhalt des aktuellen Verzeichnisses eingelesen (2). Der Inhalt wird in Dateien und Verzeichnisse getrennt. Zusätzlich findet ein rekursiver Abstieg in die Unterverzeichnisse statt (7,8). Die Dateien sind anschließend in Schema- und Instanzdokumente aufzuteilen (9,10).
Ist eine Schemadatei im betrachteten Verzeichnis vorhanden, wird ein neuer `DirContent`-Eintrag generiert. Anderenfalls wird die Liste der bisher erstellten Testfälle unverändert zurückgegeben (12).

```
1  readDir pre p = do -- :: FilePath -> FilePath -> IO Entry
2                  dir     <- getDirectoryContents $ pre ++ "/" ++ p
3                  entries <- getEntries $ clean [".", "..", "CVS"] dir
4                  return $ Dir entries
5  getEntries xs -- :: [FilePath] -> IO [Entry]
6    = do
7      files    <- mapM toEntryFiles xs
8      dirs     <- mapM toEntryDir xs
9      xmlfiles <- return $ [ f | f <- files, ".xml" `isSuffixOf` f]
10     rngfile  <- return $ fromMaybe "" $ find (".rng" `isSuffixOf`) files
11     return $ if rngfile /= ""
12              then (DirContent rngfile xmlfiles):dirs else dirs
```

Beispiel 8.4: Einlesen der Verzeichnisstruktur

8.3.2 Erstellen der Testfälle

Für das Generieren der Testfälle ist neben der erstellten Verzeichnisstruktur (2) die Relax NG Spezifikation, gegen die die Schemata geprüft werden (3), notwendig. Auf der Basis dieser beiden Informationen können die einzelnen Fälle aufgebaut, zu einer Testreihe zusammengesetzt und ausgeführt (4) werden.
Die Spezifikation sollte bereits an dieser Stelle eingelesen und normalisiert werden. Hierdurch wird erreicht, dass diese Aufgabe nicht für jeden Testfall neu durchzuführen ist, was einen erheblichen Performanzgewinn bedeutet. Die einmal erstellte Spezifikation steht unverändert sämtlichen Tests zur Verfügung.

```
1  runTest path = do -- :: FilePath -> IO Counts
2                 dir     <- readDir path ""
3                 [spezi] <- runX createSpezification
4                 runTestTT $ TestList $ generateTestCases [] [dir] spezi
```

Beispiel 8.5: Ausführen der Tests

War in einem Verzeichnis keine XML-Datei vorhanden (2), muss lediglich das Schema auf Gültigkeit geprüft werden. Existieren hingegen Instanzdokumente, wird für jedes Dokument ein eigener Testfall aufgebaut (5). Sämtliche Prüfungen werden mit einer Beschriftung, bestehend aus dem Schema- und dem Instanz-Dateinamen, versehen (10). Schlägt ein Test fehl, kann über diese Angaben das Problem schnell lokalisiert werden.

Der Validator liefert als Ergebnis eine Liste der aufgetretenen Fehler zurück (12). Handelt es sich bei dem Test um ein ungültiges Schema, dieses besitzt stets den Namen `i.rng`, darf die Fehlerliste nicht leer sein (14). Ebenso muss bei einem nicht korrekten Instanzdokument (`i.xml`) mindestens ein Fehler zurückgeliefert werden (16), damit der Test als erfolgreich durchgeführt gilt. Trifft eine der beiden Bedingungen nicht zu, wird das erhaltene Validator Ergebnis formatiert und durch HUnit zur weiteren Problemeingrenzung ausgegeben (13).

```
1  generateTestCases :: Attributes -> [Entry] -> XmlTree -> [Test]
2  generateTestCases al [ DirContent rngFile [] ] spezi
3    = [generateTestCase al rngFile "" spezi]
4  generateTestCases al [ DirContent rngFile xmlFiles ] spezi
5    = map (\x -> generateTestCase al rngFile x spezi) xmlFiles
6  ...
7
8  generateTestCase :: Attributes -> FilePath -> FilePath -> XmlTree -> Test
9  generateTestCase al rng xml spezi
10   = TestLabel "Schema=" ++ rng ++ ", instanz=" ++ xml $ TestCase $
11     do
12     res <- runX $ constA spezi >>> validateWithSpezification al xml rng
13     assertBool (concat $ map formatXmlTree res)
14                ( if ("i.rng" `isSuffixOf` rng) then res /= []
15                  else ( if ("i.xml" `isSuffixOf` xml)
16                         then res /= [] else res == []))
```

Beispiel 8.6: Erstellen der Testfälle

Die Abbildung 8.3 zeigt abschließend die Modul Struktur, die zum Ausführen der Relax NG Testfälle notwendig ist.

