XML

Guido Moerkotte

Sept. 2011

Spezifikationen rund um XML

- XML, XML Base, XML Namespaces
- XML Schema
- XPath, XPointer, XLink
- XSL
- XML Query (XQuery)
- DOM, SAX
- SOAP, XML Signature, XHTML, WSDL, BioML, UDDI, ...

siehe: http://www.w3.org

XML: Beispiel 1

<auto>Dieses Auto kostet<preis>500</preis>Euro</auto>
<auto preis = "500"> Noch so eine alte Karre </auto>

Beachte: Anreichung des Textes durch Meta-Information (Schema-Information), die normalerweise (z.B. in Datenbanksystemen) herausfaktorisiert wird.

XML in der Vorlesung

- www.w3.org/TR/2000/REC-xml-20001006
- Lesen Sie die Spezifikation!
  Sekundärliteratur mag eine gute Einführung liefern, läßt aber meistens (immer?) wichtige Details außer acht.

XML: Beispiel 2

```
<?xml version="1.0"?>
<product barcode="23948373">
  <manufacturer>Verbatim</manufacturer>
  <name>DataLife MF 2HD</name>
  <quantity>10</quantity>
  <size>3.5''</size>
  <color>black</color>
  <description>Floppy Disk</description>
</product>
```

Konzepte

- Element, Attribut
- Entity
  - General Entity (< & > " ')
  - External Parsed General Entity
  - External Unparsed Entity
  - Parameter Entity
- Comment <!– TEXT –>
- Processing Instruction <? target text ?>
- CDATA Section <![CDATA[text]]>
- Document Type Definition (DTD)

DTD: Beispiel

```
<!ELEMENT person (name, profession*)>
<!ELEMENT name   (first_name, last_name>
<!ELEMENT first_name (#PCDATA)>
<!ELEMENT last_name  (#PCDATA)>
<!ELEMENT profession (#PCDATA)>
```

In Datei oder im Kopf des Dokumentes:

```
<!DOCTYPE WurzelElement SYSTEM "URL">
<!DOCTYPE WurzelElement [
 DTD
]>
```

Elementdeklarationen

- Sequenz (“,”)
- Auswahl (“|”)
- Wiederholung (?, +, *)
- Blätter: #PCDATA, EMPTY, ANY, ElementName
- Beliebige Klammerung

DTD: Beispiel

```
<!ELEMENT sku (#PCDATA)>
<!ATTLIST sku
 list_price              CDATA  #IMPLIED
 suggested_retail_price CDATA #IMPLIED
 actual_price            CDATA #IMPLIED
>
<!ATTLIST date month (Januar | Februar) #REQUIRED >
```

Attributdeklarationen

- Attributtypen: CDATA, NMTOKEN(S), Enumeration, ENTITY, ENTITIES, ID, IDREF(S), NOTATION
- Attributdefaults: #IMPLIED, #REQUIRED, #FIXED, Literal

DTD: Beispiel

```
<!ENTITY super "sldkfjl  sldkfj  slkdkljflks  slslslslsl">
in Dokument: &super;

<!ENTITY footer SYSTEM "http://www.lssoo.com/soos/soso.xml">

<!ENTITY myPic SYSTEM "http://www.slslso.com/soso/soso.jpg"
                NDATA jpeg>
<!NOTATION jpeg SYSTEM "image/jpeg">

<!ATTLIST image source ENTITY #REQUIRED>

in Dokument: <image source = "myPic"/>
```

DTD: Beispiel: Parameterentitäten

```
<!ENTITY % residential_content "address, footage, rooms">
<!ENTITY % rental_content      "rent">
<!ENTITY % purchase_content    "price">

<!ELEMENT apartment (%residential_content;, %rental_content;)>
<!ELEMENT house (%residential_content;, %purchase_content;)>
```

Grobklassifikation von Dokumenten

- dokumentzentriert
- datenzentriert
- Alternativ: strukturiert, unstrukturiert (,semistrukturiert(?!))

Zusammenfassung

- XML ist sehr flexibel
- XML-Spezifikation ist kurz und bündig.
- XML ist leicht zu verstehen, enthält aber einige wesentliche Details.
- XML-Spezifikation fehlen einige wichtige Dinge, die jetzt folgen.

XML Namespace

Wozu XML Namespace?

- Ein XML-Dokument kann von mehreren Anwendungen bearbeitet werden
- Jede Anwendung kann eigene Element- und Attributnamen erfinden
- wichtig: diese auseinanderzuhalten

Beispiel

3 * “title” in:

```
<section><title>Book-Signing Event</title>
<signing>
  <author title="Mr" name="Vikram Seth" />
  <book title="A Suitable Boy" price="$22.95" />
</signing>
<signing>
  <author title="Dr" name="Oliver Sacks" />
  <book title="The Island of the Color-Blind" price="$12.95"
</signing>
</section>
```

XML namespace: einige Definitionen

Ein *XML namespace* ist eine Kollektion von Namen, die durch eine URI identifiziert werden.
Zwei URI-Referenzen, die je einen XML namespace identifizieren sind *identisch*, genau dann wenn ihre Zeichenketten identisch sind.
Ein *qualifizierter Name* setzt sich durch zusammen aus einem *Namensraumpräfix* und, durch Doppelpunkt (":"), einem lokalen Teil.
Der Präfix wird auf eine URI abgebildet, die dann den Namensraum bestimmt.
Des weiteren gibt es:

- attributbasierte Möglichkeiten der Namensraumassoziation mit einem URI
- scoping Regeln
- default Regeln

Namensraumdeklaration

Namensraumdeklarationen erfolgen durch Angabe eines Attributs. Es gibt zwei Möglichkeiten:

- default namespace attribute: xmlns
- prefixed namespace attribute: xmlns:NCName

Beispiel:

```
<x xmlns:edi = "http://ecommerce.org/schema">
   <!-- Präfix edi wird an URI http://ecommerce.org/schema gebunden
   <!-- Scope: das Element x und dessen Inhalt gebunden -->
</x>
```

Nutzung qualifizierter Namen

Elementnamen und Attributnamen dürfen qualifizierte Namen sein.
Beispiel:

```
<x xmlns:edi = "http://ecommerce.org/schema">
   <edi:price units = "Euro">32.18</edi:price>
</x>

<x xmlns:edi = "http://ecommerce.org/schema">
   <lineItem edi:taxClass = "exempt">Baby food</lineItem>
</x>
```

Scoping

- Innerhalb eines Elementes können mehrere Namensräume deklariert werden.
- Der Gültigkeitsraum der Deklarationens sind das aktuelle Element plus alle Attribute und Unterelement, solange man nicht auf eine weitere Namensraumdeklaration stößt, die die aktuelle überschreibt.

default namespace

Ein *default namespace* wirkt auf das Element in dem es definiert ist (falls es keinen Namensraumpräfix hat), sowie auf alle weiteren Elemente mit keinem Namensraumpräfix, die in dem Element enthalten sind.
Falls für ein Element kein default Namensraum gültig ist, so wird dieses Element **keinem Namensraum** zugeordnet.
Man beachte: Der Gültigkeitsbereich von default namespaces umfaßt keine Attribute.
Ist der Wert von xmlns die leere Zeichenkette, so bewirkt dies, dass es keinen default namespace gibt.

Eindeutigkeit von Attributen

Innerhalb eines Element-Tags dürfen keine zwei Attribute existieren, die

- identischen Namen haben oder
- einen identischen lokalen Teil haben und Präfixe, die an denselben Namensraum gebunden sind.

Beispiele

```
<x xmlns:n1="http://www.w3.org"
   xmlns:n2="http://www.w3.org" >
  <bad a="1"     a="2" />
  <bad n1:a="1"  n2:a="2" />
</x>

x xmlns:n1="http://www.w3.org"
   xmlns="http://www.w3.org" >
  <good a="1"     b="2" />
  <good a="1"     n1:a="2" />
</x>
```

XML Base

Abkürzungsmechanismus für URIs um (relative) Adressierungen zu vereinfachen.
Angabe eines Basis-URIs geschieht mittels des *xml:base*-Attributs.
Beispiel:

```
<doc xml:base="http://example.org/today/"
     xmlns:xlink="http://www.w3.org/1999/xlink">
<head> <title>Virtual Library</title> </head>
<body>
 <paragraph>
 <link xlink:type="simple" xlink:href="new.xml">NEW</link>
 </paragraph>
 <paragraph>Check out the hot picks of the day!</paragraph>
  <olist xml:base="/hotpicks/">
  <item>
  <link xlink:type="simple" xlink:href="pick1.xml">Pick #1</link>
  </item>
  <item>
  <link xlink:type="simple" xlink:href="pick2.xml">Pick #2</link>
  </item>
</olist> </body> </doc>
```

URI Auflösung im Beispiel

- "NEW" resolves to the URI "http://example.org/today/new.xml"
- "Pick #1" resolves to the URI "http://example.org/hotpicks/pick1.xml"
- "Pick #2" resolves to the URI "http://example.org/hotpicks/pick2.xml"

XPath

- Der Zweck von XPath ist es einen Teil eines XML-Dokumentes zu adressieren.
- XPath hat sich ergeben, um Gemeinsamkeiten aus XSLT und XPointer herauszufaktorisieren.
- XPointer wird bei XLink benutzt, um einen Teil eines (externen) Dokuments zu referenzieren. XLink erlaubt verschiedene Möglichkeiten zur Verbindung von Dokumenten. Einfache Verbindungen wie Hyperlinks in HTML werden erheblich verallgemeinert.

Wir betrachten nur XPath Version 1.0

Datenmodell

- Dokument ist Baum.
- Es gibt sieben Knotentypen
  1. root node
  2. element node
  3. attribute node
  4. text node
  5. comment node
  6. processing instruction node
  7. namespace node

Achtung: root node $\neq$ root element (= document node)

Ausdrücke und Ergebnistypen

Jeder Ausdruck in XPath hat ein Ergebnis. diese kann sein:

- node-set (unordered without duplicates)
- boolean (true or false)
- number (a floating-point number)
- string (a sequence of UCS characters)

Kontext

Kontext besteht aus:

- a node (context node)
- a pair of non-zero positive integers (context position and context size)
- a set of variable bindings
- a function library
- set of namespace declarations in scope for the expression

Jeder XPath-Ausdruck wird relativ zu einem Kontext ausgewertet. Dabei bleiben die letzten drei Komponenten unverändert während sich die ersten beiden für Unterausdrücke ändern.

Location Path

Location Paths sind die wichtigste Komponente von XPath.

- child::para selects the para element children of the context node
- child::* selects all element children of the context node
- attribute::name selects the name attribute of the context node
- child::*/child::para selects all para grandchildren of the context node
- /child::doc/child::chapter[position()=5]/child::section[position()=2] selects the second section of the fifth chapter of the doc document element

Location Path

LocationPath	::=	RelativeLocationPath | AbsoluteLocationPath
AbsoluteLocationPath	::=	'/' RelativeLocationPath?
		| AbbreviatedAbsoluteLocationPath
RelativeLocationPath	::=	Step
		RelativeLocationPath '/' Step
		AbbreviatedRelativeLocationPath

Location Step

Ein Location Step besteht aus drei Teilen: eine Achse (axis), einem Knotentest (node test) und keinem oder mehreren Prädikaten.

Step	::=	AxisSpecifier NodeTest Predicate
		| AbbreviatedStep
AxisSpecifier	::=	AxisName '::'
		| AbbreviatedAxisSpecifier
NodeTest	::=	NameTest | NodeType '(' ')'
NameTest	::=	'*' | NCName':*' | QName
NodeType	::=	'comment' | 'text' | 'node'
		| 'processing-instruction'
Predicate	::=	'[' Expr ']'

Achsen

- child
- descendant
- parent
- ancestor
- following-sibling
- preceding-sibling
- following
- preceding
- attribute
- namespace
- self
- descendant-or-self
- ancestor-or-self

Note: The ancestor, descendant, following, preceding and self axes partition a document (ignoring attribute and namespace nodes): they do not overlap and together they contain all the nodes in the document.

Principle Node Type

Jede Achse hat einen *principal node type*:

- attribute axis: principle node type is attribute
- naemspace axis: principle node type is namespace
- other: principle node type is element

Node Test

- '*': falls Knoten vom principle node type
- QName: falls Knoten vom principle node type und expanded-name ist gleich dem expanded-name von QName
- NCName':*': falls Knoten vom principle node type dessen expanded name eine Namensraum-URI hat, die gleich der von NCName ist.

Falls der Knotentest ein Knotentyp ist, so muss der entsprechende Knoten vom angegebenen Knotentyp sein.

Document Order

There is an ordering, document order, defined on all the nodes in the document corresponding to the order in which the first character of the XML representation of each node occurs in the XML representation of the document after expansion of general entities.
Thus, the root node will be the first node. Element nodes occur before their children. Thus, document order orders element nodes in order of the occurrence of their start-tag in the XML (after expansion of entities).

- The attribute nodes and namespace nodes of an element occur before the children of the element.
- The namespace nodes are defined to occur before the attribute nodes.
- The relative order of namespace nodes is implementation-dependent.
- The relative order of attribute nodes is implementation-dependent.

Reverse document order is the reverse of document order.

Forward and backward axis

Eine Achse ist entweder eine Vorwärts- (forward) oder Rückwärtsachse (reverse axis).

- Eine Achse, die nur den Kontextknoten oder Knoten, die dem Kontextknoten in Dokumentordnung folgen enthält, ist einen Vorwärtsachse.
- Eine Achse, die nur den Kontextknoten oder Knoten, die dem Kontextknoten in Dokumentordnung vorausgehen enthält, ist eine Rückwärtsachse.

Damit kann zu jeder Achse eine Ordnung (Dokumentordnung oder reverse Dokumentordnung) assoziiert werden.

Predicate

Ein Prädikat filtert eine Knotenmenge (node-set) und erzeugt eine neue Knotenmenge. Für jeden Knoten in der Knotenmenge wird der Prädikatausdruck ausgewertet.
Der Prädikatsausdruck wird mit dem Knoten auf den es angewendet wird als Kontextknoten ausgewertet. Die Kontextgröße ist dabei die Kardinalität der ursprünglichen Knotenmenge. Die Kontextposition des Kontextknotens ist dabei seine Stellung innerhalb der ursprünglichen Knotenmenge gemäß der assozierten Ordnung der Achse des Step zu dem das Prädikat gehört.
Das Ergebnis der Auswertung des Prädikatausdrucks wird in einen boolschen Wert konvertiert. Falls das Ergebnis der Auswertung des Prädikates eine Zahl ergibt, wird diese zu *true* konvertiert, falls sie gleich der Kontextposition ist; ansonsten zu *false*.

