# XML Schema Import for the Pathfinder XQuery Compiler

Bachelor's Thesis
zur Erlangung des Grades
Bachelor of Science

Jan Rittinger

Universität Konstanz

November 2003

# Inhaltsverzeichnis

**Einleitung** 4

**1 *XML*** 4

**2 *XQuery*** 6

**3 *XML Schema*** 9

**4 *XML Schema* und *XQuery* Typen** 10

4.1 *QNames* und *Namespaces* . . . . . 10
4.2 *XML Schema* Typen . . . . . 10
4.2.1 `schema` . . . . . 11
4.2.2 `element` . . . . . 12
4.2.3 `group` . . . . . 13
4.2.4 `all` . . . . . 14
4.2.5 `choice` . . . . . 15
4.2.6 `sequence` . . . . . 15
4.2.7 `any` . . . . . 16
4.2.8 `element occurrences` . . . . . 16
4.2.9 `attribute` . . . . . 17
4.2.10 `attributeGroup` . . . . . 18
4.2.11 `anyAttribute` . . . . . 19
4.2.12 `complexType` . . . . . 19
4.2.13 `simpleType` . . . . . 20
4.2.14 `simpleContent` und `complexContent` . . . . . 21
4.2.15 `extension` . . . . . 21
4.2.16 `restriction` . . . . . 23
4.2.17 `list` . . . . . 25
4.2.18 `union` . . . . . 25
4.2.19 *Symbol Spaces* . . . . . 26
4.2.20 Ignorierte *XML Schema*-Komponenten . . . . . 27
4.3 *XQuery* Typen . . . . . 29
4.3.1 `item types` . . . . . 29
4.3.2 `regular expression types` . . . . . 30
4.3.3 `named type` . . . . . 32
4.3.4 Weitere eingebaute Typen . . . . . 32

**5 *Mapping* von *XML Schema* nach *XQuery*** 32

5.1 `schema` . . . . . 33
5.2 `element` . . . . . 33
5.3 `group` . . . . . 35
5.4 `all` . . . . . 36
5.5 `choice` . . . . . 37
5.6 `sequence` . . . . . 37
5.7 `any` . . . . . 38
5.8 `element occurrences` . . . . . 38
5.9 `attribute` . . . . . 39
5.10 `attributeGroup` . . . . . 41
5.11 `anyAttribute` . . . . . 42
5.12 `complexType` . . . . . 42
5.13 `simpleType` . . . . . 45
5.14 `simpleContent` und `complexContent` . . . . . 45

5.15 `extension` . . . 45
5.16 `restriction` . . . 47
5.17 `list` . . . 49
5.18 `union` . . . 50
5.19 *Symbol Spaces* . . . 51
5.20 Anwendung der Abbildungsregeln . . . 51
5.21 Ignorierte *XML Schema*-Komponenten . . . 54
5.22 Vergleich: *Mapping – XQuery Formal Semantics* . . . 55

**6 *SAX*** **56**
6.1 *SAX-events* . . . 57
6.2 *SAX-callbacks* . . . 57

**7 *State Machine*** **57**
7.1 *Endlicher Automat* . . . 58
7.1.1 *Eingabewort* und *Eingabealphabet* . . . 58
7.1.2 Zustände und Übergänge . . . 58
7.2 *Mini State Machine* – Konzepte . . . 59
7.2.1 *Mini Schema* . . . 59
7.2.2 *Mini State Table* . . . 60
7.2.3 Erweiterte *Mini State Table* . . . 62
7.3 *State Machine* . . . 65
7.3.1 `complexType` – Konzepte . . . 66
7.3.2 Konzept der `attribute` Kombination . . . 67
7.3.3 `union` – Konzept . . . 67

**8 Implementation** **68**
8.1 *stacks* . . . 69
8.2 *SAX-callback functions* . . . 70
8.2.1 `start element event` . . . 70
8.2.2 `end element event` . . . 70
8.3 *State Table* . . . 70
8.4 *State Table* Aktionen . . . 70

**9 Zusammenfassung** **72**

**Literatur** **73**

**A *State Table*** **74**

**B Aktionen** **76**

**C Weitere Änderungen –** `substitutionGroup` **78**

## Einleitung

*XQuery* [2] ist eine speziell auf *XML* [1] angepasste Anfragesprache. *XQuery* - ganz analog zu SQL - ist eine deklarative Sprache. Ein *XQuery-Compiler* ist notwendig, um einen deklarativen *XQuery*-Ausdruck in ein ausführbares Programm zu übersetzen, das die Auswertung des Ausdrucks implementiert. Eine Umsetzung ist der an der Universität Konstanz in der Arbeitsgruppe für Datenbanken & Informationssyteme entwickelte *XQuery-Compiler Pathfinder* [4].

Ein Feature von *XQuery*, das *Pathfinder* unterstützt, ist der *XML Schema* Import. Mit Hilfe eines *Schema* Imports können in *XQuery* neue, in einem *XML Schema* [5] definierte, Typen sichtbar gemacht werden. Die Umsetzung des Imports ist der Inhalt dieser Arbeit.

Die ersten drei Abschnitte (Abschnitt 1 – Abschnitt 3) geben eine kurze Einführung in *XML*, *XQuery* und *XML Schema*. Da sich die Typsysteme von *XQuery* und *XML Schema* um einiges unterscheiden, ist ein wichtiger Bestandteil des *Schema* Imports die Abbildung von *XML Schema*-Typen nach *XQuery*-Typen. Abschnitt 4 stellt die einzelnen Typen der beiden Sprachen vor und bereitet damit das Verständnis für das *Mapping* in Abschnitt 5. Das *Mapping* gibt für jede mögliche Ausprägung der einzelnen *Schema*-Komponenten eine Abbildungsregel an.

Im weiteren wird *SAX* [6] vorgestellt (Abschnitt 6), das ein *XML Schema* in seine Komponenten zerlegt und damit das *Mapping* steuert. Abschnitt 7 stellt eine *State Machine* vor, die komplett durch *SAX-events* angetrieben wird. Anhand der *events* und einer Zustandstabelle (*State Table*) werden die richtigen Abbildungsregeln ausgewählt und ausgeführt. Damit ist der *XML Schema* Import vollzogen.

Zuletzt erklärt Abschnitt 8 Ausschnitte der Implementation des ***XML Schema* Import for the *Pathfinder XQuery Compiler***.

## 1 *XML*

Die *Extended Markup Language* (*XML*) [1] wurde vom *World Wide Web Consortium* (*W3C*) aus der *Standard Generalized Markup Language* (*SGML*) [7] entwickelt. Die *SGML* wird wegen ihrer hohen Komplexität selten eingesetzt. *XML* dagegen ist eine einfachere Beschreibungssprache, die dennoch die Ausdruckskraft von *SGML* erreicht. Eine Menge von wenigen, unterschiedlichen Komponenten sorgt für eine gute Verständlichkeit:

- **Elemente** (`start elements`, `end elements`):
  Elemente sind das wichtigste Ausdrucksmittel von *XML*. Mit Ihnen können Daten in Blöcke unterteilt werden. Jedes *XML*-Dokument besteht aus einem Block wobei jeder Block durch ein `start element` (<*Name*>) eingeleitet und durch ein gleichnamiges `end element` (</*Name*>) beendet wird. Innerhalb eines Blocks können weitere Blöcke geschachtelt sein, so dass mit Hilfe der Elemente eine beliebig breite und tiefe Verschachtelungs-Struktur erstellt werden kann.

- **Zeichenketten** (`character-data`):
  *XML* bietet die Möglichkeit innerhalb jedes Blocks nicht nur weitere Blöcke sondern auch Zeichenketten einzufügen. Diese Zeichenketten können z.B. als Text-, Zahlen-, Daten- oder Listen-Formate interpretiert werden.

- **Attribute** (`attributes`):
  Attribute werden in der Form *Attribut*="*Wert*" innerhalb eines `start element` angegeben. Sie dienen dazu, die durch `start element` und `end element` begrenzten Blöcke genauer zu spezifizieren, bzw. mit Inhalt zu füllen.

- **sonstige Komponenten** (`comments`, `processing-instructions`, `CDATA`): in *XML* können zusätzlich Kommentare (`comments`), Prozessor-Anweisungen (`processing-instructions`) oder uninterpretierte Zeichen (`CDATA`) eingefügt werden. Diese werden von Anwendungen gesondert behandelt und spielen in dieser Arbeit keine weitere Rolle.

Losgelöst von Anwendungen, Systemen oder Plattformen bietet *XML* nicht nur die Möglichkeit neue Dialekte wie z.B. *XHTML* [8], *XSL* [9] oder *XML Schema* [5] auszudrücken, sondern auch große Mengen an Daten (`elements`, `attributes`, `character-data`) klar strukturiert (`start elements`, `end elements`) zu speichern.

Das *XML*-Dokument in Beispiel 1.1 stellt eine Video-Sammlung dar, in der die Sammlung (`<videos> ... </videos>`) mehrere Videos enthält. Jedes Video ist darin durch seinen eigenen geschachtelten Block (`<video> ... </video>`) klar strukturiert. Die Nummerierung der Videos übernimmt das Attribut `nr` und die Informationen zu den Filmen werden durch weitere Elemente (z.B. `<title>`) bzw. Zeichenketten (z.B. `"Finding Nemo"`) dargestellt.

**Beispiel 1.1 (*XML*-Dokument (**`"videos.xml"`**)):**

```
<?xml version="1.0"?>
 <videos>
  <video nr="1">
    <title>Finding Nemo</title>
    <language>English</language>
    <extra>1h40m</extra>
    <extra>Adventure Animation Comedy Family</extra>
  </video>
  <video nr="2">
    <title>Vita è bella, La</title>
    <language>Italian</language>
    <extra>Comedy Drama Romance War</extra>
    <extra>1997</extra>
  </video>
  <video nr="3">
    <title>Iron Giant, The</title>
    <language>English</language>
  </video>
</videos>
```

Zusätzlich zu der normalen Darstellung eines *XML*-Dokuments als Text gibt es die Repräsentation durch Bäume. Dabei werden die einzelnen, durch Elemente begrenzten Blöcke als Knoten dargestellt, die den Namen der jeweiligen Elemente tragen. Der Inhalt eines Blocks wird wieder durch Knoten (*Kind-Knoten*) dargestellt, wobei diese jeweils mit einer Kante zum *Vater-Knoten* verbunden werden. Rekursiv wird somit ein Baum aufgebaut.

Die Attribute (`attributes`) werden den Knoten als Information angehängt. Die Zeichenketten (`character-data`) werden in Knoten überführt, die Blätter genannt werden, da sie die unterste Ebene des Baumes bilden.

In der Baum-Repräsentation (Abbildung 1) des *XML*-Dokuments (Beispiel 1.1) bildet `videos` den *Wurzel-Knoten* unter dem drei `video`-Knoten hängen. Diese haben wiederum jeweils einen `title`, einen `language` und `extra` *Kind-Knoten.* Auf der untersten Ebene sind die *Blatt-Knoten*, wie z.B. die Zeichenketten `"Finding Nemo"` oder `"1h40m"`, zu finden.

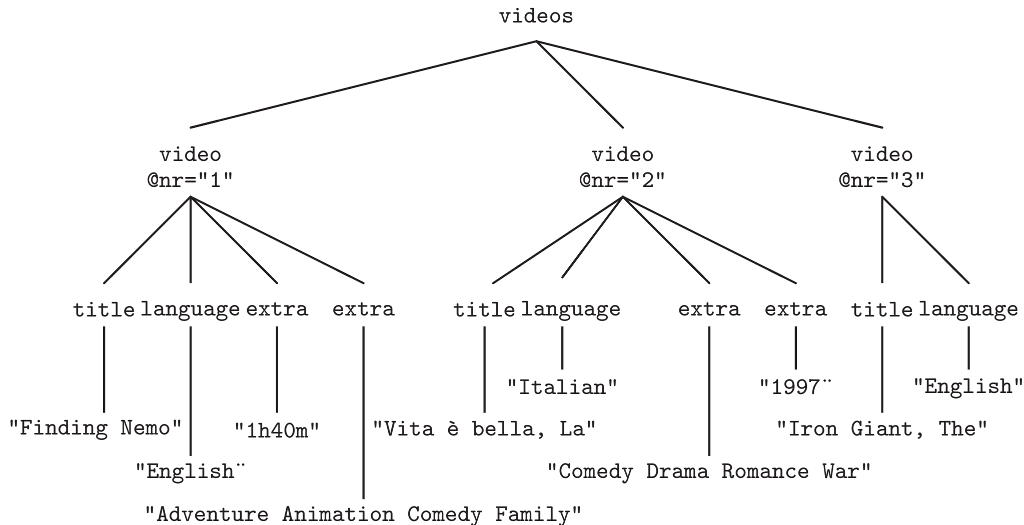


Abbildung 1: Baum-Repräsentation des *XML*-Dokuments (Beispiel 1.1)

## 2 *XQuery*

*XQuery* [2] ist eine Anfragesprache, die speziell auf *XML* angepasst ist. Sie bietet flexible Anfragemöglichkeiten um *XML*-strukturierte Daten zu durchsuchen und zu verarbeiten. Eine *XQuery*-Anfrage besteht aus einem Ausdruck, der ausgewertet wird und einen Wert als Ergebnis liefert. Dabei legt *XQuery* weder die Reihenfolge der Ausführung des Ausdrucks fest, noch löst sie Seiteneffekte aus. Ein Ausdruck kann aus einer einfachen Anfrage wie z.B. `3 + 4` bestehen oder wiederum geschachtelt weitere Ausdrücke beinhalten (z.B. `(3+4)*(3-4)`). Dabei sind die Anfragen natürlich nicht auf Zahlen beschränkt, sondern können z.B. auch Pfad-Anfragen [10] sein.

**Beispiel 2.1 (*XQuery*-Anfrage):**
Suche die Titel der Videos.

```
let $vids := doc("videos.xml")/videos
return $vids//title/text()
```

Der `videos`-Knoten wird durch das `let` an die Variable `$vids` gebunden. Ausgehend davon werden alle Knoten gesucht, die den Namen `title` tragen und deren Textinhalt zurückgegeben. Das Ergebnis ist eine Sequenz von Titeln.

**Beispiel 2.2 (Ergebnis der *XQuery*-Anfrage (Beispiel 2.1)):**
`"Finding Nemo", "Vita è bella, La", "Iron Giant, The"`

Diese Sequenz kann in *XQuery* wieder als Eingang für eine weitere Anfrage verwendet werden. Zur Iteration über Sequenzen gibt es in *XQuery* den Ausdruck `for`, der jedes Element sukzessiv an eine Variable bindet. Im Zusammenspiel mit dem Ausdruck `let` (der das ganze Ergebnis an eine Variable bindet), `where` (der weitere Einschränkungen erlaubt), `order by` (der eine Sequenz sortiert) und `return` (der das Ergebnis zurückliefert) entsteht der *FLWOR*-Ausdruck, welcher komplexere Anfragen ermöglicht.

**Beispiel 2.3 (*XQuery*-Anfrage):**
Suche alle englischen Videos und liefere diese sortiert nach Titeln ohne das `language`-Element zurück.

```
<english_videos>
  {
    for $a in doc("videos.xml")/videos/video
    where $a/language/text() = "English"
    order by $a/title
    return
      <video>
        { $a/title }
        { $a/extra }
      </video>
  }
</english_videos>
```

In Beispiel 2.3 bindet `for` an die Variable `$a` jeweils einen `video`-Knoten, `where` begrenzt die Sequenz auf die `video`-Elemente, deren `language`-Knoten den Text `"English"` enthalten. `order by` sortiert nach dem Titel der Videos und für jedes Video wird ein neues `video`-Element zurückgegeben, welches nur noch `title`- und `extra`-Knoten besitzt.

**Beispiel 2.4 (Ergebnis der *XQuery*-Anfrage (Beispiel 2.3)):**

```
<english_videos>
  <video>
    <title>Finding Nemo</title>
    <extra>1h40m</extra>
    <extra>Adventure Animation Comedy Family</extra>
  </video>
  <video>
    <title>Iron Giant, The</title>
  </video>
</english_videos>
```

Weiterhin kann nicht nur nach Knoten, wie in Pfad-Ausdrücken (Beispiel 2.1), oder nach Inhalten von Knoten, wie in dem *FLWOR*-Ausdruck (Beispiel 2.3), sondern auch nach Typen gesucht werden. *XQuery* bietet dazu einige eingebaute atomare Typen wie z.B. `integer`, `boolean`, `string`, `duration` oder `date` (siehe Abschnitt 4.3). Der Element-Test (`element(`*Name*`,` *Typ*`)`) ist eine Möglichkeit Elemente nach Namen bzw. Typen der Elemente (z.B. eingebaute Typen wie `duration`) zu überprüfen.

**Beispiel 2.5 (*XQuery*-Anfrage):**
Suche die Videos, bei denen die Länge des Filmes bekannt ist und gib den Titel zurück.

```
for $a in doc("videos.xml")/videos/video
for $b in $a/extra
  return typeswitch ($b)
    case $c as element(*,duration) return $a/title
    default return ()
```

In Beispiel 2.5 ist die Variable `$b` an ein `extra`-Element gebunden, bei dem der Element-Test den Typ `duration` für den Inhalt des `extra`-Elements erfordert. Das Ergebnis ist das `title`-Element, des an die Variable `$a` gebundenen `video`-Elements.

**Beispiel 2.6 (Ergebnis der *XQuery*-Anfrage (Beispiel 2.5)):**

```
<title>Finding Nemo</title>
```

Hier wurde bei der Suche der eingebaute Typ `duration` von *XQuery* verwendet. Es ist allerdings auch möglich neue Typen mit Hilfe eines *XML Schema* Import in die *XQuery*-Typumgebung hinzuzufügen. Diese importierten Typen können dann wie eingebaute Typen in *XQuery* verwendet werden. In Beispiel 2.7 kann z.B. eine Liste von Genres (`genre_list`) im `typeswitch`-Ausdruck erkannt und verarbeitet werden.

**Beispiel 2.7 (*XQuery*-Anfrage):**
Ersetze die `extra`-Knoten durch entsprechend passende Elemente wie `<duration/>`[1], `<year/>` oder `<genres/>`

```
import schema "videos.xsd"
<videos>
  {
  for $a in doc("videos.xml")/videos/video
  return
    <video nr="{ $a/@nr }">
      { $a/title }
      { $a/language }
      for $b in $a/extra
       return typeswitch ($b)
         case $c as element(*,genre_list) return
           <genres>{ $c }</genres>
         case $c as element(*,duration) return
           <duration>{ $c }</duration>
         case $c as element(*,integer) return
           <year>{ $c }</year>
         default return $b
    </video>
  }
</videos>
```

**Beispiel 2.8 (Ergebnis der *XQuery*-Anfrage (Beispiel 2.7)):**

```
   <videos>
    <video nr="1">
      <title>Finding Nemo</title>
      <language>English</language>
      <duration>1h40m</duration>
      <genres>Adventure Animation Comedy Family</genres>
    </video>
    <video nr="2">
      <title>Vita è bella, La</title>
      <language>Italian</language>
      <genres>Comedy Drama Romance War</genres>
      <year>1997</year>
    </video>
    <video nr="3">
      <title>Iron Giant, The</title>
      <language>English</language>
    </video>
  </videos>
```

[1] `<duration/>` ist eine abkürzende Schreibweise für `<duration></duration>`

# 3 *XML Schema*

*XML Schema* [5] ist ein *XML*-Dialekt, der wie die *Document Type Definition* (*DTD*) den Aufbau eines *XML*-Dokuments beschreibt. Allerdings kann *XML Schema XML*-Dokumente wesentlich genauer beschreiben, als eine *DTD*. Mit *XML Schema* kann sowohl die Struktur als auch der Inhalt und die Semantik von *XML*-Dokumenten definiert werden. Anhand dieser *Schema*-Definitionen kann maschinell überprüft werden, ob ein *XML*-Dokument nach den vorgeschriebenen Regeln aufgebaut ist (*validation*). Die Überprüfung eines *XML*-Dokuments mit Hilfe der *validation* ermöglicht es einen Überblick über den Aufbau des *XML*-Dokuments zu erlangen. Damit ist der Weg für einen besseren Zugriff (bekannte Typen, effizientere Algorithmen, einfachere Pfadsuche, ...) auf das *XML*-Dokument geebnet.

**Beispiel 3.1 (*XML Schema* ("videos.xsd")):**

```
1 <?xml version="1.0"?>
2 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
3   <xsd:element name="videos">
4     <xsd:complexType>
5       <xsd:sequence maxOccurs="unbounded">
6         <xsd:element name="video" type="video"/>
7       </xsd:sequence>
8     </xsd:complexType>
9   </xsd:element>
10
11  <xsd:complexType name="video">
12    <xsd:sequence>
13      <xsd:element name="title" type="xsd:string"/>
14      <xsd:element name="language" type="xsd:string"/>
15      <xsd:element name="extra" type="extra_types"
16         minOccurs="0" maxOccurs="3"/>
17    </xsd:sequence>
18    <xsd:attribute name="nr" type="xsd:integer"/>
19  </xsd:complexType>
20
21  <xsd:simpleType name="extra_types">
22    <xsd:union memberTypes="year duration genre_list"/>
23  </xsd:simpleType>
24  <xsd:simpleType name="year" type="xsd:integer"/>
25  <xsd:simpleType name="duration" type="xsd:duration"/>
26
27  <xsd:simpleType name="genre">
28    <xsd:restriction base="xsd:String">
29      <!-- komplette Auflistung aller Genres -->
30      <xsd:enumeration value="Adventure"/>
31      <xsd:enumeration value="Animation"/>
32      ...
33      <xsd:enumeration value="War"/>
34    </xsd:restriction>
35  </xsd:simpleType>
36
37  <xsd:simpleType name="genre_list">
38    <xsd:list itemType="genre"/>
39  </xsd:simpleType>
40 </xsd:schema>
```

Beispiel 3.1 zeigt das zu dem *XML*-Dokument (Beispiel 1.1, Seite 5) passende *Schema*. Es erlaubt ein *globales*[2] Element `videos` (Zeilen 2-9), das beliebig viele `video`-Elemente beinhaltet. Ein `video`-Element hat den Typ `video`, der wiederum ein `nr`-Attribut, ein `title`-, ein `language`- und null bis drei `extra`-Elemente definiert (Zeilen 11-19). Die Typen des `nr`-Attributs und der Elemente `title` und `language` sind eingebaute Typen (`built-in types`). Der Typ des Elements `extra` ist ein neuer Typ `extra_types`, der in den Zeilen 21-39 definiert wird. Auf den Typ `genre_list` (Zeilen 37-39) wird im Beispiel 2.7 zugegriffen. Er ist durch den *XML Schema* Import in *XQuery* sichtbar.

# 4 *XML Schema* und *XQuery* Typen

*XML Schema* und *XQuery* verwenden verschiedene Typen. In diesem Abschnitt werden die beiden Typsysteme genauer erklärt, um in Abschnitt 5 das Abbilden von *XML Schema* nach *XQuery* besser nachvollziehen zu können. Zuerst wird allerdings das Konzept der *QNames* und *Namespaces* [11] vorgestellt, welches sowohl in *XQuery*, als auch in *XML Schema* verwendet wird.

## 4.1 *QNames* und *Namespaces*

*XML*-Dokumente können beliebig groß werden. Ab einer gewissen Größe oder dann, wenn *XML*-Dokumente aus verschiedenen Quellen zusammengesetzt werden, würde die Namensgebung ein Problem darstellen, da die Namen entweder sehr lang werden oder in Konflikt treten würden. Um diesem Problem vorzugreifen wurde das Konzept der *Qualified Names* (*QNames*) eingeführt. Mit ihnen ist eine Unterteilung des Namensraums möglich. Ein *QName* besteht aus einem *Namespace-prefix* und einem mit Doppelpunkt getrennten *local part* (z.B. `xsd:complexType`), wobei der *local part* aus einer Zeichenkette besteht und der *Namespace-prefix* an einen *Namespace* gebunden ist.

Ein *Namespace* setzt sich aus einer Abkürzung (*prefix*) und einer daran gebundenen Adresse (*Uniform Resource Identifier – URI*) zusammen. Ein *URI* wird mittels einer *Namespace*-Deklaration `xmlns:`*prefix*`="`*URI*`"` an einen *prefix* gebunden. Die Bindung kann in jedem *XML*-Element als Attribut definiert werden und ist bis zum `end element` (`</xsd:complexType>`) sichtbar. Damit kann ein *QName* innerhalb und außerhalb eines Elements aus der gleichen Zeichenkette bestehen, aber dennoch wegen eines unterschiedlichen *URI* nicht übereinstimmen. Ist kein *prefix* vorhanden (z.B. `complexType`), dann nimmt der *QName* den Standard-*URI* des Dokuments an, der durch den Ausdruck `xmlns="`*URI*`"` festgelegt wird.

**Beispiel 4.1 (*QName* und *Namespace*):**

```
xmlns:foo="www.example.com/foo"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns="http://www.w3.org/2001/XMLSchema"

xsd:string != xsd:integer  -- local part stimmt nicht überein
xsd:string != foo:string   -- URI stimmt nicht überein
    string != foo:string   -- URI stimmt nicht überein
xsd:string  =     string   -- URI und local part stimmen überein
```

## 4.2 *XML Schema* Typen

*XML Schema* beschreibt den Aufbau von *XML*-strukturierten Daten und kann somit als Typsystem für *XML* verstanden werden. Aus diesem Grund werden in diesem

[2] *globale* Typen siehe Abschnitt 4.2

Abschnitt die *XML Schema*-Komponenten beschrieben. Bei jedem *Schema*-Element sollen ein oder mehrere *Schema*-Beispiele den Zusammenhang zwischen den einzelnen Komponenten darstellen. Zusätzlich wird der der Typsyntax durch den `Inhalt` bzw. den `Kontext` der Komponenten aufgeführt. Dabei wird für den `Inhalt` die Syntax eines regulären Ausdrucks verwendet. Der `Kontext` listet nur die Elemente auf, in denen die aktuelle Komponente direkt beinhaltet sein kann.

Da ein *Schema* ein gültiges *XML*-Dokument darstellt, besteht es hauptsächlich aus Elementen, Attributen und Zeichenketten. Um den Unterschied zwischen Elementen und der *Schema*-Komponente `element` bzw. Attributen und `attribute` deutlich zu machen wird im weiteren das *Schema*-Element 'Element' genannt und ein *Schema*-Attribut 'Attribut'. Die in *XML Schema* definierte Komponente `element` wird mit 'Element-Typ' oder '`element`' bzw. die Komponente `attribute` mit 'Attribut-Typ' oder '`attribute`' bezeichnet.

### 4.2.1 `schema`

Die formale Definition des Elements `schema` zeigt, welche Komponenten direkt darin enthalten sein können. Diese werden in den folgenden Abschnitten alle erklärt, wobei einige Typen wie z.B. `annotation` nur der Vollständigkeit halber aufgelistet sind. Sie sind durch eine kleinere Schriftart erkennbar und werden beim *Schema* Import ignoriert. In Abschnitt 4.2.20 'Ignorierte *XML Schema*-Komponenten' werden sie dennoch kurz vorgestellt.

**Typsyntax** (schema):

```
Inhalt:
( include | import | redefine | annotation)*,
( simpleType
| complexType
| element
| attribute
| attributeGroup
| group
| notation ),
( annotation )* )*
```

Jedes *Schema* wird durch das Element `schema` eingeleitet bzw. abgeschlossen. `schema` bildet somit die Hülle für alle im *Schema* definierten Typen. Damit unterscheidet `schema` die Komponenten, die direkt in einem *Schema* enthalten sind, von denen, die tiefer geschachtelt auftreten. Die Komponenten der ersten Gruppe werden als '`global`' oder '*globale*' Typen bezeichnet und sind nach außen hin sichtbar bzw. referenzierbar. Die tiefer geschachtelten *Schema*-Elemente werden als '`local`' oder '*lokale*' Typen bezeichnet und sind nicht nach außen hin sichtbar bzw. nicht referenzierbar.

Referenzen dienen der Kombination von Komponenten in *XML Schema*. Sie verbinden z.B. einzelne *globale* Typen zu einem komplexen Typ. Obwohl ein Typ so definiert werden könnte, dass er alle Definitionen in einem Typ vereinigt, ist eine Zerlegung der Typen sinnvoll: Die Modularität der Typen sorgt für eine bessere Übersicht bzw. Verständlichkeit des *Schema* und erlaubt die mehrfache Verwendung von Typen. Aus den selben Gründen unterstützt *XML Schema* einige weitere Referenzen (siehe folgende Abschnitte – Attribut `ref`).