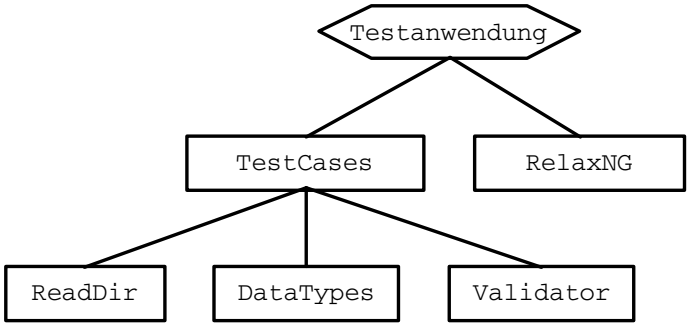

Abbildung 8.3: Anordnung der Testsuite Module

Die Testanwendung braucht zum Ausführen der Testfälle nur das Modul `TestCases` zu importieren und kann sie anschließend über die Funktion `runTest` starten. Der Parameter der Funktion `runTest` bestimmt den Pfad zum Verzeichnis, in dem die Testfälle liegen.

```
import Text.XML.HXT.RelaxNG.RelaxNG
import Text.XML.HXT.RelaxNG.TestCases
main = do -- :: IO ()
       result <- runTest "testCases"
       putStrLn $ show result
```

Beispiel 8.7: Anwendung für die Relax NG Testsuite

9 Schlussbetrachtungen

Das Ziel dieser Arbeit bestand in der Analyse der Relax NG Spezifikation und der Entwicklung eines Validator Moduls für die Haskell XML Toolbox.

9.1 Erreichtes

Der implementierte Relax NG Schema Prozessor führt alle Normalisierungsschritte sowie sämtliche Restriktionsprüfungen, die in der Spezifikation vorgeschrieben sind, erfolgreich durch. Am Ende der Transformation liegt jedes gültige Schema in vereinfachter Syntax vor. Alle unzulässigen Schemata werden als solche erkannt und der Anwender durch eine entsprechende Fehlerbeschreibung informiert.
Der entwickelte Validator ist in der Lage, ein beliebiges XML-Instanzdokument gegen ein Relax NG Schema zu prüfen und zu bestimmen, ob es der geforderten Struktur entspricht und somit Gültigkeit besitzt. Ist dies nicht der Fall, wird der Benutzer der Toolbox auch hier durch einer Fehlermeldung benachrichtigt.
Es wurde zudem eine Schnittstelle geschaffen, die es ermöglicht, das Validator Modul schnell und komfortabel um weitere Datentyp-Bibliotheken zu erweitern. Als Bestandteil dieser Arbeit sind die beiden Relax NG Datentypen `token` und `string`, ein Ausschnitt der W3C XML Schema Sprache sowie eine Bibliothek zum Abbilden von MySQL Datentypen implementiert worden.
Die entwickelten Haskell Module lassen sich unter den Betriebssystemen Linux und Windows mit dem Glasgow Haskell Compiler [GHC 05] in der Version 6.4 und der Compilerdirektive `-Wall` ohne Warnungen übersetzen.
Zudem passieren alle Testfälle der Relax NG Testsuite den Validator fehlerfrei, so dass das Ziel dieser Arbeit als vollständig erreicht angesehen werden kann.