Abkürzungen

AbbreviatedAbsoluteLocationPath	::=	'//' RelativeLocationPath
AbbreviatedRelativeLocationPath	::=	RelativeLocationPath '//'
AbbreviatedStep	::=	'.' | '..'
AbbreviatedAxisSpecifier	::=	'@'?

para	≡	child::para
*	≡	child::*
.//para	≡	self::node()/descendant-or-self::node()/child::para
../@lang	≡	parent::*/attribute::lang

Ausdrücke

Ausdrücke sind Vergleiche von Ausdrücken mit den üblichen Vergleichsoperatoren. Komplexe boolsche Ausdrücke können mit `and` und `or` gebildet werden. Klammern ist erlaubt. Auch LocationPaths sind Ausdrücke. Eine typische Operation ist dort die Vereinigung (|). Weiter können arithmetische Ausdrücke gebildet werden und Funktionsaufrufe. Für letztere ist noch eine Core Function Library von Funktionen definiert, die unterstützt werden müssen, u.a.:

- id(object)
- last(), position()
- count(node-set)
- string(object)
- contains(string,string)

Konvertierungen

Wichtig für die Auswertung von Ausdrücken sind die Konvertierungsvorschriften.
Konvertierungen finden in den meisten Fällen mittels Konvertierungsfunktionen statt:

- string(·)
- boolean(·)
- number(·)

Diese Funktionen sind in der Core Function Library definiert.

string

Konvertierung mittels `string(x)` falls x:

node-set : `string-value` des ersten Knotens von x in Dokumentordnung

number : Repräsentation der Zahl inklusive NaN, Infinity, etc.

boolean : *true* ergibt "true", *false* ergibt "false"

Andere als die vier Basistypen von XPath: Konvertierung typabhängig.

boolean

Konvertierung mittels `boolean(x)` falls x:

- node-set *true*, gdw. nicht leer
- number *true* falls nicht 0 oder NaN
- string *true* falls Länge ungleich Null.

Andere als die vier Basistypen von XPath: Konvertierung typabhängig.

number

Konvertierung mittels `number(x)` falls x:

string	x repräsentiert Zahl, dann zur nächsten IEEE 754 Zahl runden
boolean	*true* wird 1, *false* 0
node-set	`number(string(x))`

Andere als die vier Basistypen von XPath: Konvertierung typabhängig.

Vergleiche

Vergleiche auf Knotenmengen ($X \theta Y$):

- X, Y Knotenmengen: $\exists x \in X, y \in Y$ so daß deren `string-value` einem θ-Vergleich standhalten
- X Knotenmenge, Y number: $\exists x \in X$, so daß $\texttt{number}(\texttt{string-value}(x))\theta y$
- X Knotenmenge, Y string: $\exists x \in X$, so daß $\texttt{string-value}(x)\theta y$
- X Knotenmenge, Y boolean: $\texttt{boolean}(X)\theta y$

Plus symmetrische Fälle falls Y Knotenmenge.

Vergleiche

Falls weder X noch Y eine Knotenmenge ist und $\theta \in \{=, !=\}$, dann werden X und Y in einen gemeinsamen Typ konvertiert:

- X oder Y ist boolean: konvertierung beider nach boolean, sonst:
- X oder Y ist number: konvertierung beider nach number, sonst:
- konvertierung beider nach string

Für alle anderen Vergleichsfunktionen ($<=$, $<$, $>=$, $>$) werden beide Argumente in number konvertiert.

XPath 2.0: new features

- use XQuery 1.0 and XPath 2.0 data model
- new operators: `intersect`, `except`
- larger set of functions
- sequence expressions
  - ',': comma: concatenation
  - `to`: construct sequence from numeric range
  - `some`, `every`: quantification
- `for...return`: iteration
- `if` (e1) `then` e2 `else` e3
- `cast`

XPath 2.0: element test

- `element()`, `element(*)`: matches any single element node
- `element(N)`: matches element named `N`
- `element(N,T)`: matches element named `N` of type `T` and derived types
- `element(*,T)`: matches element of type `T` and derived types

Analog für Attribute (`attribute`)

XPath 2.0: incompatibilities with XPath 1.0

- arithmetic operator's argument is sequence with more than one node: error
- inequality on strings: string comparison
- arithmetic on strings: error
- '=': on child-only elements: error

(alte semantik mit XPath 1.0 compabitility mode)

XML Schema

- XML Schema erfasst mehr Semantik als DTDs
- XML Schema daher *die* Sprache für XBMS Schemadefinitionen
- Dokumente:
  - XML Schema Part 0: Primer (www.w3.org/TR/xmlschema-0)
  - XML Schema Part 1: Structures (www.w3.org/TR/xmlschema-1)
  - XML Schema Part 2: Datatypes (www.w3.org/TR/xmlschema-2)
- P. Walmsley: Definitive XML Schema (Prentice Hall)
- Harold, Means: XML in a Nutshell (O'Reilly)

(1) Dokument

```
<product effDate = "2001-04-02">
  <number>557</number>
  <size>10</size>
</product>
```

(1) Schema

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name = "product" type = "ProductType" />
  <xsd:complexType name = "ProductType">
    <xsd:sequence>
      <xsd:element name = "number" type = "xsd:integer"/>
      <xsd:element name = "size" type = "xsd:integer"/>
    </xsd:sequence>
    <xsd:attribute name = "effDate" type = "xsd:date"/>
  </xsd:complexType>
</xsd:schema>
```

(1) Schema-Organisation

- Jedes Schema hat als Wurzelelement ein xs:schema Element.
- Deklarationen, die direkte Kinder des Wurzelements sind heißen *global*.
- Elementdeklarationen, die globale Elemente sind, können als Dokumentelemente vorkommen
- Deklarationen, die nicht global sind, sind *lokal*.
- Elemente, die zu XML Schema gehören und als solche interpretiert werden sollen, müssen im Namespace http://www.w3.org/2001/XMLSchema sein.
- Wir deuten dies immer durch den Namespace xsd an.

(1.2) Warum Schemata?

- data validation
- contract between partners
- system documentation
- augmentation of data (e.g. default values)
- application information

(1.3) Schema Design: Ziele

Ziele:

- Accuracy and Precision
- Clarity
- Broad applicability
  (reusability, extensibility)

(2) XML Schema: Quick Tour

```
<product effDate = "2001-04-02">
  <number>557</number>
  <size>10</size>
</product>
```

(2) XML Schema: Quick Tour

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name = "product" type = "ProductType" />
  <xsd:complexType name = "ProductType">
    <xsd:sequence>
      <xsd:element name = "number" type = "xsd:integer"/>
      <xsd:element name = "size" type = "SizeType"/>
    </xsd:sequence>
    <xsd:attribute name = "effDate" type = "xsd:date"/>
  </xsd:complexType>
  <xsd:simpleType name = "SizeType">
    <xsd:restriction base = "xsd:integer">
      <xsd:minInclusive value = "2"/>
      <xsd:maxInclusive value = "18"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

(2.2) Declarations vs. Definitions

- Deklaration: elements, attributes, notations
  Deklaration nur für solche genutzt, von denen es Instanzen im zu validierenden Dokument geben kann.
- Definition: data types, model groups, attribute groups, identity constraints
  Definition nur für solche Komponenten, welche Schemaintern sind.

(2.2) Global vs. Local Components

- Global: direkt unter `xsd:schema`
  kann von anderen Komponenten wiederverwendet (referenziert werden)
- Lokal: nicht direkt unter `xsd:schema`
  kann nicht von anderen Komponenten wiederverwendet (referenziert werden)

(2.2) Named vs. Unnamed Components

- Nur benannte Komponenten können wiederverwendet werden
- Globale Komponenten müssen immer benannt sein
- Benannte Komponenten (außer Elementen, Attributen) müssen immer global sein

(2.2) Komponenten eines Schemas

Component	can be named?	can be unnamed?	can be global?	can
element	yes	no	yes	
attribute	yes	no	yes	
simple type	yes	yes	yes	
complex type	yes	yes	yes	
notation	yes	no	yes	
named model group	yes	no	yes	
attribute group	yes	no	yes	
identity constraint	yes	no	no	

(2.3) Elements and Attributes

- Elemente und Attribute sind die wesentlichen Bestandteile von XML-Dokumenten
- Sie können in einem Schema deklariert werden.
- Sie haben einen Namen (Elemente: tag name)
- Sie haben einen Typ
- Zwischen Name und Typ besteht eine n:m-Beziehung
- Bei globalen Elementen/Attributen: gleicher Name bei unterschiedlichem Typ geht natürlich nur in verschiedenen Schemata.

(2.4) Datentypen

- simple types
  - simple types: atomic type, list, union
  - Attribute haben immer simple type
  - Elemente können einen simple type haben.
- complex types
  - nur für Elemente
  - Sobald ein Element Attribute hat, hat es einen complex type

(2.4) Type definition hierarchy

Zwei Möglichkeiten Untertypen zu bilden:

- Restriction
  (siehe obiges Beispiel)
- Extension
  (füge neue Komponenten (z.B. Kindelemente) hinzu)

(2.5) Simple Types: built-in simple types

Category	built-in types
strings and names	string, normalizedString, token, Name, NCName, QName, lang
numeric	float, double, decimal, integer, long, int, short, byte, positiveInteger, nonPositiveInteger, negativeInteger, nonNegativeInteger, unsignedLong, unsignedInt, unsignedShort, unsignedByte
date and time	duration, dateTime, date, time, gYear, gYearMonth, gMonth, gMonthDay, gDay
legacy types	ID, IDREF, IDREFS, ENTITY, ENTITIES, NMTOKEN, NMTOKENS, NOTATION
other	boolean, hexBinary, base64Binary, anyUri

(2.5) Simple Types: restriction

Wende facets an, schränkt Wertebereich ein

Category	Facets
bounds	minInclusive, maxInclusive, minExclusive, maxExclusive
length	length, minLength, maxLength
precision	totalDigits, fractionDigits
enumerated values	enumeration
pattern matching	pattern
whitespace processing	whiteSpace

(2.5) Simple Types: List and union types

- nur listen von atomaren Werten
- Listenwert ist whitespace-separierte liste von atomaren Werten

```
<sizes>10 large 2</sizes>
```

- union: Wert eines Union ist entweder Vereinigung von
  - atomaren Werten oder
  - von Listen

(2.6) Complex Types: Content Types

Content: ist alles zwischen den TAGs eines Elements
vier Content-Typen:

- simple
- element (-only)
- mixed
- empty

ACHTUNG: Bezieht sich nur auf den INHALT, also nicht auf Attribute und deren An- bzw. Abwesenheit.

(2.6) Complex Types: Beispiele

```
<size system = "US-Dress">10</size>

<product>
  <number>557</number>
  <size>10</size>
</product>

<letter>Dear <custName>Priscilla</custName>...</letter>

<color value = "blue"/>
```

(2.6) Complex Types: Content Model

Content Model:

- Welche Kindknoten in welcher Reihenfolge mit welcher Multiplizität vorkommen

Ähnlich zu DTD, können beliebig geschachtelt werden:

- `sequence`
- `choice`
- `all` groups
  alle Kinder müssen 0 oder 1 mal vorkommen, wobei die Reihenfolge beliebig ist

zusätzlich:

- `any` beliebiges Content-Model
- `anyAttributes` beliebige Attribute möglich

(2.6) Complex Types: Beispiel

```
<xsd:complexType name = "ProductType">
  <xsd:sequence>
    <xsd:element name = "number" type = "xsd:integer"/>
    <xsd:choice minOccurs = "0" maxOccurs = "3">
      <xsd:element name = "size" type = "SizeType"/>
      <xsd:element name = "color" type = "ColorType"/>
    </xsd:choice>
    <xsd:any namespace = "##other"/>
  </xsd:sequence>
</xsd:complexType>
```

(2.6) Complex Types: Deriving complex types

Möglichkeiten:

- restriction
  Einschränkung des Wertbereichs
- extension
  mehr (Kind-) Elemente und/oder Attribute

(2.6) Complex Types: Extension: Beispiel

```
<xsd:complexType name = "ShirtType">
  <xsd:complexContent>
    <xsd:extension base = "ProductType">
    <xsd:sequence>
      <xsd:element name = "color" type = "ColorType"/>
    </xsd:sequence>
    <xsd:attribute name = "id"
                   type = "xsd:ID"
                   use = "required"/>
  </xsd:complexContent>
</xsd:complexType>
```

Content-Modelle werden hintereinandergehängt (sequence).

(2.7) XML Schema and Namespaces

1. `schema`, `element`, `complexType`, ... sind im XML Schema Namespace definiert.
2. target namespace: gibt an, welche Elemente in welchen Namespace wie validiert werden
   - Ein Schemadokument hat höchstens einen target namespace
   - Ein Schema kann aus mehreren Schemadokumenten bestehen, die jeweils einen anderen target namespace haben

(2.7) Beispielschema

```
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema"
            targetNamespace = "http://example.org/prod"
            xmlns:prod="http://example.org/prod">

<xsd:element name = "product" type = "prod:productType"/>
```

```
  <xsd:complexType name = "ProductType">
    <xsd:sequence>
      <xsd:element name = "number" type = "xsd:integer"/>
      <xsd:element name = "size" type = "prod:SizeType"/>
    </xsd:sequence>
    <xsd:attribute name = "effDate" type = "xsd:date"/>
  </xsd:complexType>

  <xsd:simpleType name = "SizeType">
    <xsd:restriction base = "xsd:integer">
      <xsd:minInclusive value = "2"/>
      <xsd:maxInclusive value = "18"/>
    </xsd:restriction>
  </xsd:simpleType>

</xsd:schema>
```

(2.7) Beispielinstanz mit Namensraum

```
<prod:product xmlns:prod = "http://example.org/prod"
              effDate = "2001-04-02">
  <number>557</number>
  <size>10</size>
</prod:product>
```

(2.7) Diskussion

- Instanz valid: Instance namespace must match target namespace
- Beachte: `number`, `size`, `effDate` haben keinen Namespace-Prefix
  - Default: Lokal deklarierte Komponenten nehmen nicht den target namespace an
  - Überschreiben des defaults mit `elementFormDefault`, `attributeFormDefault`

(2.8) Schema composition

Schemata können aus mehreren Schemadokumenten zusammengesetzt werden

- `xsd:include` inkludiert Schemadokument mir gleichem target namespace
- `xsd:import` importiert Schemadokument mit anderem taget namespace

Dies sind nicht die einzigen Möglichkeiten!
Der Standard läßt aber auch andere Möglichkeiten des Zusammensetzens von Schemata aus Schemadokumenten zu.

(2.8) Beispiel

```
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema"
            xmlns = "http://example.org/ord"
            targetNamespace = "http://example.org/ord">

<xsd:include schemaLocation = "moreOrderInfo.xsd"/>

<xsd:import namespace = "http://example.org/prod"
            schemaLocation = "productInfo.xsd"/>

<!--...-->
</xsd:schema>
```

(2.9) Instances and Schemas

- namespace für XML Schema Instanzen:

  `http://www.w3.org/2001/XMLSchema-instance`

  enthält: `type`, `nil`, `schemaLocation`, `noNamespaceSchemaLocation`
- Per Konvention wird der Präfix `xsi` verwendet
- Der Standard ist absichtlicherweise vage, über die Zuordnung von Schemata zu Instanzen.

(2.9) Assoziation von Schemata und Dokumenten

Möglichkeiten:

1. xsi:schemaLocation
   Dieses Attribut enthält eine Liste von Paaren Namespace, Schema-URI. Elemente in Namespace werden mit dem entsprechenden Schema validiert.
2. noNamespaceSchemaLocation
   Dieses Attribut gibt die URI eines Schemas an, das für Elemente in keinem Namespace zur Validierung benutzt wird.
3. Ein validierender Parser könnte versuchen Schema herauszufinden, z.B. durch Namespaces, die ihm bekannt vorkommen.
4. Dem validierenden Parser wird direkt beim Aufruf ein Schema mitgegeben, gegen das ein Dokument validiert werden soll.
5. built-in schemata, internal catalogs, ...

(2.9) Beispiel für Explizite Assoziation