**Beispiel 4.2 (*globale* bzw. *lokale* Typen):**

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="movies" type="video"/>
  <xsd:element name="films" type="video"/>
  <xsd:complexType name="video">
    <xsd:sequence>
      <xsd:element name="title" type="xsd:string"
         maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

In Beispiel 4.2 sind das `element movie`, das `element film` und der `complexType video` *globale* Typen. Deshalb darf das `element movie` den `complexType video` als Typ referenzieren (`type="video"`). Die `sequence` und das `element title` dagegen sind *lokale* Typen. Eine Referenz auf sie ist daher nicht möglich.

Die Definition eines *Namespace* ist außer der Unterscheidung von *globalen* und *lokalen* Typen die wichtigste Aufgabe. Da sämtliche Elemente in *XML Schema* den vordefinierten *Schema Namespace*[3] tragen müssen, muss dieser im ersten Element (`<schema>`) definiert werden (`xmlns:xsd="http://www.w3.org/2001/XMLSchema"`).

### 4.2.2 `element`

*XML Schema* besitzt ein Element `element`, welches ein *XML*-Element (`<../>`) beschreibt. Es kann sowohl *global* als auch *lokal* auftreten. Der `Inhalt` besteht aus einem optionalen `simpleType` bzw. `complexType`.

**Typsyntax** (`element`):

```
Kontext: schema, all, choice, sequence
Inhalt:
annotation?, ( complexType | simpleType )?,
( unique | key | keyref )*
```

Ein `element` kann zusätzlich eine Reihe von Attributen enthalten. Anhand der Attribute können zwei Konzepte unterschieden werden:

1. `element` mit dem Attribut `name`:
   `element` beschreibt ein neues *XML*-Element. Der Name des neuen Elements entspricht dem Wert des `name`-Attributs.

   **Beispiel 4.3 (Element-Typ mit `name`-Attribut):**
   ```
   <xsd:element name="foo"/>
   ```

   **Beispiel 4.4 (entsprechendes *XML*-Element (Beispiel 4.3)):**
   ```
   <foo/>
   ```

   Der Typ des `element` kann entweder mit dem Attribut `type` definiert werden oder durch einen geschachtelten `simpleType` bzw. `complexType`.
   Wenn es sich um `global elements` handelt dürfen die Attribute `ref`, `minOccurs` und `maxOccurs` nicht auftreten. In `local elements` können die Attribute `minOccurs` und `maxOccurs` verwendet werden (mehr dazu siehe Abschnitt 4.2.8).

[3] *XML Schema Namespace*: `"http://www.w3.org/2001/XMLSchema"`

**Beispiel 4.5 (*globale* bzw. *lokale* Element-Typen):**

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="videos">
    <xsd:complexType>
      <xsd:sequence maxOccurs="unbounded">
        <xsd:element name="video" type="video"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="language" type="xsd:string"/>
</xsd:schema>
```

2. `local element` mit dem Attribut `ref`:
   Ein Element-Typ mit dem Attribut `ref` verweist auf ein `global element`. Das `ref`-Attribut bedeutet somit eine Ersetzung des `local element` durch das referenzierte `global element`.
   In einem `local element` mit dem Attribut `ref` können die Attribute `minOccurs` und `maxOccurs` genauso wie in anderen `local elements` verwendet werden Die Attribute `name` und `type` dagegen sind nicht erlaubt.

   **Beispiel 4.6 (Element-Typ mit `ref`-Attribut):**

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="video">
    <xsd:sequence>
      <xsd:element ref="title"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:element name="title" type="xsd:string"/>
</xsd:schema>
```

   Beispiel 4.6 ersetzt mit dem Attribut `ref` das `local element` mit dem `global element title`.

### 4.2.3 `group`

*XML Schema* bietet verschiedene Typen von Gruppen an. Diese fassen entweder `elements` oder `attributes` zusammen. Für die Gruppierung der `elements` handelt es sich dabei zum einen um anonyme Gruppen (`all`, `choice` und `sequence` – siehe Abschnitte 4.2.4, 4.2.5 und 4.2.6), zum anderen um das Element `group` das eine benannte Gruppe repräsentiert.

**Typsyntax** (group):

```
Kontext: schema, complexType, choice, sequence,
         extension, restriction, redefine
Inhalt:
annotation?, ( all | choice | sequence)?
```

Der `Inhalt` einer `group` besteht aus einer anonymen Gruppe und die `group` kann wie ein `element` sowohl *global* als auch *lokal* verwendet werden. Aus dieser Unterscheidung folgt für eine `group` die Unterteilung in zwei Konzepte:

1. `global groups` mit dem Attribut `name`

2. `local groups` mit dem Attribut `ref`

Dabei übernimmt das Element `local group` die Referenz (ähnlich wie `ref` bei den `elements`), die auf die `global group` zugreift.

**Beispiel 4.7** (`group`):

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="video">
    <xsd:sequence>
      <xsd:group ref="video_group"/>
    </xsd:sequence>
  </xsd:element>
  <xsd:group name="video_group">
    <xsd:sequence>
      <xsd:element name="title" type="xsd:string"/>
      <xsd:element name="language" type="xsd:string"/>
    </xsd:sequence>
  </xsd:group>
</xsd:schema>
```

In Beispiel 4.7 wird auf die `global group video_group` innerhalb des `element video` verwiesen. Damit ist das `element title` innerhalb von `video` erlaubt (siehe Beispiel 4.8).

**Beispiel 4.8 (entsprechendes *XML*-Element (Beispiel 4.7)):**

```
  <video>
    <title>Finding Nemo</title>
    <language>English</language>
  </video>
```

### 4.2.4 `all`

`all` ist eine der drei anonymen Gruppen, die `elements` zusammenfasst. Eine `all`-Gruppe darf nicht in einer anderen anonymen Gruppe enthalten sein und in ihr dürfen nur `elements` auftreten.

**Typsyntax** (`all`):

```
Kontext: complexType, group, extension, restriction
Inhalt:
annotation?, ( element )*
```

Jedes `element` innerhalb einer `all`-Gruppe darf entweder gar nicht oder maximal einmal verwendet werden (`minOccurs="0"` oder `minOccurs="1"`; `maxOccurs="1"`). Eine `all`-Gruppe definiert eine Menge von `elements`, die dann in einer *XML*-Instanz in beliebiger Reihenfolge auftauchen können (siehe Beispiel 4.10 `title` nach `language`).

**Beispiel 4.9** (`all`):

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="video">
    <xsd:complexType>
      <xsd:all>
        <xsd:element name="title" type="xsd:string"/>
        <xsd:element name="language" type="xsd:string"
           minOccurs="0"/>
      </xsd:all>
      <xsd:attribute name="nr" type="xsd:integer"/>
```

```
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

**Beispiel 4.10 (entsprechendes *XML*-Element (Beispiel 4.9)):**

```
  <video>
    <language>English</language>
    <title>Finding Nemo</title>
  </video>
```

### 4.2.5 `choice`

`choice` bildet die zweite anonyme `element`-Gruppe. Der Inhalt des Elements `choice` sind `elements` oder Gruppen, die selbst wieder Gruppen oder `elements` beinhalten.

**Typsyntax** (`choice`):

```
   Kontext complexType, group, choice, sequence, extension,
           restriction
   Inhalt:
   annotation?, ( element | group | choice | sequence | any )*
```

Von den in einer `choice`-Gruppe erlaubten `elements` bzw. Gruppen darf nur ein `element` bzw. ein Gruppen-Element ausgewertet werden. Das heißt ein *XML*-Element in Beispiel 4.11 darf nur eines der in der `choice` beinhalteten `elements` nutzen (entweder `duration` oder `year`).

**Beispiel 4.11** (`choice`):

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="extra" type="extra"/>
  <xsd:complexType name="extra">
    <xsd:choice>
      <xsd:element name="duration" type="xsd:duration"/>
      <xsd:element name="year" type="xsd:integer"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:schema>
```

**Beispiel 4.12 (entsprechendes *XML*-Element (Beispiel 4.11)):**

```
<extra>
  <year>English</year>
</extra>
```

### 4.2.6 `sequence`

Die dritte anonyme `element`-Gruppe ist die `sequence`, die die gleichen Typen wie `choice` erlaubt.

**Typsyntax** (`sequence`):

```
   Kontext: complexType, group, choice, sequence, extension,
            restriction
   Inhalt:
   annotation?, ( element | group | choice | sequence | any )*
```

Allerdings werden im Gegensatz zur `all`- und `choice`-Gruppe in einer *XML*-Instanz alle aufgeführten `elements` bzw. Gruppen in der deklarierten Reihenfolge erwartet. In Beispiel 4.13 muss `title` vor dem `element language` auftreten (siehe Beispiel 4.14)

**Beispiel 4.13** (`sequence`)**:**
```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="video" type="video"/>
  <xsd:complexType name="video">
    <xsd:sequence>
      <xsd:element name="title" type="xsd:string"/>
      <xsd:element name="language" type="xsd:string"/>
      <xsd:element name="extra" type="extra_types"
         minOccurs="0" maxOccurs="3"/>
    </xsd:sequence>
    <xsd:attribute name="nr" type="xsd:integer"/>
  </xsd:complexType>
</xsd:schema>
```

**Beispiel 4.14 (entsprechendes *XML*-Element (Beispiel 4.13)):**
```
<video nr="1">
  <title>Finding Nemo</title>
  <language>English</language>
  <extra>1h40m</extra>
  <extra>Adventure Animation Comedy Family</extra>
</video>
```

### 4.2.7 `any`

Mit `any` wird ein `wildcard element` eingeführt, das in Gruppen auftreten kann. `any` entspricht damit einem beliebigen `element`.

**Typsyntax** (`any`)**:**
```
Kontext: choice, sequence
Inhalt:
annotation?
```

### 4.2.8 `element occurrences`

`minOccurs` und `maxOccurs` sind die Attribute, mit welchen bei `elements` und Gruppen die Häufigkeit gesteuert wird. Die beiden Attribute legen einen Bereich fest, bei dem `minOccurs` die untere und `maxOccurs` die obere Schranke bildet. Beide Attribute sind optional und haben im Standardfall den Wert `"1"`. Der Wertebereich von `minOccurs` beginnt bei `"0"` und beinhaltet alle positiven `integer`. Bei `"1"` beginnend kann `maxOccurs` alle positiven `integer` und zusätzlich den Wert `"unbounded"` (unendlich viele Vorkommen) annehmen.

Die beiden Attribute dürfen nur in `all`, `choice`, `sequence`, `any`, `local groups` und `local elements` auftreten. In `global elements` bzw. `global groups` sind sie verboten. Zu beachten ist außerdem der Sonderfall `all`, bei dem `minOccurs` und `maxOccurs` nur mit den Wert `"1"` benutzt werden dürfen.

**Beispiel 4.15** (`element occurrences`)**:**
```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="phonebook">
    <xsd:sequence maxOccurs="150">
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="prename" type="xsd:string" maxOccurs="10"/>
      <xsd:element name="tel" type="phone_nr" minOccurs="0"
         maxOccurs="unbounded"/>
    </xsd:sequence>
```

```
  </xsd:complexType>
</xsd:schema>
```

### 4.2.9 `attribute`

In *XML Schema* werden die Attribute eines *XML*-Elements mit Hilfe von Attribut-Typen definiert. Ihr Typ darf nicht wie bei den `elements` aus einem `complexType` bestehen, der selbst wieder `elements` und `attributes` enthalten kann, sondern nur aus einem `simpleType` (Erklärung von `complexType` und `simpleType` siehe folgende Abschnitte).

**Typsyntax** (`attribute`):

```
Kontext: schema, complexType, attributeGroup, extension,
         restriction
Inhalt:
annotation?, ( simpleType )?
```

Die `attributes` benutzen wie `elements` die Attribute `name`, `type` und `ref`. Es sind ähnlich wie bei `elements` zwei Konzepte zu unterscheiden:

1. `attribute` mit dem Attribut `name`:
   Der Attribut-Typ beschreibt ein neues Attribut eines *XML*-Elements. Der Name des neuen Attributs entspricht dabei dem Wert des `name`-Attributs. Der Typ des neuen Attributs entspricht dem Wert des `type`-Attributs.

   **Beispiel 4.16 (Attribut-Typ mit `name`- und `type`-Attribut):**
   ```
   <xsd:attribute name="nr" type="xsd:integer"/>
   ```

   **Beispiel 4.17 (entsprechendes *XML*-Attribut (Beispiel 4.16)):**
   ```
   <... nr="42" .../>
   ```

   Ist kein Attribut `type` vorhanden, dann kann ein geschachtelter `simpleType` den Typ definieren.

2. `local attributes` mit dem Attribut `ref`:
   Ähnlich wie bei `elements` stellt das Attribute `ref` die Möglichkeit einer Referenz zur Verfügung. Das `ref`-Attribut bedeutet somit eine Ersetzung des `local attribute` durch das referenzierte `global attribute`. Ein `ref`-Attribut darf nur verwendet werden, wenn `name`- und `type`-Attribut nicht verwendet werden und umgekehrt.

`use` ist ein weiteres Attribut, welches in `local attributes` erlaubt ist. Es legt fest, ob ein `attribute` vorhanden sein muss, optional verwendet werden kann oder nicht erlaubt ist. Dementsprechend kann `use` auch die drei Werte `"required"`, `"optional"` und `"prohibited"` annehmen. Wenn `use` nicht verwendet wird, ist der Standardfall `use="optional"` aktiv.

Im Beispiel 4.19 tauchen die Attribute `title` und `extra` auf. `title` ist in Beispiel 4.18 als `"required"` bestimmt und muss deswegen auch vorkommen. `language` und `extra` haben keine `use`-Attribute und sind damit `"optional"`. Deswegen kann `extra` verwendet und `language` weggelassen werden. Das Attribut `nr` darf hingegen nicht benutzt werden.

**Beispiel 4.18** (attributes):

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="video" type="video"/>
  <xsd:complexType name="video">
    <xsd:attribute name="title" type="xsd:string" use="required"/>
    <xsd:attribute name="language" type="xsd:string"/>
    <xsd:attribute ref="extra"/>
    <xsd:attribute name="nr" type="xsd:integer" use="prohibited"/>
  </xsd:complexType>
  <xsd:attribute name="extra">
    <xsd:simpleType>
      <xsd:union memberTypes="year duration genre_list"/>
    </xsd:simpleType>
  </xsd:attribute>
</xsd:schema>
```

**Beispiel 4.19 (entsprechendes *XML*-Element (Beispiel 4.18)):**

```
<video title="Finding Nemo" extra="1h40m"/>
```

### 4.2.10 `attributeGroup`

Das Element `attributeGroup` fasst `attributes` zusammen, ähnlich wie eine `group elements`.

**Typsyntax** (attributeGroup):

```
Kontext: schema, complexType, attributeGroup, extension,
         restriction, redefine
Inhalt:
annotation?, ( attribute | attributeGroup )*, ( anyAttribute )?
```

Im Gegensatz zu `elements` ist die Reihenfolge von `attributes` nicht definiert (ähnlich einer `all`-Gruppe). Deswegen entfallen bei den `attributes` die anonymen Gruppen.

`attributeGroup` kann wie `group` in zwei Konzepte unterteilt werden:

1. `global attributeGroup` mit dem Attribut `name`
2. `local attributeGroup` mit dem Attribut `ref`

In Beispiel 4.20 übernimmt die `local attributeGroup` die referenzierende Rolle (genau wie eine `local group`). Die `global attributeGroup` sammelt die `attributes` bzw. weitere `local attributeGroups`, um von einer `local attributeGroup` referenziert zu werden.

**Beispiel 4.20** (attributeGroup):

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="video">
    <xsd:complexType>
      <xsd:attributeGroup ref="video"/>
    </xsd:complexType>
  </xsd:element>
  <xsd:attributeGroup name="video">
    <xsd:attribute name="title" type="xsd:string" use="required"/>
    <xsd:attribute name="language" type="xsd:string"/>
  </xsd:attributeGroup>
</xsd:schema>
```

**Beispiel 4.21 (entsprechendes XML-Element (Beispiel 4.20)):**
```
<video title="Finding Nemo" language="English"/>
```

### 4.2.11 `anyAttribute`

`anyAttribute` ist das `any` entsprechende `wildcard attribute`. Es kann an allen Stellen verwendet werden, an denen auch `attributes` auftreten dürfen und steht für beliebige `attributes`.

**Typsyntax** (`anyAttribute`):
```
Kontext: complexType, attributeGroup, extension, restriction
Inhalt:
annotation?
```

### 4.2.12 `complexType`

In den Abschnitten 4.2.2 – 4.2.11 wurden `elements`, `attributes` und ihre jeweiligen Gruppen vorgestellt. Diese Komponenten sind jeweils für einzelne Konzepte in *XML* zuständig. Die Kombination der Konzepte in einer gemeinsamen Struktur übernimmt das Element `complexType`. Es verbindet `element`-Gruppen mit `attributes` und `attributeGroups` zu einem Typ. Weitere `Inhalte` eines `complexType` sind `simpleContent` bzw. `complexContent`, die in Abschnitt 4.2.14 behandelt werden.

**Typsyntax** (`complexType`):
```
Kontext: schema, element, redefine
Inhalt:
annotation?,
( simpleContent
| complexContent
| (( all | choice | sequence | group )?,
   ( attribute | attributeGroup )*, anyAttribute? ))
```

In Beispiel 4.22 hat das `element videos` den Typ `complexType`, der ein `element video` enthält. Das `element video` bezieht seinen Typ auf den `global complexType video`, der die `elements title`, `language` und `extra` sowie das `attribute nr` kombiniert.

**Beispiel 4.22** (`complexType`):
```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="videos">
    <xsd:complexType>
      <xsd:sequence maxOccurs="unbounded">
        <xsd:element name="video" type="video"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:complexType name="video">
    <xsd:sequence>
      <xsd:element name="title" type="xsd:string"/>
      <xsd:element name="language" type="xsd:string"/>
      <xsd:element name="extra" type="extra_types"
         minOccurs="0" maxOccurs="3"/>
    </xsd:sequence>
    <xsd:attribute name="nr" type="xsd:integer"/>
  </xsd:complexType>
</xsd:schema>
```

Ein `local complexType` wird auch als anonymer `complexType` bezeichnet, da er kein `name`-Attribut trägt. Ein `global complexType` dagegen benötigt das Attribut `name`, damit er als Typ referenziert werden kann (siehe Beispiel 4.22 `type="video"`).

**Beispiel 4.23 (entsprechendes *XML*-Element (Beispiel 4.22)):**

```
<videos>
  <video nr="1">
    <title>Finding Nemo</title>
    <language>English</language>
    <extra>1h40m</extra>
    <extra>Adventure Animation Comedy Family</extra>
  </video>
</videos>
```

### 4.2.13 simpleType

Wie `complexType` stellt auch `simpleType` eine Struktur zur Verfügung, die Typen kombiniert. Allerdings treten in einem `simpleType` keine `elements` oder `attributes` auf, sondern Typen die durch Zeichenketten in *XML* repräsentiert werden können. Diese Typen sind Ableitungen eingebauter Typen und können als Typen von `elements`, `attributes`, oder weiteren Veränderungen von `simpleTypes` verwendet werden.

**Typsyntax** (`simpleType`):

```
Kontext: schema, element, attribute, list, union, restriction,
         redefine
Inhalt:
annotation?, ( list | union | restriction )
```

Ein `simpleType` ist mit einem `complexType` im Bezug auf Referenzierung mit Hilfe des Attributs `name` und der Anonymität bei *lokalen* Typen vergleichbar (siehe Beispiel 4.24 `type="genre_list"`).

**Beispiel 4.24** (`simpleType`):

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="genres" type="genre_list"/>
  <xsd:simpleType name="genre_list">
    <xsd:list>
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="Adventure"/>
          ...
          <xsd:enumeration value="War"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:list>
  </xsd:simpleType>
</xsd:schema>
```

Der Inhalt des *XML*-Elements `genres` besteht nun aus einer Liste von bestimmten `strings`.

**Beispiel 4.25 (entsprechendes *XML*-Element (Beispiel 4.24)):**

```
<genres>Adventure Animation Comedy Family</genres>
```

### 4.2.14 `simpleContent` und `complexContent`

`simpleContent` und `complexContent` können nur in dem Element `complexType` direkt enthalten sein und erfordern beide als Inhalt entweder ein `restriction`- oder ein `extension`-Element.

**Typsyntax** (`simpleContent`):

```
Kontext: complexType
Inhalt:
annotation?, ( restriction | extension )
```

**Typsyntax** (`complexContent`):

```
Kontext: complexType
Inhalt:
annotation?, ( restriction | extension )
```

Sie haben damit theoretisch keine Unterscheidungsmöglichkeit und könnten auch einfach weggelassen werden. In der Praxis werden sie allerdings dazu verwendet `restriction` bzw. `extension` zu unterteilen. Im weiteren wird deswegen `restriction` innerhalb von `complexContent` als `complex restriction` bezeichnet, und als `simple restriction`, falls es innerhalb eines `simpleContent` auftritt. Für `extension` gilt die selbe Unterteilung in `simple extension` und `complex extension`.

### 4.2.15 `extension`

`extension` erweitert die Definition eines schon definierten Typs. Dabei wird der bereits definierte Typ mit dem Attribut `base` referenziert und mit dem Inhalt der `extension` erweitert. `extension` kann dabei in zwei Konzepte aufgeteilt werden (siehe Abschnitt 4.2.14):

1. `simple extension`:

   **Typsyntax** (`simple extension`):

   ```
   Kontext: simpleContent
   Inhalt:
   annotation?, ( attribute | attributeGroup )*, anyAttribute?
   ```

   Eine `simple extension` kann `elements`, deren Typ aus einem `simpleType` besteht, um `attributes` bzw. `attributeGroups` erweitern. Ein `element` kann einen `simpleType` ansonsten nur mit dem Attribut `type` referenzieren, weshalb kein weiteres `attribute` geschachtelt sein könnte.

   **Beispiel 4.26** (`simple extension`):

   ```
   <?xml version="1.0"?>
   <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
     <xsd:element name="title">
       <xsd:complexType>
         <xsd:simpleContent>
           <xsd:extension base="xsd:string">
             <xsd:attribute name="language" type="xsd:string"/>
           </xsd:extension>
         </xsd:simpleContent>
       </xsd:complexType>
     </xsd:element>
   </xsd:schema>
   ```

   **Beispiel 4.27 (entsprechendes *XML*-Element (Beispiel 4.26)):**

   ```
   <title language="English">Finding Nemo</title>
   ```

Wäre `title` nicht mit einer `simple extension` erweitert worden, dann wäre entweder `<title language="English"/>` (mit einem `complexType`) oder (mit `type`-Attribut) `<title>Finding Nemo</title>`, aber nicht die Kombination aus `simpleType` und `attributes` (siehe Beispiel 4.26) möglich.

2. `complex extension`:
   Im Gegensatz zur `simple extension` erweitert eine `complex extension` einen `complexType`. Die hinzugefügten Typen können sich aus einer `element`-Gruppe, `attributes` und `attributeGroups` zusammensetzen.

   **Typsyntax** (`complex extension`):

   ```
   Kontext: complexContent, simpleContent
   Inhalt:
   annotation?, ( all | choice | sequence | group )?,
   ( attribute | attributeGroup )*, anyAttribute?
   ```

   **Beispiel 4.28** (`complex extension`):

   ```
   <?xml version="1.0"?>
   <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
     <xsd:complexType name="simple_video">
       <xsd:sequence>
         <xsd:element name="title" type="xsd:string"/>
         <xsd:element name="language" type="xsd:string"/>
       </xsd:sequence>
     </xsd:complexType>
     <xsd:element name="video">
       <xsd:complexType>
         <xsd:complexContent>
           <xsd:extension base="simple_video">
             <xsd:sequence>
               <xsd:element name="extra" type="extra_types"/>
             </xsd:sequence>
             <xsd:attribute name="nr" type="xsd:integer"/>
           </xsd:extension>
         </xsd:complexContent>
       </xsd:complexType>
     </xsd:element>
   </xsd:schema>
   ```

   Die Definitionen des `base`-Typs und der `extension` werden zusammengeführt, indem die beiden `element`-Gruppen in einer `sequence` kombiniert werden (Die `video elements` in Beispiel 4.28 und Beispiel 4.29 definieren beide das gleichen `element video`).

   **Beispiel 4.29 (Beispiel 4.28 ohne complex extension):**

   ```
   <?xml version="1.0"?>
   <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
     <xsd:element name="video">
       <xsd:complexType>
         <xsd:sequence>
           <xsd:element name="title" type="xsd:string"/>
           <xsd:element name="language" type="xsd:string"/>
           <xsd:element name="extra" type="extra_types"/>
         </xsd:sequence>
         <xsd:attribute name="nr" type="xsd:integer"/>
       </xsd:complexType>
   ```

```
  </xsd:element>
</xsd:schema>
```

### 4.2.16 `restriction`

Im Gegensatz zur `extension`, die einen Typ erweitert, schränkt `restriction` einen Typ ein. Genau wie `extension` besitzt `restriction` das Attribut `base`, das den bereits definierten Typ referenziert. Zusätzlich zu `simple` und `complex restriction` gibt es eine `simpleType restriction`, da eine `restriction` auch der direkte Inhalt eines `simpleType` sein kann. Die drei `restriction`-Arten werden im weiteren einzeln vorgestellt:

1. `simpleType restriction`
   Eine `simpleType restriction` schränkt nur `simpleTypes` ein. Dabei muss entweder auf das `base`-Attribut zugegriffen werden oder, falls dieses nicht vorhanden ist, ersatzweise ein geschachtelter `simpleType` die Rolle der `base` vertreten. Der `base`-Typ kann innerhalb der `restriction` durch `facets` (siehe Abschnitt 4.2.20 – Ignorierte *XML Schema*-Komponenten) eingeschränkt werden.

   **Typsyntax** (`simpleType restriction`):
   ```
   Kontext: simpleType
   Inhalt:
   annotation?, simpleType?, facet*,
   ```

   **Beispiel 4.30** (`simpleType restriction`):
   ```
   <?xml version="1.0"?>
   <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
     <xsd:element name="genre">
       <xsd:simpleType>
         <xsd:restriction base="xsd:string">
           <xsd:enumeration value="Adventure"/>
           <xsd:enumeration value="Animation"/>
           <xsd:enumeration value="Comedy"/>
           <xsd:enumeration value="War"/>
         </xsd:restriction>
       </xsd:simpleType>
     </xsd:element>
   </xsd:schema>
   ```

   **Beispiel 4.31 (entsprechendes *XML*-Element (Beispiel 4.30)):**
   ```
   <genre>Animation</genre>
   ```

   Das *XML*-Element in Beispiel 4.32 verletzt z.B. das in Beispiel 4.30 definierte *Schema*, da nur noch die aufgelisteten `strings` als Inhalt des `element genre` erlaubt sind.

   **Beispiel 4.32 (nicht passendes *XML*-Element (Beispiel 4.30)):**
   ```
   <genre>chair</genre>
   ```

2. `simple restriction`:
   Die `simple restriction` funktioniert genauso wie die `simpleType restriction`. Sie beschränkt ebenfalls einen `simpleType` durch `facets` und das `base`-Attribut kann auch durch einen `simpleType` innerhalb der `restriction` ersetzt werden. Allerdings sind in einer `simple restriction` wie in einer `simple extension` auch `attributes` erlaubt.

**Typsyntax** (`simple restriction`):

```
Kontext: simpleContent
Inhalt:
annotation?, simpleType?, facet*,
( attribute | attributeGroup )*, anyAttribute?
```

**Beispiel 4.33** (`simple restriction`):

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="genre">
    <xsd:complexType>
      <xsd:simpleContent>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="Abenteuer"/>
          <xsd:enumeration value="Animation"/>
          <xsd:enumeration value="Krieg"/>
          <xsd:attribute name="language" type="xsd:string"/>
        </xsd:restriction>
      </xsd:simpleContent>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

**Beispiel 4.34 (entsprechendes *XML*-Element (Beispiel 4.33)):**

```
<genre language="German">Abenteuer</genre>
```

3. `complex restriction`:
   In einer `complex restriction` wird ein `complexType` eingeschränkt, weshalb der `Inhalt` einer `complex restriction` dem eines `complexType` entspricht.

   **Typsyntax** (`restriction`):

   ```
   Kontext: complexContent
   Inhalt:
   annotation?, ( all | choice | sequence | group )?,
   ( attribute | attributeGroup )*, anyAttribute?
   ```

   Das Attribut `base` muss auf einen definierten `complexType` verweisen. Dessen Definitionen müssen in der `restriction` alle wiederholt werden und können z.B. durch Attribute wie `minOccurs`, `maxOccurs`, `use` oder durch spezifischere Typen beschränkt werden. Eine `complex restriction` erstellt also eine Untermenge des `base`-Typs. Ein Programm, das den `base`-Typ akzeptiert, versteht somit auch die, durch die `restriction` eingeschränkte, Untermenge. Die `element`- bzw. `attribute`-Definition, die im `base`-Typ aufgeführt sind, müssen wiederholt werden, da sich ansonsten die Struktur des `complexType` und damit die der Untermenge ändern würde.
   In Beispiel 4.35 schränkt `top50_videos` den Typ `videos` durch eine festgesetzte Anzahl von 50 `video elements` ein.

   **Beispiel 4.35** (`complex restriction`):

   ```
   <?xml version="1.0"?>
   <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
     <xsd:complexType name="videos">
       <xsd:sequence maxOccurs="unbounded">
         <xsd:element name="video" type="video"/>
       </xsd:sequence>
     </xsd:complexType>
   ```

```
    <xsd:element name="top50_videos">
      <xsd:complexType>
        <xsd:complexContent>
          <xsd:restriction base="videos">
            <xsd:sequence minOccurs="50" maxOccurs="50">
              <xsd:element name="video" type="video"/>
            </xsd:sequence>
          </xsd:restriction>
        </xsd:complexContent>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
```

### 4.2.17 `list`

`list` ist der Konstruktor für eine Liste, deren Elemente einen gemeinsamen Typ haben. `list` wird innerhalb eines `simpleType` verwendet und besitzt das Attribut `itemType`, das den Typ der Listenelemente beschreibt. Fehlt dieses Attribut, muss stattdessen ein `simpleType` in dem `list`-Element geschachtelt sein.

**Typsyntax** (`list`):

```
Kontext: simpleType
Inhalt:
annotation?, simpleType?
```

**Beispiel 4.36** (`list`):

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="genres">
    <xsd:simpleType>
      <xsd:list itemType="genre"/>
    </xsd:simpleType>
  </xsd:element>
  <xsd:simpleType name="genre">
    <xsd:restriction base="xsd:String">
      <xsd:enumeration value="Adventure"/>
      <xsd:enumeration value="Animation"/>
      ...
      <xsd:enumeration value="War"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

In einer *XML*-Instanz werden die einzelnen Elemente einer Liste durch ein Leerzeichen getrennt. Ein Element das ein Leerzeichen enthält würde in zwei Elemente aufgeteilt werden (`"Science Fiction"` würde z.B. als `"Science"` und `"Fiction"` aufgeteilt werden). Darum sollten *strings* Leerzeichen vermeiden oder durch eine `restriction` untersagen.

**Beispiel 4.37 (entsprechendes *XML*-Element (Beispiel 4.36)):**

```
<genres>Adventure Animation Comedy Family</genres>
```

### 4.2.18 `union`

`union` vereinigt verschiedene `simpleTypes` in einem einzigen `simpleType`. Aus der Vereinigung kann damit je nach Anwendung der passende `simpleType` gewählt werden. Die unterschiedlichen Typen werden in dem Attribut `memberTypes` als Liste

gespeichert. Dabei sind Leerzeichen (bzw. alle *white spaces*) die Trennzeichen. Ist das Attribut `memberTypes` nicht vorhanden wird stattdessen für jeden Typ ein geschachtelter `simpleType` erwartet.

**Typsyntax** (`union`):

```
Kontext: simpleType
Inhalt:
annotation?, simpleType*
```

**Beispiel 4.38** (`union`):

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="extra">
    <xsd:simpleType>
      <xsd:union memberTypes="year duration genre_list"/>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>
```

Die in `union` definierten Typen verhalten sich ähnlich wie eine `choice`-Gruppe. In Beispiel 4.38 darf z.B. nur einer der drei Typen in einem `extra element` benutzt werden.