9.2 Vor- und Nachteile der Arrow-Notation

Die neu eingeführte Arrow-Notation der Haskell XML Toolbox hatte zu Beginn der Implementierungsphase einige Probleme zur Folge. Es war erst eine intensive Einarbeitung in die Schreibweise und die zur Verfügung stehenden Kombinatoren notwendig, bevor eine effektive Programmierung möglich war. Die bereits vorhandenen Kenntnisse über die zugrunde liegenden Konzepte funktionaler Programmiersprachen sowie der Sprachelemente von Haskell ermöglichten jedoch eine rasche Überwindung der Einstiegshürden.
Durch die während der Umsetzung der Module gewonnenen praktischen Erfahrungen und der intensiven Zusammenarbeit mit dem Masterstudenten Manuel Ohlendorf und Prof. Dr. Schmidt von der Fachhochschule Wedel gelang es, die benötigten Algorithmen schnell zu entwickeln und effiziente Lösungen für die auftretenden Probleme zu finden.

Als großer Vorteil der Arrow-Struktur ist die zusätzlich erhaltene Typsicherheit zu sehen. Die bisher verwendete Technik der Haskell XML Toolbox bildet alle Funktionen für die Transformation von XML-Dokumenten über den Datentyp `XmlFilter` ab. Diese Filter können durch den einheitlichen Typ beliebig miteinander kombiniert werden, was eine sehr hohe Flexibilität zur Folge hat. Der Nachteil des Ansatzes besteht jedoch darin, dass

sich Filterfunktionen verknüpfen lassen, deren Aus- beziehungsweise Eingabe nicht sinnvoll zueinander passen.
Durch das Fehlen unterschiedlicher Typen ist der Haskell Compiler nicht mehr in der Lage, solche Anwendungsfehler aufzudecken. Die zusätzliche Sicherheit, die eine streng typisierte Sprache wie Haskell dem Programmierer bietet, ist durch den Filteransatz verloren gegangen. Die Arrow-Notation behebt dieses Problem und ergänzt die flexiblen Kombinationsmöglichkeiten der Filter um die Eigenschaft der Typsicherheit. In diesem Zuge wird auch die Lesbarkeit der implementierten Funktionen erhöht, da weder der Ein- noch der Ausgabewert auf `XmlTree` beschränkt ist, sondern ein beliebig benannter und problemspezifischer Typ sein kann.

Die entwickelten Module für den Relax NG Validator der Haskell XML Toolbox umfassen 3.500 Zeilen Quellcode. Der Ausschnitt, aus dem Validator Jing, der in etwa denselben Funktionsumfang abdeckt, besteht aus über 10.000 Javacode Zeilen. Diese beiden Werte machen deutlich, dass mit dem Einsatz von Arrows in Haskell eine sehr kompakte Beschreibung der Algorithmen möglich ist.
An ihre Grenzen stößt die Arrow-Notation allerdings, wenn für einen Arrow weitere Parameter notwendig sind beziehungsweise der Eingabewert an mehreren Stellen benötigt wird. Die Haskell XML Toolbox besitzt für diese Fälle spezielle Kombinatoren, die das geforderte Verhalten nachbilden können. Ihre Notwendigkeit ist jedoch häufig nicht direkt einsehbar und vor allem am Beginn der Programmierung mit Arrows nur schwierig nachzuvollziehen. Die Komplexität vieler bereits implementierter Funktionen konnte aber im Laufe der Modul Entwicklung durch ein detaillierteres Wissen über die Arrow-Notation erheblich reduziert werden.

Zusammenfassend kann gesagt werden, dass die Implementierung des Relax NG Validators in Haskell zielgerichteter und effizienter durchgeführt werden konnte, als dies mit vielen anderen Programmiersprachen möglich gewesen wäre.

9.3 Ausblick

Relax NG besitzt neben der XML konformen Notation noch ein weitere, kompaktere Syntax zur Beschreibung von Schemata. Eine mögliche Erweiterung dieser Arbeit besteht in der Entwicklung eines Parsers für diese Schreibweise. Ebenso ist ein, auf dem Parser aufbauendes, Modul zur Transformation der XML konformen Syntax in die kompakte Syntax und umgekehrt denkbar.