```
<prod:product
    xmlns:prod = "http://example.org/prod"
    xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation = "http://example.org/prod prod.xsd"
    effDate = "2001-04-02">
  <number>557</number>
  <size>10</size>
</prod:product>
```

(2.10) Annotations

- Als erstes Element in einem XML Schema Element kann ein xsd:annotation Element vorkommen.
- Dessen Inhalt dient der Dokumentation:
  - xsd:documentation Elemente sind für menschenlesbare Dokumentation
  - xsd:appinfo Elemente sind für maschinenlesbare Dokumentation

(2.10) Beispiel

```
<!--...-->
<xsd:element name="fullName" type="xsd:string">
 <xsd:annotation>
  <xsd:documentation>
     contains first and last name
  </xsd:documentation>
  <xsd:app-info>
    <help-text>Enter person's full name.</help-text>
  </xsd:app-info>
 </xsd:annotation>
</xsd:element>
<!--...-->
```

(2.11) XML Schema: advanced features

1. Reusable groups
   Gruppen von Elementen oder Attributen können zusammengefasst definiert werden, um danach an mehreren Stellen einfach benutzt zu werden.
   - named model groups: Fragmente eines Inhaltsmodells
   - named attribute groups: Menge von Attributdeklarationen
2. Identity constraints
   - key constraints: eindeutige Identifizierung von Knoten
   - references: Einhaltung der referenziellen Integrität
3. Substitution groups
   - substitution groups definieren Elemente, die für andere Elemente substituiert werden dürfen
   - Zweck:
     - Alternativenbehandlung
     - einfaches Hinzufügen von Alternativen ohne originale Deklarationen ändern zu müssen
4. Redefinition zur Umdefinition. Zweck:
   - Erweiterung/Einschränkung eines Schemas
   - Modifikationen über die Zeit hinweg (Schemaevolution)

(3) XML Schemata und Namensräume

- Namespaces und Schemata stehen in einer n:m-Beziehung zueinander
- Ein Namespace kann durch 0 oder mehrere Schemata beschrieben werden
- Ein Schema kann Namen für einen oder mehrere Namespaces deklarieren
- Es kann Schemata ohne Namespace geben
- Schemata können aus mehreren Schemadokumenten bestehen, jedes mit einem eigenen target namespace

(3.3) XML Schema Namensraum

- enthält `schema`, `element`, `simpleType`, ...
- Kein Attribute im XML Schema Namespace ist global definiert. Daher: diese haben im Schemadokument keinen Präfix:

```
<xsd:element name = "number" type = "xsd:integer"/>
```

  (s. `name` und `type`)
- built-in simple types sind aber im XML Schema Namespace (s. obiges Beispiel) (dies ist der target namespace des Schemas von XML Schema)

(3.3) XML Schema Instance Namespace

- namespace für XML Schema Instanzen:

  `http://www.w3.org/2001/XMLSchema-instance`

- Per Konvention wird der Präfix `xsi` verwendet
- Der XML Schema Instance Namespace enthält die Attribute:
  - `type`
  - `nil`
  - `schemaLocation`
  - `noNamespaceSchemaLocation`

(3.3) Alternativen zum Namespace-Handling in Schemadokumenten

1. Benutze Präfix für XML Schema Namespace (hier und im Buch)
2. Benutze Präfix für Target Namespace
3. Benutze Präfix für beides

(3.3) 1. Benutze Präfix für XML Schema Namespace

```
<xsd:schema xmlns:xsd = "http:www.w3.org/2001/XMLSchema"
  xmlns = "http://example.org/prod"
  targetNamespace = "http://example.org/prod">

<xsd:element name = "number" type = "xsd:integer"/>

<xsd:element name = "size" type = "SizeType"/>

<xsd:simpleType name = "SizeType">
  <!--...-->
</xsd:simpleType>

</xsd:schema>
```

(3.3) Falsche Benutzung

Falls kein target namespace angegeben ist, müssen XMLSchema Komponenten mit Präfix versehen werden:

```
<schema xmlns = "http://www.w3.org/2001/XMLSchema">
  <element name = "number" type = "integer"/>
  <element name = "size" type = "SizeType"/>
  <simpleType name = "SizeType">
    <!--...-->
  </simpleType>
</schema>
```

Problem: `size` referenziert `SizeType`, welches der Validator im XML Schema Namespace sucht (= default namespace), dort aber nicht findet!

(3.3) 2. Benutze Präfix für Target Namespace

```
<schema xmlns = "http://www.w3.org/2001/XMLSchema"
        xmlns:prod = "http://example.org/prod"
        targetNamespace = "http://example.org/prod">

  <element name = "number" type = "integer"/>

  <element name = "size" type = "prod:SizeType"/>

  <simpleType name = "SizeType>
    <!--...-->
  </simpleType>
</schema>
```

(3.3) 3. Benutze Präfix für beides

```
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema"
            xmlns:prod = "http://example.org/prod"
            targetNamespace = "http://example.org/prod">

 <xsd:element name = "number" type = "xsd:integer"/>

  <xsd:element name = "size" type = "prod:SizeType"/>

  <xsd:simpleType name = "SizeType>
    <!--...-->
  </simpleType>
</schema>
```

(4) Schema Composition

- Ein Schema kann aus mehreren Schemadokumenten bestehen.
- Beispiel: order, customer, product
- Warum Schema auf mehrere Dokumente verteilen?
  - Einfachere Wiederbenutzung
  - Einfachere Wartung
  - Reduzierte Gefahr von Namenskollisionen
  - Verbesserte Zugriffskontrolle

(4.1) Schema Composition

Möglichkeiten für Dekompositionskriterien:

- nach Gebiet
- allgemein/spezifisch
  generische Bestellungen/Rechnungen in einem Schemadokument,
  branchenspezifische Ausprägungen in jeweils einem anderen
- basic/advance
  Pflichtanteile in einem Dokument, optionale Komponenten in anderen
- Faktorisierung
  Häufig benutzte Komponenten (z.B. Geldbeträge, Geokoordinaten) in ein Dokument zur leichteren Wiederverwendung herausfaktorisieren

(4.2) Schemadokumentaufbau

Rootelement eines Dokuments ist `schema`

Elementname: `schema`

Parents: none.

Attribute name	Type	Description
id	ID	unique ID
version	normalizedString	version of the schema document
xml:lang	language	natural language of the schema document
targetNamespace	anyURI	namespace for global schema components

(4.2) Schemadokumentaufbau (Fortsetzung)

Attribute name	**Type/Value**	**Description**
attribute-FormDefault	"qualified" | "unqualitifed"	local attributes should use qualified names ?
element FormDefault	"qualified" | "unqualitifed"	local elements should use qualified names?

(4.2) Schemadokumentaufbau (Fortsetzung)

Attribute name	Type/Value	Description
blockDefault	"#all" |list of ("substitution" | "extension" | "restriction")	whether to block element substitution or type substitution
finalDefault	"#all" | list of ("extension" | "restrictions" | "list" | "union")	whether to disallow type derivation

(4.2) Schemadokumentaufbau (Fortsetzung)

Content:

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

Content

1. (include | import | redefine | annotation)*
2. ((attribute | attributeGroup | complexType | element | group | notation | simpleType), annotation*)*

(4.3) Schemaaufbau

Schemadokumenten können mittels `include` und `import` zusammengeführt werden.
Andere Möglichkeiten:

- Instanz: spezifiziere mehrere Schemadokumente (schema locations)
- Prozessor: mehrere Schemadokumente als Parameter
- Prozessor: mehrere vordefinierte Orte, an denen er Schemadokumente finden kann

(4.3.2) Eindeutigkeit von qualifizierenden Namen

WICHTIG:

- Qualifizierte Namen von global deklarierten Schemakomponenten müssen in einem Schema (und nicht nur in einem Schemadokument) eindeutig sein.

Bemerkungen:

- Qualifizierte Namen können sehrwohl in unterschiedlichen Schemadokumenten gleich sein. Diese dürfen dann nur nicht gleichzeitig in einem Schema verwendet werden.
- Eindeutigkeit ist pro Schemakomponentenkategorie zu sehen. Bspw. dürfen ein Element und ein Attribute den gleichen qualifizierten Namen haben, ABER:
- simple und complex types dürfen nicht denselben qualifizierten Namen haben.

(4.3.3) Fehlende Schemakomponenten

- Referenzierte Schemakomponenten dürfen ausserhalb des Schema liegen, solange diese Komponenten nicht für die Validierung des vorliegenden Dokumentes notwendig sind.

(4.3.4) Schema document defaults

Betrifft: `attributeFormDefault`, `elementFormDefault`, `blockDefault`, `finalDefault`.

- Festgelegter Wert gilt immer für die entsprechenden Komponenten des Schemadokuments.
- Falls es inkludiert/importiert wird, ändert sich dies nicht, selbst wenn das inkludierende/importierende Dokument etwas anderes spezifiziert!

(4.4) include, redefine, import

- include
- redefine machen wir nicht
- import

(4.4) `include`

- `include` darf nur auf oberster Ebene vorkommen
- `include` darf nur am Anfang des Schemadokuments vorkommen (zusammen mit `redefine` und `import`)
- Es ist nicht verboten das gleiche Schema (direkt oder indirekt) mehrfach zu inkludieren. (Es kostet nur Zeit)

(4.4) include

Falls include verwendet wird, muss eine der folgenden Bedingungen gelten:

1. Beide Schemadokumente haben denselben target namespace
2. Keines der beiden Schemadokumente hat einen target namespace
3. Das inkludierende Schema hat einen target namespace und das inkludierte Schema hat keinen

Fall 3:

- chameleon components
- nehmen target namespace des inkludierenden Schemadokumentes an

(4.4) `redefine`

- `redefine` ist sehr ähnlich zu `include`
- ABER:
  - Keine, einige, alle Schemakomponenten dürfen redefiniert werden.

(4.4) `import`

- `import` gibt dem Schemaprozessor einen Hinweis darauf, dass Komponenten aus einem anderen Namensraum referenziert werden. Der Prozessor muss diesem Hinweis nicht nachgehen.
- `import` darf nur als top-level Element unter `schema` vorkommen.
- Es muss am Anfang stehen, zusammen mit den `include` und `redefine`

(4.4) `import`

- **Name:** `import`
- **Parent:** `schema`

Attribute name	Type	Description
id	ID	unique ID
namespace	anyURI	namespace to be imported
schemaLocation	anyURI	location of schema document describing components in the imported namespace

- **Content:** annotation?

(4.4) `import`

- `namespace` gibt an, welchen namespace man importieren möchte
- ist `namespace` nicht angegeben, so gibt man an, dass man sich für Komponenten in keinem Namensraum interessiert
- Der importierte Namensraum darf nicht derselbe sein wie der target namespace des importierenden Schemadokuments
- Falls das importierende Schemadokument keinen target namespace hat, so muss das importierte Schemadokument einen haben.

(4.4) `import`

weitere Bedingungen:

- falls keine `schemaLocation` angegeben wird, geht man davon aus, dass der Prozessor weiss, wo man die Schemadokumente findet
- falls `schemaLocation` angegeben wird, so muss die Derefernzierung ein Schemadokument ergeben.
- Der target namespace dieses Schemadokuments muss gleich dem im `namespace`-Attribute von `import` angegebenen sein

Freiheiten:

- Es ist legal mehrmals den gleichen Namensraum zu importieren.
- Sogar zyklische imports sind erlaubt.

(5) Instanzen und Schemata

- Ein Schema kann mehrere gültige Instanzen beschreiben.
- Diese können unterschiedliche Wurzelelemente haben.
- Eine Instanz kann durch mehrere Schemata beschrieben werden, abhängig von den Umständen.

(5.1) Instanzattribute

Die folgenden Attribute sind global deklariert:

Attribute name	**Type**	**Default**	**Description**
nil	boolean	false	whether element's value is nil
type	QName		name of datatype substituted for the element's declared type
schemaLocation	list of anyURI		list of locations of schema documen for designated namespaces
noNamespace-SchemaLocation	anyURI		location of a schema document with no target namespace

Assoziieren von Schemata zu Instanzen

- **Hinweise in der Instanz** `schemaLocation`, `noNamespaceSchemaLocation`
- **Anwendungssache** Programm weiss wie es das macht
- **Benutzersache** Angabe zur Prozessierungszeit (e.g. command line)
- **Namensraumdereferenzierung** (5.5)

(5.4) xsi:schemaLocation

- Wert: Liste von Paaren `namespaceURI, schemaURI`)
- Falls Schema andere Schemata importiert wird: Aufnahme in Liste unnötig
- `xsi:schemaLocation` darf in jedem Element auftauchen
- Prozessoren können einige/alle `xsi:schemaLocation` ignorieren

Beispiel:

```
xsi:schemaLocation="http://example.org/prod prod.xsd
                    http://example.org/ord  ord.xsd"
```

(5.4) xsi:noNamespaceSchemaLocation

- Wert: eine URI für eine Schemalocation
- Das referenzierte Schemadokument darf keinen target namespace haben.

(5.5) Namensraumdereferenzierung

- Namensraum ist URI
- Man kann gucken, ob es da etwas gibt.
- Was es da gibt, könnte ein Schemadokument sein, muss es aber nicht

und sollte es nicht:

- mehrere Schemadokumente, die einen Namensraum beschreiben
- andere Dokumente (z.B. Dokumentation) könnten da liegen

Bessere Lösung:

- resource directory beschrieben in RDDL

(5.6) Wurzelelement

- Jedes global definierte Element kann Wurzelelement sein.
- Es gibt keine Möglichkeit, dies einzuschränken.

(5.7) DTS und Schemata

DTDs und Schemata können zusammen benutzt werden

1. DTD processing findet zuerst statt (validieren, entities expandieren, whitespace normalisieren)
2. es folgt die Validierung bzgl. des Schemas

(6) Schema documentation and extension

machen wir nicht

(7.1) Globale Elementdeklaration

Name: `element`
Parent: `schema`

Attribute	Type	Required Default	Description
id	ID		unique ID
name	NCName	required	element-type name
type	QName		data type
default	string		default value
fixed	string		fixed value
nillable	boolean	false	whether `xsi:nil` can be used

Attribute	Type	Required Default	Description
abstract	boolean	false	whether it can be instantiated
substitution- Group	QName		head of substitution group to which it belongs
block	"#all" or list of ("substitution" | "extension" | "restriction")	defaults to `blockDefault` of `schema`	whether type and/or element substitutions should be blocked from the instance
final	"#all" or list of ("substitution" | "restriction")	defaults to `finalDefault` of `schema`	whether the declaration can be head of a substitution group

Content

```
annotation?,
(simpleType | complexType)?,
(key | keyref | unique)*
```

(7.1) Beispiel

```
<xsd:schema xmlns: xsd = "http://www.w3.org/2001/XMLSchema"
            xmlns = "http://example.org/prod"
            targetNamespace = "http://example.org/prod">

<xsd:element name = "name" type = "xsd:string"/>
<xsd:element name = "size" type = "xsd:integer"/>

<xsd:complexType name = ="ProductType">
  <xsd:sequence>
    <xsd:element ref = "name"/>
    <xsd:element ref = "size" minOccurs = "0"/>
  </xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

(7.1) Globale Elementdeklaration

- Die globalen Elementnamen eines Schemas müssen eindeutig sein.
- Der Wert des `name`-Attributs muss eine Name ohne Doppelpunkt sein, dass heißt, ein unqualifizierter.
- Der qualifizierte Name des Elements besteht aus dem target namespace und dem im `name`-Attribut angegebenen lokalen Namen.
- Vorsicht: `minOccurs` und `maxOccurs` in Elementreferenz und nicht in der Elementdeklaration.