**Beispiel 4.39 (entsprechendes *XML*-Element (Beispiel 4.38)):**

```
<extra>2003</extra>
```

### 4.2.19 *Symbol Spaces*

In *XML Schema* wird der Namensraum nicht nur mit Hilfe der *QNames* aufgeteilt. Damit zum Beispiel ein `element` mit dem Namen `video` nicht mit dem `complexType video` in Konflikt gerät, werden fünf verschiedene Namensbereiche (*Symbol Spaces*) [12] bereitgestellt:

- *Type Symbol Space*
- *Element Symbol Space*
- *Attribute Symbol Space*
- *Group Symbol Space*
- *AttributeGroup Symbol Space*

Der *Type Symbol Space* fasst sämtliche `simpleTypes`, `complexTypes` und `type`-Attribute zusammen. Die restlichen vier *Symbol Spaces* sind so eingeteilt, dass für jede *Schema*-Komponente, die ein `ref`-Attribut benutzen kann, der gleiche *QName* zur Referenz verwendet werden kann und dennoch keinen Namenskonflikt verursacht wird.

Beispiel 4.40 verwendet `video` in vier verschiedenen *Symbol Spaces*. Dabei entsteht kein Namenskonflikt – nur die Verständlichkeit leidet.

**Beispiel 4.40 (*Symbol Spaces):***
```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="video" type="video">

  <xsd:complexType name="video">
    <xsd:sequence>
      <xsd:group ref="video"/>
    </xsd:sequence>
    <xsd:attributeGroup ref="video"/>
  </xsd:complexType>

  <xsd:attributeGroup name="video">
    <xsd:attribute name="nr" type="xsd:integer"/>
  </xsd:attributeGroup>

  <xsd:group name="video">
    <xsd:sequence>
      <xsd:element name="title" type="xsd:string"/>
      <xsd:element name="language" type="xsd:string"/>
      <xsd:element name="extra" type="extra_types"
         minOccurs="0" maxOccurs="3"/>
    </xsd:sequence>
  </xsd:group>
</xsd:schema>
```

**Beispiel 4.41 (entsprechendes *XML*-Element (Beispiel 4.40)):**
```
<video nr="1">
  <title>Finding Nemo</title>
  <language>English</language>
  <extra>1h40m</extra>
  <extra>Adventure Animation Comedy Family</extra>
</video>
```

### 4.2.20 Ignorierte *XML Schema*-Komponenten

**Ignorierte Elemente**

- `annotation`, `notation` und `documentation`
  In *XML Schema* stellen `annotation`, `notation` und `documentation` die Komponenten für Kommentare dar und werden daher nicht als Typen interpretiert. (siehe `http://www.w3.org/TR/xmlschema-0/#ref16`)

- `facet`
  `facet` ist ein Konstrukt, das stellvertretend für eine Liste von Einschränkungen steht, die in `restriction` benutzt werden um `simpleTypes` einzuschränken. Dabei können auf einen `simpleType` mehrere `facets` angewendet werden.

  **Typsyntax** (`facets`):
  ```
  facet ::= length, minLength, maxLength, pattern,
            enumeration, whiteSpace, maxInclusive,
            maxExclusive, minExclusive, minInclusive,
            totalDigits, fractionDigits
  ```

  (siehe `http://www.w3.org/TR/xmlschema-0/#SimpleTypeFacets`)

- `include`
  Das Element `include` erlaubt das Hinzufügen von *Schema*-Definitionen zum aktuellen *Schema*. Das Attribut `schemaLocation` gibt die Position des *Schema* an. Allerdings müssen die eingefügten *Schema*-Definitionen aus dem gleichen `targetNamespace` stammen.
  (siehe `http://www.w3.org/TR/xmlschema-0/#SchemaInMultDocs`)

- `redefine`
  `redefine` importiert, genauso wie `include` sämtliche *Schema*-Definitionen. Dabei können allerdings die hinzugefügten Typen geändert (`redefine`) werden, ohne dass ihre Namen dabei geändert werden müssen.
  (siehe `http://www.w3.org/TR/xmlschema-0/#Redefine`)

- `import`
  `import` erlaubt das Importieren eines *XML Schema* mit anderen *Namespaces*. Innerhalb eines *Schema* können die *globalen* Definitionen der importierten *Schemas* verwendet werden.
  (siehe `http://www.w3.org/TR/xmlschema-0/#import`)

- `unique`
  `unique` ermöglicht es, die Einzigartigkeit (*uniqueness*) eines `attribute` oder des Wertes eines `element` sicherzustellen.
  (siehe `http://www.w3.org/TR/xmlschema-0/#specifyingUniqueness`)

- `keyref, key`
  Innerhalb eines `key`-Elements kann wie in `unique` ein Bereich festgelegt werden, der mit `keyref` später referenziert werden kann.
  (siehe `http://www.w3.org/TR/xmlschema-0/#specifyingKeys&theirRefs`)

- `selector, field`
  `selector` und `field` können innerhalb eines `unique`-, `key`- oder `keyref`-Elements auftreten. `selector` schränkt dabei den Bereich mit Hilfe des `xpath`-Attributs (einer *XPath*-Anfrage) ein. `field` identifiziert die `elements` oder `attributes` relativ zu dem Ergebnis von `selector`, auf das `unique`, `key` oder `keyref` angewendet wird.
  (siehe `http://www.w3.org/TR/xmlschema-0/#specifyingUniqueness`)

**Ignorierte Attribute**

- `final`
  `final` unterbindet in einem `complexType` eine weitere Ableitung durch `extension` oder `restriction`. Dazu stehen dem Attribut `final` die Werte `"all"`, `"restriction"` und `"extension"` zur Verfügung.

- `fixed`
  `fixed` kann sowohl in `elements` als auch in `attributes` verwendet werden. Der Wert des Attributs `fixed` legt dabei den Wert des `element` bzw. des `attribute` fest.

- `mixed`
  Das Attribut `mixed` kann in einem `complexType` auftauchen und erlaubt mit dem Wert `mixed="true"` beliebige Zeichenketten zwischen den Elementen.

- `substitutionGroup`
  Mit Hilfe des Attributs `substitutionGroup` können `global elements` durch andere `global elements` ersetzt werden. Diesen `elements` ist es möglich an jeder Stelle aufzutreten, an der das referenzierte `global element` erlaubt ist.

Dabei muss der Typ dem des `global element` oder eines daraus abgeleiteten Typs entsprechen. Mit der Änderung des *Schema* aus Beispiel 4.42 kann somit an jeder Stelle eines *XML*-Dokuments statt `video` auch das Element `movie` stehen.

**Beispiel 4.42** (`element` **mit** `substitutionGroup`**-Attribut):**

```
<xsd:element name="movie" type="video"
             substitutionGroup="video"/>
```

## 4.3 *XQuery* Typen

Im vorherigen Abschnitt wurden die Typen von *XML Schema* vorgestellt. Da diese sich von den in *XQuery* bekannten Typen unterscheiden wird in diesem Abschnitt das Typsystem von *XQuery* [3] genauer erklärt.

Das Typsystem von *XQuery* kann in drei Gruppen unterteilt werden (siehe Abbildung 2). Die erste Gruppe behandelt die sogenannten `item types`, welche die eingebauten Typen (`atomic types`) und die Knoten-Typen (`node types`) zusammenfassen. Die zweite Gruppe behandelt die `regular expression types`, welche die Häufigkeit und die Kombinationen von Typen beeinflussen. Die dritte Gruppe besteht aus einem Typ, dem `named type`, der eine Referenz auf Typen in der Typumgebung zulässt.

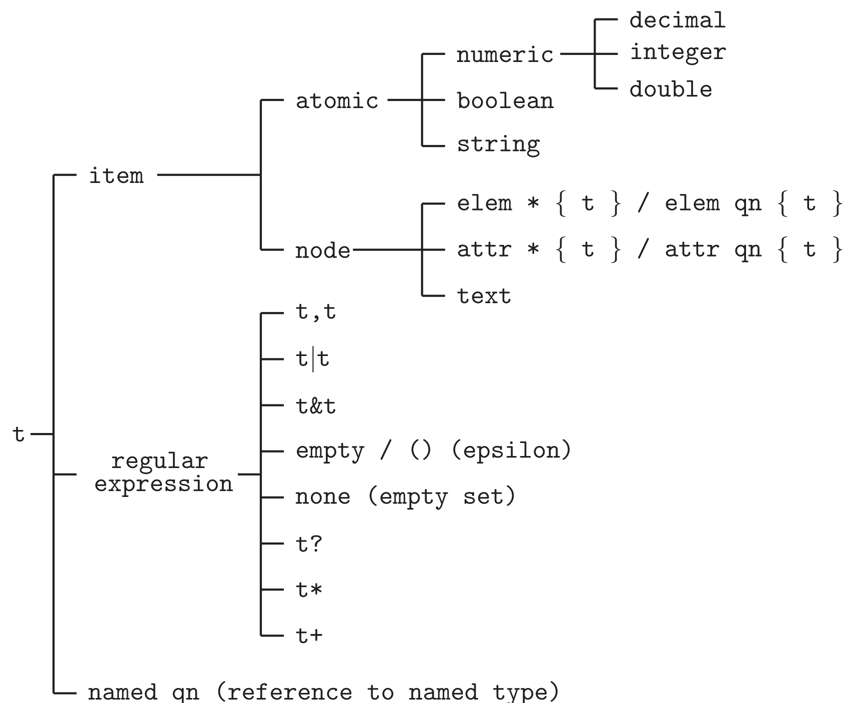


Abbildung 2: Hierarchie *XQuery*-Typen

### 4.3.1 `item types`

Die Gruppe der `item types` setzt sich aus den eingebauten Typen (`atomic types`) und den `node types` zusammen. `item types` repräsentieren jeweils einen einzelnen Wert.

`atomic types`

Die `atomic types` fassen die in *XQuery* eingebauten Typen zusammen. In Abbildung 2 sind beispielhaft die Typen `string`, `boolean` und `numeric` aufgeführt. Diese Typen können selbst nochmals in spezifischere Typen unterteilt sein, wie z.B. der Typ `numeric`, welcher in die Typen `decimal`, `integer`, `double`, ... unterteilt werden kann. Die selben `atomic types` bzw. `built-in types` werden sowohl in *XQuery* als auch in *XML Schema* verwendet. Die vollständige Liste der `atomic types` ist in der Definition von *XML Schema (Part 2, Datatypes)*[4] zusammengefasst.

`node types`

Die `node types` fassen die aus *XML* bekannten Komponenten (Elemente, Attribute und Zeichenketten) zusammen. Zeichenketten entsprechen dem Typ `text` in *XQuery*. Elemente bestehen, ähnlich wie in *XML Schema*, aus einem Namen und einem Typ. Ein Element kann dabei entweder einen *QName* oder einen beliebigen Namen ( `*` ) (*wildcard*) besitzen. Der Typ ist durch geschweifte Klammern begrenzt und kann wie in Beispiel 4.43 jeder konstruierbare Typ sein.

**Beispiel 4.43 (*XQuery*-Typ `elem`):**
```
elem "video" { named "video" }
elem "title" { xsd:string }
```

Die Attribute sind ebenso wie die Elemente in zwei Ausführungen möglich: anonym ( `*` ) oder mit *QName*. Ein Attribut wird mit `attr` gekennzeichnet und sein Typ in geschweifte Klammern gesetzt.

**Beispiel 4.44 (*XQuery*-Typ `attr`):**
```
attr "nr" { xsd:integer }
attr * { xsd:string }
```

### 4.3.2 `regular expression types`

*XQuery* bietet die Möglichkeit Typen zu kombinieren um dadurch z.B. Wiederholungen auszudrücken. Die Kombination von Typen führen die Komponenten `all` (`"&"`), `choice` (`"|"`) und `sequence` (`","`) durch. Sie sind vergleichbar mit den in *XML Schema* verwendeten `all`, `choice` und `sequence`. Die Häufigkeit eines Typs bestimmen die Typkonstruktoren `"?"`, `"*"` und `"+"`.

`sequence`

Mit Hilfe des Typkonstruktors `","` kann eine Sequenz von Typen erzeugt werden. In Beispiel 4.45 kann z.B. mit einer `sequence` das Element `"video"` ein Attribut und zwei Elemente beinhalten.

**Beispiel 4.45 (*XQuery*-Typ `sequence`):**

```
elem "video" { attr "nr" { xsd:integer },
               elem "title" { xsd:string },
               elem "language" { xsd:string } }
```

`choice`

Der *choice*-Typkonstruktor in *XQuery* ermöglicht genauso wie `choice` in *XML Schema* eine Vereinigung mehrerer Typen. So kann in Beispiel 4.46 der Typ des Attributs `"extra"` aus einem der drei mit dem *choice*-Typkonstruktor `"|"` verbundenen Typen ausgewählt werden.

[4] *XML Schema – Part 2, Datatypes*: `http://www.w3.org/TR/xmlschema-2/`

**Beispiel 4.46 (*XQuery*-Typ `choice`):**
```
attr "extra" { xsd:integer | xsd:duration | named "genres" }
```

`all`

Der Typkonstruktor `all` steht stellvertretend für eine Permutation der mit `"&"` verbundenen Typen. Dies ist z.B. bei Attributen sinnvoll, da deren Reihenfolge von *XML* nicht festgelegt ist und mit Hilfe von `all` auch variabel bleibt. Eine `all`-Gruppe wird in *XQuery* intern in eine Kombination von `choice` und `sequence` zerlegt (siehe Beispiel 4.47).

**Beispiel 4.47 (*XQuery*-Typ `all`):**
```
  (attr "title" { xsd:string } & attr "language" { xsd:string })
                                  =
  (attr "title" { xsd:string } , attr "language" { xsd:string })
| (attr "language" { xsd:string } , attr "title" { xsd:string })
```

`empty`

Der Typ `empty` beschreibt eine leere Sequenz (`"()"`) und ist das neutrale Element der Sequenzbildung mit `","`.

**Beispiel 4.48 (empty type):**
```
elem "title" { xsd:string }, () = elem "title" { xsd:string }
elem "title" { xsd:string }, empty = elem "title" { xsd:string }
```

`none`

Der Typ `none` bezeichnet die leere Menge von Typen bzw. Werten. Er ist das neutrale Element der `choice` mit `"|"`.

**Beispiel 4.49 (empty type):**
```
  attr "extra" { xsd:integer | xsd:duration | none }
= attr "extra" { xsd:integer | xsd:duration }
```

`occurrences`

*XQuery* stellt drei Typkonstruktoren zur Verfügung mit denen die Häufigkeit eines Typs (`occurrence`) beeinflusst werden kann.

**Typsyntax** (`occurrence`):
```
  occurrence ::=  "?" | "*" | "+"
```

Der erste Typkonstruktor `"?"` erlaubt null oder ein Vorkommen eines Typs. `"*"`, der zweite Konstruktor, steht für null bis unendlich viele Auftreten und der dritte Typkonstruktor `"+"` für ein bis unendlich viele.

Beispiel 4.50 zeigt die Typkonstruktoren `"?"` und `"*"` in Aktion und Beispiel 4.51 ist eine entsprechende, erlaubte *XML*-Instanz.

**Beispiel 4.50 (*XQuery*-`occurrences`):**
```
elem "videos"
{ (elem "video"
  { (attr "nr" { xsd:integer })?, named "video" }
  )*
}
```

**Beispiel 4.51 (*XML*-Instanz zu Beispiel 4.50):**

```
<videos>
  <video nr="1">
    <title>Finding Nemo</title>
    <language>English</language>
    ...
  </video>
  <video>
    ...
  </video>
  <video nr="3">
    ...
  </video>
</videos>
```

### 4.3.3 `named type`

In der Typumgebung von *XQuery* können nicht nur die eingebauten Typen, sondern auch beliebige andere konstruierte Typen eingetragen sein. Natürlich sind in die Typumgebung eingetragene Typen nur sinnvoll, wenn wieder auf sie zugegriffen werden kann. Dies ist möglich, indem an jeden Typ ein Name gebunden wird. Der `named type` stellt die Schnittstelle zwischen einem Typ und anderen in der Typumgebung eingetragenen Typen mit Hilfe einer Referenz auf den Namen dar. Somit entspricht der `named type` in *XQuery* dem Attribut `ref` in *XML Schema*.

### 4.3.4 Weitere eingebaute Typen

In der Typumgebung sind zusätzlich zu den `atomic types` weitere Typen definiert. Diese setzen sich aus den vorgestellten Typen zusammen und bilden durch Kombination neue bzw. anders benannte Typen. Dabei entspricht z.B. `anyElement` dem Typ `any` in *XML Schema*.

**Typsyntax:**

```
anyType ::= item *
anySimpleType ::= atomic
anyElement ::= elem * { anyType }
anyAttribute = attr * { anySimpleType }
```

# 5 *Mapping* von *XML Schema* nach *XQuery*

Einer der zentralen Punkte dieser Arbeit ist das *Mapping*, welches nun erklärt wird. Da sich die Typen von *XML Schema* und *XQuery* unterscheiden, ist es wichtig, Abbildungsregeln zu finden, die die Typen von *XML Schema* möglichst genau nach *XQuery* abbilden. Ohne korrekte Abbildungsregeln wäre auch der *Schema* Import sinnlos.

Das Abbilden wird formal durch einen Pfeil nach unten dargestellt ( ⇓ ). In jeder Regel wird eine Komponente von *XML Schema*, die durch doppelte Klammern ( ⟦ `type` ⟧ ) begrenzt ist, auf den entsprechenden *XQuery*-Typ abgebildet.

**Beispiel 5.1 (Abbildungsregel für ein `local element` mit Teil-Typ):**

```
⟦ <element name="n"> type </element> ⟧
                  ⇓
         elem "n" { ⟦ type ⟧ }
```

Das `element` in Beispiel 5.1 kann eine solche *Schema*-Komponente sein, die in ein entsprechendes *XQuery* Element (`elem`) übersetzt wird. Allerdings können innerhalb eines `element` weitere Typen geschachtelt sein (z.B. ein `complexType`), die durch andere Regeln rekursiv übersetzt werden müssen. Diese Typen werden im weiteren als '`type`' bzw. Teil-Typ bezeichnet. Die *Schema*-Typen werden in ihrer *XML*-Syntax dargestellt, wobei deren Inhalt meist durch `type` abgekürzt wird und die Werte von Attributen (im Beispiel 5.1 `"n"`) als Variablen verstanden werden müssen.

Eine weitere Komponente der Abbildung ist der Pfeil nach rechts ( $\Mapsto$ ), der den Eintrag eines *globalen* Typs in die Typumgebung (*type environment*) von *XQuery* markiert. Auf der linken Seite steht der Name des eingetragenen Typs, direkt rechts unterhalb des Pfeils der *Symbol Space*, dem der Name zugeordnet wird und auf der rechten Seite der übersetzte *XQuery*-Typ.

In Beispiel 5.2 wird der Wert des `name`-Attributs bezogen auf den Teil-Typ in den *Type Symbol Space* der Typumgebung eingetragen.

**Beispiel 5.2 (Abbildungsregel für einen `global simpleType`):**

$$\llbracket \texttt{<simpleType name="n"> type </simpleType>} \rrbracket$$
$$\Downarrow$$
$$\texttt{"n"} \Mapsto_{type} \llbracket \texttt{ type } \rrbracket$$

In den folgenden Unter-Abschnitten wird nun zu jeder in Abschnitt 4.2 aufgeführten *XML Schema*-Komponente die einzelnen Abbildungsregeln vorgestellt:

## 5.1 `schema`

Das Element `schema` selbst trägt keine Information, die in *XQuery* übertragen werden kann.

## 5.2 `element`

Jedes `element` in *XML Schema* kann in *XQuery* durch ein entsprechendes Element (`elem`) dargestellt werden. Die erste Abbildungsregel eines `element` gibt keinen Typ für das `element` vor, da weder ein `type`-Attribut noch ein Teil-Typ vorhanden ist. Dementsprechend werden bei der Abbildung alle Typen zugelassen, was in *XQuery* dem Typ `anyType` entspricht.

***Mapping* 5.1 (`global element` ohne Teil-Typ und ohne `type`-Attribut):**

$$\llbracket \texttt{<element name="n"/>} \rrbracket$$
$$\Downarrow$$
$$\texttt{"n"} \Mapsto_{element} \texttt{elem "n"} \; \{ \texttt{ anyType } \}$$

Da Regel 5.1 ein `global element` behandelt, muss der Wert des `name`-Attributs sowie der Typ `anyType` mit dem *Element Symbol Space* in die Typumgebung von *XQuery* eingetragen werden. Der Wert des `name`-Attributs bildet dabei den Suchschlüssel für das `elem`, damit dieses später referenziert werden kann.

**Beispiel 5.3 (Abbildung eines `global element` ohne Typ):**

```
⟦ <?xml version="1.0"?>
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="title">
  </xsd:schema>                                            ⟧
```

$$\Downarrow$$
$$\texttt{"title"} \Mapsto_{element} \texttt{elem "title"} \; \{ \texttt{ anyType } \}$$

***Mapping* 5.2 (`global element` mit `type`-Attribut):**

$$\llbracket\ \texttt{<element name="n" type="t"/>}\ \rrbracket$$
$$\Downarrow$$
$$\texttt{"n"} \Mapsto_{element} \texttt{elem "n"}\ \{\ \texttt{named}_{type}\ \texttt{"t"}\ \}$$

Die zweite Regel (Mapping 5.2) bildet ein `global element` mit dem Attribut `type` ab. Die Abbildung ist der ersten Abbildung sehr ähnlich. Die einzige Veränderung ist, dass anstatt des Typs `anyType` der Wert des `type`-Attributs mit Hilfe von `named` als Typ referenziert wird. In Beispiel 5.4 ist dies der Typ `string`, der als `named type` referenziert wird.

**Beispiel 5.4 (Abbildung eines `global element` mit `type`-Attribut):**

```
⟦ <?xml version="1.0"?>
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="title" type="xsd:string">
  </xsd:schema> ⟧
```

$$\Downarrow$$
$$\texttt{"title"} \Mapsto_{element} \texttt{elem "title"}\ \{\ \texttt{named}_{type}\ \texttt{"xsd:string"}\ \}$$

Abbildungsregel 5.3 behandelt die dritte Möglichkeit ein `global element` abzubilden. Statt des Attributs `type` ist ein Teil-Typ vorhanden (z.B. ein `complexType` in Beispiel 5.5, der mit anderen Regel abgebildet wird).

***Mapping* 5.3 (`global element` mit Teil-Typ):**

$$\llbracket\ \texttt{<element name="n"> type </element>}\ \rrbracket$$
$$\Downarrow$$
$$\texttt{"n"} \Mapsto_{element} \texttt{elem "n"}\ \{\ \llbracket\ \texttt{type}\ \rrbracket\ \}$$

Das `elem` (Mapping 5.3) wird allerdings erst in die Typumgebung eingetragen, wenn alle beinhalteten *XML Schema*-Typen mit Hilfe der Abbildungsregeln rekursiv aufgelöst wurden.

**Beispiel 5.5 (Abbildung eines `global` und eines `local element`):**

```
⟦ <?xml version="1.0"?>
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="video">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="title" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema> ⟧
```

$$\Downarrow$$
$$\texttt{"video"} \Mapsto_{element}$$
$$\texttt{elem "video"}\ \{\ \texttt{elem "title"}\ \{\ \texttt{named}_{type}\ \texttt{"xsd:string"}\ \}\ \}$$

Mit den folgenden drei Abbildungsregeln 5.4, 5.5 und 5.6 werden `local elements` behandelt. Im Gegensatz zu den `global elements` werden sie nicht in die Typumgebung eingetragen, sondern lediglich rekursiv übersetzt.

***Mapping* 5.4 (local element ohne Teil-Typ und ohne type-Attribut):**

⟦ <element name="n"/> ⟧
⇓
elem "n" { anyType }

***Mapping* 5.5 (local element mit type-Attribut):**

⟦ <element name="n" type="t"/> ⟧
⇓
elem "n" { $\texttt{named}_{type}$ "t" }

Der Teil-Typ (<element name="title" .../>) wurde in Beispiel 5.5 mit Hilfe der Regel 5.5 aufgelöst.