Die Verbreitung von Relax NG zum Definieren von XML Schemata ist in der Wirtschaft und Forschung bisher deutlich geringer als bei der W3C XML Schema Sprache. Um die Einsatzfähigkeit der Haskell XML Toolbox innerhalb von verschiedenen Projekten zu steigern, könnte in einem nächsten Schritt die Umwandlung eines Relax NG Schemas in ein äquivalentes W3C XML Schema implementiert werden.
Auch der vollständige Ausbau der W3C Datentyp Bibliothek sollte den Verwendungsbereich des entwickelten Validators deutlich erhöhen.

Das HaXml-Projekt [HaXml 05] besitzt mit dem Werkzeug `DtdToHaskell` die Möglichkeit, eine DTD in einen gleichwertigen Haskell Datentyp zu transformieren und entsprechende Ein- und Ausgabefunktionen zu erzeugen. Dieser Ansatz kann auf das entwickelte Relax NG Modul übertragen werden, um damit eine Anwendung zu schaffen, die auf Basis der strengen Typisierung von Haskell ausschließlich gültige XML-Dokumente erzeugt.

A Anhang

A.1 Relax NG Schema für Relax NG

```
<grammar datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes"
  ns="http://relaxng.org/ns/structure/1.0"
  xmlns="http://relaxng.org/ns/structure/1.0">
  <start> <ref name="pattern"/> </start>

  <define name="pattern">
    <choice>
      <element name="element">
        <choice>
          <attribute name="name"> <data type="QName"/> </attribute>
          <ref name="open-name-class"/>
        </choice>
        <ref name="common-atts"/>
        <ref name="open-patterns"/>
      </element>
      <element name="attribute">
        <ref name="common-atts"/>
        <choice>
          <attribute name="name"> <data type="QName"/> </attribute>
          <ref name="open-name-class"/>
        </choice>
        <interleave>
          <ref name="other"/>
          <optional> <ref name="pattern"/> </optional>
        </interleave>
      </element>
      <element name="group">
        <ref name="common-atts"/> <ref name="open-patterns"/>
      </element>
      <element name="interleave">
        <ref name="common-atts"/> <ref name="open-patterns"/>
      </element>
      <element name="choice">
        <ref name="common-atts"/> <ref name="open-patterns"/>
      </element>
      <element name="optional">
        <ref name="common-atts"/> <ref name="open-patterns"/>
      </element>
      <element name="zeroOrMore">
        <ref name="common-atts"/> <ref name="open-patterns"/>
      </element>
      <element name="oneOrMore">
        <ref name="common-atts"/> <ref name="open-patterns"/>
```

```
</element>
<element name="list">
  <ref name="common-atts"/> <ref name="open-patterns"/>
</element>
<element name="mixed">
  <ref name="common-atts"/> <ref name="open-patterns"/>
</element>
<element name="ref">
  <attribute name="name"> <data type="NCName"/> </attribute>
  <ref name="common-atts"/>
</element>
<element name="parentRef">
  <attribute name="name"> <data type="NCName"/> </attribute>
  <ref name="common-atts"/>
</element>
<element name="empty">
  <ref name="common-atts"/> <ref name="other"/>
</element>
<element name="text">
  <ref name="common-atts"/> <ref name="other"/>
</element>
<element name="value">
  <optional>
    <attribute name="type"> <data type="NCName"/> </attribute>
  </optional>
  <ref name="common-atts"/>
  <text/>
</element>
<element name="data">
  <attribute name="type"> <data type="NCName"/> </attribute>
  <ref name="common-atts"/>
  <interleave>
    <ref name="other"/>
    <group>
      <zeroOrMore>
        <element name="param">
          <attribute name="name"> <data type="NCName"/> </attribute>
          <text/>
        </element>
      </zeroOrMore>
      <optional>
        <element name="except">
          <ref name="common-atts"/> <ref name="open-patterns"/>
        </element>
      </optional>
    </group>
  </interleave>
</element>
<element name="notAllowed">
  <ref name="common-atts"/> <ref name="other"/>
</element>
<element name="externalRef">