(7.1) Lokale Elementdeklaration

Name: `element`
Parents: `all`, `choice`, `sequence`

Attribute	Type	Required Default	Description
id	ID		unique ID
name	NCName	required	element-type name
type	QName		data type
default	string		default value
fixed	string		fixed value
nillable	boolean	false	whether `xsi:nil` can be used

Attribute	Type	Required Default	Description
form	"qualified | "unqualified"	defaults to `elementForm-` `Default` of `schema`, which defaults to unqualified	whether element-type name must be qualified in the instance
minOccurs	`nonNegative-` `Integer`	1	minimum number of occurrences
maxOccurs	`nonNegative-` `Integer`	1	maximum number of occurrences

Attribute	Type	Required Default	Description
block	"#all" or list of ("substitution" | "extension" | "restriction")	defaults to `blockDefault` of `schema`	whether type and/or element substitutions should be blocked from the instance

Content:

```
annotation?,
(simpleType | complexType)?,
(key | keyref | unique)*
```

Beispiel

```
<xsd:schema xmlns: xsd = "http://www.w3.org/2001/XMLSchema"
            xmlns = "http://example.org/prod"
            targetNamespace = "http://example.org/prod">

<xsd:complexType name = "ProductType">
  <xsd:sequence>
    <xsd:element name = "name" type = "xsd:string"/>
    <xsd:element name = "size" type = "xsd:integer"
                 minOccurs = "0"/>
  </xsd:sequence>
</xsd:complexType>

</xsd:schema>
```

(7.1) Lokale Elementdeklaration

- `form` wie gehabt
- scope of names: local element declaration

(7.1) Globale vs. Lokale Elementdeklaration

Globale Elementdeklaration bevorzugen, falls:

- Elementdeklaration für Wurzelelement benötigt
- Wiederverwendbarkeit erwünscht
- Elementdeklaration soll in substitution group vorkomment

Lokale Elementdeklaration bevorzugen, falls

- unqualifizierte Elementtypnamen in Instanz erwünscht
- der gleiche Elementtypname soll öfter verwendet werden

(7.2) Deklaration des Datentyps von Elementen

Assoziation von Typ zu Elementtypnamen:

1. Referenzieren eines benannten Datentyps
2. Definiere anonymen Typ mit `simpleType` oder `complexType` Kind
3. Ohne beides: Dann wird der Typ automatisch `anyType`

Beispiele

```
<xsd:element name = "size" type = "SizeType"/>

<xsd:element name = "name" type = "xsd:string"/>

<xsd:element name = "product">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref = "name"/>
      <xsd:element ref = "size"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name = "anything"/>
```

(7.3) Default and fixed values

- Falls Element leer: Schemaprozessor fügt default oder fixed Wert ein
- Falls Element fehlt: Schemaprozessor fügt es nicht ein!
- Nur ein Attribut `default` oder `fixed` darf vorkommen (gegenseitiger Ausschluss)

(7.3) Default and fixed values

Default und fixed Werte dürfen nur benutzt werden, falls Elementtyp:

- simple type
- complex type mit simple content
- complex type mit mixed content, falls alle Kindelemente optional sind

(7.3) Default values

```
<xsd:element name = "product">
  <xsd:complexType>
    <xsd:choice minOccurs = "0" maxOccurs = "unbounded">
      <xsd:element name = "name" type = "xsd:string"
                   default = "N/A"/>
      <xsd:element name = "size" type = "xsd:integer"
                   default = "12"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>
```

(7.3) Default values

Beachte:

- falls `xsi:nil` *true* ist, wird default Wert nicht eingesetzt
- falls der Wert nur aus Leerzeichen besteht und `whiteSpace` facet auf `collapse` gesetzt ist (z.B. bei `xsd:integer`), so wird der default Wert eingesetzt

(7.3) Fixed values

- fixed Werte werden genau dann eingefügt, wenn default Werte eingefügt werden
- Falls ein Wert vorhanden ist, so muss er gleich dem fixed Wert sein Gleichheit abstrahiert von verschiedenen Darstellungen des gleichen Werts und beinhaltet Whitespace-Behandlung

(7.4) Nils and nillability

Gründe für fehlende Werte:

- Information ist nicht anwendbar (z.B. Regenschirm hat keine Größe)
- Wert ist unbekannt
- Wert ist für Anwendung irrelevant
- Es ist der default und daher nicht spezifiziert
- Element vorhanden, aber Inhalt ist leer (z.B. Art der Geschenkverpackung: leer heisst es gibt keine Geschenkverpackung)
- es fehlt fehlerhafterweise

(7.4) Nils and nillability

Ein Element mit `xsi:nil` gleich *true* bedeutet:

- Obwohl der Wert des Elements fehlt, soll es als gültig betrachtet werden.

Beachte:

- Falls `xsi:nil` gleich *true* ist, so muss das Element leer sein.

(7.4) Nils and nillability

Vorteile von `xsi:nil`

- der Datentyp muss nicht abgeschwächt werden (z.B. durch optionale Kindelemente)
- explizite Aussage über fehlenden Wert
- Falls Anwendung von der Anwesenheit des Elementes abhängt, so ist dies kein Problem
- Default value processing kann nach belieben für ein Element abgeschaltet werden

(7.4) Nils and nillability

Beispielinstanzen:

```
<size xsi:nil = "true"/>
<size xsi:nil = "true"></size>
<size xsi:nil = "true" system = "US-Dress"/>
<size xsi:nil = "false">10</size>
<size xsi:nil = "true">10</size> <!-- INVALID -->
```

(7.4) Nils and nillability

- Verwendung von `xsi:nil` setzt voraus, dass Element im Schema als *nillable* gekennzeichnet wurde
  (selbst wenn `xsi:nil` auf *false* gesetzt wurde)
- Dies geschieht durch setzten des Attributs `xsd:nillable` auf *true*
- Falls `xsd:nillable` *true* ist, so darf man nicht `xsd:fixed` verwenden
  (default Werte bleiben davon unberührt)
- Elemente, die nillable sind dürfen nicht in Schlüsseln vorkommen

(7.4) Nils and nillability

Beispiel:

```
<xsd:element name = "size"
             type = "xsd:integer"
             nillable = "true"/>
```

(7.4) Qualified vs. unqualified forms

- global deklarierte Elemente: immer mit Namespaceprefix
- lokal deklarierte Elemente: Wahl durch `form`
  - `qualified`
  - `unqualified`

  defaults to `elementFormDefault` of `schema`, which in turn defaults to `unqualified`

(8) Attributdeklaration

Wie bei Elementdeklarationen gibt es

- globale Attributdeklarationen und
- lokale Attributdeklarationen.

(8.1) Elemente vs. Attribute

Vorteile Attribute:

- kürzer
- Fehlende Attribute mit default-values können nachträglich hinzugefügt werden

Vorteile Elemente:

- leichter erweiterbar
- können strukturiert sein
- können mehrfach auftreten
- können in Substitutionsgruppen auftreten
- können nillable sein
- type substitution ist möglich
- Ordnung ist signifikant
- für große (lange) Werte besser lesbar

Generelle Regel:

- Benutze Elemente für Daten
- Benutze Attribute für Metadaten

Beispiel:

```
<price currency = "EUR">9.98</price>
```

(8.1) Globale Attributdeklaration

Name: `attribute`
Parent: `schema`

Attribute Name	Type	Required/ Default	Description
id	ID		unique ID
name	NCName	required	attribute name
type	QName		data type
default	string		default value
fixed	string		fixed value

Content:

```
annotation?, simpleType?
```

(8.1) Beispiel

```
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema"
            xmlns = "http://example.org/prod"
            targetNamespace = "http://example.org/prod">

  <xsd:attribute name = "system" type = "xsd:string"/>
  <xsd:attribute name = "dim"    type = "xsd:integer"/>

  <xsd:complexType name = "SizeType>
    <xsd:attribute ref = "system" use = "required"/>
    <xsd:attribute ref = "dim"/>
  </xsd:complexType>

</xsd:schema>
```

(8.1) Lokale Attributdeklaration

Name: `attribute`
Parents: `complexType`, `restriction`, `extension`, `attributeGroup`

Attribute Name	**Type**	**Required/ Default**	**Description**
id	ID		unique ID
name	NCName	required	attribute name
type	QName		data type

form	"qualified" | "unqualified"	defaults to `attributeFormDefault` of `schema`, which in turn defaults to "unqualified"	whether the attribute must be qualified in the instance
use	"optional" | "requried" | "prohibited"	`optional`	whether it is required or optional
default	string		default value
fixed	string		fixed value

Content: `annotation?, simpleType?`

(8.1) Lokale Attributedeklaration

Namensraum, in dem ein lokale definierter Attributname liegt:

- Falls `form` `qualified` ist, so ist es im `targetNamespace` des Schemas.
- Falls `form` `unqualified` ist, so ist es in keinem Namensraum.

(8.1) Globale vs. lokale Attributdeklaration

- am besten keine globalen Attributdeklarationen verwenden
- falls man dennoch in Versuchung gerät, betrachte Alternativen:
  - Attributgruppen
  - lokale Deklaration mit benanntem (globalen) simple type

(8.2) Zuweisung von Typen zu Attributen

Möglichkeiten:

1. named type spezifiziert durch `type`-Attribut
2. anonymous type spezifiziert durch `simpleType`-Kind
3. weder noch. Dann: `anySimpleType`

(8.2) Zuweisung von Typen zu Attributen

```
<xsd:attribute name = "color" type = "ColorType"/>

<xsd:attribute name = "dim" type = "xsd:integer"/>

<xsd:attribute name = "system">
  <xsd:simpleType>
    <xsd:restriction base = "xsd:string">
      <xsd:enumeration value = "US-Dress/>
      <!-- ... -->
    </xsd:restriction>
  </xsd:simpleType>
</xsd:attribute>

<xsd:attribute name = "anything"/>
```

(8.3) Default and fixed values

- Falls attribut in Element fehlt und default oder fixed angegeben wurde: Attribut wird durch Prozessor eingefügt.
- `fixed` und `default` schließen sich gegenseitig aus
- Falls default value spezifiziert, kann das Attribut nicht `required` sein.
- Ein default Wert wird nur dann eingefügt, falls das Attribut fehlt. Er wird nicht mal eingeführt, wenn Attributwert leer ist (`a = ""`)
- fixed values werden genau dann *eingeführt*, wenn default value eingefügt wird.
- Falls für ein fixed Attribut ein Wert in einem Element spezifiziert wird, so muss dieser gleich dem fixed value sein.
  (Gleichheit modulo verschiedener Repräsentationen gleicher Werte)

(8.4) Qualified vs. unqualified forms

- default namespaces gelten nicht für Attribute
- `form` gibt also an, ob Präfixe für Attribute benutzt werden müssen oder nicht
- mehr über Designentscheidungen diesbezüglich später (20.2)

(9) Simple types

1. atomic types
2. list types
3. union types

(9.1) Wie klein sollten Daten heruntergebrochen werden?

- so kleine Einheiten wie möglich:
  - zusammensetzen einfach, auseinanderdividieren schwierig
  - Typisierung/Validierung kann sonst schwierig werden
  - Ausschluß arithmetischer Operationen
- Beispiel

```
<price>5EUR</pice>
<price currency = "EUR">5</price>
```

(9.2) Simple type definitions

- Einfache Typen können benannt oder anonym sein.
- benannte einfache Typen: nur global definierbar
- Name muss eindeutig unter allen Typnamen sein (egal ob simple oder complex)

(9.2) Benannter einfacher Typ

Name: `simpleType`
Parents: `schema`, `redefine`

Attribute Name	**Type**	**Required/ Default**	**Description**
id	ID		unique ID
name	NCName	required	simple type name
final	"#all" | list of ("restriction" | "list" | "union")	defaults to `finalDefault` of `schema`	whether other types can be derived from this one

Content: `annotation?, (restriction | list | union)`

Beispiel

```
<xsd:simpleType name = "DressSizeType">
  <xsd:restriction base = "xsd:integer">
    <xsd:minInclusive value = "2"/>
    <xsd:maxInclusive value = "18"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:element name = "size" type = "DressSizeType"/>
```

(9.2) Anonymer einfacher Typ

Name: `simpleType`

Parents: `element`, `attribute`, `restriction`, `list`, `union`

Attribute Name	Type	Required/ Default	Description
id	ID		unique ID

Content: `annotation?, (restriction | list | union)`

(9.2) Benannte vs. anonyme einfache Typen

Vorteile benannte einfache Typen:

- reuse
- consistency
- reducing possibility of error
- saving time developing new schema
- simplifying maintenance

(9.3) Einschränkung einfacher Typen

- Jeder einfache Typ ist Einschränkung (restriction) eines anderen einfachen Typen (= Basistyp)
- Es ist nicht möglich einfache Typen zu erweitern (extension)
- außer um Attribute: ergibt dann komplexen Typen
- Einschränkung durch Anwendung von Facetten
- nicht jede Facette ist auf jeden Typ anwendbar

(9.3) Definieren einer Restriktion

Name: `restriction`
Parents: `simpleType`

Attribute Name	Type	Required/ Default	Description
id	ID		unique ID
base	QName	either a `base` attribute or a `simpleType` child is required	simple type to be restricted

Content [hauptsächlich Facetten]:

```
annotation?, simpleType?, (minExclusive | maxExclusive |
minInclusive | maxInclusive | length | minLength | maxLength
| totalDigits | fractionDigits | enumeration | pattern |
whiteSpace)
```

(9.3) Beispiel

```
<xsd:simpleType name = "DressSizeType">
  <xsd:restriction base = "xsd:integer">
    <xsd:minInclusive value = "2"/>
    <xsd:maxInclusive value = "18"/>
    <xsd:pattern value = "\d{1,2}"/>
  </xsd:restriction>
</xsd:simpleType>
```

(9.3) Facetten

Sei x der in der Facette angegebene Wert.