***Mapping* 5.6 (local element mit Teil-Typ):**

⟦ <element name="n"> type </element> ⟧
⇓
elem "n" { ⟦ type ⟧ }

Regel 5.7 nimmt eine Sonderstellung bei den Abbildungen von elements ein. Da das Attribut ref eine Referenz zu einem anderem global element herstellt, wird das element in einen named type abgebildet, der auf das von ref verwiesene element im *Element Symbol Space* verweist.

***Mapping* 5.7 (local element mit ref-Attribut):**

⟦ <element ref="r"/> ⟧
⇓
$\texttt{named}_{element}$ "r"

## 5.3 group

Die global group stellt in *XML Schema* ein benannte, referenzierbare Gruppe für elements zur Verfügung und hat keinen direkt vergleichbaren *XQuery*-Typ. Deswegen wird empty für die leere group bzw. der Teil-Typ der group als Typ und der Name der group mit dem eigenen *Group Symbol Space* in die Typumgebung eingetragen (Regel 5.8, Regel 5.9).

***Mapping* 5.8 (global group ohne Teil-Typ):**

⟦ <group name="n"/> ⟧
⇓
"n" $\mapsto_{group}$ empty

***Mapping* 5.9 (global group):**

⟦ <group name="n"> type </group> ⟧
⇓
"n" $\mapsto_{group}$ ⟦ type ⟧

Die Referenzierung des Inhalts einer global group durch eine local group funktioniert wie bei den elements. Der *Group Symbol Space* ist dabei die einzige Veränderung (vergleiche Mapping 5.7).

***Mapping* 5.10** (`local group`):

$$[\![ \texttt{<group ref="r"/>} ]\!] \\ \Downarrow \\ \texttt{named}_{group}\ \texttt{"r"}$$

## 5.4 `all`

Eine `all`-Gruppe muss, genau wie eine `group`, nicht unbedingt einen Teil-Typ enthalten. `all` ohne Teil-Typ wird auf den *XQuery*-Typ `none` abgebildet, da eine `all`-Gruppe intern durch eine `choice`-Gruppe repräsentiert werden kann (siehe Abschnitt 4.3.2) und `none` das neutrale Element des *XQuery*-Typkonstruktors `choice` (`"|"`) ist.

***Mapping* 5.11 (`all` ohne Teil-Typ):**

$$[\![ \texttt{<all/>} ]\!] \\ \Downarrow \\ \texttt{none}$$

Handelt es sich um einen einzigen Teil-Typ erzeugt `all` keinen zusätzlichen *XQuery*-Typ sondern es wird lediglich der Teil-Typ übersetzt.

***Mapping* 5.12 (`all` mit Teil-Typ):**

$$[\![ \texttt{<all> type </all>} ]\!] \\ \Downarrow \\ [\![ \texttt{type} ]\!]$$

Im Gegensatz zu `group`, kann `all` mehr als einen Teil-Typ enthalten. Die einzelnen Teil-Typen werden verbunden, indem jeweils zwei Teil-Typen mit Hilfe des `all`-Typkonstruktors (`"&"`) zu einem Teil-Typ kombiniert werden.

**Beispiel 5.6 (Abbildung einer `all`-Gruppe mit mehreren Teil-Typen):**

```
⟦<?xml version="1.0"?>
 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:complexType name="video">
     <xsd:all>
       <xsd:element name="title" type="xsd:string"/>
       <xsd:element name="language" type="xsd:string"/>
       <xsd:element name="duration" type="xsd:duration"/>
       <xsd:element name="year" type="xsd:integer"/>
     </xsd:all>
   </xsd:complexType>
 </xsd:schema>⟧
                              ⇓
   "video" ⤇type
      (((elem "title" { named_type "xsd:string" }
         & elem "language" { named_type "xsd:string" })
        & elem "duration" { named_type "xsd:duration" })
       & elem "year" { named_type "xsd:integer" })
```

***Mapping* 5.13 (all mit mehreren Teil-Typen):**

$$[\![\ \texttt{<all> type}_1\ \texttt{type}_2\ \texttt{...}\ \texttt{type}_n\ \texttt{</all>}\ ]\!]$$
$$\Downarrow$$
$$\texttt{(...(}[\![\ \texttt{type}_1\ ]\!]\ \texttt{\&}\ [\![\ \texttt{type}_2\ ]\!]\texttt{)}\ \texttt{\&}\ \texttt{...}\ \texttt{\&}\ [\![\ \texttt{type}_n\ ]\!]\texttt{)}$$

## 5.5 choice

Eine choice kann ebenfalls leer sein. Die Abbildung dieses Falles sieht wie bei einer all-Gruppe den Type none vor, da none das neutrale Element des *XQuery*-Typkonstruktors choice ("|") ist.

***Mapping* 5.14 (choice ohne Teil-Typ):**

$$[\![\ \texttt{<choice/>}\ ]\!]$$
$$\Downarrow$$
$$\texttt{none}$$

choice mit einem Teil-Typ erzeugt analog zu all keinen weiteren *XQuery*-Typ, sondern der Teil-Typ wird rekursiv übersetzt.

***Mapping* 5.15 (choice mit Teil-Typ):**

$$[\![\ \texttt{<choice> type </choice>}\ ]\!]$$
$$\Downarrow$$
$$[\![\ \texttt{type}\ ]\!]$$

choice geht mit mehreren Teil-Typen ähnlich wie all vor. Der einzige Unterschied zu einer all-Gruppe ist der verwendete Typkonstruktor choice ("|").

***Mapping* 5.16 (choice mit mehreren Teil-Typen):**

$$[\![\ \texttt{<choice> type}_1\ \texttt{type}_2\ \texttt{...}\ \texttt{type}_n\ \texttt{</choice>}\}\ ]\!]$$
$$\Downarrow$$
$$\texttt{(...(}[\![\ \texttt{type}_1\ ]\!]\ |\ [\![\ \texttt{type}_2\ ]\!]\texttt{)}\ |\ \texttt{...}\ |\ [\![\ \texttt{type}_n\ ]\!]\texttt{)}$$

## 5.6 sequence

Die leere sequence-Gruppe wird ähnlich wie eine choice-Gruppe auf das neutrale Element abgebildet. Im Falle einer sequence handelt es sich um den *XQuery*-Typ empty.

***Mapping* 5.17 (sequence ohne Teil-Typ):**

$$[\![\ \texttt{<sequence/>}\ ]\!]$$
$$\Downarrow$$
$$\texttt{empty}$$

***Mapping* 5.18 (sequence mit Teil-Typ):**

$$[\![\ \texttt{<sequence> type </sequence>}\ ]\!]$$
$$\Downarrow$$
$$[\![\ \texttt{type}\ ]\!]$$

Die sequence erstellt ähnlich wie all und choice bei mehreren Teil-Typen eine Liste. Allerdings handelt es sich bei dem verwendeten Konstruktor einer sequence um den *XQuery*-Typkonstruktor (",").

***Mapping* 5.19 (`sequence` mit mehreren Teil-Typen):**

$$[\![\ \texttt{<sequence> type}_1\ \texttt{type}_2\ \texttt{...}\ \texttt{type}_n\ \texttt{</sequence>}\ ]\!] \\ \Downarrow \\ \texttt{(...(}[\![\ \texttt{type}_1\ ]\!]\ \texttt{,}\ [\![\ \texttt{type}_2\ ]\!]\texttt{)}\ \texttt{,}\ \texttt{...,}\ [\![\ \texttt{type}_n\ ]\!]\texttt{)}$$

## 5.7 `any`

Für `any` gibt es den entsprechenden eingebauten *XQuery*-Typ `anyElement`.

***Mapping* 5.20 (`any`):**

$$[\![\ \texttt{<any/>}\ ]\!] \\ \Downarrow \\ \texttt{anyElement}$$

## 5.8 `element occurrences`

Die Attribute `minOccurs` und `maxOccurs` können sowohl in `elements`, als auch in Gruppen eingesetzt werden. Stellvertretend steht in den folgenden Abbildungsregeln ein `"x"` für jedes mögliche *Schema*-Element. *XQuery* kann im Gegensatz zu *XML Schema* nicht jede mögliche Häufigkeit darstellen, sondern die Vorkommen nur durch `"?"`, `"*"` und `"+"` beeinflussen.

Wenn weder `minOccurs` noch `maxOccurs` vorkommen, dann tritt Regel 5.22 in Kraft. Ist nur eines der Attribute vorhanden dann treten entsprechend entweder Regel 5.21 oder Regel 5.24 in Kraft.

Die Abbildung der Regel 5.21 von *XML Schema* nach *XQuery* findet ohne Informationsverlust statt, da `"?"` genau eine Häufigkeit von `"0"` oder `"1"` definiert.

***Mapping* 5.21 (`minOccurs`- und `maxOccurs`-Attribut):**

$$[\![\ \texttt{<x minOccurs="0" maxOccurs="1"/>}\ ]\!] \\ \Downarrow \\ [\![\ \texttt{x}\ ]\!]\ \texttt{?}$$

Regel 5.22 stellt den Standardfall dar, bei dem die Häufigkeiten nicht verändert wurden.

***Mapping* 5.22 (`minOccurs`- und `maxOccurs`-Attribut):**

$$[\![\ \texttt{<x minOccurs="1" maxOccurs="1"/>}\ ]\!] \\ \Downarrow \\ [\![\ \texttt{x}\ ]\!]$$

In Regel 5.23 ist `n > 1`. Dabei wird `"2"` genauso wie `"1000000"` oder `"unbounded"` auf beliebig viele Vorkommen abgeschätzt.

***Mapping* 5.23 (`minOccurs`- und `maxOccurs`-Attribut):**

$$[\![\ \texttt{<x minOccurs="0" maxOccurs="n"/>}\ ]\!] \\ \Downarrow \\ [\![\ \texttt{x}\ ]\!]\ \texttt{*}$$

Regel 5.24 wird ähnlich wie Regel 5.23 nach oben hin abgeschätzt. Da allerdings `minOccurs="1"` ist, wird der Typkonstruktor `"+"` verwendet. Genauso wie jede positive Zahl (`n > 1`), wird auch der Sonderfall `maxOccurs="unbounded"` mit der Regel 5.24 abgebildet (siehe Beispiel 5.7).

***Mapping* 5.24 (`minOccurs`- und `maxOccurs`-Attribut):**

$$
\begin{gathered}
[\![\ \texttt{<x minOccurs="1" maxOccurs="n"/>}\ ]\!] \\
\Downarrow \\
[\![\ \texttt{x}\ ]\!]\ \texttt{+}
\end{gathered}
$$

**Beispiel 5.7 (Abbildung des `minOccurs`- und `maxOccurs`-Attributs):**

$$
\begin{gathered}
\left[\!\!\left[ \begin{array}{l}\texttt{<xsd:element name="extra" type="xsd:string"} \\ \quad\texttt{maxOccurs="unbounded"/>}\end{array} \right]\!\!\right] \\
\Downarrow \\
\texttt{( elem "extra" } \{\ \texttt{xsd:string}\ \}\texttt{ ) +}
\end{gathered}
$$

Die letzte der `occurrence`-Abbildungsregeln (Regel 5.25) geht davon aus, dass `1 <= n <= m` ist. Demnach kommt `"x"` mindestens einmal vor und wird mit der Häufigkeit `"+"` abgeschätzt.

***Mapping* 5.25 (minOccurs- und maxOccurs-Attribut):**

$$
\begin{gathered}
[\![\ \texttt{<x minOccurs="n" maxOccurs="m"/>}\ ]\!] \\
\Downarrow \\
[\![\ \texttt{x}\ ]\!]\ \texttt{+}
\end{gathered}
$$

## 5.9 `attribute`

*XML Schema* und *XQuery* kennen beide das Konzept `attribute`. Somit wird ein `attribute` direkt auf ein `attr` abgebildet.

Die Abbildung der `attributes` verhält sich ähnlich, wie die der `elements`. Es gibt die Unterscheidung zwischen *globalen* und *lokalen* Typen, sowie die Unterscheidung zwischen keinem Typ, `type`-Attribut oder einem Teil-Typ. Die erste Abbildungsregel (Regel 5.26) befasst sich mit einem `attribute`, dessen Typ weder durch das `type`-Attribut noch durch einen Teil-Typ bestimmt ist. In die Typumgebung von *XQuery* wird deshalb `anySimpleType` eingetragen, was jedem möglichen Typ eines `attribute` entspricht.

***Mapping* 5.26 (`global attribute` ohne Teil-Typ und ohne `type`-Attribut):**

$$
\begin{gathered}
[\![\ \texttt{<attribute name="n"/>}\ ]\!] \\
\Downarrow \\
\texttt{"n"}\ \Mapsto_{attribute}\ \texttt{attr "n"}\ \{\ \texttt{anySimpleType}\ \}
\end{gathered}
$$

Die Abbildung eines `global attribute` mit `type`-Attribut verhält sich wie erwartet. Das `attr` hat den Namen des `name`-Attributs und den Typ `named type`, der auf den Typ verweist, den das `type`-Attribut referenziert. Das `attr` wird in den *Attribute Symbol Space* eingetragen.

***Mapping* 5.27 (`global attribute` mit `type`-Attribut):**

$$
\begin{gathered}
[\![\ \texttt{<attribute name="n" type="t"/>}\ ]\!] \\
\Downarrow \\
\texttt{"n"}\ \Mapsto_{attribute}\ \texttt{attr "n"}\ \{\ \texttt{named}_{type}\ \texttt{"t"}\ \}
\end{gathered}
$$

Im Gegensatz zur vorherigen Abbildung (Regel 5.27) wird in der nächsten (Regel 5.28) der komplette Teil-Typ des `attr` in die Typumgebung eingetragen.

***Mapping* 5.28 (`global attribute` mit Teil-Typ):**

$$[\![\ \texttt{<attribute name="n"> type </attribute>}\ ]\!]$$
$$\Downarrow$$
$$\texttt{"n"} \mapsto_{attribute} \texttt{attr "n"} \{ [\![\ \texttt{type}\ ]\!] \}$$

Das Attribut `use` kann nur bei `local attributes` auftreten. Da es im Standardfall `"optional"` ist, wird jedes `local attribute`, das kein `use`-Attribut enthält, durch den Typkonstruktor `"?"` optional gemacht. Ist das `use`-Attribut vorhanden, dann wird im Fall `use="required"` der Typkonstruktor `"?"` nicht benutzt. Falls `use="prohibited"`, dann wird das `attribute` nicht nach *XQuery* übersetzt (`empty`), da ein Verbot eines Typs in *XQuery* nicht ausgedrückt werden kann und die Abbildung des `attribute` ohne Verbot der Logik von `"prohibited"` widersprechen würde.

***Mapping* 5.29 (`local attribute` mit `use`-Attribut):**

$$[\![\ \texttt{<attribute name="n" use="required"> type </attribute>}\ ]\!]$$
$$\Downarrow$$
$$\texttt{attr "n"} \{ [\![\ \texttt{type}\ ]\!] \}$$

$$[\![\ \texttt{<attribute name="n" use="optional"> type </attribute>}\ ]\!]$$
$$\Downarrow$$
$$\texttt{( attr "n"} \{ [\![\ \texttt{type}\ ]\!] \} \texttt{ ) ?}$$

$$[\![\ \texttt{<attribute name="n" use="prohibited"> type </attribute>}\ ]\!]$$
$$\Downarrow$$
$$\texttt{empty}$$

Die folgenden Abbildungsregeln gehen davon aus, dass das Attribut `use` nicht vorhanden ist, ansonsten würde Regel 5.29 eine andere Verarbeitung vorschlagen.

Analog zu den Abbildungsregeln der `global attributes` werden die Fälle der `local attributes` abgebildet.

***Mapping* 5.30 (`local attribute` ohne Teil-Typ und ohne `type`-Attribut):**

$$[\![\ \texttt{<attribute name="n"/>}\ ]\!]$$
$$\Downarrow$$
$$\texttt{( attr "n"} \{ \texttt{anySimpleType} \} \texttt{ ) ?}$$

**Beispiel 5.8 (Abbildung eines `attribute` mit `type`-Attribut):**

$$[\![\ \texttt{<xsd:attribute name="nr" type="xsd:integer"/>}\ ]\!]$$
$$\Downarrow$$
$$\texttt{( attr "nr"} \{ \texttt{named}_{type}\ \texttt{"xsd:integer"} \} \texttt{ ) ?}$$

Beispiel 5.8 zeigt die Abbildung eines `attribute` mit `type`-Attribut nach der Abbildungsregel 5.31.

***Mapping* 5.31 (`local attribute` mit `type`-Attribut):**

$$[\![\ \texttt{<attribute name="n" type="t"/>}\ ]\!]$$
$$\Downarrow$$
$$\texttt{( attr "n"} \{ \texttt{named}_{type}\ \texttt{"t"} \} \texttt{ ) ?}$$

***Mapping* 5.32** (`local attribute` **mit Teil-Typ**):

$$\llbracket\ \texttt{<attribute name="n"> type </attribute>}\ \rrbracket$$
$$\Downarrow$$
$$\texttt{( attr "n" \{ } \llbracket\ \texttt{type}\ \rrbracket \texttt{ \} ) ?}$$

Das `attribute` mit `ref`-Attribut verhält sich genauso wie jede Referenz (z.B. `element` mit `ref`-Attribut – Regel 5.7). Es referenziert den Typ des `ref`-Attributs im *Attribute Symbol Space*.

***Mapping* 5.33** (`local attribute` **mit** `ref`**-Attribut**):

$$\llbracket\ \texttt{<attribute ref="r"/>}\ \rrbracket$$
$$\Downarrow$$
$$\texttt{( named}_{attribute}\ \texttt{"r" ) ?}$$

## 5.10 attributeGroup

Die `attributeGroup` stellt ähnlich wie die `group` bei den `elements` eine benannte Gruppe zur Verfügung, die in einen eigenen *Symbol Space* eingetragen wird, aber selbst nicht übersetzt wird. In die Typumgebung wird nur der Typ `empty` bzw. der Teil-Typ eingetragen. Bei dem Teil-Typ handelt es sich um `local attributes`, `anyAttribute` oder `local attributeGroups`. Diese werden hier stellvertretend durch '`attr`' repräsentiert.

***Mapping* 5.34** (`global attributeGroup` **ohne Teil-Typ**):

$$\llbracket\ \texttt{<attributeGroup name="n"/>}\ \rrbracket$$
$$\Downarrow$$
$$\texttt{"n"}\ \mapsto_{attributeGroup}\ \texttt{empty}$$

***Mapping* 5.35** (`global attributeGroup` **mit einem Teil-Typ**):

$$\llbracket\ \texttt{<attributeGroup name="n"> attr </attributeGroup>}\ \rrbracket$$
$$\Downarrow$$
$$\texttt{"n"}\ \mapsto_{attributeGroup}\ \llbracket\ \texttt{attr}\ \rrbracket$$

***Mapping* 5.36** (`global attributeGroup` **mit mehreren Teil-Typen**):

$$\llbracket\ \texttt{<attributeGroup name="n"> attr}_1\ \texttt{attr}_2\ \texttt{...}\ \texttt{attr}_n\ \texttt{</attributeGroup>}\ \rrbracket$$
$$\Downarrow$$
$$\texttt{"n"}\ \mapsto_{attributeGroup}$$
$$\texttt{(...(}\llbracket\ \texttt{attr}_1\ \rrbracket\ \texttt{\&}\ \llbracket\ \texttt{attr}_2\ \rrbracket\texttt{) \& ... \&}\ \llbracket\ \texttt{attr}_n\ \rrbracket\texttt{)}$$

Enthält eine `global attributeGroup` mehrere Teil-Typen, dann werden diese mit dem Typkonstruktor `"&"` verbunden. `"&"` wird gewählt, weil die Teil-Typen `attributes` sind, deren Reihenfolge undefiniert ist. Beispiel 5.9 zeigt die Abbildung von mehreren Teil-Typen nach der Regel 5.36.

**Beispiel 5.9 (Abbildung einer `attributeGroup` mit mehreren Teil-Typen):**

```
⟦<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:attributeGroup name="video">
    <xsd:attribute name="title" type="xsd:string" use="required"/>
    <xsd:attribute name="language" type="xsd:string"/>
  </xsd:attributeGroup>
</xsd:schema>⟧
```

$$\Downarrow$$

$$\texttt{"video"} \Mapsto_{attributeGroup}$$
$$\texttt{((attr "title" \{ named}_{type}\ \texttt{"xsd:string" \})}$$
$$\texttt{\& (attr "language" \{ named}_{type}\ \texttt{"xsd:string" \} ?))}$$

***Mapping* 5.37** (`local attributeGroup`):

$$[\![\ \texttt{<attributeGroup ref="r"/>}\ ]\!]$$
$$\Downarrow$$
$$\texttt{named}_{attributeGroup}\ \texttt{"r"}$$

Ein Typ wie in Beispiel 5.9 kann durch eine `local attributeGroup` referenziert werden. In der Abbildung aus Beispiel 5.10 greift das `element video` auf die `attributeGroup` aus Beispiel 5.9 zu.

**Beispiel 5.10 (Abbildung einer `local attributeGroup`):**

```
⟦<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="video">
    <xsd:complexType>
      <xsd:attributeGroup ref="video"/>
    <xsd:complexType>
  </xsd:element>
</xsd:schema>⟧
```

$$\Downarrow$$

$$\texttt{"video"} \Mapsto_{element}$$
$$\texttt{elem "video" \{ named}_{attributeGroup}\ \texttt{"video" \}}$$

## 5.11 `anyAttribute`

`anyAttribute` hat genauso wie `any` eine direkte Abbildung in *XQuery*: `anyAttribute`. In *XQuery* steht `anyAttribute` für ein einziges `attr`, in *XML Schema* allerdings für beliebig viele `attributes`. Deswegen wird `anyAttribute` auf beliebig viele (`"*"`) `anyAttributes` in *XQuery* abgebildet.

***Mapping* 5.38** (`anyAttribute`):

$$[\![\ \texttt{<anyAttribute/>}\ ]\!]$$
$$\Downarrow$$
$$\texttt{anyAttribute *}$$

## 5.12 `complexType`

Die `complexTypes` übernehmen in *XML Schema* die Strukturierung des *Schema*. In *XQuery* gibt es keine entsprechende Komponente, weshalb die Strukturierung

die Typkonstruktoren "," (zwischen `attributes` und Teil-Typ) und "&" (für die Kombination der `attributes`) übernehmen.

***Mapping* 5.39 (global complexType ohne Attribute und ohne Teil-Typ):**

$$[\![\ \texttt{<complexType name="n"/>}\ ]\!]$$
$$\Downarrow$$
$$\texttt{"n"} \mapsto_{type} \texttt{empty}$$

Ein `complexType` mit Teil-Typ enthält selbst keine Information. Deshalb wird nur der Teil-Typ rekursiv übersetzt und in die Typumgebung eingetragen.

***Mapping* 5.40 (global complexType mit Teil-Typ):**

$$[\![\ \texttt{<complexType name="n"> type </complexType>}\ ]\!]$$
$$\Downarrow$$
$$\texttt{"n"} \mapsto_{type} [\![\ \texttt{type}\ ]\!]$$

Ein `complexType` mit `attributes` verfährt genauso, wie die vorherige Regel, wenn nur ein einziges `attribute` vorhanden ist. Gibt es mehrere `attributes` werden diese wie in einer `attributeGroup` mit Hilfe des "&"-Typkonstruktors zu einem Typ verschmolzen.

***Mapping* 5.41 (global complexType mit Attributen):**

$$[\![\ \texttt{<complexType name="n"> attr}_1\ \texttt{attr}_2\ \texttt{...}\ \texttt{attr}_n\ \texttt{</complexType>}\ ]\!]$$
$$\Downarrow$$
$$\texttt{"n"} \mapsto_{type}$$
$$\texttt{(...(}[\![\ \texttt{attr}_1\ ]\!]\ \texttt{\&}\ [\![\ \texttt{attr}_2\ ]\!]\texttt{)}\ \texttt{\&}\ \texttt{...}\ \texttt{\&}\ [\![\ \texttt{attr}_n\ ]\!]\texttt{)}$$

Sind sowohl `attributes` als auch ein Teil-Typ in einem `complexType` enthalten, dann werden die `attributes` wie in der vorherigen Abbildung zusammengefasst und mit dem Teil-Typ in einer Sequenz verbunden (`attributes, type`).

***Mapping* 5.42 (global complexType mit Attributen und Teil-Typ):**

$$[\![\ \texttt{<complexType name="n"> type attr}_1\ \texttt{attr}_2\ \texttt{...}\ \texttt{attr}_n\ \texttt{</complexType>}\ ]\!]$$
$$\Downarrow$$
$$\texttt{"n"} \mapsto_{type}$$
$$\texttt{(...(}[\![\ \texttt{attr}_1\ ]\!]\ \texttt{\&}\ [\![\ \texttt{attr}_2\ ]\!]\texttt{)}\ \texttt{\&}\ \texttt{...}\ \texttt{\&}\ [\![\ \texttt{attr}_n\ ]\!]\texttt{),}\ [\![\ \texttt{type}\ ]\!]$$

Beispiel 5.11 zeigt einen größeren `complexType`, welcher mit den Abbildungsregeln, die bisher vorgestellt wurden, in einen *XQuery*-Typ abgebildet werden kann.

**Beispiel 5.11 (Abbildung `complexType` mit Teil-Typen und Attributen):**

```
⟦<?xml version="1.0"?>
 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:complexType name="video">
     <xsd:sequence>
       <xsd:element name="title" type="xsd:string"/>
       <xsd:element name="language" type="xsd:string"/>
     </xsd:sequence>
     <xsd:attribute name="duration" type="xsd:duration"/>
     <xsd:attribute name="year" type="xsd:integer"/>
   <xsd:complexType>
 </xsd:schema>⟧
                              ⇓
    "video" ⤇_type
      ((attr "duration" { named_type "xsd:duration" } &
        attr "year" { named_type "xsd:integer" })
       ,
       (elem "title" { named_type "xsd:string" },
        elem "language" { named_type "xsd:string" }))
```

Die einzelnen `elements` können mit der Regel 5.2 in `elem` abgebildet werden. Die `sequence` verbindet die beiden `elem` mit der Regel 5.19. Dies stellt den zweiten Block des Abbildungsbeispiels 5.11 dar. Die `attributes` werden mit der Regel 5.27 in `attr` umgeformt und mit Hilfe der Regel 5.42 mit dem Typkonstruktor `"&"` zum ersten Block der Abbildung zusammengefasst. Die gleiche Abbildungsregel (Regel 5.42) kombiniert auch die `attributes` und die `sequence` mit Hilfe des `","`-Typkonstruktors und trägt den ganzen Block in die Typumgebung ein.

Die Abbildungsregeln für die `local complexTypes` verhalten sich analog, zu den Regeln der `global complexType`. Sie werden nur nicht in die Typumgebung eingetragen, rekursiv übersetzt.

***Mapping* 5.43 (`local complexType` ohne Attribute und ohne Teil-Typ):**

```
⟦ <complexType/> ⟧
        ⇓
      empty
```

***Mapping* 5.44 (`local complexType` mit Teil-Typ):**

```
⟦ <complexType> type </complexType> ⟧
                 ⇓
             ⟦ type ⟧
```

***Mapping* 5.45 (`local complex` Type mit Attributen):**

```
⟦ <complexType> attr_1 attr_2 ... attr_n </complexType> ⟧
                           ⇓
    (...(⟦ attr_1 ⟧ & ⟦ attr_2 ⟧) & ... & ⟦ attr_n ⟧)
```

***Mapping* 5.46 (`local complexType` mit Attributen und Teil-Typ):**

```
⟦ <complexType> type attr_1 attr_2 ... attr_n </complexType> ⟧
                             ⇓
  (...(⟦ attr_1 ⟧ & ⟦ attr_2 ⟧) & ... & ⟦ attr_n ⟧), ⟦ type ⟧
```

## 5.13 simpleType

Die Abbildung der simpleTypes muss nur zwischen global simpleType und local simpleType unterschieden werden, da ein simpleType immer genau einen beinhalteten type besitzt. Der global simpleType wird in die Typumgebung mit dem rekursiv übersetzten Teil-Typ eingetragen und im local simpleType wird nur der Teil-Typ rekursiv übersetzt.

***Mapping* 5.47** (global simpleType **mit Teil-Typ**)**:**

$$[\![ \texttt{<simpleType name="n"> type </simpleType>} ]\!]$$
$$\Downarrow$$
$$\texttt{"n"} \mapsto_{type} [\![ \texttt{type} ]\!]$$

***Mapping* 5.48** (local simpleType **mit Teil-Typ**)**:**

$$[\![ \texttt{<simpleType> type </simpleType>} ]\!]$$
$$\Downarrow$$
$$[\![ \texttt{type} ]\!]$$

## 5.14 simpleContent und complexContent

Sowohl simpleContent als auch complexContent dienen nur zur Strukturierung des *XML Schema*. Dementsprechend wird nur jeweils der Teil-Typ rekursiv übersetzt.