  <attribute name="href"> <data type="anyURI"/> </attribute>
```

```
      <ref name="common-atts"/> <ref name="other"/>
    </element>
    <element name="grammar">
      <ref name="common-atts"/> <ref name="grammar-content"/>
    </element>
  </choice>
</define>

<define name="grammar-content">
  <interleave>
    <ref name="other"/>
    <zeroOrMore>
      <choice>
        <ref name="start-element"/>
        <ref name="define-element"/>
        <element name="div">
          <ref name="common-atts"/> <ref name="grammar-content"/>
        </element>
        <element name="include">
          <attribute name="href"> <data type="anyURI"/> </attribute>
          <ref name="common-atts"/> <ref name="include-content"/>
        </element>
      </choice>
    </zeroOrMore>
  </interleave>
</define>

<define name="include-content">
  <interleave>
    <ref name="other"/>
    <zeroOrMore>
      <choice>
        <ref name="start-element"/>
        <ref name="define-element"/>
        <element name="div">
          <ref name="common-atts"/> <ref name="include-content"/>
        </element>
      </choice>
    </zeroOrMore>
  </interleave>
</define>

<define name="start-element">
  <element name="start">
    <ref name="combine-att"/> <ref name="common-atts"/>
    <ref name="open-pattern"/>
  </element>
</define>

<define name="define-element">
  <element name="define">
    <attribute name="name"> <data type="NCName"/> </attribute>
    <ref name="combine-att"/> <ref name="common-atts"/>
```

```
    <ref name="open-patterns"/>
  </element>
</define>

<define name="combine-att">
  <optional>
    <attribute name="combine">
      <choice>
        <value>choice</value> <value>interleave</value>
      </choice>
    </attribute>
  </optional>
</define>

<define name="open-patterns">
  <interleave>
    <ref name="other"/>
    <oneOrMore> <ref name="pattern"/> </oneOrMore>
  </interleave>
</define>

<define name="open-pattern">
  <interleave>
    <ref name="other"/> <ref name="pattern"/>
  </interleave>
</define>

<define name="name-class">
  <choice>
    <element name="name">
      <ref name="common-atts"/> <data type="QName"/>
    </element>
    <element name="anyName">
      <ref name="common-atts"/> <ref name="except-name-class"/>
    </element>
    <element name="nsName">
      <ref name="common-atts"/> <ref name="except-name-class"/>
    </element>
    <element name="choice">
      <ref name="common-atts"/> <ref name="open-name-classes"/>
    </element>
  </choice>
</define>

<define name="except-name-class">
  <interleave>
    <ref name="other"/>
    <optional>
      <element name="except"> <ref name="open-name-classes"/> </element>
    </optional>
  </interleave>
</define>
```

```
  <define name="open-name-classes">
    <interleave>
      <ref name="other"/>
      <oneOrMore> <ref name="name-class"/> </oneOrMore>
    </interleave>
  </define>

  <define name="open-name-class">
    <interleave>
      <ref name="other"/> <ref name="name-class"/>
    </interleave>
  </define>

  <define name="common-atts">
    <optional> <attribute name="ns"/> </optional>
    <optional>
      <attribute name="datatypeLibrary"> <data type="anyURI"/> </attribute>
    </optional>
    <zeroOrMore>
      <attribute>
        <anyName> <except> <nsName/> <nsName ns=""/> </except> </anyName>
      </attribute>
    </zeroOrMore>
  </define>

  <define name="other">
    <zeroOrMore>
      <element>
        <anyName> <except> <nsName/> </except> </anyName>
        <zeroOrMore>
          <choice>
            <attribute> <anyName/> </attribute>
            <text/>
            <ref name="any"/>
          </choice>
        </zeroOrMore>
      </element>
    </zeroOrMore>
  </define>

  <define name="any">
    <element>
      <anyName/>
      <zeroOrMore>
        <choice>
          <attribute> <anyName/> </attribute>
          <text/>
          <ref name="any"/>
        </choice>
      </zeroOrMore>
    </element>
  </define>
</grammar>
```