Facet	**Meaning**
`minExclusive`	Wert muss $> x$ sein
`maxExclusive`	Wert muss $< x$ sein
`minInclusive`	Wert muss $\geq x$ sein
`maxInclusive`	Wert muss $\leq x$ sein
`length`	Länge des Werts muss gleich x sein
`minLength`	Länge des Werts muss $\geq x$ sein
`maxLength`	Länge des Werts muss $\leq x$ sein
`totalDigits`	maximale Anzahl signifikanter Stellen
`fractionDigits`	maximale Anzahl der Nachkommastellen
`whiteSpace`	`preserve`, `replace`, `collapse`
`enumeration`	x ist einer der gültigen Werte
`pattern`	x ist regulärer Ausdruck und Wert muss matchen

(9.3) Facetten

Name: eine der obigen facetten

Parent: `restriction`

Attribute Name	**Type**	**Required/ Default**	**Description**
id	ID		unique ID
value	various		value of the restricting facet
fixed	boolean	false n/a for pattern and enumeration	whether facet can be further restricted

(9.3) Beispiel

```
<xsd:simpleType name = "DressSizeType">
  <xsd:restriction base = "xsd:integer">
    <xsd:minInclusive value = "2"/>
    <xsd:maxInclusive value = "18"/>
    <xsd:pattern value = "\d{1,2}"/>
  </xsd:restriction>
</xsd:restriction>
```

(9.3) Inheriting and restricting facets

- Facetten des eingeschränkten Typen müssen restriktiver sein als die ererbten
- bei `enumeration` entspricht dies einer Teilmengenbeziehung

(9.3) Beispiel

```
<xsd:simpleType name = "MediumDressSizeType">
  <xsd:restriction base = "DressSizeType">
    <xsd:minInclusive value = "8"/>
    <xsd:maxInclusive value = "12"/>
  </xsd:restriction>
</xsd:restriction>
```

(9.3) Fixed facets

- alle Facetten außer `enumeration` und `pattern` können fixiert werden (`fixed` = *true*)
- dann dürfen diese nicht weiter eingeschränkt werden
- fixieren nur falls die Bedeutung des Typs sonst erheblich leidet (Bsp: Währung: Änderung der Anzahl der Nachkommastellen)

(9.4) Nullwerte (leere Werte)

- mit Element kein Problem
- Attribute: union von Originaltyp und leerem String

(9.4) Nullwerte (leere Werte)

```
<xsd:simpleType name = "DressSizeType">
  <xsd:union>
    <xsd:simpleType>
      <xsd:restriction base = "xsd:integer">
        <xsd:minInclusive value = "2"/>
        <xsd:maxInclusive value = "18"/>
      </xsd:restriction>
    </xsd:simpleType>
    <xsd:simpleType>
      <xsd:restriction base = "xsd:token">
        <xsd:enumeration value = ""/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:union>
</xsd:simpleType>
```

(9.4) Facetten

- length ist nicht auf boolean, numeric types, date/time anwendbar
- totalDigits muss eine positive ganze Zahl sein
- fractionDigits muss kleiner gleich totalDigits sein
- enumerate-Werte müssen eindeutig sein und gültig für den Typ
- pattern dürfen mehrfach vorkommen. Semantik: Disjunktion

(9.4) Facetten: `whiteSpace`

`preserve`	alles bleibt wie es ist
`replace`	tab, line feed, carriage return durch blank ersetzen, aufeinanderfolgende blanks durch ein blank ersetzen
`collapse`	wie replace und dann blank Präfix und Suffix abschneiden

(9.4) Facetten: `whiteSpace`

- für eingebaute Typen ist whitespace handling fixed auf `collapse`
- nur string-basierte Typen können `whiteSpace` aufweisen: ABER NICHT MACHEN
- statt dessen: richtigen string-Typen aussuchen:

String type	**whiteSpace**
`string`	`preserve`
`normalizedString`	`replace`
`token`	`collapse`

(9.5) Blockieren der Ableitung von einfachen Typen

Wert von `final` :

- `#all`:
  - keine Ableitung mehr möglich
- explizite Liste (`restriction, extension, list, union`):
  - nur diese werden blockiert
- wird `final` nicht gesetzt:
  - Wert wird durch `finalDefault` gesetzt
    - falls dies nicht spezifiziert: keinerlei Blockierung
    - falls dieser gesetzt: durch `final=""` wieder alles erlaubt

(10) Regular expressions

Machen wir nicht, müssen Sie aber trotzdem können

(11) Union- und Listentypen

- Listentypen haben als Wert eine Liste von durch Leerzeichen getrennten atomaren Werten
- Uniontypen haben als Wert atomare Werte oder Listen von atomaren Werten, wobei der Wertebereich zusammengesetzt ist.

(11.1) Union- und Listentypen

Jeder neu definierte einfache Typ muß wie folgt gewonnen werden:

1. Restriktion eines existierenden Typen
2. Liste eines existierenden Typen
3. Union eines existierenden Typen

(11.1) Union- und Listentypen

		Derived Type		
		restriction	list	union
Base Kind	atomic	atomic	list	union
	list	list	not legal	union
	union	union	list[1]	union

1) nur legal, falls Union-Typ keinen Listentyp enthält

(11.2) Uniontypen

Name: `union`
Parent: `simpleType`

Attribute Name	Type	Required/ Default	Description
id	ID		unique ID
memberTypes	list of QNames	either a `memberTypes` attribute or at least one `simpleType` child is required	member types making up the union type

Content: `annotation?, simpleType*`

(11.2) Beispiel

```
<xsd:simpleType name = "SizeType">
  <xsd:union>
    <xsd:simpleType>
      <xsd:restriction base = "xsd:integer">
        <xsd:minInclusive value = "2"/>
        <xsd:maxInclusive value = "18"/>
      </xsd:restriction>
    </xsd:simpleType
    <xsd:simpleType>
      <xsd:restriction base = "xsd:token">
        <xsd:enumeration value = "small"/>
        <xsd:enumeration value = "medium"/>
        <xsd:enumeration value = "large"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:union>
</xsd:simpleType>
```

(11.2) Beispiel

```
<xsd:simpleType name = "SizeType">
  <xsd:union memberTypes = "DressSizeType SMLSizeType"/>
</xsd:simpleType>
```

(2.2) Uniontyp

- Es sind nur zwei Facetten zur Restriktion eines Uniontyps erlaubt:
  - `pattern`
  - `enumeration`
- union von union ist möglich
- Instanzelemente: optional: `xsi:type` um base-type von Uniontype, der in Instanz genutzt wird, zu spezifizieren

(2.3) Listentypen

Name: `list`
Parent: `simpleType`

Attribute Name	**Type**	**Required/ Default**	**Description**
id	ID		unique ID
itemType	QName	either a `itemType` attribute or a `simpleType` child is required	member type that make up the list type

Content: `annotation?, simpleType?`

(2.3) Beispiel

Schema:

```
<xsd:simpleType name = "AvailableSizesType">
  <xsd:list itemType = "DressSizeType"/>
</xsd:simpleType>
```

Instanz:

```
<availableSizes>10 12 14</availableSizes>
```

(2.3) Listentypen

- `whiteSpace`-Facette für Listentypen ist `collapse`

(2.3) Listentypalternative

```
<availableSizes>10 12 14</availableSizes>
```

oder

```
<availableSizes>
  <size>10</size>
  <size>12</size>
  <size>14</size>
</availableSizes>
```

(2.3) Nachteile von Listentypen

- Listenelemente können keine Whitespaces enthalten
- Erweiterbarkeit fällt flach
- keine Möglichkeit nil-Werte zu spezifizieren
- Momentan kein Support für Listentypen in XPath oder XSLT

(11.3) Restriktion von Listentypen

- Facetten gelten für die gesamte Liste und nicht für deren Elemente

(11.3) Restriktion von Listentypen

Name: `restriction`
Parent: `simpleType`

Attribute Name	Type	Required/ Default	Description
id	ID		unique ID
base	QName	either a `base` attribute or a single `simpleType` child is required	base type of the restriction. HERE: list type

Content:

```
annotation?, simpleType?,
(length | minLength | maxLength | pattern | enumeration)*
```

(11.3) Restriktion von Listentypen

- Längenfacetten beziehen sich auf die Listenlänge
- `enumeration` enthält Listen
- `pattern` gilt für die gesamte Liste

(11.3) Listentypen

- Listen von Unions ist o.k.
- Listen von Listen sind verboten
- der Elementtyp einer Liste kann nachträglich nicht mehr eingeschränkt werden

(12) Built-in simple types

wir überspringen alles und behandeln nur

(12.7) Wertgleichheit

- Wertvergleiche sind mehr als Stringvergleiche
- Nur Typen, die durch Restriktion oder Unionbildung miteinander in Beziehung stehen können gleiche Werte haben
- verschiedene Repräsentationen eines Wertes führen zur Wertgleichheit
- vor Stringvergleich wird Whitespacebehandlung durchgeführt

(13) Komplexe Typen

- nur für Content-Model und Attribute von Elementen
- 4 content-types:
  1. simple content
  2. element-only content
  3. mixed content
  4. empty content

(13.1) Komplexe Typen

Elemente von komplexem Typ:

```
<size system = "US-DRESS">10</size>

<product>
  <number>557</number>
  <name>Short-Sleeved Linen Blouse</name>
</product>

<letter>Dear <custName>Priscilla Walmsley</custName>...</letter>

<color value = "blue" />
```

(13.2) Benannte komplexe Typen

Name: `complexType`
Parents: `schema`, `redefine`

Attribute	Type	Required/default	Description
`id`	`ID`		unique ID
`name`	`NCName`	required	complex type name
`mixed`	`boolean`	`false`	allow mixed content?
`abstract`	`boolean`	`false`	whether the type can be used in an instance
`block`	`"#all"` | list of (`"extension"` | `"restriction"`)	defaults to `blockDefault` of `schema`	block type substitution in instance?
`final`	`"#all"` | list of (`extension"` | `"restriction"`)	defaults to `finalDefault` of `schema`	whether other types can be derived from this one

Content:

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

(13.2) Beispiel

```
<xsd:complexType name="ProductType">
  <xsd:sequence>
    <xsd:element name="number" type="ProdNumType"/>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="size" type="SizeType"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:element name="product" type="ProductType"/>
```

(13.2) Anonyme komplexe Typen

Name: `complexType`
Parents: `element`

Attribute name	Type	Required/default	Description
`id`	`ID`		unique ID
`mixed`	`boolean`	`false`	allow mixed content?

Content:

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

Beispiel

```
<xsd:element name="product">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="number" type="ProdNumType"/>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="size" type="SizeType"/>
    </xsd:sequence>
  </xsd:complexType>
<xsd:element>
```

(13.2) Alternative Möglichkeiten

Drei Möglichkeiten zur Definition eines komplexen Typs, gesteuert durch Kindelement von `complexType`:

1. `complexContent`-Kind: Ableitung aus bestehendem komplexen Typen
2. `simpleContent`-Kind: Ableitung aus bestehendem einfachen Typen
3. `group`, `all`, `choice`, `sequence`-Kind: direkte Definition (ohne Ableitung)

(13.3) Inhaltstypen: Simple Content

```
<xsd:complexType name="SizeType">
  <xsd:simpleContent>
    <xsd:extension base="xsd:integer">
      <xsd:attribute name="system" type="xsd:token"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
```

(13.3) Inhaltstypen: Element-only Content

```
<xsd:complexType name="ProductType">
  <xsd:sequence>
    <xsd:element name="number" type="ProdNumType"/>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="size" type="SizeType"/>
    <xsd:element name="color" type="ColorType"/>
  </xsd:sequence>
</xsd:complexType>
```

(13.3) Inhaltstypen: Mixed Content

```
<xsd:complexType name="LetterType" mixed="true">
  <xsd:sequence>
    <xsd:element name="custName" type="CustNameType"/>
    <xsd:element name="prodName" type="xsd:string"/>
    <xsd:element name="prodSize" type="SizeType"/>
    <!--...-->
  </xsd:sequence>
</xsd:complexType>
```

Beispielinstanz

```
<letter>
Dear <custName>Priscilla Walmsley</custName>,

Unfortunately,we are out of stock of the
<prodName>Short-Sleeved Linen Blouse</prodName>
in size
<prodSize>10</prodSize>
that you ordered...
</letter>
```

(13.3) Inhaltstypen: Empty Content

```
<xsd:complexType name="ColorType">
  <xsd:attribute name="value" type="ColorValueType"/>
</xsd:complexType>
```

[Kein Inhaltsmodell = Empty Content]

(13.4) Benutzen von Elementtypen

Definition komplexer Typen kann

- lokale Elementdeklarationen enthalten
- globale Elementdeklarationen referenzieren (`ref`-Attribut)
  - Achtung: Facetten `min/maxOccurs` nur bei Verwendung des Elements, d.h. nicht in globalen Elementdeklarationen
- Was wir micht machen: Element wildcards

(13.4) Benutzen von Elementtypen

Name: `element` [Hier: Elementreferenz]
Parents: `all, choice, sequence`

Attribute name	**Type**	**Required/ default**	**Description**
`id`	`ID`		unique ID
`ref`	`QName`	required	name of the global element declaration being referenced
`minOccurs`	`nonNegativeInteger`	1	minimum number of times the element may occur
`maxOccurs`	`nonNegativeInteger` | `"unbounded"`	1	maximum number of times the element may occur

Content `annotation?`

Beispiel

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name = "number" type = "ProdNumType"/>
  <xsd:element name = "name"   type = "xsd:string"/>
  <xsd:element name = "size"   type = "SizeType"/>
  <xsd:element name = "color"  type = "ColorType"/>

  <xsd:complexType name="ProductType">
    <xsd:sequence>
      <xsd:element ref = "number"/>
      <xsd:element ref = "name"/>
      <xsd:element ref = "size"  minOccurs = "0"/>
      <xsd:element ref = "color" minOccurs = "0"/>
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

(13.5) Benutzung von Modellgruppen (Model Groups)

- erlauben das Gruppieren von Elementen
- Möglichkeiten:
  - `sequence`: offensichtlich Sequenz
  - `choice`: Auswahl: genau ein Element muss vorkommen
  - `all`: alle Elemente müssen vorkommen, Reihenfolge egal
- Jeder komplexe Type hat genau eine model-group-Kind
- Elementdeklarationen sind niemals direkt Kind von `complexType`

(13.5) Model groups: sequence

Name: sequence
Parents: complexType, restriction, extension, group, sequence

Attribute	Type	Default/	Description
id	ID		unique ID
minOccurs	nonNegativeInteger | "unbounded"	1	wie gehabt
maxOccurs	nonNegativeInteger | "unbounded"	1	wie gehabt

Content: annotation?, (element | group | choice | sequence | any)*

(13.5) Beispiel

```
<xsd:complexType name="ProductType">
  <xsd:sequence>
    <xsd:element name="number" type="ProdNumType"/>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="size" type="SizeType" minOccurs="0"/>
    <xsd:element name="color" type="ColorType" minOccurs="0"/>
  <xsd:sequence>
</xsd:complexType>
```

(13.5) Model groups: `choice`

Name: `choice`
Parents: `complexType, restriction, extension, group, sequence`

Attribute	Type	Default/	Description
`id`	`ID`		unique ID
`minOccurs`	`nonNegativeInteger` | `"unbounded"`	`1`	wie gehabt
`maxOccurs`	`nonNegativeInteger` | `"unbounded"`	`1`	wie gehabt

Content: `annotation?, (element | group | choice | sequence | any)*`

Beispiel

```
<xsd:complexType name="ItemsType">
  <xsd:choice>
    <xsd:element name="shirt" type="ShirtType"/>
    <xsd:element name="hat" type="HatType"/>
    <xsd:element name="umbrella" type="UmbrellaType"/>
  </xsd:choice>
</xsd:complexType>
```

Beispiel

```
<xsd:complexType name="ItemsType">
  <xsd:choice minOccurs="0" maxOccurs="unbounded">
    <xsd:element name="shirt" type="ShirtType"/>
    <xsd:element name="umbrella" type="UmbrellaType"/>
    <xsd:element name="hat" type="HatType"/>
  </xsd:choice>
</xsd:complexType>
```

(13.5) Model groups: `all`

- Alle Kinder von `all` müssen vorkommen, egal in welcher Reihenfolge
- Einschränkungen:
  - nur Elementdeklarationen und Elementreferenzen als Kinder
  - jedes vorkommende Kind muß `maxOccurrences = "1"` haben
  - `all` group kann nicht in anderen Gruppen vorkommen
    $\Longrightarrow$ `all` group immer auf oberster Ebene unter `complexType`