***Mapping* 5.49** (simpleContent)**:**

$$[\![ \texttt{<simpleContent> type </simpleContent>} ]\!]$$
$$\Downarrow$$
$$[\![ \texttt{type} ]\!]$$

***Mapping* 5.50** (complexContent)**:**

$$[\![ \texttt{<complexContent> type </complexContent>} ]\!]$$
$$\Downarrow$$
$$[\![ \texttt{type} ]\!]$$

## 5.15 extension

extension wurde im Abschnitt 4.2.15 in simple und complex extension unterschieden. Beim Abbilden werden diese auch bis auf den Fall ohne Teil-Typ getrennt behandelt. Da das Attribut base zwingend erforderlich ist, wird der Typ einer extension ohne Teil-Typ auf den named type der base abgebildet.

***Mapping* 5.51** (extension **ohne Attribute und ohne Teil-Typ**)**:**

$$[\![ \texttt{<extension base="b"/>} ]\!]$$
$$\Downarrow$$
$$\texttt{named}_{type}\ \texttt{"b"}$$

Eine simple extension kann zusätzlich zum base-Typ attributes enthalten. Da der base-Typ der simple extension aus einem eingebauten Typ bzw. einem simpleType besteht, wird dieser bei der Abbildung genau wie bei einem complexType an die attributes angehängt.

***Mapping* 5.52** (`simple extension` **mit Attributen**):

$$[\![\ \texttt{<extension base="b"> attr}_1\ \texttt{attr}_2\ \ldots\ \texttt{attr}_n\ \texttt{</extension>}\ ]\!]$$
$$\Downarrow$$
$$\texttt{(...(}[\![\ \texttt{attr}_1\ ]\!]\ \texttt{\&}\ [\![\ \texttt{attr}_2\ ]\!]\texttt{)}\ \texttt{\&}\ \ldots\ \texttt{\&}\ [\![\ \texttt{attr}_n\ ]\!]\texttt{)}\ ,\ \texttt{named}_{type}\ \texttt{"b"}$$

**Beispiel 5.12 (Abbildung einer `simple extension`):**

```
[[ <xsd:element name="title">
     <xsd:complexType>
       <xsd:simpleContent>
         <xsd:extension base="video">
           <xsd:attribute name="language" type="xsd:string"/>
         </xsd:extension>
       </xsd:simpleContent>
     </xsd:complexType>
   </xsd:element> ]]
```

$$\Downarrow$$

$$\texttt{"title"}\ \mapsto_{element}\ \texttt{elem "title"}$$
$$\{\ \texttt{attr "language"}\ \{\ \texttt{named}_{type}\ \texttt{"xsd:string"}\ \}\texttt{?},$$
$$\texttt{named}_{type}\ \texttt{"xsd:string"}\ \}$$

In einer `complex extension` kann der `base`-Typ aus einer Sequenz von Attributen und einem Teil-Typ bestehen. Eine `complex extension`, die nun um eine neue Gruppe von Typen erweitert wird, hängt diese an den `base`-Typ, bzw. die darin definierten Typen, mittels des Sequenz-Typkonstruktors an.

***Mapping* 5.53** (`complex extension` **mit Teil-Typ**):

$$[\![\ \texttt{<extension base="b"> type </extension>}\ ]\!]$$
$$\Downarrow$$
$$\texttt{named}_{type}\ \texttt{"b"},\ [\![\ \texttt{type}\ ]\!]$$

Da die `complex extension` auch die Definition von Attributen zulässt, müssten die Attribute des `base`-Typs und die neuen Attribute mit dem Typkonstruktor `"&"` verbunden werden, um sicherzustellen, dass die Reihenfolge der Attribute beliebig bleibt. Allerdings kann der `base`-Typ nicht ohne größeren Aufwand zerlegt werden. Aus diesem Grund wird in Regel 5.54 eine ungenaue Abbildung nach *XQuery* zugelassen. Mit dem Typkonstruktor `"&"`, der intern in `choices` und `sequences` zerlegt wird, geht der richtige Typ nicht verloren (siehe Beispiel 5.13). Allerdings können dabei auch Typen zugelassen werden, die aus einem Teil-Typ gefolgt von Attributen bestehen.

**Beispiel 5.13 (Auflösung des Typkonstruktors `"&"`):**

```
(base-atts, base-type) ::= base
atts & base = atts, base | base, atts
            = atts, base-atts, base-type |
              base-atts, atts, base-type |
              ...
```

***Mapping* 5.54** (`complex extension` **mit Attributen**):

$$[\![\ \texttt{<extension base="b"> attr}_1\ \texttt{attr}_2\ \ldots\ \texttt{attr}_n\ \texttt{</extension>}\ ]\!]$$
$$\Downarrow$$
$$\texttt{(...(}[\![\ \texttt{attr}_1\ ]\!]\ \texttt{\&}\ [\![\ \texttt{attr}_2\ ]\!]\texttt{)}\ \texttt{\&}\ \ldots\ \texttt{\&}\ [\![\ \texttt{attr}_n\ ]\!]\texttt{)}\ \texttt{\&}\ \texttt{named}_{type}\ \texttt{"b"}$$

Wie Regel 5.54 bildet Regel 5.55 eine `extension` mit Attributen und Teil-Typ nur ungenau ab. Auch hier werden durch den Typkonstruktor `"&"` mehr Möglichkeiten zugelassen, um die Zerlegung des Attributs `base` zu umgehen.

***Mapping* 5.55 (`complex extension` mit Attributen und Teil-Typ):**

$$\llbracket \texttt{<extension base="b"> type attr}_1\ \texttt{attr}_2\ \texttt{...}\ \texttt{attr}_n\ \texttt{</extension>} \rrbracket$$
$$\Downarrow$$
$$\texttt{((...(}\llbracket \texttt{attr}_1 \rrbracket\ \texttt{\&}\ \llbracket \texttt{attr}_2 \rrbracket\texttt{)}\ \texttt{\&}\ \texttt{...}\ \texttt{\&}\ \llbracket \texttt{attr}_n \rrbracket\texttt{)}$$
$$\texttt{\& named}_{type}\ \texttt{"b"),}\ \llbracket \texttt{type} \rrbracket$$

## 5.16 `restriction`

Die Abbildungsregeln einer `restriction` können, wie die der `extension`, nicht zusammengelegt werden. Deswegen werden die drei Konzepte `simpleType restriction`, `simple restriction` und `complex restriction` aus Abschnitt 4.2.16 einzeln abgebildet.

Die `simpleType restriction` kann nur `facets` enthalten, die nicht sinnvoll in *XQuery* dargestellt werden können (siehe Abschnitt 5.21). Aus diesem Grund wird nur der `base`-Typ abgebildet.

***Mapping* 5.56 (`simpleType restriction` mit `base`-Attribut):**

$$\llbracket \texttt{<restriction base="b"> type </restriction>} \rrbracket$$
$$\Downarrow$$
$$\texttt{named}_{type}\ \texttt{"b"}$$

Anstatt des Attributs `base` kann der einzuschränkende Typ auch durch einen `simpleType`, der in einer `restriction` beinhaltet ist, ersetzt werden. Deshalb wird in Regel 5.57 der Teil-Typ der `restriction` abgebildet.

***Mapping* 5.57 (`simpleType restriction` ohne `base`-Attribut):**

$$\llbracket \texttt{<restriction> type </restriction>} \rrbracket$$
$$\Downarrow$$
$$\llbracket \texttt{type} \rrbracket$$

Die `simple restriction` kann genau wie die `simpleType restriction` entweder das Attribut `base` oder einen `simpleType` als einzuschränkenden Typen besitzen. Allerdings sind zusätzlich noch `attributes` erlaubt. Sind keine `attributes` definiert, dann stimmen die Abbildungsregeln mit denen der `simpleType restriction` überein (Regel 5.56 und Regel 5.58, Regel 5.57 und Regel 5.59).

***Mapping* 5.58 (`simple restriction` mit `base` und ohne Attribute):**

$$\llbracket \texttt{<restriction base="b"/>} \rrbracket$$
$$\Downarrow$$
$$\texttt{named}_{type}\ \texttt{"b"}$$

***Mapping* 5.59 (`simple restriction` ohne `base` und ohne Attribute):**

$$\llbracket \texttt{<restriction> type </restriction>} \rrbracket$$
$$\Downarrow$$
$$\llbracket \texttt{type} \rrbracket$$

Sind in der `simple restriction attributes` definiert, dann werden diese in einer Sequenz vor den einzuschränkenden Typ gehängt.

***Mapping* 5.60 (`simple restriction` mit `base` und mit Attributen):**

$$\llbracket\ \texttt{<restriction base="b"> attr}_1\ \texttt{attr}_2\ \ldots\ \texttt{attr}_n\ \texttt{</restriction>}\ \rrbracket$$
$$\Downarrow$$
$$\texttt{(...(}\llbracket\ \texttt{attr}_1\ \rrbracket\ \texttt{\&}\ \llbracket\ \texttt{attr}_2\ \rrbracket\texttt{)}\ \texttt{\&}\ \ldots\ \texttt{\&}\ \llbracket\ \texttt{attr}_n\ \rrbracket\texttt{),}\ \texttt{named}_{type}\ \texttt{"b"}$$

***Mapping* 5.61 (`simple restriction` ohne `base` und mit Attributen):**

$$\llbracket\ \texttt{<restriction> type attr}_1\ \texttt{attr}_2\ \ldots\ \texttt{attr}_n\ \texttt{</restriction>}\ \rrbracket$$
$$\Downarrow$$
$$\texttt{(...(}\llbracket\ \texttt{attr}_1\ \rrbracket\ \texttt{\&}\ \llbracket\ \texttt{attr}_2\ \rrbracket\texttt{)}\ \texttt{\&}\ \ldots\ \texttt{\&}\ \llbracket\ \texttt{attr}_n\ \rrbracket\texttt{),}\ \llbracket\ \texttt{type}\ \rrbracket$$

Die `complex restriction` schränkt einen `complexType` ein und muss, wie in Abschnitt 4.2.16 schon erklärt, sämtliche, im `base`-Typ definierten Typen wiederholen. Aus diesem Grund wird der `base`-Typ nicht mit abgebildet, sondern nur die Teil-Typen wie die eines `complexType` behandelt.

Eine Ausnahme bildet dabei die `complex restriction` ohne Attribute und ohne Teil-Typ, die genau wie die `simple restriction` ohne Attribute auf den `base`-Typ abgebildet wird.

***Mapping* 5.62 (`complex restriction` ohne Attribute und ohne Teil-Typ):**

$$\llbracket\ \texttt{<restriction base="b"/>}\ \rrbracket$$
$$\Downarrow$$
$$\texttt{named}_{type}\ \texttt{"b"}$$

***Mapping* 5.63 (`complex restriction` mit Teil-Typ):**

$$\llbracket\ \texttt{<restriction base="b"> type </restriction>}\ \rrbracket$$
$$\Downarrow$$
$$\llbracket\ \texttt{type}\ \rrbracket$$

***Mapping* 5.64 (`complex restriction` mit Attributen):**

$$\llbracket\ \texttt{<restriction base="b"> attr}_1\ \texttt{attr}_2\ \ldots\ \texttt{attr}_n\ \texttt{</restriction>}\ \rrbracket$$
$$\Downarrow$$
$$\texttt{(...(}\llbracket\ \texttt{attr}_1\ \rrbracket\ \texttt{\&}\ \llbracket\ \texttt{attr}_2\ \rrbracket\texttt{)}\ \texttt{\&}\ \ldots\ \texttt{\&}\ \llbracket\ \texttt{attr}_n\ \rrbracket\texttt{)}$$

***Mapping* 5.65 (`complex restriction` mit Attributen und Teil-Typ):**

$$\llbracket\ \texttt{<restriction base="b"> type attr}_1\ \texttt{attr}_2\ \ldots\ \texttt{attr}_n\ \texttt{</restriction>}\ \rrbracket$$
$$\Downarrow$$
$$\texttt{(...(}\llbracket\ \texttt{attr}_1\ \rrbracket\ \texttt{\&}\ \llbracket\ \texttt{attr}_2\ \rrbracket\texttt{)}\ \texttt{\&}\ \ldots\ \texttt{\&}\ \llbracket\ \texttt{attr}_n\ \rrbracket\texttt{),}\ \llbracket\ \texttt{type}\ \rrbracket$$

**Beispiel 5.14 (Abbildung einer `complex restriction`):**

```
⟦<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="video">
    <xsd:sequence>
      <xsd:element name="title" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute name="language" type="xsd:string"/>
  </xsd:complexType>
  <xsd:complexType name="video2">
    <xsd:complexContent>
      <xsd:restriction base="video">
        <xsd:sequence>
          <xsd:element name="title" type="xsd:string"/>
        </xsd:sequence>
        <xsd:attribute name="language" type="xsd:string"
           use="required"/>
      </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:schema>⟧
```

$$\Downarrow$$

```
"video"  ⤇type  attr "language" { named_type "xsd:string" }?,
                elem "title" { named_type "xsd:string" }
"video2" ⤇type  attr "language" { named_type "xsd:string" },
                elem "title" { named_type "xsd:string" }
```

In Beispiel 5.14 wird der Typ `video` eingeschränkt, indem das `attribute language` mit dem Attribut `use` spezifiziert wird. Damit unterscheidet sich die `restriction video2` nun nur durch den Typkonstruktor `"?"` von `video`. Wäre der `base`-Typ mit abgebildet worden, dann hätte `video2` zweimal das `elem title` und zweimal das `attr language`.

## 5.17 `list`

`list` erlaubt eine beliebig lange Folge von Werten eines vom Attribut `itemType` bestimmten Typs. Die Funktion einer Liste kann mit dem `"*"`-Typkonstruktor erreicht werden, der beliebig oft den `named type` des `itemType`-Attributs zulässt.

***Mapping* 5.66 (`list` mit `itemType`-Attribut):**

```
⟦ <list itemType="i"/> ⟧
          ⇓
   ( named_type "i" ) *
```

**Beispiel 5.15 (Abbildung einer `list` nach Regel 5.66):**

```
⟦ <xsd:list itemType="genre"/> ⟧
             ⇓
     (named_type "genre")*
```

Es ist möglich, statt des `itemType`-Attributs auch direkt einen Typ zu konstruieren. Auch dieser Typ wird mit dem Typkonstruktor `"*"` in eine Liste übersetzt.

***Mapping* 5.67 (`list` ohne `itemType`-Attribut):**

$$[\![\ \texttt{<list> type </list>}\ ]\!] \\ \Downarrow \\ [\![\ \texttt{type}\ ]\!]\ \texttt{*}$$

## 5.18 union

`union` kann in seinem Attribut `memberTypes` sowohl einen einzelnen Typ, als auch mehrere Typen speichern. Besteht das `memberTypes`-Attribut aus nur einem einzelnen Typ, dann wird dieser als `named type` abgebildet.

***Mapping* 5.68 (`union` mit `memberTypes`-Attribut (ein Eintrag)):**

$$[\![\ \texttt{<union memberTypes="m"/>}\ ]\!] \\ \Downarrow \\ \texttt{named}_{type}\ \texttt{"m"}$$

Sind in dem `memberTypes`-Attribut mehrere Typen zusammengefasst, dann sind in der Abbildung die einzelnen Typen miteinander durch eine `choice` ("`|`") verbunden. Jeder Typ wird dabei durch einen `named type` repräsentiert. Bei der Auswahl des Typs kann wegen der Bedeutung des Typkonstruktors "`|`" nur einer der Typen gewählt werden.

***Mapping* 5.69 (`union` mit `memberTypes`-Attribut (mehrere Einträge)):**

$$[\![\ \texttt{<union memberTypes="m}_1\ \texttt{m}_2\ \texttt{...}\ \texttt{m}_n\texttt{"/>}\ ]\!] \\ \Downarrow \\ \texttt{(...(named}_{type}\ \texttt{"m}_1\texttt{"}\ |\ \texttt{named}_{type}\ \texttt{"m}_2\texttt{"})\ |\ \texttt{...}\ |\ \texttt{named}_{type}\ \texttt{"m}_n\texttt{")}$$

**Beispiel 5.16 (Abbildung einer `union` nach Regel 5.69):**

$$[\![\ \texttt{<xsd:union memberTypes="year duration genrelist"/>}\ ]\!] \\ \Downarrow \\ \texttt{(named}_{type}\ \texttt{"year"}\ |\ \texttt{named}_{type}\ \texttt{"duration"}\ | \\ \texttt{named}_{type}\ \texttt{"genrelist")}$$

Es ist möglich, dass anstatt des Attributs `memberTypes` ein `simpleType` definiert ist. Statt eines `named type` wird dann der geschachtelte `simpleType` abgebildet.

***Mapping* 5.70 (`union` ohne `memberTypes`-Attribut (ein Teil-Typ)):**

$$[\![\ \texttt{<union> type </union>}\ ]\!] \\ \Downarrow \\ [\![\ \texttt{type}\ ]\!]$$

Enthält die `union` mehrere `simpleTypes`, dann wird jeder als einzelner Teil-Typ aufgefasst und mit dem Typkonstruktor "`|`" verbunden, genau wie bei dem Attribut `memberTypes`.

***Mapping* 5.71 (`union` ohne `memberTypes`-Attribut (mehrere Teil-Typen)):**

$$[\![\ \texttt{<union>}\ \texttt{type}_1\ \texttt{type}_2\ \texttt{...}\ \texttt{type}_n\ \texttt{</union>}\ ]\!] \\ \Downarrow \\ \texttt{(...(}[\![\ \texttt{type}_1\ ]\!]\ |\ [\![\ \texttt{type}_2\ ]\!]\texttt{)}\ |\ \texttt{...}\ |\ [\![\ \texttt{type}_n\ ]\!]\texttt{)}$$

## 5.19 *Symbol Spaces*

*Symbol Spaces* werden durch die verschiedenen `named types` in *XQuery* übernommen. *XQuery* hat dafür fünf verschiedene Typ-Umgebungen vorgesehen: *type*, *element*, *group*, *attribute* und *attributeGroup*.

## 5.20 Anwendung der Abbildungsregeln

In den vorherigen Abschnitten wurde ein komplettes *XML Schema* immer in einem Schritt nach *XQuery* abgebildet. Hier findet die Abbildung eines *Schema* nun Regel für Regel nach *XQuery* statt.

```
⟦<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="videos">
    <xsd:complexType>
      <xsd:sequence maxOccurs="unbounded">
        <xsd:element name="video">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="title" type="xsd:string"/>
              <xsd:element name="language" type="xsd:string"/>
            </xsd:sequence>
            <xsd:attribute name="nr" type="xsd:integer"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>⟧
```

$\Downarrow$ Abbildungsregel -- (`schema`)

```
⟦<xsd:element name="videos">
  <xsd:complexType>
    <xsd:sequence maxOccurs="unbounded">
      <xsd:element name="video">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="title" type="xsd:string"/>
            <xsd:element name="language" type="xsd:string"/>
          </xsd:sequence>
          <xsd:attribute name="nr" type="xsd:integer"/>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>⟧
```

$\Downarrow$ Abbildungsregel 5.3 (`global element` mit Teil-Typ)

```
"videos" ⤇type elem "videos"
   ⟦<xsd:complexType>
      <xsd:sequence maxOccurs="unbounded">
        <xsd:element name="video">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="title" type="xsd:string"/>
{             <xsd:element name="language" type="xsd:string"/>   }
            </xsd:sequence>
            <xsd:attribute name="nr" type="xsd:integer"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>⟧
```

⇓ Abbildungsregel 5.44 (`local complexType` mit Teil-Typ)

```
"videos" ⤇type elem "videos"
   ⟦<xsd:sequence maxOccurs="unbounded">
      <xsd:element name="video">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="title" type="xsd:string"/>
{           <xsd:element name="language" type="xsd:string"/>   }
          </xsd:sequence>
          <xsd:attribute name="nr" type="xsd:integer"/>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>⟧
```

⇓ Abbildungsregel 5.24 (`occurrences`)

```
"videos" ⤇type elem "videos"
{
     ⟦<xsd:sequence>
        <xsd:element name="video">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="title" type="xsd:string"/>
  (           <xsd:element name="language" type="xsd:string"/>   )+
            </xsd:sequence>
            <xsd:attribute name="nr" type="xsd:integer"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>⟧
}
```

⇓ Abbildungsregel 5.18 (`sequence` mit einem Teil-Typ)

```
"videos" ⤇type elem "videos"
{
      ⟦<xsd:element name="video">
         <xsd:complexType>
           <xsd:sequence>
             <xsd:element name="title" type="xsd:string"/>
   (         <xsd:element name="language" type="xsd:string"/>⟧ )+
           </xsd:sequence>
           <xsd:attribute name="nr" type="xsd:integer"/>
         </xsd:complexType>
       </xsd:element>
}
```

⇓ Abbildungsregel 5.6 (`local element` mit Teil-Typ)

```
"videos" ⤇type elem "videos"
{ (elem "video"
      ⟦<xsd:complexType>
         <xsd:sequence>
           <xsd:element name="title" type="xsd:string"/>
   {         <xsd:element name="language" type="xsd:string"/>⟧ }
           </xsd:sequence>e
         <xsd:attribute name="nr" type="xsd:integer"/>
       </xsd:complexType>
   )+
}
```

⇓ Abbildungsregel 5.46 (`local complexType` mit Attributen und Teil-Typ)

```
"videos" ⤇type elem "videos"
{ (elem "video"
   { (⟦ <xsd:attribute name="nr" type="xsd:integer"/> ⟧,
       ⟦<xsd:sequence>
          <xsd:element name="title" type="xsd:string"/>
          <xsd:element name="language" type="xsd:string"/>
        </xsd:sequence>⟧
      )
   })+
}
```

⇓ Abbildungsregel 5.31 (`local attribute` mit `type`-Attribut)

```
"videos" ⤇_type elem "videos"
{ (elem "video"
   { (attr "nr" { named_type "xsd:integer" },
      ⟦<xsd:sequence>
         <xsd:element name="title" type="xsd:string"/>
         <xsd:element name="language" type="xsd:string"/>
       </xsd:sequence>                                    ⟧
      )
   })+
}
```

⇓ Abbildungsregel 5.19 (`sequence` mit mehreren Teil-Typen)

```
"videos" ⤇_type elem "videos"
{ (elem "video"
   { (attr "nr" { named_type "xsd:integer" },
      (⟦ <xsd:element name="title" type="xsd:string"/> ⟧,
       ⟦ <xsd:element name="language" type="xsd:string"/> ⟧
     ))
   })+
}
```

⇓ Abbildungsregel 5.5 (`local element` mit `type`-Attribut)

```
"videos" ⤇_type elem "videos"
{ (elem "video"
   { (attr "nr" { named_type "xsd:integer" },
      (elem "title" { named_type "xsd:string" },
       ⟦ <xsd:element name="language" type="xsd:string"/> ⟧
     ))
   })+
}
```

⇓ Abbildungsregel 5.5 (`local element` mit `type`-Attribut)

```
"videos" ⤇_type elem "videos"
{ (elem "video"
   { (attr "nr" { named_type "xsd:integer" },
      (elem "title" { named_type "xsd:string" },
       elem "language" { named_type "xsd:string" }
     ))
   })+
}
```

## 5.21 Ignorierte *XML Schema*-Komponenten

### Ignorierte Elemente

- `annotation`, `notation` und `documentation`
  Die drei Kommentare werden nicht übersetzt, da beim *Schema* Import nach *XQuery* nur die Typen von Interesse sind.

- `facet`
  `facets` können nicht sinnvoll nach *XQuery* abgebildet werden, da die Ausdruckskraft der `facets` die der Typen in *XQuery* übersteigt. Die meisten `facets` befassen sich mit Längen von Zeichenfolgen bzw. Zahlenbereichen die in *XQuery* nur mit `"?"`, `"*"` oder `"+"` abgebildet werden könnten.

- `include`, `redefine` und `import`
  Die Komponenten `include`, `redefine` und `import` gehören nicht direkt zum Abbilden der Typen. Da ein mehrfacher *Schema* Import in *XQuery* möglich ist, wird dieser den *XML Schema*-Komponenten `include`, `redefine` und `import` vorgezogen.

- `unique`, `keyref`, `key`, `selector` und `field`
  Die Konzepte `unique` und `key` werden nicht übersetzt, da sie keine Typen, sondern nur die Werte von *XML*-Instanzen beeinflussen.

**Ignorierte Attribute**

- `final`
  Das Attribut `final` soll *Schema*-Typen davor schützen, erweitert oder eingeschränkt zu werden. Dies wird durch Überprüfung eines *Schema* auf Validität eingehalten und spielt beim *Schema* Import keine weitere Rolle, da die Typen in *XQuery* nicht mehr verändert werden.

- `fixed`
  `fixed` wird nicht übersetzt, da beim *Schema* Import nicht Instanzen von Typen, sondern nur die Typen selbst entscheidend sind. Außerdem stellt das Typsystem von *XQuery* keine Möglichkeit zur Verfügung, um Instanzen zu spezifizieren.

- `mixed`
  `mixed` erlaubt beliebige Zeichenketten zwischen Elementen innerhalb eines *XML*-Element. Eine Übersetzung müsste in *XQuery* zwischen allen `elem` Text-Typen einfügen. Da dies einen unverhältnismäßig hohen Aufwand darstellt, ist sinnvoller auf den Text-Typ bei der Übersetzung zu verzichten.

- `substitutionGroup`
  Das Attribut `substitutionGroup` wird nicht abgebildet, da eine Abbildung nicht mit den Konzepten dieser Arbeit, die in den folgenden Abschnitten beschriebenen werden, möglich ist. Außerdem kann die selbe Funktionalität mit Hilfe einer `choice`-Gruppe ausgedrückt werden. Anhang C bietet dennoch einen Vorschlag für die Umsetzung des Attributs `substitutionGroup`.

## 5.22 Vergleich: *Mapping – XQuery Formal Semantics*

In dem *Working Draft* des *W3C XQuery 1.0 and XPath 2.0 Formal Semantics*[5] werden Abbildungsregeln zum Import von *XML Schema* nach *XQuery* vorgeschlagen. Die Regeln werden in diesem Abschnitt ausschnittsweise mit dem in Abschnitt 5 beschriebenen *Mapping* verglichen und bei Abweichungen die Unterschiede begründet. Die Abbildungsregeln der *Formal Semantics* [13] verwenden eine andere Syntax. Aus diesem Grund werden im weiteren die Regeln der *Formal Semantics* auf die hier verwendete Syntax abgeändert.

Ein Großteil der Abbildungsregeln der *Formal Semantics* entsprechen genau den hier verwendeten Regeln. Dies betrifft z.B. die *Schema*-Komponenten `attribute`,

[5] *Working Draft* der *XQuery 1.0 and XPath 2.0 Formal Semantics* vom 22. August 2003: `http://www.w3.org/TR/2003/WD-xquery-semantics-20030822/`

`element`, `complexType`, `simpleType`, `all`, `choice`, `sequence`, `list` und `union`. Allerdings sind für die meisten *Schema*-Komponenten nur die Standardfälle abgebildet. In Abschnitt 5 dieser Arbeit sind dagegen alle möglichen Abbildungsregeln aufgeführt. In der *Formal Semantics* gibt es z.B. drei Abbildungsregeln für `restriction` (`simpleType`, `simple` und `complex`). Im *Mapping* dagegen sind es 10 `restriction`-Regeln, die damit alle Fälle berücksichtigen.

Für die Komponenten `include`, `redefine` und `import` wird vorgeschlagen die Auflösung der Funktionen einem *XML Schema-Processor* zu übergeben und danach erst einen *Schema* Import nach *XQuery* durchzuführen. Damit ignorieren sowohl *Mapping* als auch *Formal Semantics* diese Komponenten. Eine weitere Übereinstimmung ist die Behandlung von `facets`, die in beiden Abbildungen ignoriert werden.