A.2 Relax NG Grammatik

A.2.1 Vollständige Syntax

```
pattern          ::= <element name="QName"> pattern+ </element>
                   | <element> nameClass pattern+ </element>
                   | <attribute name="QName"> [pattern] </attribute>
                   | <attribute> nameClass [pattern] </attribute>
                   | <group> pattern+ </group>
                   | <interleave> pattern+ </interleave>
                   | <choice> pattern+ </choice>
                   | <optional> pattern+ </optional>
                   | <zeroOrMore> pattern+ </zeroOrMore>
                   | <oneOrMore> pattern+ </oneOrMore>
                   | <list> pattern+ </list>
                   | <mixed> pattern+ </mixed>
                   | <ref name="NCName"/>
                   | <parentRef name="NCName"/>
                   | <empty/>
                   | <text/>
                   | <value [type="NCName"]> string </value>
                   | <data type="NCName"> param* [exceptPattern] </data>
                   | <notAllowed/>
                   | <externalRef href="anyURI"/>
                   | <grammar> grammarContent* </grammar>
param            ::= <param name="NCName"> string </param>
exceptPattern    ::= <except> pattern+ </except>
grammarContent   ::= start
                   | define
                   | <div> grammarContent* </div>
                   | <include href="anyURI"> includeContent* </include>
includeContent   ::= start
                   | define
                   | <div> includeContent* </div>
start            ::= <start [combine="method"]> pattern </start>
define           ::= <define name="NCName" [combine="method"]>pattern+</define>
method           ::= choice
                   | interleave
nameClass        ::= <name> QName </name>
                   | <anyName> [exceptNameClass] </anyName>
                   | <nsName> [exceptNameClass] </nsName>
                   | <choice> nameClass+ </choice>
exceptNameClass ::= <except> nameClass+ </except>
```

A.2.2 Vereinfachte Syntax

```
grammar         ::= <grammar> <start> top </start> define* </grammar>
define          ::= <define name="NCName">
                      <element> nameClass top </element>
                    </define>
top             ::= <notAllowed/>
                  | pattern
pattern         ::= <empty/>
                  | nonEmptyPattern
nonEmptyPattern ::= <text/>
                  | <data type="NCName" datatypeLibrary="anyURI">
                      param* [exceptPattern]
                    </data>
                  | <value datatypeLibrary="anyURI" type="NCName" ns="string">
                      string
                    </value>
                  | <list> pattern </list>
                  | <attribute> nameClass pattern </attribute>
                  | <ref name="NCName"/>
                  | <oneOrMore> nonEmptyPattern </oneOrMore>
                  | <choice> pattern nonEmptyPattern </choice>
                  | <group> nonEmptyPattern nonEmptyPattern </group>
                  | <interleave> nonEmptyPattern nonEmptyPattern </interleave>
param           ::= <param name="NCName"> string </param>
exceptPattern   ::= <except> pattern </except>
nameClass       ::= <anyName> [exceptNameClass] </anyName>
                  | <nsName ns="string"> [exceptNameClass] </nsName>
                  | <name ns="string"> NCName </name>
                  | <choice> nameClass nameClass </choice>
exceptNameClass ::= <except> nameClass </except>
```

Literatur und Quellenverzeichnis

[Brzozowski 64] Janusz A. Brzozowski: *Derivatives of Regular Expressions*, Journal of the ACM, Volume 11, Issue 4, 1964

[Clark NCA 03] James Clark: *Name class analysis*, http://www.thaiopensource.com/relaxng/nameclass.html

[Clark ARV 02] James Clark: *An algorithm for RELAX NG validation*, http://www.thaiopensource.com/relaxng/derivative.html

[DCD 98] World Wide Web Consortium: *DCD (Document Content Description for XML)*, http://www.w3.org/TR/NOTE-dcd/

[DocBook 05] OASIS: *DocBook Technical Committee Document Repository*, http://www.oasis-open.org/docbook/

[EBNF] International Organization for Standardization: *Erweiterte Backus-Naur-Form: ISO/IEC 14977:1996(E)*, http://www.iso.org/

[GHC 05] *Homepage des Glasgow Haskell Compilers*, http://www.haskell.org/ghc/

[HaXml 05] *Homepage von HaXml*, http://www.cs.york.ac.uk/fp/HaXml/

[HUnit 02] HUnit: *Haskell Unit Testing*, http://hunit.sourceforge.net/

[Jing 03] Jing: *A RELAX NG validator in Java*, http://www.thaiopensource.com/relaxng/jing.html

[MathML 03] World Wide Web Consortium: *Mathematical Markup Language (MathML) Version 2.0 (Second Edition)*, http://www.w3.org/Math/