(13.5) Model groups: `all`

Name: `all`
Parents: `complexType, restriction, extension, group`

Attribute	Type	Default/	Description
`id`	`ID`		unique ID
`minOccurs`	`nonNegativeInteger`	1 | 2	wie gehabt
`maxOccurs`	`nonNegativeInteger`	1	wie gehabt

Content: `annotation?, element*`

(13.5) Model groups: named model group references

(definition von named model groups später)

```
<xsd:complexType name="ProductType">
  <xsd:sequence>
    <xsd:group ref="DescriptionGroup"/>
    <xsd:element name="number" type="ProdNumType"/>
    <xsd:element name="name" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
```

(13.5) Determinismus des Inhaltsmodels

- Nur ein Zweig des Inhaltsmodels darf zu jeden Zeitpunkt beim Parsen gültig sein.
- kein Vorausschauen notwendig

Ungültiges Beispiel

```
<xsd:complexType name="AOrBOrBothType">
  <xsd:choice>
    <xsd:element name="a" type="xsd:string"/>
    <xsd:element name="b" type="xsd:string"/>
    <xsd:sequence>
      <xsd:element name="a" type="xsd:string"/>
      <xsd:element name="b" type="xsd:string"/>
    </xsd:sequence>
  </xsd:choice>
</xsd:complexType>
```

Gleiches Inhaltsmodell deterministisch Modelliert

```
<xsd:complexType name="AOrBOrBothType">
  <xsd:choice>
    <xsd:sequence>
      <xsd:element name="a" type="xsd:string"/>
      <xsd:element name="b" type="xsd:string" minOccurs="0"/>
    </xsd:sequence>
    <xsd:element name="b" type="xsd:string"/>
  </xsd:choice>
</xsd:complexType>
```

(13.6) Attribute

Lokale Attributdeklaration:

```
<xsd:complexType name="ProductType">
  <xsd:sequence>
    <!--...-->
  </xsd:sequence>
  <xsd:attribute name="effDate" type="xsd:date"
                 default="1900-01-01"/>
</xsd:complexType>
```

Merke: Attributdeklarationen nach Inhaltsmodell

(13.6) Attributreferenz

- Definitionen komplexer Typen können Referenzen auf globale Attribute enthalten
- `use`: `required`, `optional`, oder, nur im Fall von Restriktion, `prohibited` (eliminiert dieses Attribut)
- `default` und `mixed` schliessen sich gegenseitig aus

(13.6) Attributreferenzen

Name: `attribut`
Parents: `complexType`, `restriction`, `textttextension`, `attributeGroup`

Attribute	Type	Required/default	Description
`id`	`ID`		unique ID
`name`	`NCName`	required	complex type name
`used`	"optional" | "required" | "prohibited"	`optional`	required?
`default`	`string`		default value
`fixed`	`string`		fixed value

Content: `annotation?`

Beispiel

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <xsd:attribute name="effDate" type="xsd:date"
                default="1900-01-01/>

 <xsd:complexType name="ProductType">
   <xsd:sequence>
   <!--...-->
   </xsd:sequence>
   <xsd:attribute ref="effDate" default="2000-12-31"/>
  </xsd:complexType>

</xsd:schema>
```

(13.6) Attribute wildcards

machen wir nicht

(13.6) Attribute group references

- Referenzen zur Wiederbenutzung von Attributgruppen in Definitionen komplexer Typen
- Definition von Attributgruppen später

Beispiel

```
<xsd:complexType name="ProductType">
  <xsd:sequence>
    <!--...-->
  </xsd:sequence>
  <xsd:attributeGroup ref="IdentifierGroup"/>
  <xsd:attribute name="effDate" type="xsd:date"/>
</xsd:complexType>
```

(14) Ableitung komplexer Typen

Vorteile Ableitung

- subsetting (garantiere Teilmengeneigenschaft)
- safe extension (kontrollierte Erweiterbarkeit)
- substitution (Substituierbarkeit für Basistypen)
- reuse
- future type-aware XML technologies

(14.2) Restriktion und Erweiterung

- Restriction: Einschränkung auf Teilmenge
- Extension: Erweiterung um neue Kindknoten/Attribute
- beides geht nicht gleichzeitig

(14.3) Simple content, complex content

- komplexer Typ hat entweder einfachen oder komplexen Inhalt
- Ableitung eines komplexen Typen aus einem anderen mittels `simpleContent` oder `complexContent`

(14.3) `simpleContent`-Elemente

`simpleContent` wird benutzt, um komplexe Typen aus

- einfachen oder
- komplexen Typen mit einfachem Inhalt

abzuleiten.

- Falls ein komplexer Typ einfachen Inhalt hat, so auch alle von ihm abgeleiteten Typen

(14.3) simpleContent

Name: simpleContent
Parents: complexType

Attribute name	Type	Required/default	Description
id	ID		unique ID

Content: annotation?, (extension | restriction)

(14.3) complexContent-Elemente

Verwendung zur Ableitung von

- komplexen Typen mit komplexem Inhalt aus ebensolchen

(14.3) complexContent-Elemente

Name: `complexContent`
Parents: `complexType`

Attribute name	Type	Required/default	Description
`id`	`ID`		unique ID
`mixed`	`boolean`	overrides `mixed` value of `complexType`	whether the complex type allows mixed content

Content: `annotation?, (extension | restriction)`

(14.4) Complex type extension

DERIVED TYPE		BASE TYPE				
		Simple type	Complex type			
			Simple content	Element-only	Mixed	Empty
Simple type		no	no	no	no	no
Complex type	Simple content	yes	yes	no	no	no
	Element-only	no	no	yes	no	yes
	Mixed	no	no	no	yes	yes
	Empty	no	no	no	no	yes

(14.4) Simple content extension

- einziger Zweck: Attribute hinzufügen

(14.4) Simple content extension

Name: `extension`
Parents: `simpleContent`

Attribute name	Type	Required/default	Description
`id`	`ID`		unique ID
`base`	`Qname`	required	base type being extended

Content: `annotation?, (attribute | attributeGroup)*, anyAttribute?`

Beispiel

Schema:

```
 %E 14-1
<xsd:complexType name="SizeType">
  <xsd:simpleContent>
    <xsd:extension base="xsd:integer">
      <xsd:attribute name="system" type="xsd:token"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
```

Instanz:

```
<size system="US-DRESS">10</size>
```

(14.4) Complex content extension

- Hinzufügen von neuen Kindern am Ende des Inhaltsmodells (als wären neues und altes Inhaltsmodell die beiden Kinder einer neuen Sequenz)
- Hinzufügen von Attributen
- Erweiterung funktioniert nicht für `all`-Gruppen (sic!)
- Mixed content: alle abgeleiteten Typen haben ebenfalls mixed content
- element-only content: alle abgeleiteten Typen haben ebenfalls element-only content
- empty content: man kann Kinder oder Attribute hinzufügen (nothing special)

(14.4) Complex content extension

Name: `extension`
Parents: `complexContent`

Attribute name	Type	Required/default	Description
`id`	`ID`		unique ID
`base`	`Qname`	required	base type being extended

Content:

```
annotation?,
(group | all | choice | sequence)?,
(attribute | attributeGroup)*,
anyAttribute?
```

Beispiel

```
<xsd:complexType name="ProductType">
  <xsd:sequence>
    <xsd:element name="number" type="ProdNumType"/>
    <xsd:element name="name" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ShirtType">
  <xsd:complexContent>
    <xsd:extension base="ProductType"/>
      <xsd:choice maxOccurs="unbounded">
        <xsd:element name="size" type="SizeType"/>
        <xsd:element name="color" type="ColorType"/>
      </xsd:choice>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

. . . verhält sich wie

```
<xsd:complexType name="ShirtType">
  <xsd:sequence>
    <xsd:sequence>
      <xsd:element name="number" type="ProdNumType"/>
      <xsd:element name="name" type="xsd:string"/>
    </xsd:sequence>
    <xsd:choice maxOccurs="unbounded">
      <xsd:element name="size" type="SizeType"/>
      <xsd:element name="color" type="ColorType"/>
    </xsd:choice>
  <xsd:sequence>
</xsd:complexType>
```

(14.4) Extending `choice` groups

- nothing special here
- Aber Vorsicht: es werden nicht mehr Auswahlmöglichkeiten durch Erweiterung hinzugefügt

(14.4) Attributerweiterungen

- es können nach belieben Attribute hinzugefügt werden
- ABER:
  - Typ/Wertbereiche können nicht geändert werden
  - Attribute können nicht entfernt

(14.5) Complex type restriction

Einschränkung durch

- Einschränken des erlaubten Inhaltsmodells
- Einschränken von Attributen
- Eliminieren von Attributen

(14.5) Complex type restriction

DERIVED TYPE		BASE TYPE				
		Simple type	**Complex type**			
			Simple content	**Element-only**	**Mixed**	**Empty**
Simple type		yes	no	no	no	no
Complex type	**Simple content**	no	yes	no	yes*	no
	Element-only	no	no	yes	yes	no
	Mixed	no	no	no	yes	no
	Empty	no	no	yes*	yes*	yes

* If all children are optional

(14.4) Simple content restrictions

Zweck:

- Einschränkung des einfachen Inhalts
- oder der Attributwertebereiche

Beides geschieht durch Facetten, die ererbte Facetten einschränken müssen

(14.4) Simple content restriction

Name: `restriction`
Parents: `simpleContent`

Attribute name	Type	Required/default	Description
`id`	`ID`		unique ID
`base`	`Qname`	either `base` attribute or `simpleType` child is required	base type being restricted

Content:

```
{annotation?, simpleType?,
(enumeration | fractionDigits | totalDigits
| minInclusive | minExclusive | maxInclusive | maxExclusive
| length | minLength | maxLength
| pattern | whiteSpace)*,
(attribute | attributeGroup)*, anyAttribute?
```

Beispiel: hier Basistyp

```
<xsd:complexType name="SizeType">
  <xsd:simpleContent>
    <xsd:extension base="xsd:integer">
      <xsd:attribute name="system" type="xsd:token"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
```

Beispiel: hier Restriktion

```
<xsd:complexType name="SmallSizeType">
  <xsd:simpleContent>
    <xsd:restriction base="SizeType">
      <xsd:minInclusive value="2"/>
      <xsd:maxInclusive value="6/>
      <xsd:attribute name="system" type="xsd:token"
                     use="required"/>
    </xsd:restriction>
  </xsd:simpleContent>
</xsd:complexType>
```

(14.5) Complex content restriction

- Einschränkung von Inhaltsmodell und Attributen
- Alle relevanten Teile des Inhaltsmodells müssen wiederholt werden
- Das neue muß eine Einschränkung des alten sein

(14.5) Complex content restriction

Name: `restriction`
Parents: `complexContent`

Attribute name	Type	Required/default	Description
`id`	`ID`		unique ID
`base`	`Qname`	required	base type being restricted

Content:

```
annotation?,
(group | all | choice | sequence)?,
(attribute | attributeGroup)*,
anyAttribute?
```

```
<xsd:complexType name="ProductType">
  <xsd:sequence>
    <xsd:element name="number" type="ProdNumType"/>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="size" type="SizeType" minOccurs="0"/>
    <xsd:element name="color" type="ColorType" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="RestrictedProductType">
  <xsd:complexContent>
    <xsd:restriction base="ProductType">
      <xsd:sequence>
        <xsd:element name="number" type="ProdNumType"/>
        <xsd:element name="name" type="xsd:string"/>
      </xsd:sequence>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>
```

(14.5) Attribute restrictions

- Alle Attribute werden vererbt.
- Sie müssen nur in der Einschränkung auftauchen, wenn sie eingeschränkt werden sollen.
- Mögliche Einschränkungen:
  - Typeinschränkung
  - add, change, remove default-value
  - add fixed value, if not present
  - make optional attributes required
  - make optional attributes prohibited (remove them)

```
<xsd:complexType name="BaseType">
  <xsd:attribute name="a" type="xsd:integer"/>
  <xsd:attribute name="b" type="xsd:string"/>
  <xsd:attribute name="c" type="xsd:string" default="c"/>
  <xsd:attribute name="d" type="xsd:string"/>
  <xsd:attribute name="e" type="xsd:string" fixed="e"/>
  <xsd:attribute name="f" type="xsd:string"/>
  <xsd:attribute name="g" type="xsd:string"/>
  <xsd:attribute name="x" type="xsd:string"/> </xsd:complexType>
<xsd:complexType name="DerivedType"> <xsd:complexContent>
    <xsd:restriction base="BaseType">
      <xsd:attribute name="a" type="xsd:positiveInteger"/>
      <xsd:attribute name="b" type="xsd:string" default="b"/>
      <xsd:attribute name="c" type="xsd:string" default="c2"/>
      <xsd:attribute name="d" type="xsd:string" fixed="d"/>
      <xsd:attribute name="e" type="xsd:string" fixed="e"/>
      <xsd:attribute name="f" type="xsd:string" use="required"/>
      <xsd:attribute name="g" type="xsd:string" use="prohibited"/>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>
```

(14.6) Type substitution

- Substituierbarkeit
  Instanzen von abgeleitete Typen können für Instanzen ihre Basistypen eingesetzt werden
- `xsi:type` kann optional substituierten Typ kennzeichnen

Beispiel

```
<xsd:complexType name="ProductType">
  <xsd:sequence>
    <xsd:element name="number" type="ProdNumType"/>
    <xsd:element name="name" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:element name="product" type="ProductType"/>

<xsd:complexType name="ShirtType">
  <xsd:complexContent>
    <xsd:extension base="ProductType">
      <xsd:choice maxOccurs="unbounded">
        <xsd:element name="size" type="SizeType"/>
        <xsd:element name="color" type="ColorType"/>
      </xsd:choice>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Beispielinstanz

```
<items xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <product xsi:type="ShirtType">
    <number>557</number>
    <name>Short-Sleeved Linen Blouse</name>
    <color value="blue"/>
  </product>

  <!--...-->
</items>
```

(14.7) Controlling type derivation and substitution

- `final` schränkt Typableitung im Schema ein
- `block` schränkt Substituierbarkeit ein
- `abstract` erzwingt vor Benutzung Typableitung

(14.7) Controlling type derivation and substitution

Werte für `final` oder `block`

- `"extension"` verhindert Erweiterung
- `"restriction` verhindert Restriktion
- `"#all` verhindert beides
- `""` keine Einschränkungen, überschreibt `finalDefault` bzw. `blockDefault`

Beispiel: Verhinderung von Typableitung

```
<xsd:complexType name="ProductType" final="#all">
  <xsd:sequence>
    <xsd:element name="number" type="ProdNumType"/>
    <xsd:element name="name" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
```

Beispiel: Verhinderung von Substituierbarkeit

```
<xsd:complexType name="ProductType" block="extension">
  <xsd:sequence>
    <xsd:element name="number" type="ProdNumType"/>
    <xsd:element name="name" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:element name="product" type="ProductType"/>

<xsd:complexType name="ShirtType">
  <xsd:complexContent>
    <xsd:extension base="ProductType">
      <xsd:choice maxOccurs="unbounded">
        <xsd:element name="size" type="SizeType"/>
        <xsd:element name="color" type="ColorType"/>
      </xsd:choice>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="shirt" type="ShirtType"/>
```

Beispiel: abstrakter komplexer Typ

```
<xsd:complexType name="ProductType" abstract="true">
  <xsd:sequence>
    <xsd:element name="number" type="ProdNumType"/>
    <xsd:element name="name" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:element name="product" type="ProductType"/>