Ein Unterschied entsteht dagegen bei den Komponenten `group` und `attributeGroup`, die im Gegensatz zum *Mapping* in dem *Working Draft* der *Formal Semantics* vom 22. August 2003 noch nicht behandelt werden. Ein weiterer kleiner Unterschied in den Abbildungsregeln ist Regel 5.25 (*occurrence*-Attribute mit `minOccurs="n"` und `maxOccurs="m"`, `m >= n >= 1`). Die *Formal Semantics* schlagen eine Übersetzung mit `"*"` vor. In Regel 5.25 wird dagegen `"+"` verwendet, da `n` nach Voraussetzung mindestens den Wert `1` annimmt.

Auch die Abbildung des Attributs `use` wird unterschiedlich gehandhabt. Die Abbildungsregeln der *Formal Semantics* übersetzen nur den Fall `"optional"`, während die anderen noch ein ungeklärter Punkt (*open issue*) sind. In Regel 5.29 des *Mapping* werden auch die fehlenden Werte des Attributs `use` (`"required"` und `"prohibited"`) abgebildet.

Die Attribute `substitutionGroup` und `mixed` sind die einzigen Punkte, bei denen die *Formal Semantics* genauer abbilden, als das *Mapping*. Beide Attribute greifen dabei auf *XQuery*-Typen zu, die in *Pathfinder* nicht bekannt sind und dementsprechend nicht verwendet werden. Anhang C liefert für die Abbildung des `substitutionGroup`-Attributs eine Lösung ohne `"substitution of ..."`.

***Formal Semantics* 5.1 (`element` mit `substitutionGroup`-Attribut):**

```
⟦ <element name="n" type="t" substitutionGroup="s"/> ⟧
                          ⇓
     elem "n" substitution of "s" { named "t" }
```

***Formal Semantics* 5.2 (`complexType` mit `mixed`-Attribut):**

```
⟦ <complexType mixed="true"> type </complexType> ⟧
                        ⇓
                 mixed ⟦ type ⟧
```

# 6 *SAX*

Ein *XML*-Dokument (bzw. ein *XML Schema*) muss geparsed werden, bevor es von einem Prozessor interpretiert werden kann. Das *Simple API for XML* (*SAX*) [6] unterstützt die Übersetzung, indem ein *SAX-Parser* das *XML*-Dokument einmal sequentiell einliest und es in kleine Stücke (*tokens*) zerlegt. Aus diesen *tokens* werden einzelne Ereignisse (*events*) erzeugt, die an das übersetzende Programm geschickt werden. Die Verbindung zwischen *SAX* und dem übersetzenden Programm geschieht über sogenannte *SAX-callbacks*, die die verschiedenen *events* mit Übersetzungfunktionen verbinden.

## 6.1 *SAX-events*

Ein *SAX-Parser* erkennt beim Zerlegen verschiedene *tokens*, die von ihrem syntaktischen Ballast befreit und als *events* verschickt werden. Ein solches *token* kann z.B. der Start eines *XML*-Elements sein (`<xsd:group name="foo">`), aus dem ein `start element event` erzeugt wird, das als Parameter den Namen des Elements, sowie die Attribute und deren Werte übermittelt (`xsd:group, name="foo"`).

**Beispiel 6.1 (Beispiel-*Schema*):**

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="video">
    <xsd:sequence>
      <xsd:element name="title" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

**Beispiel 6.2 (*SAX-events* zum Beispiel-*Schema* (Beispiel 6.1)):**

```
event:          parameters:
start document  -
start element   xsd:schema,
                xmlns:xsd="http://www.w3.org/2001/XMLSchema"
start element   xsd:complexType, name="video"
start element   xsd:sequence
start element   xsd:element, name="title", type="xsd:string"
end element     xsd:element
end element     xsd:sequence
end element     xsd:complexType
end element     xsd:schema
end document    -
```

Beispiel 6.2 zeigt die Zerlegung eines *Schema* (Beispiel 6.1), bei der in einem Durchgang nacheinander die aufgelisteten *events* erzeugt wurden.

Bei der Übersetzung eines *XML Schema* werden nur das `start element event`, sowie das `end element event` benötigt. Andere *events* wie z.B. `start document`, `end document`, `comment`, `characters` oder `processing instructions` sind durch die Struktur von *XML Schema* vernachlässigbar.

## 6.2 *SAX-callbacks*

Eine effiziente Verbindung zwischen *SAX* und einer Anwendung wird durch die *function callbacks* erreicht. Dabei werden vor dem Start des *Parsing* Funktionen an *events* gebunden. Während des *Parsing* wird bei jedem *event* die entsprechende Funktion mit den Parametern der *events* ausgeführt. Sobald die Funktion beendet ist, wird das nächste *event* interpretiert.

# 7 *State Machine*

In Abschnitt 5 wurde das *Mapping*, das die einzelnen *Schema*-Komponenten abbildet, vorgestellt. Im vorigen Abschnitt wurde die Zerlegung eines *Schema* in seine einzelnen Komponenten mit Hilfe eines *SAX-Parser* erklärt. Nun fehlt nur noch eine Koppelung zwischen *SAX* und dem *Mapping* um den *Schema* Import zu vervollständigen. Diese Koppelung übernimmt die *State Machine*, die in diesem Abschnitt vorgestellt wird. Sie wird allein durch die *SAX-events* angetrieben und führt die Abbildungsregeln aus.

## 7.1 *Endlicher Automat*

Die *State Machine* ist vergleichbar mit einem *endlichen Automaten* (*DFA*), der eine endliche Folge von Symbolen (*Eingabewort*) aus einer endlichen Menge von Symbolen (*Eingabealphabet*) einliest und diese am Ende akzeptiert oder zurückweist.

### 7.1.1 *Eingabewort* und *Eingabealphabet*

In der *State Machine* setzt sich ein *Eingabewort* aus den *events* zusammen, die ein *SAX-Parser* aus einem *XML Schema* generiert.

**Beispiel 7.1 (ein *Eingabewort*):**

```
start element   xsd:schema,
                xmlns:xsd="http://www.w3.org/2001/XMLSchema"
start element   xsd:element, name="video", type="video"
end element     xsd:element
end element     xsd:schema
```

Dementsprechend besteht das *Eingabealphabet* in der *State Machine* aus den einzelnen *SAX-events*, die ein *XML Schema* liefert. Für jede *Schema*-Komponente, die übersetzt wird, gibt es dabei jeweils zwei Symbole im *Eingabealphabet* (`start element event` und `end element event`):

**Beispiel 7.2 (das *Eingabealphabet*):**

```
start element  all, ...                 end element  all, ...
start element  any, ...                 end element  any, ...
start element  anyAttribute, ...        end element  anyAttribute, ...
start element  attribute, ...           end element  attribute, ...
start element  attributeGroup, ...      end element  attributeGroup, ...
start element  choice, ...              end element  choice, ...
start element  complexContent, ...      end element  complexContent, ...
start element  complexType, ...         end element  complexType, ...
start element  element, ...             end element  element, ...
start element  extension, ...           end element  extension, ...
start element  group, ...               end element  group, ...
start element  list, ...                end element  list, ...
start element  restriction, ...         end element  restriction, ...
start element  schema, ...              end element  schema, ...
start element  sequence, ...            end element  sequence, ...
start element  simpleContent, ...       end element  simpleContent, ...
start element  simpleType, ...          end element  simpleType, ...
start element  union, ...               end element  union, ...
```

### 7.1.2 Zustände und Übergänge

Ein *endlicher Automat* besteht aus einer festen Anzahl von Zuständen, die mit Hilfe von Übergängen untereinander verbunden sind. Diese Übergänge werden durch Symbole aus dem *Eingabealphabet* (also durch *SAX-events*) hergestellt, wobei bei jedem Symbol in einem Zustand genau ein anderer Zustand aufgerufen werden kann. Gibt es für ein eingelesenes Symbol in einem Zustand keinen Übergang, so wird die gesamte Eingabe zurückgewiesen. Der *endliche Automat* startet im Startzustand, der das erste Symbol des *Eingabeworts* einliest. Das letzte Symbol des *Eingabeworts* kann den *endlichen Automaten* entweder in einen besonders gekennzeichneten Endzustand oder in einen 'normalen' Zustand überführen. Handelt es sich um einen 'normalen' Zustand dann wird die Eingabe zurückgewiesen. Überführt das letzte Symbol des *Eingabeworts* den *endlichen Automaten* in einen Endzustand, dann

wird die Eingabe akzeptiert. Wird in der *State Machine* eine Eingabe zurückgewiesen, so bedeutet dies, dass das eingelesene *XML Schema* nicht *valid* ist bzw. eine *Schema*-Komponente verwendet wird, die nicht unterstützt wird (z.B. `facets`).

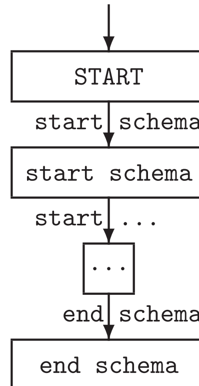


Abbildung 3: *endlicher Automat*

In Abbildung 3 startet der *endliche Automat* im Zustand `Start` und weist die Eingabe zurück, falls nicht das Symbol `start schema` auftritt. Der *endliche Automat* wird, durch die *events* des *Schema* bestimmt, von Zustand zu Zustand solange durchlaufen bis das letzte *event*, im Normalfall `end schema`, den *endlichen Automaten* in den Endzustand überführt. Die Konstruktion der einzelnen *XQuery*-Typen entsteht dabei als Seiteneffekt.

## 7.2 *Mini State Machine* – Konzepte

Die *State Machine* bildet einen *endlichen Automaten* mit Hilfe einer Zustandstabelle (*State Table*) ab. Diese Zustandstabelle ist allerdings relativ groß und daher nicht das geeignete Mittel, um die Konzepte der *State Machine* zu erklären. Deshalb wird im weiteren zur Erklärung der Konzepte eine *Mini Schema* Sprache mit der entsprechenden *Mini*-Zustandstabelle eingeführt.

### 7.2.1 *Mini Schema*

Die *Mini Schema* Sprache kennt nur die Elemente `schema`, `element` und `group`. Der Inhalt von `schema` besteht aus einem einzigen `element`. Dieses kann entweder ein `type`-Attribut oder eine `group` beinhalten. Eine `group` besteht wiederum aus einem oder mehreren `elements`.

**Typsyntax (*Mini Schema*):**

```
schema ::= element
element (@name, @type) ::= group ?
group ::= element +
```

**Beispiel 7.3 (*Mini Schema*):**

```
<schema>
  <element name="video">
    <group>
      <element name="title" type="string"/>
      <element name="language" type="string"/>
    </group>
  </element>
</schema>
```

Da das *Mini Schema* ebenfalls in *XQuery* abgebildet werden soll, gibt es auch vier Abbildungsregeln. Die einzelnen Regeln sind mit denen des *XML Schema* vergleichbar.

***Mapping* 7.1 (*Mini* element mit type-Attribut):**

$$\begin{array}{c} [\![ \; \texttt{<element name="n" type="t"/>} \; ]\!] \\ \Downarrow \\ \texttt{elem "n"} \; \{ \; \texttt{named}_{type} \; \texttt{"t"} \; \} \end{array}$$

***Mapping* 7.2 (*Mini* element mit Teil-Typ):**

$$\begin{array}{c} [\![ \; \texttt{<element name="n"> type </element>} \; ]\!] \\ \Downarrow \\ \texttt{elem "n"} \; \{ \; [\![ \; \texttt{type} \; ]\!] \; \} \end{array}$$

***Mapping* 7.3 (*Mini* group mit einem Teil-Typ):**

$$\begin{array}{c} [\![ \; \texttt{<group> type </group>} \; ]\!] \\ \Downarrow \\ [\![ \; \texttt{type} \; ]\!] \end{array}$$

***Mapping* 7.4 (*Mini* group mit mehreren Teil-Typen):**

$$\begin{array}{c} [\![ \; \texttt{<group> type}_1\texttt{, type}_2\texttt{, ..., type}_n \; \texttt{</group>} \; ]\!] \\ \Downarrow \\ \texttt{(...(}[\![ \; \texttt{type}_1 \; ]\!] \; \texttt{,} \; [\![ \; \texttt{type}_2 \; ]\!]\texttt{)} \; \texttt{, ... ,} \; [\![ \; \texttt{type}_n \; ]\!]\texttt{)} \end{array}$$

Die Abbildung des Beispiels 7.3 nach den vier Abbildungsregeln ergibt Beispiel 7.4:

**Beispiel 7.4 (auf *XQuery* abgebildetes *Mini Schema* (Beispiel 7.3):**

$$\begin{array}{l} \texttt{elem "video"} \; \{ \; \texttt{(elem "title"} \; \{ \; \texttt{named}_{type} \; \texttt{"string"} \; \}\texttt{,} \\ \qquad\qquad\quad \texttt{elem "language"} \; \{ \; \texttt{named}_{type} \; \texttt{"string"} \; \}\texttt{)} \; \} \end{array}$$

### 7.2.2 *Mini State Table*

Die Zeilen der *Mini State Table* (siehe Abbildung 4) beschreiben wie die originale *State Table* die einzelnen Zustände. Die einzelnen Spalten der Zustandstabelle stehen jeweils für ein `element event`, welches ein *SAX*-Interpreter erzeugt. Für jedes *Schema*-Elements gibt es eine `start-event`- und eine `end-event`-Spalte (siehe Abschnitt 7.1.1). Im *Mini Schema* reagieren damit sechs Spalten auf die *SAX-events*. Die Zahlen in der Tabelle entsprechen den Übergängen eines *endlichen Automaten* zu anderen Zuständen und geben Auskunft, welche *events* von einem bestimmten Zustand akzeptiert werden. Die Löcher in der Zustandstabelle kennzeichnen damit entsprechend, welche *events* in einem Zustand nicht erlaubt sind.

Beispiel (bezogen auf Abbildung 4): Wenn sich die *Mini State Machine* gerade im Zustand des `start element` (`3`) befindet, dann wird entweder ein `group start-event` oder ein `element end-event` akzeptiert. Ist das nächste *event* ein `group start-event`, dann ist `5` der neue Zustand (`start group`).

Die *Mini State Table* aus Abbildung 4 kann auch wieder als *endlicher Automat* dargestellt werden. Abbildung 5 zeigt den zur *Mini State Table* passenden *endlichen Automaten*.

Zum Abbilden eines *Schema* auf *XQuery* müssen nun nur die Zustandstabelle durchlaufen und dabei die Abbildungsregeln ausgelöst werden. Indem an jeden Zustand eine Aktion, die automatisch beim Erreichen des Zustandes aufgerufen wird, angehängt wird können die Abbildungsregeln ausgeführt werden. Die

| | event | start | | | end | | |
|---|---|---|---|---|---|---|---|
| | | element | group | schema | element | group | schema |
| 0 | START | | | 1 | | | |
| 1 | start schema | 3 | | | | | |
| 2 | end schema | | | | | | |
| 3 | start element | | 5 | | 4 | | |
| 4 | end element | 3 | | | | 6 | 2 |
| 5 | start group | 3 | | | | | |
| 6 | end group | | | | 4 | | |

Abbildung 4: *Mini State Table* (1. Versuch)

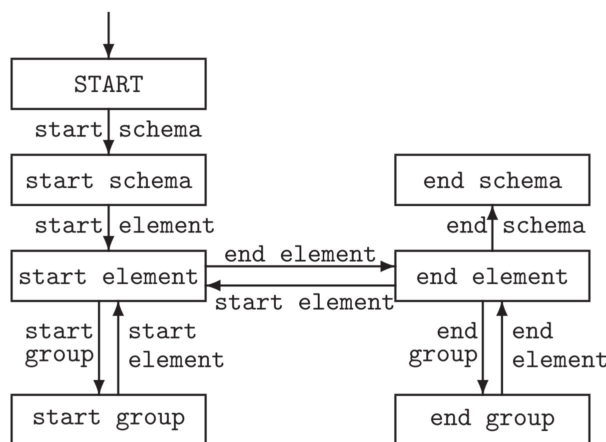


Abbildung 5: *endlicher Automat* zur *Mini State Table*

Wahl der Zustände, deren Aktionen die Abbildungsregeln ausführen, fällt auf die `end-event`-Zustände, da zum Zeitpunkt eines `start-event` der Inhalt und somit der Typ der Komponente noch unbekannt ist (Beispiel: Zum Zeitpunkt des `group start-event` ist nicht bekannt, welche Teil-Typen die `group` enthält). Beim Erreichen eines `end-event`-Zustandes sind dagegen alle Teil-Typen bekannt.

Die Aktion eines `end-event`-Zustandes muss nun die richtige Abbildungsregel für das jeweilige Element auswählen. Im *Mini Schema* muss zum Beispiel sowohl für `element` als auch für `group` jeweils eine der zwei Abbildungsregeln ausgewählt werden. In *XML Schema* ist die Auswahl noch schwieriger, da z.B. für das Element `complexType` eine der acht Regeln auszuwählen ist.

- Zustand: `4` (`end element`)
  Aktion: Abbildungsregel 7.1 vs. Abbildungsregel 7.2

- Zustand: `6` (`end group`)
  Aktion: Abbildungsregel 7.3 vs. Abbildungsregel 7.4

Eine bessere Lösung wäre eine Zustandstabelle, die auf der einen Seite aus einer größeren Anzahl von Zuständen besteht, auf der anderen Seite aber die Zuordnung einer Abbildungsregel zu einem Zustand erlaubt.

Ein weiteres Problem entsteht beim Verarbeiten der Regel 7.4. Bis zum Zeitpunkt des `group end-event` können eine unbekannte Anzahl Teil-Typen angesam-

melt worden sein, die nun alle auf einmal verarbeitet werden müssen. Es wäre sinnvoller, nach jedem `element` innerhalb einer `group` die Sequenz zu bilden, anstatt erst beim `end-event` die ganze Arbeit zu verrichten. Damit wäre z.B. die Aktion eines `element end-event` abhängig vom vorherigen Zustand (Inhalt einer `group` oder eines `schema`). Eine größere Tabelle, bei der jeweils der vorherige Zustand beachtet wird, kann allerdings nicht entworfen werden, da die Größe der Tabelle von der möglichen Verschachtelungstiefe des *Schema* abhinge. Die Verschachtelungstiefe des *Mini Schema* ist aber, ebenso wie die des *XML Schema*, wegen der Rekursivität beliebig tief.

Die Lösung des Problems ist ein *stack*, der die Zustände der *State Table* speichert, die zuvor aktiv waren. Der *stack* hilft zu erkennen, ob z.B. das beendete `element` innerhalb einer `group` oder eines `schema` geschachtelt war und kann dementsprechend Aktionen ausführen.

### 7.2.3 Erweiterte *Mini State Table*

Abbildung 6 stellt eine überarbeite Version der *Mini*-Zustandstabelle dar. Sie hat fünf Zustände mehr als die 1. Version und eine zusätzliche Spalte, die den Umgang mit dem *stack* erleichtert. Außerdem beruht das Konzept der Zustandstabelle nun nicht mehr auf einem *endlichen Automat* sondern auf einem *Stack Automat*.

| | event | start | | | end | | | |
|---|---|---|---|---|---|---|---|---|
| | | element | group | schema | element | group | schema | stack |
| 0 | START | | | 1 | | | | |
| 1 | start schema | 4 | | | | | | 2 |
| 2 | zw. schema | | | | | | 3 | |
| 3 | end schema | | | | | | | |
| 4 | start element | | 8 | | 6 | | | 5 |
| 5 | zw. element | | | | 7 | | | |
| 6 | end element 1 | | | | | | | |
| 7 | end element 2 | | | | | | | |
| 8 | start group | 4 | | | | | | 9 |
| 9 | zw. group 1 | 4 | | | | 11 | | 10 |
| 10 | zw. group 2 | 4 | | | | 11 | | |
| 11 | end group | | | | | | | |

Abbildung 6: erweiterte *Mini State Table*

Hinzugekommen ist ein zweiter Zustand für das `end-event` des `element` (Zeile `6` und `7`). Anhand dieses neuen Zustandes kann beim `element start-event` ohne Teil-Typ das erste `element end-event` gewählt werden, während mit Teil-Typ das zweite gewählt wird. Damit kann an jeden der beiden Zustände jeweils eine der Abbildungsregeln für `elements` angehängt werden.

Der vorherige Zustand kann gespeichert werden, indem bei jedem `start-event` der Inhalt der `stack`-Spalte des neuen, aktuellen Zustandes auf den *stack* gelegt wird. Der vorherige Zustand wird nach jedem `end-event` wieder aktiviert, indem das oberste Element des *stack* gelöscht wird, da das aktuelle Element beendet wurde, und die *stack*-Spitze zum neuen aktuellen Zustand gemacht wird. Diese vom *stack* gewonnenen Zustände (z.B. Zustände `2`, `5`, `8` und `10` der Abbildung 6) sind sogenannte Zwischenzustände. Sie führen wichtige Aktionen im Zusammenhang mit den Abbildungsregeln aus (z.B. die Verbindung der einzelnen `element`-Typen innerhalb einer `group`) und legen neue Übergänge fest. Für jedes Element, das einen Teil-Typ enthalten könnte, wird mindestens ein Zwischenzustand eingeführt, damit nach jedem `end-event` immer ein weiterer Zustand ausgeführt werden kann.

Die Zusammenfassung der Übergänge in einem Zwischenzustand und die damit gewonnene Übersichtlichkeit, sowie geringe Fehleranfälligkeit ist dabei nur ein

kleiner, zusätzlicher Bonus.

Abbildung 7 zeigt den, der erweiterten *Mini State Table* (Abbildung 6) entsprechenden, *Stack Automat*. Dieser ist aufgeteilt in drei Blöcke: schema (**0**), element (**1**) und group (**2**). Jeder Block beschreibt jeweils eine Ebene bzw. eine *Schema*-Komponente, die durch ein `start-event` eingeleitet, durch beinhaltete Elemente unterbrochen und durch den Aufruf von Zwischenzuständen beendet wird.

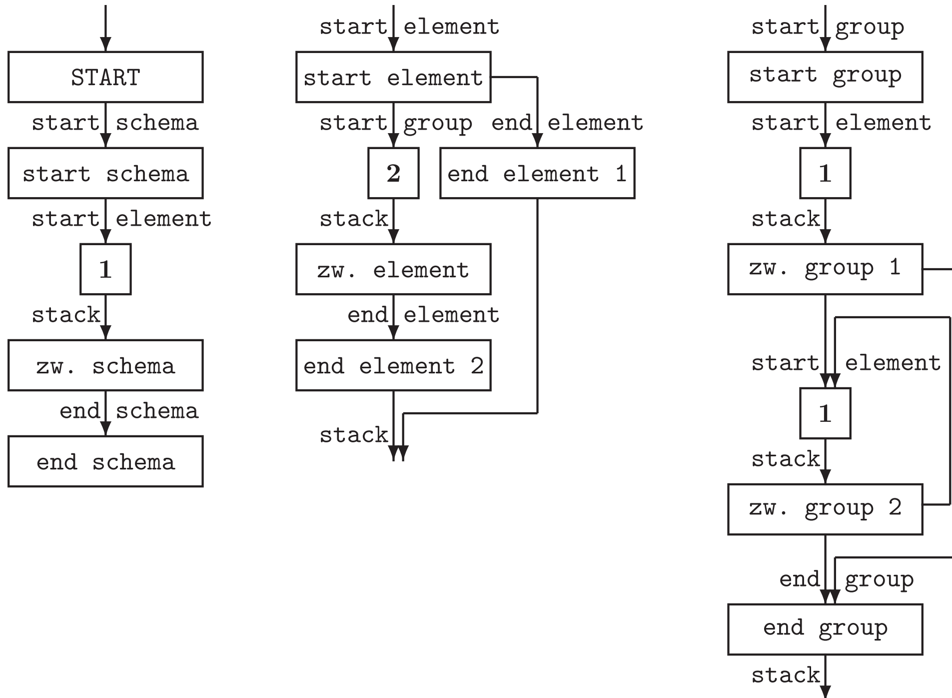


Abbildung 7: *Mini Stack Automat* (**0** `schema`, **1** `element`, **2** `group`)

Mit Hilfe der Änderungen in der erweiterten *Mini State Table* ist es nun möglich jeder der vier Abbildungsregeln jeweils einen Zustand als Aktion zuzuordnen:

- Zustand: `6` (`end element 1`)
  Aktion: Abbildungsregel 7.1
- Zustand: `7` (`end element 2`)
  Aktion: Abbildungsregel 7.2
- Zustand: `9` (`zw. group 1`)
  Aktion: speichere `elem` als `type`
  Aktion: ändere `stack`-Spitze auf Wert der `stack`-Spalte (von `9` nach `10`)
- Zustand: `10` (`zw. group 2`)
  Aktion: bilde Sequenz: (`type`, `elem`)
  Aktion: speichere Sequenz als neuen `type`
- Zustand: `11` (`end group`)
  Aktion: Abbildungsregel 7.3

Abbildungsregel 7.4 wird hier durch die Zwischenzustände `9` und `10` gelöst, indem Zustand `9` den ersten zurückgegebenen Teil-Typ (`elem`) speichert und durch die Änderung des *stack* bewirkt, dass statt ihm nur noch Zustand `10` aktiv wird. Zustand `10` verarbeitet jeden neuen Teil-Typ (`elem`) sofort zu einer Sequenz aus dem

bisherigen Teil-Typ (`type`) und dem `elem`. Dadurch besteht beim `end-event` der `group` der Ergebnis-Typ schon aus einer Sequenz von `elements`, bzw. einem einzigen `element`, falls das `end-event` der `group` im Zustand `9` geliefert wurde.

**Beispiel 7.5 (Anwendung der *Mini State Machine*):**
Anhand des *Mini Schema* (Beispiel 7.3) wird im Folgenden Schritt für Schritt die Anwendung der *Mini State Table* durchlaufen.

1. aktueller Zustand: `0` (`START`)
   Erklärung: Startzustand

2. event: `start schema`
   aktueller Zustand: `1` (`start schema`)
   Aktion: lege `2` auf den Zustand-*stack*

3. event: `start element`
   aktueller Zustand: `4` (`start element`)
   Aktion: lege `5` auf den Zustand-*stack*
   Erklärung: `element "video"` gesehen

4. event: `start group`
   aktueller Zustand: `8` (`start group`)
   Aktion: lege `9` auf den Zustand-*stack*

5. event: `start element`
   aktueller Zustand: `4` (`start element`)
   Aktion: lege `5` auf den Zustand-*stack*
   Erklärung: `element "title"` gesehen

6. event: `end element`
   aktueller Zustand: `6` (`end element 1`)
   Aktion:

   - führe Abbildungsregel 7.1 aus
   - lösche oberstes Element vom Zustand-*stack* (`5`)
   - setze aktuellen Zustand auf `9` (`zw. group 1`)
   - führe Aktionen aus:
     – `type := elem "title" {` $\texttt{named}_{type}$ `"string" }`
     – lösche oberstes Element vom Zustand-*stack* (`9`)
     – lege `10` auf den Zustand-*stack*

   Erklärung: die Aktionen für ein `end-event` werden ausgeführt und danach der Zwischenzustand des vorherigen Zustandes aktiviert und ausgeführt. Dieser speichert den neuen *XQuery*-Typ und ändert den Zustand-*stack*.