[MySQL 05] MySQL *The world's most popular open source database*, http://www.mysql.com/

[Namespaces 99] World Wide Web Consortium: *Namespaces in XML*, http://www.w3.org/TR/REC-xml-names/

[Parsec 00] Daan Leijen: *Parsec: a free monadic parser combinator library for Haskell*, http://www.cs.uu.nl/∼daan/parsec.html

[Relax 01] RELAX NG Spezification: *RELAX NG Spezification, Committee Specification 3 December 2001*, http://www.oasis-open.org/committees/relax-ng/spec-20011203.html

[RelaxCore 00] RELAX Core: *RELAX (Regular Language description for XML)*, http://www.xml.gr.jp/relax/

[Schmidt 02] Martin Schmidt: *Master Thesis: Design and Implementation of a validating XML parser in Haskell*, 2002

[SGML 95] World Wide Web Consortium: *SGML (ISO 8879)*, http://www.w3.org/MarkUp/SGML/

[SMIL 98] World Wide Web Consortium: *SMIL (Synchronized Multimedia Intergration Language)*, http://www.w3.org/AudioVideo/

[SOX 99] World Wide Web Consortium: *Schema for Object-Oriented XML 2.0*, http://www.w3.org/TR/NOTE-SOX/

[SVG 03] World Wide Web Consortium: *SVG (Scalable Vector Graphics)*, http://www.w3.org/TR/SVG/

[TestSuite 03] James Clark: *Relax NG Testsuite*, http://www.relaxng.org/#test-suites

[Toolbox 02] *Homepage der Haskell XML Toolbox*, http://www.fh-wedel.de/~si/HXmlToolbox/index.html

[TREX 99] TREX: *TREX - Tree Regular Expressions for XML*, http://www.thaiopensource.com/trex/

[Vlist 03] Eric van der Vlist: *RELAX NG, A Simpler Schema Language for XML*, O'Reilly, ISBN 0-596-00421-4, 2003

[W3C Schema 04] World Wide Web Consortium: *XML Schema, W3C Recommendation 28 October 2004 (Second Edition)*, http://www.w3.org/XML/Schema/

[XHTML 05] World Wide Web Consortium: *XHTML 2.0*, http://www.w3.org/TR/xhtml2/

[XML 04] World Wide Web Consortium: *Extensible Markup Language (XML) 1.0 (Third Edition)*, http://www.w3.org/TR/2004/REC-xml-20040204/

[XPath 99] World Wide Web Consortium: *XML Path Language (XPath) Version 1.0*, http://www.w3.org/TR/xpath/

Weitere Quellen

In diesem Abschnitt werden die nicht in das Literaturverzeichnis passenden Quellen aufgeführt, die aber mindestens genauso viel zu dieser Arbeit beigetragen haben, wie die dort aufgeführten.

LaTeX

Helmut Kopka: *LaTeX: Eine Einführung*, Addison-Wesley, ISBN 3-89319-199-2, 1990

Haskell

- Simon Thompson: *Haskell: The Craft of Functional Programming, Second Edition*, Addison-Wesley, ISBN 0-201-34275-8, 1999
- Zvon: *Haskell Reference*, http://www.zvon.org/other/haskell/Outputglobal/

Wörterbücher

- *Der Duden, Die deutsche Rechtschreibung*, DUDENVERLAG, ISBN 3-411-04013-0 , 2004
- *Duden Online*, http://duden.de/
- *Oxford Advanced Learner's Dictionary*, Cornelsen Verlag, ISBN 3-464-11223-3, 1995
- *Dictionary LEO*, http://dict.leo.org/

Antworten auf alle möglichen (mehr oder weniger) fachlichen Fragen gaben:

- Prof. Dr. Uwe Schmidt, Fachhochschule Wedel
- Dipl.-Wi.-Inform. (BA) Manuel Ohlendorf

Korrekturleser:

- Dipl.-Wi.-Inform. (FH) Sven Dietze
- Dipl.-Medieninform. (FH) Wiebke Leander

seid bedankt...

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe.

Hamburg, 01.09.2005

(Torben Kuseler)