<xsd:complexType name="ShirtType">
  <xsd:complexContent>
    <xsd:extension base="ProductType">
      <xsd:choice maxOccurs="unbounded">
        <xsd:element name="size" type="SizeType"/>
        <xsd:element name="color" type="ColorType"/>
      </xsd:choice>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
<xsd:element name="shirt" type="ShirtType"/>
```

(15) Resusable Groups

Es gibt wiederverwendbare Gruppen von

- Elementen
- Attributen

(15.1) Wozu wiederverwendbare Gruppen?

- Konsistenz über Schemakomponenten hinweg
- Vereinfachte Wartbarkeit (Konzentration der Änderungen auf wenige Stellen)
- Explizite Kennzeichnung der Wiederverwendung
- Schema kürzer halten
- Geringerer Schemaentwicklungsaufwand

(15.2) Benannte Modellgruppen

Eine benannte Modellgruppe (named model group)

- ist ein wiederverwendbares Fragment eines Inhaltsmodells.
- kann keine Attribute enthalten.

(15.2) Benannte Modellgruppen: Definition

Name: `group`
Parents: `schema`, `redefine`

Attribute	Type	Required/default	Description
`id`	`ID`		unique ID
`name`	`NCName`	required	name of named model group

Content:

```
annotation?, (all | choice | sequence)
```

(15.2) Beispiel mit lokalen Elementdeklarationen

```
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema">
  <xsd:group name = "DescriptionGroup">
    <xsd:sequence>
      <xsd:element name = "description" type = "xsd:string"/>
      <xsd:element name = "comment" type = "xsd:string"
                   minOccurs = "0"/>
    </xsd:sequence>
  </xsd:group>
</xsd:schema>
```

(15.2) Beispiel mit Elementreferenzen

```
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema">

  <xsd:element name = "description" type = "xsd:string"/>
  <xsd:element name = "comment" type = "xsd:string"/>

  <xsd:group name = "DescriptionGroup">
    <xsd:sequence>
      <xsd:element ref = "description"/>
      <xsd:element ref = "comment" minOccurs = "0"/>
    </xsd:sequence>
  </xsd:group>
</xsd:schema>
```

(15.2) Referenzieren einer benannten Modellgruppe

Name: `group`
Parents: `complexType`, `restriction`, `extension`, `choice`, `sequence`

Attribute	Type	Required/default	Description
`id`	`ID`		unique ID
`ref`	`QName`	required	name of referenced group
`minOccurs`	nonNegativeInteger	1	wie gehabt
`maxOccurs`	nonNegativeInteger | `unbounded`	1	wie gehabt

Content: `annotation?`

(15.2) Beispiel

```
<xsd:complexType name = "PurchaseOrder">
  <xsd:sequence>
    <xsd:group ref = "DescriptionGroup" minOccurs = "0"/>
    <xsd:element ref = "items"/>
    <!-- ... -->
  </xsd:sequence>
</xsd:complexType>
```

(15.2) Referenzieren einer benannten Modellgruppe

Eine benannte Modellgruppe kann auch

- als top-level Element von `complexType` vorkommen

```
<xsd:complexType name = "DescriptionType">
  <xsd:group ref = "DescriptionGroup"/>
  <xsd:attribute ref = "xml:lang"/>
</xsd:complexType>
```

- Eine `all`-Gruppe darf nur Elementdeklarationen und -referenzen enthalten
- Eine `all`-Gruppe darf keine benannte Modellgruppe referenzieren
- Da eine `all`-Gruppe nur als top-level Element vorkommen kann, kann eine benannte Modellgruppe, die eine `all`-Gruppe enthält nur wie im vorangegangenen Beispiel als top-level Element vorkommen.

(15.2) Referenzieren einer benannten Modellgruppe

Benannte Modellgruppen dürfen andere referenzieren:

```
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema">
  <xsd:group name = "ProductPropertyGroup">
    <xsd:sequence>
      <xsd:group ref = "DescriptionGroup"/>
      <xsd:element name = "number" type = "ProdNumType"/>
      <xsd:element name = "name" type = "xsd:string"/>
    </xsd:sequence>
  </xsd:group>
```

(15.3) Attributgruppen

Name: `attributeGroup`
Parents: `schema`, `redefine`

Attribute	Type	Required/default	Description
`id`	`ID`		unique ID
`name`	`NCName`	required	attribute group name

Contents:

```
annotation?, (attribute | attributeGroup)*, anyAttribute?
```

(15.3) Attributgruppe mit lokalen Attributdeklarationen

```
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema">
  <xsd:attributeGroup name = "IdentifierGroup">
    <xsd:attribute name = "id" type = "xsd:ID"
                   use = "required"/>
    <xsd:attribute name = "version" type = "xsd:decimal"/>
  </xsd:attributeGroup>
</xsd:schema>
```

(15.3) Attributgruppe mit Attributreferenzen

```
\begin{verbatim}
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema">

  <xsd:attribute name = "id" type = "xsd:ID"/>
  <xsd:attribute name = "version" type = "xsd:decimal"/>

  <xsd:attributeGroup name = "IdentifierGroup">
    <xsd:attribute ref = "id" use = "required"/>
    <xsd:attribute ref = "version"/>
  </xsd:attributeGroup>

</xsd:schema>
```

(15.3) Referenzieren einer Attributgruppe

Name: `attributeGroup`

Parents: `complexType`, `restriction`, `extension`, `attributeGroup`

Attribute	Type	Required/default	Description
`id`	`ID`		unique ID
`name`	`QName`	required	attribute group being referenced

Content: `annotatation?`

(15.3) Beispiel

```
<xsd:complexType name = "ProductType">
  <xsd:sequence>
    <!-- ... -->
  </xsd:sequence>
  <xsd:attributeGroup ref = "IdentifierGroup"/>
  <xsd:attribute name = "effDate" type = "xsd:date"/>
</xsd:complexType>
```

(15.4) Wiederverwendbare Gruppen vs. Typableitung

Benutze Gruppen, falls:

- die Elemente nicht als erstes im Inhaltsmodell vorkommen
- Typen sehr unterschiedliche Semantik haben, aber trotzdem gemeinsame Konzepte beinhalten

Benutze Typableitung falls

- Elemente als erstes im Inhaltsmodell vorkommen
- Inhaltsmodell überschneidet sich stark

(16) Substitutionsgruppen

- Elemente als substituierbar für andere Deklarieren
- Bsp.: verschiedene Elemente für verschiedene Produkte
- Jede Substitutionsgruppe besteht aus Kopf (head) und Mitgliedern (member)
- Kopf muss global deklariert sein
- Jedes Mitglied kann nur Mitglied *einer* Substitutionsgruppe sein
- Aber Mitglieder können selbst wieder Kopf weiterer Substitutionsgruppen sein
- Ergibt Hierachie

(16.3) Deklaration von Substitutionsgruppen

Kopf einer Substitutionsgruppe (als solches nicht ausgezeichnet):

```
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema">
  <xsd:element name = "items" type = "ItemsType"/>
  <xsd:complexType name = "ItemsType">
    <xsd:sequence>
      <xsd:element ref = "product" maxOccurs = "unbounded"/>
    </xsd:sequence>
  <xsd:element name = "product" type = "ProductType">
  <xsd:complexType name = "ProductType">
    <xsd:sequence>
      <xsd:element ref = "number"/>
      <xsd:element ref = "name"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

(16.3) Deklaration von Substitutionsgruppen: Member

```
<xsd:schema xmlns:xsd = "http://www.w3.org/2001/XMLSchema">

  <xsd:element name = "shirt" type = "ShirtType"
               substitutiongroup = "product"/>

  <xsd:complexType name = "ShirtType">
    <xsd:complexContent>
      <xsd:extension base = "ProductType">
        <xsd:sequence>
          <xsd:element name = "size" type = "ShirtSizeType"/>
          <xsd:element name = "color" type = "ColorType"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <!-- weiter auf naechster Folie -->
```

```
<!-- Fortsetzung -->

<xsd:element name = "hat" substitutionGroup = "product">
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:extension base = "ProductType">
        <xsd:sequence>
          <xsd:element name = "size" type = "HatSizeType"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>

<xsd:element name = "umbrella" substitutionGroup = "product/

<!-- ... -->

</xsd:schema>
```

(16.4) Typebedingungen für Substitutionsgruppen

- Der Typ der Mitglieder muss entweder
  - vom Typ der Kopfgruppe sein oder
  - vom Typ der Kopfgruppe durch Restriktion oder Erweiterung abgeleitet sein.

  Die Ableitung kann direkt oder indirekt sein.
- Falls für ein Mitglied kein Typ angegeben ist, so wird automatisch der Typ des Kopfelements eingesetzt.
- Es können auch Substitutionsgruppen für Elemente von einfachem Typ gebildet werden.

(16.5) Alternativen zu Substitutionsgruppen

- `choice`-Gruppen
  - Es gibt Ähnlichkeit mit `choice`-Gruppen
  - `choice`-Gruppen können aber nicht ohne weiteres um weitere Alternative erweitert werden
- Abgeleitete Typen
  - bei abgeleiteten Typen: `xsi:type` zwingend
  - bei Substitutionsgruppen: `xsi:type` optional

(16.6) Kontrolle von Substitutionsgruppen

Kontrolle durch folgende Attribute:

- `final`: begrenzt Ableitbarkeit
- `block`: blockiert Substituierbarkeit
- `abstract`: erzwingt Ableitung

(16.6) `final`

Mögliche Werte von `final`

- `"#all"`: Element kann nicht als Kopf vorkommen
- `"extension"`: Typableitung durch Erweiterung verboten
- `"restriction"`: Typableitung durch Restriktion verboten
- `"extension" "restriction"`: beides
- `""`: aufheben von `finalDefault`

(16.6) `block`

- Werte analog zu `final`
- blockiert aber die Substituierbarkeit auf Instanzebene

(16.6) `abstract`

- Element hat `abstract="true"`: es dürfen keine Instanzen gebildet werden
- kann dann noch als Kopf einer Substitutionsgruppe dienen

(17) Eindeutigkeitsbedingungen und Referenzielle Integrität

- uniqueness constraint: erzwinge Eindeutigkeit von Werten
- key constraint: erzwinge Eindeutigkeit von Werten und deren Anwesenheit
  (in RDB: alle Attribute != NULL)
- key references: referenzielle Integrität

(17.2) ID/IDREF vs. key/keyref

- ID funktioniert nur bei Attributen
- scope ist ganzes Dokument, nicht nur Elemente einer Sorte
- ID beschränkt auf einzelnes Attribut, nicht mehrere Attribute oder gar Elemente
- ID basiert auf string-Gleichheit, nicht auf Wertgleichheit
- ID-Werte basieren auf XML names (also eingeschränkter Zeichenvorrat)

(17.3) Struktur von Identitätsbedingungen

- Identitätsbedingungen werden am Ende des Inhaltsmodells angegeben
- Jede Identitätsbedingung hat einen Namen (berücksichtigt `targetNamespace`
- Identitätsbedingungsnamen müssen eindeutig sein (im gesamten Schema)
- Identitätsbedingungen bestehen aus 3 Teilen:
  1. scope: Element, dessen Deklaration die Identitätsbedingung enthält (e.g. `catalog`)
  2. selector: XPath-Ausdrücke, die Gültigkeitsbereich identifizieren (e.g. `*/product`)
  3. fields: Schlüsselkomponenten (e.g. `number`)

Unter `catalog` identifiziert `number` ein `*/product` eindeutig.

(17.4) Uniqueness constraints

Name: `unique`

Parent: `element`

Attribute	Type	Required/default	Description
`id`	`ID`		unique ID
`name`	`NCName`	unique name	

Content:

```
annotation?, selector, field+
```

(17.4) Beispiel

```
<xsd:element name = "catalog" type = "CatalogType">
  <xsd:unique name = "dateAndProdNumKey">
    <xsd:selector xpath = "*/product"/>
    <xsd:field xpath = "number"/>
    <xsd:field xpath = "@effDate"/>
  </xsd:unique>
</xsd:element>
```

(17.5) Key constraints

Wie uniqueness constraint plus

- alle Felder müssen vorhanden und definiert sein

Also

- keine optionalen Elemente oder Attribute
- keine nillable Elemente

(17.5) Key constraints

Name: `key`

Parent: `element`

Attribute	Type	Required/default	Description
`id`	`ID`		unique ID
`name`	`NCName`	unique name	

Content:

```
annotation?, selector, field+
```

(17.5) Beispiel

```
<xsd:element name = "catalog" type = "CatalogType">
  <xsd:key name = "prodNumKey">
    <xsd:selector xpath = "*/product"/>
    <xsd:field xpath = "number"/>
  </xsd:unique>
</xsd:element>
```

(17.6) Key references

- Werte müssen da sein (wie foreign key references in RDB)
- der `key`, der durch `keyref` referenziert wird muß
  - im gleichen Element definiert werden
  - oder im Kindelement

(17.6) Beispiel

```
<xsd:element name = "order" type = "OrderType">
  <xsd:keyref name = "prodNumKeyRef" refer = "prodNumKey">
    <xsd:selector xpath = "items/*"/>
    <xsd:field xpath = "@number"/>
  </xsd:keyref>
  <xsd:key name = "prodNumKey">
    <xsd:selector xpth = ".//product"/>
    <xsd:field xpath = "number"/>
  </xsd:key>
</xsd:element>
```

(17.8) XPath Einschränkungen

Nur erlaubt: Beginne mit

- '.'
- './/'

Danach nur noch child-steps

(20.2) Naming considerations: Qualified vs. unqualified names

Falls die Deklaration eines Elements

- global ist: im Dokument muss es qualifiziert vorkommen
  - Dokumentersteller muß wissen in welchem Namespace welches Element vorkommt
- lokal ist: Wahl, ob es im Dokument qualifiziert oder unqualifiziert vorkommen soll

Zunächst zwei Beispiele:

(20.2) Qualifizierte lokale Namen

```
<ord:order xmlns:ord = "http:example.org/ord">
           xmlns:prod = "http:example.org/prod">
  <ord:number>123ABBCC123</number>
  <ord:items>
    <prod:product>
      <prod:number>557</number>
    </prod:product>
  </ord:items>
</ord:order>
```

(20.2) Unqualifizierte lokale Namen

```
<ord:order xmlns:ord = "http:example.org/ord">
  <number>123ABBCC123</number>
  <items>
    <product>
      <number>557</number>
    </product>
  </items>
</ord:order>
```

(20.2) Diskussion

- lokal unqualifiziert sieht weniger kompliziert aus
- Benutzer muss nicht wissen welchem namespace was zugeordnet ist
- Benutzer muss nicht mal wissen, dass es den `prod` namespace überhaupt gibt

(20.2) `elementFormDefault`

- `elementFormDefault`-Attribut in `schema` steuert default
- hat selbst als default `unqualified`
- wird durch `form`-Attribut auf Elementebene überschrieben

(20.2) Qualifiziert vs. unqualifiziert lokale Namen

- Mixen von globalen und lokalen unqualified Elementdeklarationen: Benutzer muß wissen, was global und was lokal ist
- Default-namespaces und unqualified passen nicht zusammen!