7. event: `start element`
   aktueller Zustand: `4` (`start element`)
   Aktion: lege `5` auf den Zustand-*stack*
   Erklärung: `element "language"` gesehen

8. event: `end element`
   aktueller Zustand: `6` (`end element 1`)
   Aktion:

   - führe Abbildungsregel 7.1 aus
   - lösche oberstes Element vom Zustand-*stack* (`5`)

- setze aktuellen Zustand auf `10` (`zw. group 2`)
- führe Aktionen aus:
  - ```
    type := (type,
              elem "language" { named_type "string" })
    ```

Erklärung: die Aktionen für ein `end-event` werden ausgeführt und danach der Zwischenzustand des vorherigen Zustandes aktiviert und ausgeführt. Dieser kombiniert die beiden *XQuery*-Typen zu einer Sequenz:

```
type := (elem "title" { named_type "string" },
         elem "language" { named_type "string" })
```

9. event: `end group`
aktueller Zustand: `11` (`end group`)
Aktion:

   - führe Abbildungsregel 7.3 aus
   - lösche oberstes Element vom Zustand-*stack* (`10`)
   - setze aktuellen Zustand auf `5` (`zw. element`)
   - führe Aktionen aus: -

   Erklärung: die Aktionen für ein `end-event` werden ausgeführt und danach der Zwischenzustand des vorherigen Zustandes aktiviert und ausgeführt. Dieser hat keine Aktionen.

10. event: `end element`
aktueller Zustand: `7` (`end element 2`)
Aktion:

   - führe Abbildungsregel 7.2 aus
   - lösche oberstes Element vom Zustand-*stack* (`5`)
   - setze aktuellen Zustand auf `2` (`zw. schema`)
   - führe Aktionen aus: -

   Erklärung: die Aktionen für ein `end-event` werden ausgeführt und danach der Zwischenzustand des vorherigen Zustandes aktiviert und ausgeführt.

11. event: `end schema`
aktueller Zustand: `3` (`end schema`)
Aktion: trage Typ in Typumgebung ein
Erklärung: Ende des Programms

## 7.3 *State Machine*

Die Konzepte der *Mini State Machine* stimmen grundsätzlich mit denen der *State Machine* für *XML Schema* überein. Im Gegensatz zur *Mini State Machine* besteht die *State Machine* allerdings aus 173 Zeilen und 37 Spalten (18 `start element events`, 18 `end element events`, 1 `stack`-Spalte – siehe Anhang A und B). Deshalb werden im Weiteren nur drei *Schema*-Komponenten (`complexType`, `attributes` und `union`) genauer vorgestellt, da diese zusätzliche, erwähnenswerte Konzepte verwenden.
```

### 7.3.1 complexType – Konzepte

Zum Abbilden von complexTypes gibt es acht Abbildungsregeln. Davon sind vier Regeln für global complexTypes und vier für local complexTypes. Eine Unterscheidung kann durch die Verdoppelung der complexType Zustände erreicht werden. Dabei helfen die Übergänge (bei den start-event-) und der Zustand-*stack* (bei den end-event-Zuständen) den richtigen complexType Block auszuwählen. Ohne den Zustand-*stack* mit den gespeicherten Zwischenzuständen wäre nicht klar, ob ein complexType end-event zu einem *lokalen* oder einem *globalen* complexType gehört.

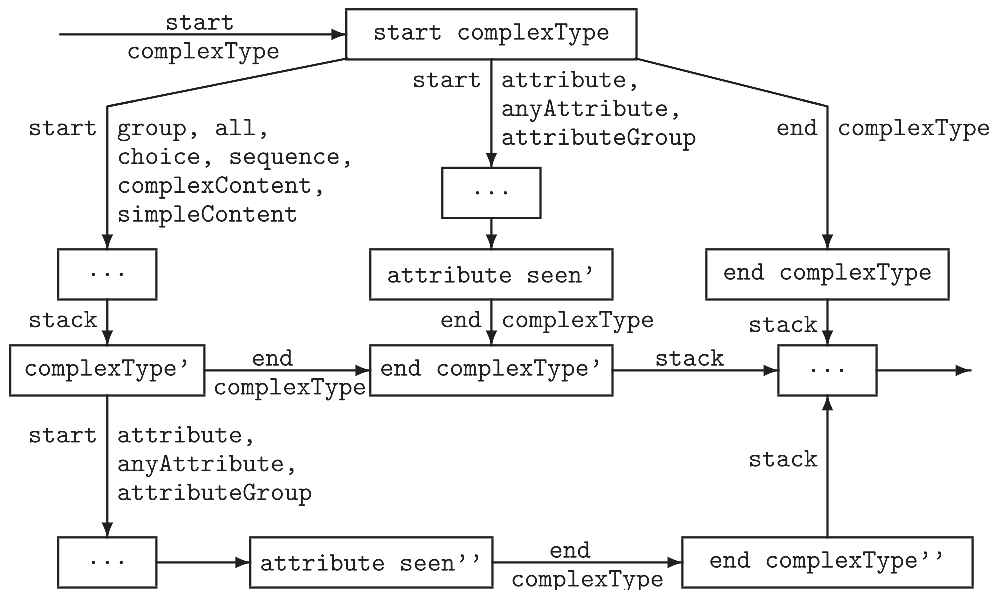


Abbildung 8: *Stack Automat* für complexType

Abbildung 8 zeigt den *Stack Automat* für die verbliebenen vier Regeln, die unterschieden werden müssen. Er besteht aus fünf Zuständen für den complexType:

- start complexType
  wird beim Auftreten des complexType start-event ausgelöst. Der Zustand complexType' wird auf den Zustand-*stack* gelegt. Die folgenden Komponenten können in drei Gruppen eingeteilt werden: Teil-Typen (group, all, choice, sequence, simpleContent, complexContent), Attribute (attribute, anyAttribute, attributeGroup) und das end-event für den complexType. Entweder endet der complexType gleich oder es werden eine Reihe von Zuständen durchlaufen bis der complexType weiter behandelt wird.

- end complexType
  wird durch das Auftreten des complexType end-event im Zustand start complexType ausgelöst. Die Aktion des Zustandes end complexType löst die Abbildungsregeln eines complexType ohne Teil-Typ und ohne Attribute aus (Regel 5.39, Regel 5.43).

- complexType'
  wird ausgeführt, wenn beim end-event-Zustand eines Teil-Typs des complex-

Type der Zwischenzustand vom *stack* geholt wird. Im Zustand complexType' sind nur noch die Übergänge zum end-event des complexType und zu attributes, anyAttribute, und attributeGroups erlaubt.

- end complexType'
  wird beim Auftreten des complexType end-event ausgelöst, das attributes oder complexType' folgt. Dies funktioniert sowohl für Regeln, die nur einen Teil-Typ haben (Regel 5.40 und Regel 5.44), als auch für complexTypes, die mehrere attributes beinhalten, da die attributes, wie eine group im *Mini Schema*-Beispiel, schon während des Durchlaufs des Zustandes attribute seen zu einem einzigen Typ verarbeitet werden (siehe Abschnitt 7.3.2). Entsprechend kann der Zustand end complexType' auch die Regeln 5.41 und 5.45 verarbeiten.

- end complexType''
  wird beim Auftreten eines complexType end-event ausgelöst, das sowohl einen Teil-Typ als auch attributes beinhaltet. Hier wird die letzte Abbildungsregel für zwei Teil-Typen ausgeführt (Regel 5.42 und Regel 5.46). Es wird nur eine Sequenz aus attributes und element-Gruppe erstellt, da die attributes wie im Zustand end complexType' zu einem Teil-Typ vorverarbeitet wurden.

### 7.3.2 Konzept der attribute Kombination

local attributes, anyAttribute und local attributeGroups sind Komponenten von *XML Schema*, die in complexTypes, global attributeGroups, extensions und restrictions in beliebiger Kombination und Anzahl vorkommen können. Sie sind in allen aufgezählten Komponenten die letzten möglichen Typen, bevor deren end-event auftaucht. Da es für complexType unmöglich ist, eine element-Gruppe von attributes zu unterscheiden, werden die attributes gesondert behandelt. Ein kompletter Block an Zuständen (siehe Abbildung 9) sorgt für die Verarbeitung von local attributes, local attributeGroups und anyAttribute. Der Zustand attribute seen kombiniert die drei Komponenten, indem er von ihnen nach ihren Aktionen anstelle eines Zwischenzustandes manuell aufgerufen wird. Das Konzept von attribute seen funktioniert ähnlich wie die group im *Mini Schema*-Beispiel. In einem ersten Schritt wird in attribute seen' keine Aktion ausgelöst, sondern nur Übergänge zu einer neuen Instanz von attributes bereitgestellt. Im dann folgenden attribute seen'' wird der bisherige Typ mit dem neuen attribute bzw. der neuen attributeGroup kombiniert. Dies geschieht so lange, bis das end-event des umschließenden Elements gesehen wird. Die attributes werden damit als ein einziger Teil-Typ zurückgegeben.

Leider reicht ein Zustand-Block, der local attributes, anyAttribute und local attributeGroups kombiniert, nicht aus. Ein local complexType z.B. kann bei einem complexType end-event nicht von einem global complexType unterschieden werden. Außerdem ist die Wahl der Abbildungsregel in einem complexType abhängig davon, ob eine element-Gruppe vor den attributes gesehen wurde. Dies bedeutet, dass zwei verschiedene Übergänge vorgesehen werden müssen. Um nun eine einfache Unterscheidung zu ermöglichen, kommt der Zustand-Block (siehe Abbildung 9) vier Mal mit verschiedenen Übergängen in der *State Table* vor.

### 7.3.3 union – Konzept

union nimmt eine Sonderrolle ein, da vom Zustand start union aus der Übergang von dem Attribut memberTypes abhängt. Ist das Attribut vorhanden, so wird automatisch vom Zustand start union in den Zustand union@mT gewechselt, in

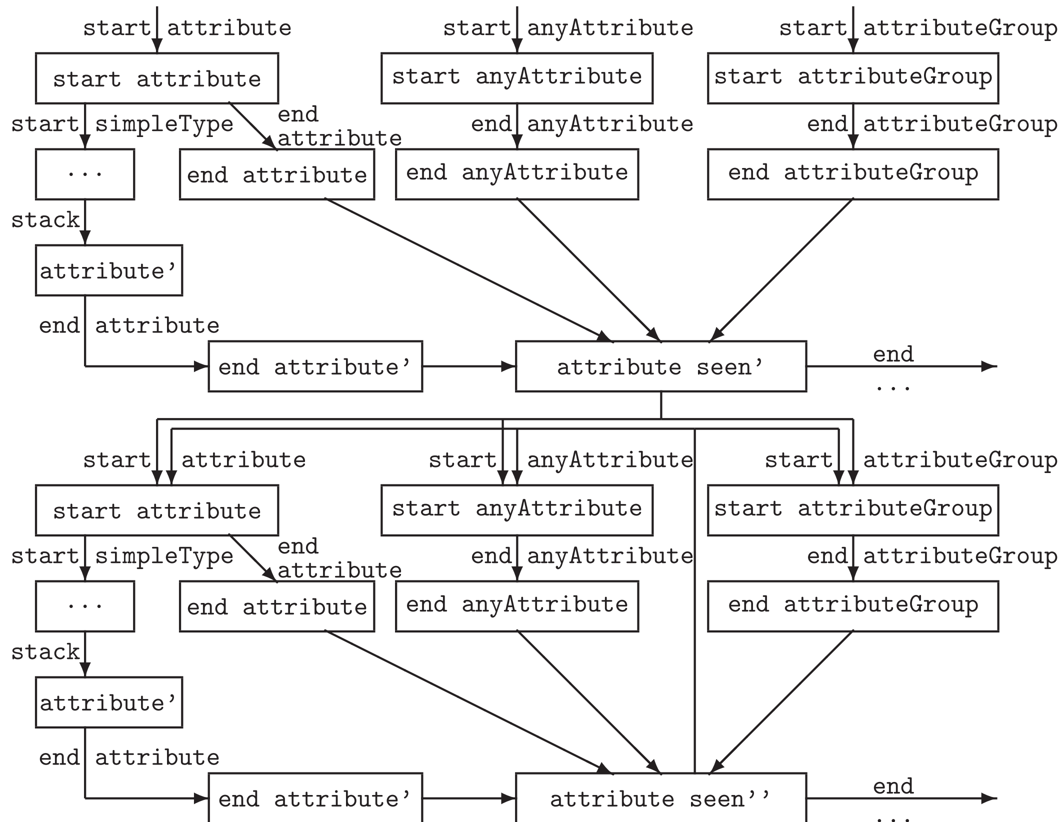


Abbildung 9: *Stack Automat* für `attributes`, `anyAttribute` und `attributeGroup`

welchem ein Teil-Typ (anders als in `start union`) nicht erlaubt ist. Ist das Attribut nicht vorhanden, so werden die einzelnen Teil-Typen wie eine `choice`-Gruppe kombiniert und verarbeitet (siehe *Stack Automat* Abbildung 10).

Die Kombination der Teil-Typen wird durch die Zustände `union'` und `union''` gelöst. Es wäre stattdessen auch möglich, diese Kombination mit einem einzigen Zustand zu bewerkstelligen. Der Zustand `union'` wird überflüssig, indem der erste Teil-Typ mit dem neutralen Element `none` des Typkonstruktors `"|"` kombiniert wird. Das neutrale Element entfällt in einer Optimierung des *XQuery*-Typen wieder und die Aktion kann im Zustand `union''` ausgeführt werden. Diese Änderung könnte entsprechend auch bei `all`, `choice` und `sequence` angewendet werden.

Da allerdings die `attributes` zwei Schleifen durchlaufen, die nicht zusammengefasst werden können, werden auch `union`, `all`, `choice` und `sequence` mit zwei Schleifen (z.B. `union'` und `union''`) durchlaufen.

# 8 Implementation

In diesem Abschnitt wird näher auf die Implementation des *XML Schema* Import für *XQuery* eingegangen. Der Import ist nur eine kleine Komponente des *Pathfinder XQuery-Compiler* [4], die wie das ganze *Pathfinder*-Projekt in *C* implementiert ist. Er wird mit der Funktion `PFschema_import`, deren Argument der *URI* des zu importierenden *XML Schema* ist, aufgerufen. Die Implementation setzt die Konzepte um, die in den vorherigen Abschnitten beschrieben wurden. Von `PFschema_import` werden deswegen zwei *SAX-callback*-Funktionen (`start element event` und `end`

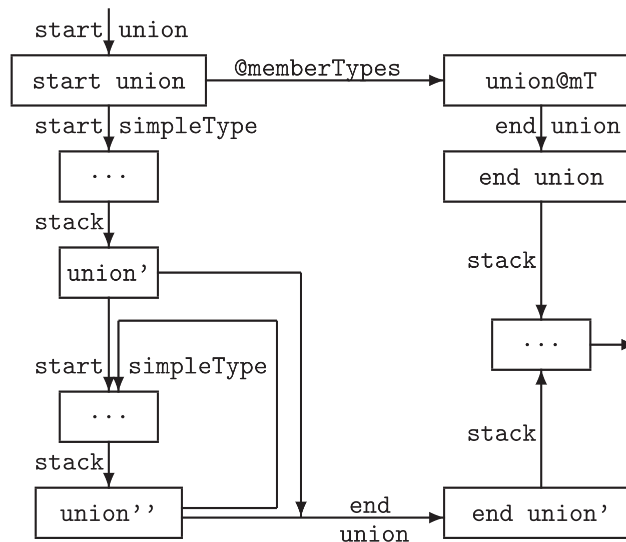


Abbildung 10: *Stack Automat* für `union`

`element event`) registriert, die *stacks* initialisiert und ein *SAX-Parser* mit der Adresse des *Schema* gestartet. Der weitere Ablauf des *Schema* Imports wird allein durch die *SAX-events* und die in Abschnitt 7 vorgestellte *State Machine* gesteuert.

## 8.1 *stacks*

In der Implementation des *XML Schema* Imports gibt es vier verschiedene *stacks*, die die Abbildung unterstützen:

- `state stack`
  Der `state stack` speichert die Zwischenzustände, die bei jedem `start-event` auf den `state stack` gelegt und bei jedem `end-event` wieder gelöscht werden. Nach dem Löschen wird außerdem der Zwischenzustand des umschließenden Zustandes, welcher der neue oberste Zustand des `state stack` ist, aktiviert und dessen Aktionen ausgeführt (siehe z.B. *Mini* `group` in Abschnitt 7.2.3). Damit werden zusätzlich die Unklarheiten, die durch ein `end-event` einer *globalen* bzw. *lokalen Schema*-Komponente entstehen, beseitigt.

- `attribute stack`
  Der `attribute stack` wird benötigt, da die Abbildungsregeln erst bei den `end-event`-Zuständen ausgeführt werden, die Attribute allerdings mit dem `start-event` als Parameter geliefert werden.

- `namespace stack`
  *XML* erlaubt in jedem Element die Definition von *Namespaces* mit Hilfe des Konstrukts `xmlns:foo="fooURI"`. Der `namespace stack` sorgt dafür, dass die definierten *Namespaces-prefixes* an der richtigen Stelle verwendbar und die *Namespace-URIs* somit für die Erstellung von *QNames* verfügbar sind.

- `type stack`
  Der `type stack` ist für das Speichern der Teil-Typen verantwortlich. Mit seiner Hilfe werden die beinhalteten Teil-Typen einer *Schema*-Komponente solange gespeichert, bis das `end-event` der Komponente verursacht, dass die

Teil-Typen vom `type stack` geholt und zu einem neuen Typ verarbeitet werden.

## 8.2 *SAX-callback functions*

Es gibt zwei *SAX-callback functions*, welche die einzelnen *events* anhand des *callback* und ihres Namens erkennen und dadurch die *State Table* steuern.

### 8.2.1 `start element event`

Jedes Mal, wenn ein `start element event` ausgelöst wird, werden zuerst die Attribute auf den `attribute stack` und die *Namespace*-Definitionen auf den `namespace stack` gelegt. Danach wird anhand des Namens des Elements und der *State Table* der neue aktuelle Zustand ermittelt und die daran gebundenen Aktionen ausgelöst. Nach den Aktionen wird der Wert der `stack`-Spalte des aktuellen Zustandes auf den `state stack` gelegt.

### 8.2.2 `end element event`

Beim Auftreten eines `end element event` wird mit Hilfe des Namens der neue aktuelle Zustand ermittelt und die Attribute werden vom `attribute stack` geholt. Danach werden die Aktionen (Abbildungsregeln) ausgelöst, welche die Teil-Typen vom `type stack` holen, die Attribute verarbeiten und in einem neuen Typ kombinieren. Wenn alle Aktionen des `end-event` Zustands ausgeführt sind, werden die *Namespace*-Definitionen vom `namespace stack` gelöscht. Zuletzt wird die umschließende Komponente anhand des `state stack` ermittelt, dessen Zwischenzustand aktiviert und die entsprechenden Aktionen ausgelöst.

## 8.3 *State Table*

Die Konzepte der Zustandstabelle wurden schon in Abschnitt 7 beschrieben. Die Einträge bestehen aus Zahlen (`int`). Die Beschriftung der Zustände ist so gewählt, dass ein Zustand, der einen Teil-Typ verarbeitet, mit einem ( `'` ) und ein Zustand, der zwei Teil-Typen verarbeitet, mit ( `''` ) markiert ist. Den `start-event` Zuständen ist ein `start`, den `end-event` Zuständen ein `end` und den Zwischenzuständen ein `between` vorangestellt.

Beispiel 8.1 zeigt den Übergang eines `state` anhand des `event` in einen neuen `state`. Dabei wird der Name des Elements (`tagn`) mit Hilfe von `getName` in einen *QName* übersetzt und die Spalte in der *State Table* durch die Funktion `get_start_event` aus dem *QName* ermittelt.

**Beispiel 8.1 (Übergang in neuen Zustand):**
```
event = get_start_event (getName (tagn));
state = stateTable [state][event];
```

## 8.4 *State Table* Aktionen

An etwa die Hälfte der Zustände in der *State Table* sind Aktionen gebunden, die in den meisten Fällen die Abbildungsregeln ausführen. Ist an einen Zustand keine Aktion gebunden, so handelt es sich entweder um einen `start-event`-Zustand, um einen Zwischenzustand oder um eine Abbildungsregel, die nur den Teil-Typ rekursiv übersetzt. Im letzten Fall bleibt der Teil-Typ auf dem `type stack` und die Abbildung wird somit automatisch ausgeführt. (siehe Beispiel 8.2).

Die Aktionen eines `end element event` tragen den neu konstruierten Typ eines *globalen Schema*-Element in die Typumgebung ein. Für *lokale Schema*-Elemente dagegen wird der neue Typ als neuer Teil-Typ auf den `type stack` gelegt

**Beispiel 8.2 (`local simpleType` mit Teil-Typ (Regel 5.48)):**

$$\begin{array}{c} \llbracket \ \texttt{<simpleType> type </simpleType>} \ \rrbracket \\ \Downarrow \\ \llbracket \ \texttt{type} \ \rrbracket \end{array}$$

Die meisten Aktionen bzw. Regeln sind relativ einfach aufgebaut. In Beispiel 8.3 werden z.B. nur die beiden Teil-Typen vom `type stack` geholt und als Sequenz wieder auf den `type stack` gelegt.

**Beispiel 8.3 (Aktion: `local complexType` mit Teil-Typ und Attributen):**

```
static void combine_atts_partial_type (PFarray_t *ignored)
{
   PFty_t partial_type, atts_partial_type;

   atts_partial_type = popType ();
   partial_type = popType ();

   pushType (PFty_seq (atts_partial_type, partial_type));
}
```

In einigen Zuständen liegt die Wahl der Abbildungsregel an den Attributen (`name`, `type`, `ref`). In diesen Fällen müssen in einem Zustand mehrere Abbildungsregeln von einer Aktion behandelt werden. In Beispiel 8.4 werden drei Regeln der `local elements` ( 5.4, 5.5 und 5.7) in einer Aktion unterschieden.

**Beispiel 8.4 (Aktion: `local element`):**

```
static void end_local_element (PFarray_t *atts)
{
   PFty_t result;
   char *name, *type, *ref;

   if ((name = findAttribute (atts, "name") )) {
      if ((type = findAttribute (atts, "type") ))
         result = PFty_elem (create_Qname (name),
                             PFty_named (create_Qname (type)));
      else
         result = PFty_elem (create_Qname (name), PFty_xs_anyType ());
   } else if ((ref = findAttribute (atts, "ref"))) {
      result = PFty_named_elem (create_Qname (ref));
   } else
      PFoops (OOPS_SCHEMAIMPORT,
              "name & type or ref in local element expected");

   result = occurrences (atts, result);
   pushType (result);
}
```

Außer den Aktionen, welche die Abbildungsregeln ausführen, gibt es noch drei Arten von Sonderfällen. Der erste tritt im `start schema` Zustand auf und interpretiert das Attribut `targetNamespace`. Der zweite Sonderfall verursacht einen Zustandswechsel aufgrund eines Attributs (z.B. in Zustand `start union` durch das

Attribut `memberTypes`). Die dritte Ausnahme ist die im *Mini*-Beispiel schon angesprochene Veränderung des `state stack`. (siehe Zustand `zw. group 1`).

**Beispiel 8.5 (Aktion:** `start union`**):**

```
static void start_list (PFarray_t *atts)
{
   /* if "itemType" goto LIST */
   if (findAttribute (atts, "itemType"))
      /* changes the actual state, to ignore any partial_types */
      state = LIST;
}
```

# 9 Zusammenfassung

Das Ziel dieser Arbeit war die Umsetzung eines, von anderer Software (wie z.B. *libxml Schema Support*) unabhängigen, *XML Schema* Import für *XQuery*. In der hier vorgestellten Lösung bestehen außer *XML* und *SAX* keine Abhängigkeiten, was diesen *XML Schema* Import losgelöst vom verwendeten *XQuery-Compiler* macht.

Mit der allein durch *SAX-events* gesteuerten *State Table* ist eine einfache Schnittstelle zwischen *SAX* und den Abbildungsregeln gelungen. In ihr stecken sämtliche Information über die Verarbeitung eines *XML Schema* im Bezug auf einen Import für *XQuery*. Die *State Table* geht auf alle Eventualitäten eines gültigen *Schema* ein und senkt somit maßgeblich die Komplexität des Abbildungsprozesses. Die Abbildungsregeln behandeln sämtliche Fälle aller *Schema*-Komponenten, die nach *XQuery* übersetzt werden können. Damit bietet diese Arbeit gegenüber der *XQuery Formal Semantics* eine größere Genauigkeit bezüglich der Anzahl der möglichen Abbildungen und greift außerdem dem Abbildungsprozess des *W3C* voraus.

Hiermit ist die erste vollständige und zugleich effiziente Umsetzung eines *XML Schema* Import für *XQuery* entstanden, die Anreiz schaffen soll, auch in anderen *XQuery-Compiler* das *XML Schema* Import Feature zu integrieren.

# Literatur

[1] *XML*
http://www.w3.org/XML/

[2] *XQuery*
http://www.w3.org/XML/Query
http://www.w3.org/TR/xquery/
*XQuery* Tutorial
http://www.datadirect.com/news/whatsnew/xquerybook.asp

[3] *XQuery*-Typen (*Formal Semantics*)
http://www.w3.org/TR/xquery-semantics/#sec_types

[4] *Pathfinder XQuery-Compiler*
http://www.inf.uni-konstanz.de/dbis/research/pathfinder/

[5] *XML Schema*
http://www.w3.org/XML/Schema
Part 0: Primer
http://www.w3.org/TR/xmlschema-0/
Part 1: Structures
http://www.w3.org/TR/xmlschema-1/
Part 2: Datatypes
http://www.w3.org/TR/xmlschema-2/

[6] *SAX*
http://www.saxproject.org/

[7] *SGML*
http://www.w3.org/MarkUp/SGML/

[8] *XHTML*
http://www.w3.org/TR/xhtml1/

[9] *XSL*
http://www.w3.org/TR/xsl/

[10] *XPath*
http://www.w3.org/TR/xpath
http://www.w3.org/TR/xpath20/

[11] *QNames* und *Namespaces*
http://www.w3.org/TR/REC-xml-names/

[12] *Symbol Spaces*
http://www.w3.org/TR/xmlschema-1/#concepts-nameSymbolSpaces
http://www.w3.org/TR/xmlschema-formal/#section-structures-names

[13] Abbildung der *XQuery Formal Semantics*
http://www.w3.org/TR/xquery-semantics/#sec_importing_schema

# A *State Table*

| | START-EVENT | | | | | | | | | | | | | | | | | | END-EVENT | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 all | 1 any | 2 anyAttribute | 3 attribute | 4 attributeGroup | 5 choice | 6 complexContent | 7 complexType | 8 element | 9 extension | 10 group | 11 list | 12 restriction | 13 schema | 14 sequence | 15 simpleContent | 16 simpleType | 17 union | 18 all | 19 any | 20 anyAttribute | 21 attribute | 22 attributeGroup | 23 choice | 24 complexContent | 25 complexType | 26 element | 27 extension | 28 group | 29 list | 30 restriction | 31 schema | 32 sequence | 33 simpleContent | 34 simpleType | 35 union | 36 stack |
| START 0 | | | | | | | | | | | | | | 1 | | | | | | | | | | | | | | | | | | | | | | | 0 |
| start schema 1 | | | | 97 | 94 | | | 12 | 4 | | 60 | | | | | | 22 | | | | | | | | | | | | | | | 3 | | | | | 2 |
| between schema' 2 | | | | 97 | 94 | | | 12 | 4 | | 60 | | | | | | 22 | | | | | | | | | | | | | | | 3 | | | | | -1 |
| end schema' 3 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 0 |
| start global element 4 | | | | | | | | 17 | | | | | | | | | 25 | | | | | | | | | | 6 | | | | | | | | | | 5 |
| between global element' 5 | | | | | | | | | | | | | | | | | | | | | | | | | | | 7 | | | | | | | | | | -1 |
| end global element 6 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| end global element' 7 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| start local element 8 | | | | | | | | 17 | | | | | | | | | 25 | | | | | | | | | | 10 | | | | | | | | | | 9 |
| between local element' 9 | | | | | | | | | | | | | | | | | | | | | | | | | | | 11 | | | | | | | | | | -1 |
| end local element 10 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| end local element' 11 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| start global complexType 12 | 66 | | 101 | 103 | 107 | 71 | 31 | | | | 64 | | | | 76 | 28 | | | | | | | | | | 14 | | | | | | | | | | | 13 |
| between global complexType' 13 | | | 119 | 121 | 125 | | | | | | | | | | | | | | | | | | | | | 15 | | | | | | | | | | | -1 |
| end global complexType 14 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| end global complexType' 15 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| end global complexType" 16 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| start local complexType 17 | 66 | | 137 | 139 | 143 | 71 | 31 | | | | 64 | | | | 76 | 28 | | | | | | | | | | 19 | | | | | | | | | | | 18 |
| between local complexType' 18 | | | 155 | 157 | 161 | | | | | | | | | | | | | | | | | | | | | 20 | | | | | | | | | | | -1 |
| end local complexType 19 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| end local complexType' 20 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| end local complexType" 21 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| start global simpleType 22 | | | | | | | | | | | | 81 | 43 | | | | | 86 | | | | | | | | | | | | | | | | | | | 23 |
| between global simpleType' 23 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 24 | | -1 |
| end global simpleType' 24 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| start local simpleType 25 | | | | | | | | | | | | 81 | 43 | | | | | 86 | | | | | | | | | | | | | | | | | | | 26 |
| between local simpleType' 26 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 27 | | -1 |
| end local simpleType' 27 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| start simpleContent 28 | | | | | | | | | | 34 | | | 48 | | | | | | | | | | | | | | | | | | | | | | | | 29 |
| between simpleContent' 29 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 30 | | | -1 |
| end simpleContent' 30 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| start complexContent 31 | | | | | | | | | | 37 | | | 55 | | | | | | | | | | | | | | | | | | | | | | | | 32 |
| between complexContent' 32 | | | | | | | | | | | | | | | | | | | | | | | | | 33 | | | | | | | | | | | | -1 |
| end complexContent' 33 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| start extension (simple) 34 | | | 137 | 139 | 143 | | | | | | | | | | | | | | | | | | | | | | | 35 | | | | | | | | | 34 |
| end extension (simple) 35 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| end extension' (simple) 36 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| start extension (complex) 37 | 66 | | 101 | 103 | 107 | 71 | | | | | 64 | | | | 76 | | | | | | | | | | | | | 39 | | | | | | | | | 38 |
| between extension' (complex) 38 | | | 119 | 121 | 125 | | | | | | | | | | | | | | | | | | | | | | | 40 | | | | | | | | | -1 |
| end extension (complex) 39 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| end extension' (type) (complex) 40 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| end extension' (atts) (complex) 41 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| end extension" (complex) 42 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| start restriction (sT) 43 | | | | | | | | | | | | | | | | | 25 | | | | | | | | | | | | | | | | | | | | 44 |
| between restriction' (sT) 44 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 47 | | | | | | -1 |
| betw. restr. @base (sT) 45 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 46 | | | | | | 45 |
| end restriction (sT) 46 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| end restriction' (sT) 47 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| start restriction (simple) 48 | | | | | | | | | | | | | | | | | 25 | | | | | | | | | | | | | | | | | | | | 49 |
| between restriction' (simple) 49 | | | 137 | 139 | 143 | | | | | | | | | | | | | | | | | | | | | | | | | | 52 | | | | | | -1 |
| betw. restr. @base 50 | | | 155 | 157 | 161 | | | | | | | | | | | | | | | | | | | | | | | | | | 51 | | | | | | 50 |
| end restriction (simple) 51 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| end restriction' (type) (simple) 52 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| end restriction' (atts) (simple) 53 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| end restriction" (simple) 54 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| start restriction (complex) 55 | 66 | | 101 | 103 | 107 | 71 | | | | | 64 | | | | 76 | | | | | | | | | | | | | | | | 57 | | | | | | 56 |
| between restriction' (complex) 56 | | | 119 | 121 | 125 | | | | | | | | | | | | | | | | | | | | | | | | | | 58 | | | | | | -1 |
| end restriction (complex) 57 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| end restriction' (complex) 58 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| end restriction" (complex) 59 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| start global group 60 | 66 | | | | | 71 | | | | | | | | | 76 | | | | | | | | | | | | | | 62 | | | | | | | | 61 |
| between global group' 61 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 63 | | | | | | | | -1 |
| end global group 62 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| end global group' 63 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| start local group 64 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 65 | | | | | | | | 64 |
| end local group 65 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| start all 66 | | | | | | | | | 8 | | | | | | | | | | 69 | | | | | | | | | | | | | | | | | | 67 |
| between all' 67 | | | | | | | | | 8 | | | | | | | | | | 70 | | | | | | | | | | | | | | | | | | 68 |
| between all" 68 | | | | | | | | | 8 | | | | | | | | | | 70 | | | | | | | | | | | | | | | | | | -1 |
| end all 69 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| end all' 70 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| start choice 71 | | 92 | | | | 71 | | | 8 | | 64 | | | | 76 | | | | | | | | | 74 | | | | | | | | | | | | | 72 |
| between choice' 72 | | 92 | | | | 71 | | | 8 | | 64 | | | | 76 | | | | | | | | | 75 | | | | | | | | | | | | | 73 |
| between choice" 73 | | 92 | | | | 71 | | | 8 | | 64 | | | | 76 | | | | | | | | | 75 | | | | | | | | | | | | | -1 |
| end choice 74 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| end choice' 75 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |

| | | START-EVENT | | | | | | | | | | | | | | | | | | END-EVENT | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 | 36 |
| start sequence | 76 | | 92 | | | | 71 | | | 8 | | 64 | | | | 76 | | | | | | | | | | | | | | | | | | 79 | | | | 77 |
| between sequence' | 77 | | 92 | | | | 71 | | | 8 | | 64 | | | | 76 | | | | | | | | | | | | | | | | | | 80 | | | | 78 |
| between sequence" | 78 | | 92 | | | | 71 | | | 8 | | 64 | | | | 76 | | | | | | | | | | | | | | | | | | 80 | | | | -1 |
| end sequence | 79 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| end sequence' | 80 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| start list | 81 | | | | | | | | | | | | | | | | | 25 | | | | | | | | | | | | | | | | | | | | 82 |
| between list' | 82 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 85 | | | | | | | -1 |
| between list @itemType | 83 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 84 | | | | | | | 83 |
| end list | 84 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| end list' | 85 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| start union | 86 | | | | | | | | | | | | | | | | | 25 | | | | | | | | | | | | | | | | | | | | 87 |
| between union' | 87 | | | | | | | | | | | | | | | | | 25 | | | | | | | | | | | | | | | | | | | 91 | 88 |
| between union" | 88 | | | | | | | | | | | | | | | | | 25 | | | | | | | | | | | | | | | | | | | 91 | -1 |
| between union @memberTypes | 89 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 90 | 89 |
| end union | 90 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| end union' | 91 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| start any | 92 | | | | | | | | | | | | | | | | | | | | 93 | | | | | | | | | | | | | | | | | 92 |
| end any | 93 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| start global attributeGroup | 94 | | | 101 | 103 | 107 | | | | | | | | | | | | | | | | | | 95 | | | | | | | | | | | | | | 94 |
| end global attributeGroup | 95 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| end global attributeGroup' | 96 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| start global attribute | 97 | | | | | | | | | | | | | | | | | 25 | | | | | 99 | | | | | | | | | | | | | | | 98 |
| between global attribute' | 98 | | | | | | | | | | | | | | | | | | | | | | 100 | | | | | | | | | | | | | | | -1 |
| end global attribute | 99 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| end global attribute' | 100 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | -1 |
| start anyAttribute (1) | 101 | | | | | | | | | | | | | | | | | | | | | 102 | | | | | | | | | | | | | | | | 101 |
| end anyAttribute (1) | 102 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 109 |
| start local attribute (1) | 103 | | | | | | | | | | | | | | | | | 25 | | | | | 105 | | | | | | | | | | | | | | | 104 |
| between local attribute' (1) | 104 | | | | | | | | | | | | | | | | | | | | | | 106 | | | | | | | | | | | | | | | -1 |
| end local attribute (1) | 105 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 109 |
| end local attribute' (1) | 106 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 109 |
| start local attributeGroup (1) | 107 | | | | | | | | | | | | | | | | | | | | | | | 108 | | | | | | | | | | | | | | 107 |
| end local attributeGroup (1) | 108 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 109 |
| attribute seen' (1) | 109 | | | 110 | 112 | 116 | | | | | | | | | | | | | | | | | | 96 | | | 15 | | 41 | | | 58 | | | | | | 0 |
| start anyAttribute (1') | 110 | | | | | | | | | | | | | | | | | | | | | 111 | | | | | | | | | | | | | | | | 110 |
| end anyAttribute (1') | 111 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 118 |
| start local attribute (1') | 112 | | | | | | | | | | | | | | | | | 25 | | | | | 114 | | | | | | | | | | | | | | | 113 |
| between local attribute' (1') | 113 | | | | | | | | | | | | | | | | | | | | | | 115 | | | | | | | | | | | | | | | -1 |
| end local attribute (1') | 114 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 118 |
| end local attribute' (1') | 115 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 118 |
| start local attributeGroup (1') | 116 | | | | | | | | | | | | | | | | | | | | | | | 117 | | | | | | | | | | | | | | 116 |
| end local attributeGroup (1') | 117 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 118 |
| attribute seen" (1') | 118 | | | 110 | 112 | 116 | | | | | | | | | | | | | | | | | | 96 | | | 15 | | 41 | | | 58 | | | | | | 0 |
| start anyAttribute (2) | 119 | | | | | | | | | | | | | | | | | | | | | 120 | | | | | | | | | | | | | | | | 119 |
| end anyAttribute (2) | 120 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 127 |
| start local attribute (2) | 121 | | | | | | | | | | | | | | | | | 25 | | | | | 123 | | | | | | | | | | | | | | | 122 |
| between local attribute' (2) | 122 | | | | | | | | | | | | | | | | | | | | | | 124 | | | | | | | | | | | | | | | -1 |
| end local attribute (2) | 123 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 127 |
| end local attribute' (2) | 124 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 127 |
| start local attributeGroup (2) | 125 | | | | | | | | | | | | | | | | | | | | | | | 126 | | | | | | | | | | | | | | 125 |
| end local attributeGroup (2) | 126 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 127 |
| attribute seen' (2) | 127 | | | 128 | 130 | 134 | | | | | | | | | | | | | | | | | | | | | 16 | | 42 | | | 59 | | | | | | 0 |
| start anyAttribute (2') | 128 | | | | | | | | | | | | | | | | | | | | | 129 | | | | | | | | | | | | | | | | 128 |
| end anyAttribute (2') | 129 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 136 |
| start local attribute (2') | 130 | | | | | | | | | | | | | | | | | 25 | | | | | 132 | | | | | | | | | | | | | | | 131 |
| between local attribute' (2') | 131 | | | | | | | | | | | | | | | | | | | | | | 133 | | | | | | | | | | | | | | | -1 |
| end local attribute (2') | 132 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 136 |
| end local attribute' (2') | 133 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 136 |
| start local attributeGroup (2') | 134 | | | | | | | | | | | | | | | | | | | | | | | 135 | | | | | | | | | | | | | | 134 |
| end local attributeGroup (2') | 135 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 136 |
| attribute seen" (2') | 136 | | | 128 | 130 | 134 | | | | | | | | | | | | | | | | | | | | | 16 | | 42 | | | 59 | | | | | | 0 |
| start anyAttribute (3) | 137 | | | | | | | | | | | | | | | | | | | | | 138 | | | | | | | | | | | | | | | | 137 |
| end anyAttribute (3) | 138 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 145 |
| start local attribute (3) | 139 | | | | | | | | | | | | | | | | | 25 | | | | | 141 | | | | | | | | | | | | | | | 140 |
| between local attribute' (3) | 140 | | | | | | | | | | | | | | | | | | | | | | 142 | | | | | | | | | | | | | | | -1 |
| end local attribute (3) | 141 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 145 |
| end local attribute' (3) | 142 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 145 |
| start local attributeGroup (3) | 143 | | | | | | | | | | | | | | | | | | | | | | | 144 | | | | | | | | | | | | | | 143 |
| end local attributeGroup (3) | 144 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 145 |
| attribute seen' (3) | 145 | | | 146 | 148 | 152 | | | | | | | | | | | | | | | | | | | | | 20 | | 36 | | | 54 | | | | | | 0 |
| start anyAttribute (3') | 146 | | | | | | | | | | | | | | | | | | | | | 147 | | | | | | | | | | | | | | | | 146 |
| end anyAttribute (3') | 147 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 154 |
| start local attribute (3') | 148 | | | | | | | | | | | | | | | | | 25 | | | | | 150 | | | | | | | | | | | | | | | 149 |
| between local attribute' (3') | 149 | | | | | | | | | | | | | | | | | | | | | | 151 | | | | | | | | | | | | | | | -1 |
| end local attribute (3') | 150 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 154 |
| end local attribute' (3') | 151 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 154 |
| start local attributeGroup (3') | 152 | | | | | | | | | | | | | | | | | | | | | | | 153 | | | | | | | | | | | | | | 152 |
| end local attributeGroup (3') | 153 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 154 |
| attribute seen" (3') | 154 | | | 146 | 148 | 152 | | | | | | | | | | | | | | | | | | | | | 20 | | 36 | | | 54 | | | | | | 0 |
| start anyAttribute (4) | 155 | | | | | | | | | | | | | | | | | | | | | 156 | | | | | | | | | | | | | | | | 155 |
| end anyAttribute (4) | 156 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 163 |
| start local attribute (4) | 157 | | | | | | | | | | | | | | | | | 25 | | | | | 159 | | | | | | | | | | | | | | | 158 |
| between local attribute' (4) | 158 | | | | | | | | | | | | | | | | | | | | | | 160 | | | | | | | | | | | | | | | -1 |
| end local attribute (4) | 159 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 163 |
| end local attribute' (4) | 160 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 163 |
| start local attributeGroup (4) | 161 | | | | | | | | | | | | | | | | | | | | | | | 162 | | | | | | | | | | | | | | 161 |
| end local attributeGroup (4) | 162 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 163 |
| attribute seen' (4) | 163 | | | 164 | 166 | 170 | | | | | | | | | | | | | | | | | | | | | 21 | | | | | 53 | | | | | | 0 |
| start anyAttribute (4') | 164 | | | | | | | | | | | | | | | | | | | | | 165 | | | | | | | | | | | | | | | | 164 |
| end anyAttribute (4') | 165 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 172 |
| start local attribute (4') | 166 | | | | | | | | | | | | | | | | | 25 | | | | | 168 | | | | | | | | | | | | | | | 167 |
| between local attribute' (4') | 167 | | | | | | | | | | | | | | | | | | | | | | 169 | | | | | | | | | | | | | | | -1 |
| end local attribute (4') | 168 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 172 |
| end local attribute' (4') | 169 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 172 |
| start local attributeGroup (4') | 170 | | | | | | | | | | | | | | | | | | | | | | | 171 | | | | | | | | | | | | | | 170 |
| end local attributeGroup (4') | 171 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | 172 |
| attribute seen" (4') | 172 | | | 164 | 166 | 170 | | | | | | | | | | | | | | | | | | | | | 21 | | | | | 53 | | | | | | 0 |

# B Aktionen

| Zustand | | Aktionen |
|---|---|---|
| START | 0 | - |
| start schema | 1 | integriert das Attribut `targetNamespace` |
| between schema' | 2 | - |
| end schema' | 3 | - |
| start global element | 4 | - |
| between global element' | 5 | - |
| end global element | 6 | Abbildungsregel 5.1 oder Abbildungsregel 5.2 |
| end global element' | 7 | Abbildungsregel 5.3 |
| start local element | 8 | - |
| between local element' | 9 | - |
| end local element | 10 | Abbildungsregel 5.4, Abbildungsregel 5.5 oder Abbildungsregel 5.7 |
| end local element' | 11 | Abbildungsregel 5.6 |
| start global complexType | 12 | - |
| between global complexType' | 13 | - |
| end global complexType | 14 | Abbildungsregel 5.39 |
| end global complexType' | 15 | Abbildungsregel 5.40 oder Abbildungsregel 5.41 |
| end global complexType" | 16 | Abbildungsregel 5.42 |
| start local complexType | 17 | - |
| between local complexType' | 18 | - |
| end local complexType | 19 | Abbildungsregel 5.43 |
| end local complexType' | 20 | Abbildungsregel 5.44 oder Abbildungsregel 5.45 |
| end local complexType" | 21 | Abbildungsregel 5.46 |
| start global simpleType | 22 | - |
| between global simpleType' | 23 | - |
| end global simpleType' | 24 | Abbildungsregel 5.47 |
| start local simpleType | 25 | - |
| between local simpleType' | 26 | - |
| end local simpleType' | 27 | Abbildungsregel 5.48 |
| start simpleContent | 28 | - |
| between simpleContent' | 29 | - |
| end simpleContent' | 30 | Abbildungsregel 5.49 |
| start complexContent | 31 | - |
| between complexContent' | 32 | - |
| end complexContent' | 33 | Abbildungsregel 5.50 |
| start extension (simple) | 34 | - |
| end extension (simple) | 35 | Abbildungsregel 5.51 |
| end extension' (simple) | 36 | Abbildungsregel 5.52 |
| start extension (complex) | 37 | - |
| between extension' (complex) | 38 | - |
| end extension (complex) | 39 | Abbildungsregel 5.51 |
| end extension' (type) (complex) | 40 | Abbildungsregel 5.53 |
| end extension' (atts) (complex) | 41 | Abbildungsregel 5.54 |
| end extension" (complex) | 42 | Abbildungsregel 5.55 |
| start restriction (sT) | 43 | falls das Attribut `base` vorhanden ist wird in den Zustand 45 gewechselt |
| between restriction' (sT) | 44 | - |
| between restriction @base (sT) | 45 | - |
| end restriction (sT) | 46 | Abbildungsregel 5.56 |
| end restriction' (sT) | 47 | Abbildungsregel 5.57 |
| start restriction (simple) | 48 | falls das Attribut `base` vorhanden ist wird in den Zustand 50 gewechselt |
| between restriction' (simple) | 49 | - |
| betw restr @base (simple) | 50 | - |
| end restriction (simple) | 51 | Abbildungsregel 5.58 |
| end restriction' (type) (simple) | 52 | Abbildungsregel 5.59 |
| end restriction' (atts) (simple) | 53 | Abbildungsregel 5.61 |
| end restriction" (simple) | 54 | Abbildungsregel 5.60 |
| start restriction (complex) | 55 | - |
| between restriction' (complex) | 56 | - |
| end restriction (complex) | 57 | Abbildungsregel 5.62 |
| end restriction' (complex) | 58 | Abbildungsregel 5.63 oder Abbildungsregel 5.64 |
| end restriction" (complex) | 59 | Abbildungsregel 5.65 |
| start global group | 60 | - |
| between global group' | 61 | - |
| end global group | 62 | Abbildungsregel 5.8 |
| end global group' | 63 | Abbildungsregel 5.9 |
| start local group | 64 | - |
| end local group | 65 | Abbildungsregel 5.10 |
| start all | 66 | - |
| between all' | 67 | ändert die *stack*-Spitze des Zustand-*stack* auf den Inhalt der `stack`-Spalte |
| between all" | 68 | Abbildungsregel 5.13 |
| end all | 69 | Abbildungsregel 5.11 |
| end all' | 70 | Abbildungsregel 5.12 |
| start choice | 71 | - |
| between choice' | 72 | ändert die *stack*-Spitze des Zustand-*stack* auf den Inhalt der `stack`-Spalte |
| between choice" | 73 | Abbildungsregel 5.16 |
| end choice | 74 | Abbildungsregel 5.14 |
| end choice' | 75 | Abbildungsregel 5.15 |
| start sequence | 76 | - |
| between sequence' | 77 | ändert die *stack*-Spitze des Zustand-*stack* auf den Inhalt der `stack`-Spalte |
| between sequence" | 78 | Abbildungsregel 5.19 |
| end sequence | 79 | Abbildungsregel 5.17 |
| end sequence' | 80 | Abbildungsregel 5.18 |
| start list | 81 | falls das Attribut `itemType` vorhanden ist wird in den Zustand 83 gewechselt |
| between list' | 82 | - |
| between list @itemType | 83 | - |
| end list | 84 | Abbildungsregel 5.66 |
| end list' | 85 | Abbildungsregel 5.67 |
| start union | 86 | falls das Attribut `memberTypes` vorhanden ist wird in den Zustand 89 gewechselt |
| between union' | 87 | ändert die *stack*-Spitze des Zustand-*stack* auf den Inhalt der `stack`-Spalte |
| between union" | 88 | Abbildungsregel 5.71 |
| between union @memberTypes | 89 | - |
| end union | 90 | Abbildungsregel 5.68 oder Abbildungsregel 5.69 |
| end union' | 91 | Abbildungsregel 5.70 |
| start any | 92 | - |

| | | |
|---|---|---|
| end any | 93 | Abbildungsregel 5.20 |
| start global attributeGroup | 94 | - |
| end global attributeGroup | 95 | Abbildungsregel 5.34 |
| end global attributeGroup' | 96 | Abbildungsregel 5.36 |
| start global attribute | 97 | - |
| between global attribute' | 98 | - |
| end global attribute | 99 | Abbildungsregel 5.26 oder Abbildungsregel 5.27 |
| end global attribute' | 100 | Abbildungsregel 5.28 |
| start anyAttribute (1) | 101 | - |
| end anyAttribute (1) | 102 | Abbildungsregel 5.38 |
| start local attribute (1) | 103 | - |
| between local attribute' (1) | 104 | - |
| end local attribute (1) | 105 | Abbildungsregel 5.30, Abbildungsregel 5.31 oder Abbildungsregel 5.33 gefolgt von der Abbildungsregel 5.29 für das Attribut use |
| end local attribute' (1) | 106 | Abbildungsregel 5.32 gefolgt von der Abbildungsregel 5.29 für das Attribut use |
| start local attributeGroup (1) | 107 | - |
| end local attributeGroup (1) | 108 | Abbildungsregel 5.37 |
| attribute seen' (1) | 109 | - |
| start anyAttribute (1') | 110 | - |
| end anyAttribute (1') | 111 | Abbildungsregel 5.38 |
| start local attribute (1') | 112 | - |
| between local attribute' (1') | 113 | - |
| end local attribute (1') | 114 | Abbildungsregel 5.30, Abbildungsregel 5.31 oder Abbildungsregel 5.33 gefolgt von der Abbildungsregel 5.29 für das Attribut use |
| end local attribute' (1') | 115 | Abbildungsregel 5.32 gefolgt von der Abbildungsregel 5.29 für das Attribut use |
| start local attributeGroup (1') | 116 | - |
| end local attributeGroup (1') | 117 | Abbildungsregel 5.37 |
| attribute seen" (1') | 118 | Kombination der attributes nach Abbildungsregel 5.45 |
| start anyAttribute (2) | 119 | - |
| end anyAttribute (2) | 120 | Abbildungsregel 5.38 |
| start local attribute (2) | 121 | - |
| between local attribute' (2) | 122 | - |
| end local attribute (2) | 123 | Abbildungsregel 5.30, Abbildungsregel 5.31 oder Abbildungsregel 5.33 gefolgt von der Abbildungsregel 5.29 für das Attribut use |
| end local attribute' (2) | 124 | Abbildungsregel 5.32 gefolgt von der Abbildungsregel 5.29 für das Attribut use |
| start local attributeGroup (2) | 125 | - |
| end local attributeGroup (2) | 126 | Abbildungsregel 5.37 |
| attribute seen' (2) | 127 | - |
| start anyAttribute (2') | 128 | - |
| end anyAttribute (2') | 129 | Abbildungsregel 5.38 |
| start local attribute (2') | 130 | - |
| between local attribute' (2') | 131 | - |
| end local attribute (2') | 132 | Abbildungsregel 5.30, Abbildungsregel 5.31 oder Abbildungsregel 5.33 gefolgt von der Abbildungsregel 5.29 für das Attribut use |
| end local attribute' (2') | 133 | Abbildungsregel 5.32 gefolgt von der Abbildungsregel 5.29 für das Attribut use |
| start local attributeGroup (2') | 134 | - |
| end local attributeGroup (2') | 135 | Abbildungsregel 5.37 |
| attribute seen" (2') | 136 | Kombination der attributes nach Abbildungsregel 5.45 |
| start anyAttribute (3) | 137 | - |
| end anyAttribute (3) | 138 | Abbildungsregel 5.38 |
| start local attribute (3) | 139 | - |
| between local attribute' (3) | 140 | - |
| end local attribute (3) | 141 | Abbildungsregel 5.30, Abbildungsregel 5.31 oder Abbildungsregel 5.33 gefolgt von der Abbildungsregel 5.29 für das Attribut use |
| end local attribute' (3) | 142 | Abbildungsregel 5.32 gefolgt von der Abbildungsregel 5.29 für das Attribut use |
| start local attributeGroup (3) | 143 | - |
| end local attributeGroup (3) | 144 | Abbildungsregel 5.37 |
| attribute seen' (3) | 145 | - |
| start anyAttribute (3') | 146 | - |
| end anyAttribute (3') | 147 | Abbildungsregel 5.38 |
| start local attribute (3') | 148 | - |
| between local attribute' (3') | 149 | - |
| end local attribute (3') | 150 | Abbildungsregel 5.30, Abbildungsregel 5.31 oder Abbildungsregel 5.33 gefolgt von der Abbildungsregel 5.29 für das Attribut use |
| end local attribute' (3') | 151 | Abbildungsregel 5.32 gefolgt von der Abbildungsregel 5.29 für das Attribut use |
| start local attributeGroup (3') | 152 | - |
| end local attributeGroup (3') | 153 | Abbildungsregel 5.37 |
| attribute seen" (3') | 154 | Kombination der attributes nach Abbildungsregel 5.45 |
| start anyAttribute (4) | 155 | - |
| end anyAttribute (4) | 156 | Abbildungsregel 5.38 |
| start local attribute (4) | 157 | - |
| between local attribute' (4) | 158 | - |
| end local attribute (4) | 159 | Abbildungsregel 5.30, Abbildungsregel 5.31 oder Abbildungsregel 5.33 gefolgt von der Abbildungsregel 5.29 für das Attribut use |
| end local attribute' (4) | 160 | Abbildungsregel 5.32 gefolgt von der Abbildungsregel 5.29 für das Attribut use |
| start local attributeGroup (4) | 161 | - |
| end local attributeGroup (4) | 162 | Abbildungsregel 5.37 |
| attribute seen' (4) | 163 | - |
| start anyAttribute (4') | 164 | - |
| end anyAttribute (4') | 165 | Abbildungsregel 5.38 |
| start local attribute (4') | 166 | - |
| between local attribute' (4') | 167 | - |
| end local attribute (4') | 168 | Abbildungsregel 5.30, Abbildungsregel 5.31 oder Abbildungsregel 5.33 gefolgt von der Abbildungsregel 5.29 für das Attribut use |
| end local attribute' (4') | 169 | Abbildungsregel 5.32 gefolgt von der Abbildungsregel 5.29 für das Attribut use |
| start local attributeGroup (4') | 170 | - |
| end local attributeGroup (4') | 171 | Abbildungsregel 5.37 |
| attribute seen" (4') | 172 | Kombination der attributes nach Abbildungsregel 5.45 |

Die einzigen Abbildungsregeln, die nicht aufgeführt wurden sind die Regeln für `occurrences` (Regel 5.21 – Regel 5.25). Die Attribute `minOccurs` und `maxOccurs` werden in den Zuständen end local element (10), end local element' (11), end local group (65), end choice' (75), end sequence' (80) und end any (93) ausgewertet und eine der fünf `occurrence`-Regeln ausgeführt.

# C Weitere Änderungen – `substitutionGroup`

Es wäre möglich, das Attribut `substitutionGroup` auch zu übersetzen. Dies ist allerdings nicht mit dem Konzept der *State Table* bzw. der *State Machine* machbar. Dazu müsste während des Durchlaufs des *Schema* jedes `substitutionGroup`-Paar (Wert des `name`-Attribut, Wert des `substitutionGroup`-Attribut) gespeichert werden. Nachdem das *Parsing* abgeschlossen wäre, müsste jedes Paar verarbeitet werden, indem der von `substitutionGroup` referenzierte Typ in der *XQuery*-Typumgebung zu einer `choice` aus dem ursprünglichen Typ und einem `named type` des gespeicherten `name`-Attribut geändert werden würde. Diese Änderungen dürften erst nach dem *Parsing* vorgenommen werden, da es erlaubt ist, dass die referenzierten `elements` zu einem späteren Zeitpunkt als das `element` mit dem Attribut `substitutionGroup` im *Schema* definiert sind.

**Beispiel C.1 (Abbildung von `global elements` ohne `substitutionGroup`):**

```
⟦<?xml version="1.0"?>
 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="video" type="videoType">>
   <xsd:element name="movie" type="videoType"/>
 </xsd:schema>⟧
```

$$\Downarrow$$

"video" $\mapsto_{element}$ `elem "video" {` named$_{type}$ `"videoType" }`
"movie" $\mapsto_{element}$ `elem "movie" {` named$_{type}$ `"videoType" }`

In Beispiel C.1 würde das Attribut `substitutionGroup` den Eintrag von `"video"` in der Typumgebung mit einer `choice` um den `named type "movie"` erweitern (siehe Änderung von Beispiel C.1 nach Beispiel C.2).

**Beispiel C.2 (Abbildung von `global elements` mit `substitutionGroup`):**

```
⟦<?xml version="1.0"?>
 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="video" type="videoType"/>
   <xsd:element name="movie" type="videoType"
      substitutionGroup="video"/>
 </xsd:schema>⟧
```

$$\Downarrow$$

"video" $\mapsto_{element}$ `elem "video" {` named$_{type}$ `"videoType" }`
`|` named$_{type}$ `"movie"`
"movie" $\mapsto_{element}$ `elem "movie" {` named$_{type}$ `"videoType" }`