(20.2) Qualifiziert vs. unqualifiziert lokale Namen

Vorteil qualifizierte lokale Namen:

- Namensraum in Dokument direkt ersichtlich
- keine Mehrdeutigkeiten von Namensraumzugehörigkeiten
- Mix von globalen und lokalen Elementen funktioniert gut: alles qualifiziert

Vorteil unqualifizierte lokale Namen:

- Dokumentautor muß die Namensräume nicht kennen
- Dokument sieht einfacher aus (ohne die ganzen Präfixe)

Generell: qualfizierte Namen vorziehen

(20.2) Qualifizierte vs. unqualifizierte Attributnamen

- globale Attribute: immer qualifiziert
  (default namespace greift nicht!)
- d.h.: qualified vs. unqualified = mit oder ohne Präfix

globale Attribute nur dann verwenden, wenn Sie für viele Elemente Sinn machen!
sonst: beste Möglichkeit:

- ignoriere `attributeFormDefault` und `form`

(20.3) Strukturieren von Namensräumen

Beispiel: product, customer, order
Alternativen:

- alle Deklarationen in den gleichen Namensraum (zusammensetzen geht dann mit `include`)
- alle Deklarationen in unterschiedlichen Namensräumen (zusammensetzen geht dann mit `import`)
- Chameleon Namensräume
  (zur Erinnerung: `include` sorgt für entsprechenden `target namespace`)

End Of XML Schema

XML Linking Language (XLink)

XLink verallgemeinert die bekannten Hyperlinks. XLink erlaubt

- das Verbinden von mehr als zwei Ressourcen,
- das Assoziieren von Metadaten mit einer Verbindung und
- das Ausdrücken von Verbdindungen außerhalb der verbundenen Ressourcen.

Hyperlinks

- Ein Hyperlink benutzt URIs zur Lokalisierung.
- Ein Hyperlink ist an einem Ende der Verbindung spezifiziert.
- Ein Hyperlink identifiziert das andere Ende.
- Benutzer können einen Hyperlink von der Quelle zum anderen Ende traversieren.

Diese Eigenschaften sind sehr mächtig, aber eine Verallgemeinerung der letzten drei Punkte kann ebenfalls nützlich sein. Dies geschieht in XLink.

Konzepte von XLink

resource Adressierbare Einheit (Information oder Service; bsp: XML-Fragment)

link repräsentiert eine explizite Beziehung zwischen (Teil-) Ressourcen

participate Durch einen Link in Beziehung gesetzte Ressourcen *nehmen* an einen Link *teil*.

traversal Benutzen oder Verfolgen eines Links;
Traversieren von Startressource zu Endressource

arc Information die angibt, wie ein Link traversiert werden kann, bzw. was in einem solchen Fall passiert wird durch eine Kante (arc) spezifiziert.
Bidirektionaler Link: zwei Kanten

in/outbound arc Je nachdem, ob die Start- oder Endressource remote bzw. in der jeweils anderen Ressource liegt.

third-party Falls weder das eine noch das andere zutrifft.

linkbase Dokumente, die viele inbound und third-party links haben heißen *link databases*.

XLink namespace http://w3.org/1999/xlink. Wir gehen davon aus, dass

Globale Attribute

XLink stellt folgende globalen Attribute zur Verfügung: `type`, `href`, `role`, `arcrole`, `title`, `show`, `actuate`, `label`, `from` und `to`. Das Attribut `type` spezifiziert mittels der fest vorgegebenen Werte `simple`, `extended`, `locator`, `arc`, `resource`, `title` welche Bedingungen erfüllt sein müssen (constraints). Darunter fällt unter anderem, welche anderen Attribute vorhanden sein müssen/können.

Einfaches Beispiel

Folgendes Beispiel modelliert einen einfachen Hyperlink:

```
<novel xlink:type = "simple"
       xlink:href = "ftp://archive.org/pub/etext/wizozoz.pdf" >
       <author>L. Frank Baum</author>
       <title>The Wonderful Wizard of Oz</title>
       <year>1900</year>
</novel>
```

Attributbenutzung

Welche Attribute bei welchen *type* vorhanden sein müssen (R) oder können (O):

	simple	extended	locator	arc	resource	title
type	R	R	R	R	R	R
href	O		R			
role	O	O	O		O	
arcrole	O			O		
title	O	O	O	O	O	
show	O			O		
actuate	O			O		
label			O		O	
from				O		
to				O		

xxx-type Element und Kindsignifikanz

Falls ein Element ein Attribute `xlink:type` mit einem Wert `xxx` hat, so spricht man auch von einem xxx-type Element.
Ein yyy-type Kindelement kann bestimmte Relevanz für ein xxx-type Elternelement haben. Die Fälle:

parent	significant child types
simple	none
extended	locator, arc, resource, title
locator	title
arc	title
resource	none
title	none

Simple and Extended Links

XLink unterstützt *zwei* Arten von Verbindungen:

extended links Unterstützen die volle Funktionalität

simple links Spezialfall für Hyperlinks.

Anmerkung: Die anderen möglichen Werte des `xlink:type`-Attributs werden nicht für die Unterscheidung weiterer Link-Typen herangezogen.

Extended Links

Ein *extended link* setzt eine beliebige Anzahl von Ressource in Beziehung zueinander. Dabei ist es egal, ob die Ressourcen remote oder local sind. Ein Extended-link-Element kann folgende Elemente in beliebiger Ordnung und gemischt mit anderem Inhalt enthalten:

- locator-type elements address remote resources
- resource-type elements address local resources
- arc-type elements provide traversal rules among resources
- title-type elements provide human-readable labels for the link

simple-type und extended-type Elemente mit extended-type Elternelement haben keine XLink-spezifische Bedeutung. locator-, arc- und resource-Elemente haben keine Bedeutung, falls das Elternelement nicht ein extended-type Element ist.
Falls ein extended-type Element ein title-Attribut hat, so gibt dieses eine Beschreibung des gesamten Links an; das role-Attribut deutet eine Eigenschaft des gesamten Links an.

Beispiel

Es werden Personen mit Kursen und anderen Informationen verlinkt. Dabei kann eine Person verschiedenen Rollen annehmen, zum Beispiel Betreuer.

Beispiel DTD

```
<!ELEMENT courseload ((tooltip|person|course|gpa|go)*)>
<!ATTLIST courseload
  xmlns:xlink     CDATA           #FIXED "http://www.w3.org/19
  xlink:type      (extended)      #FIXED "extended"
  xlink:role      CDATA           #IMPLIED
  xlink:title     CDATA           #IMPLIED>

<!ELEMENT tooltip ANY>
<!ATTLIST tooltip
  xlink:type      (title)         #FIXED "title"
  xml:lang        CDATA           #IMPLIED>
```

```
<!ELEMENT person EMPTY>
<!ATTLIST person
  xlink:type      (locator)  #FIXED "locator"
  xlink:href      CDATA      #REQUIRED
  xlink:role      CDATA      #IMPLIED
  xlink:title     CDATA      #IMPLIED
  xlink:label     NMTOKEN    #IMPLIED>

<!ELEMENT course EMPTY>
<!ATTLIST course
  xlink:type   (locator) #FIXED "locator"
  xlink:href   CDATA   #REQUIRED
  xlink:role  CDATA #FIXED "http://www.example.com/linkprops/c
  xlink:title  CDATA   #IMPLIED
  xlink:label  NMTOKEN #IMPLIED>
```

```
<!-- GPA = "grade point average" -->
<!ELEMENT gpa ANY>
<!ATTLIST gpa
  xlink:type   (resource) #FIXED "resource"
  xlink:role   CDATA  #FIXED "http://www.example.com/linkprops
  xlink:title  CDATA      #IMPLIED
  xlink:label  NMTOKEN    #IMPLIED>
```

```
<!ELEMENT go EMPTY>
<!ATTLIST go
  xlink:type      (arc)          #FIXED "arc"
  xlink:arcrole   CDATA          #IMPLIED
  xlink:title     CDATA          #IMPLIED
  xlink:show      (new
                  |replace
                  |embed
                  |other
                  |none)         #IMPLIED
  xlink:actuate   (onLoad
                  |onRequest
                  |other
                  |none)         #IMPLIED
  xlink:from      NMTOKEN        #IMPLIED
  xlink:to        NMTOKEN        #IMPLIED>
```

Beispiel Dokument

```
<courseload>

  <tooltip>Course Load for Pat Jones</tooltip>
```

```
<person
  xlink:href="students/patjones62.xml"
  xlink:label="student62"
  xlink:role="http://www.example.com/linkprops/student"
  xlink:title="Pat Jones" />

<person
  xlink:href="profs/jaysmith7.xml"
  xlink:label="prof7"
  xlink:role="http://www.example.com/linkprops/professor"
  xlink:title="Dr. Jay Smith" />
<!-- more remote resources for profs, teaching assistants, e
```

```
<course
  xlink:href="courses/cs101.xml"
  xlink:label="CS-101"
  xlink:title="Computer Science 101" />
<!-- more remote resources for courses, seminars, etc. -->

<gpa xlink:label="PatJonesGPA">3.5</gpa>
```

```
<go
  xlink:from="student62"
  xlink:to="PatJonesGPA"
  xlink:show="new"
  xlink:actuate="onRequest"
  xlink:title="Pat Jones's GPA" />
<go
  xlink:from="CS-101"
  xlink:arcrole="http://www.example.com/linkprops/auditor"
  xlink:to="student62"
  xlink:show="replace"
  xlink:actuate="onRequest"
  xlink:title="Pat Jones, auditing the course" />
```

```
  <go
    xlink:from="student62"
    xlink:arcrole="http://www.example.com/linkprops/advisor"
    xlink:to="prof7"
    xlink:show="replace"
    xlink:actuate="onRequest"
    xlink:title="Dr. Jay Smith, advisor" />

</courseload>
```

resources

- Die resource-type Elemente unter einem extended-link-type Element zählen die an der Beziehung (Link) teilnehmenden lokalen Ressourcen auf.
- Die locator-type Elemente unter einem extended-link-type Element zählen die an der Beziehung (Link) teilnehmenden remote Ressourcen auf.

Arc

arc-type Elemente unter einem extended-link Element geben die Traversierungsregeln vor.
Offensichtlich werden die Traversierungsattribute `to` und `from` hierzu benutzt. Eingetragen werden hier die `label`-Attributwerte der Ressourcen.
Da `label` von Ressourcen nicht eindeutig sein müssen kann eine arc-Spezifikation mehrere gerichtete Traversierungskanten implizieren.
Falls für `to` oder `from` kein Wert spezifiziert wurde, so heißt dies *alle Ressourcen*!
Falls kein arc-type Element spezifiziert wird, so enspricht dies einem arc-Element ohne `to` und `from`.
Doppelte Kanten sind nicht erlaubt.

locating linkbases

`arcrole` Attribut mit Wert
http://www.w3.org/1999/xlink/properties/linkbase

Simple Link

- Setzt lokale und nicht-lokale Ressource in Beziehung zueinander.
- Führt Kante (arc) von lokaler zu nicht-lokaler Ressource ein.
- Falls kein `href` vorhanden ist, so ist dies kein Fehler, der Link ist schlichtweg nur nicht traversierbar.

Attributwerte

href	URI, XPtr, relative und Auflösung mittels XML Base
role	URI
arcrole	URI
title	string
show	eins von `new`, `replace`, `embed`, `other`, `none`
actuate	eins von `onLoad`, `onRequest`, `other`, `none`

ps: XPtr: URI ∘ '#' ∘ 'xpointer(' XPath-Expr ')'

XML Include

XML Include spezifiziert eine Möglichkeit ein XML-Dokument in ein anderes zu inkludieren.
Das zu inkludierende Dokument wird in einem `include`-Element im inkludierenden Dokument mittels des Attributes `href` angegeben.
Ein zusätzlicher XLink erlaubt es nur Teile zu inkludieren.
Die Inklusion selbst erfolgt durch einklinken des Infosets des inkludierten (Teil-) Dokuments in das Infoset des inkludierenden Dokuments.

APIs for XML

- Wiederverwendbarkeit von Parsern erzwingt eindeutige Repräsentation von XML Dokumenten.
- Auch für andere Anwendungen ist eine solche Repräsentation hilfreich.
- Für XML Dokumente gibt es zwei weitverbreitete APIs:
  1. Document Object Model (DOM)
     Baumrepräsentation
  2. Simple API for XML (SAX)
     Event-Stream-Repräsentation

DOM

- Menge von Schnittstellen (Interfaces in IDL) für Knotentypen
- Zwei Sichtweisen
  - Generischer Knotentyp (Node)
  - Spezifische Knotentypen (Vererbungshierarchie)
- bei Parsern: ganzes Dokument wird geparsed und der ganze Dokumentbaum in Hauptspeicher aufgebaut.
- Erlaubt lesenden und modifizierenden Zugriff auf Dokumente
- Erzeugung von Dokumenten und -knoten durch Factory

DOM Level 1, 2, 3

Kindknotentypen

Document	Element (≤ 1), ProcessingInstruction, Comment
DocumentType	keine Kindknoten
EntityReference	Element, X
Element	Element, X
Attribute	Text, EntityReference
ProcessingInstruction	keine Kindknoten
Comment	keine Kindknoten
Text	keine Kindknoten
CDATASection	keine Kindknoten
Entity	Element, X
Notation	keine Kindknoten

X = ProcessingInstruction, Comment, Text, CDATASection, EntityReference

Weitere Interfaces

- NodeList (geordnete Liste von Nodes)
- NamedNodeMap (ungeordnete Menge von Nodes, referenziert durch Name-Attribut)
- Basisdatentypen (DOMString, DOMTimeStamp, DOMException)
- DOMImplementation als Factory für Dokumente
- Document ist Factory für andere Knotentypen
- DocumentType (Zugriff auf Entities und Notations, noch unausgegoren da XML Schema Einfluß ungeklärt)

Interfaces: DOMImplementation

```
interface DOMImplementation {
  boolean      hasFeature(in DOMString feature,
                          in DOMString version);
  // Introduced in DOM Level 2:
  DocumentType createDocumentType(in DOMString qualifiedName,
                                  in DOMString publicId,
                                  in DOMString systemId)
                                  raises(DOMException);
  // Introduced in DOM Level 2:
  Document     createDocument(in DOMString namespaceURI,
                              in DOMString qualifiedName,
                              in DocumentType doctype)
                              raises(DOMException);
};
```

Interfaces: Document

```
interface Document : Node {
  readonly attribute DocumentType     doctype;
  readonly attribute DOMImplementation  implementation;
  readonly attribute Element          documentElement;
  Element             createElement(in DOMString tagName)
                                        raises(DOMException);
  DocumentFragment    createDocumentFragment();
  Text                createTextNode(in DOMString data);
  Comment             createComment(in DOMString data);
  CDATASection        createCDATASection(in DOMString data)
                                        raises(DOMException);
  ProcessingInstruction
  createProcessingInstruction(in DOMString target,
                              in DOMString data)
                              raises(DOMException);
  Attr                createAttribute(in DOMString name)
                                       raises(DOMException);
  EntityReference     createEntityReference(in DOMString name)
```

Interfaces: Node

```
interface Node {
  // NodeTypes
  const unsigned short ELEMENT_NODE                = 1;
  const unsigned short ATTRIBUTE_NODE              = 2;
  const unsigned short TEXT_NODE                   = 3;
  const unsigned short CDATA_SECTION_NODE          = 4;
  const unsigned short ENTITY_REFERENCE_NODE       = 5;
  const unsigned short ENTITY_NODE                 = 6;
  const unsigned short PROCESSING_INSTRUCTION_NODE = 7;
  const unsigned short COMMENT_NODE                = 8;
  const unsigned short DOCUMENT_NODE               = 9;
  const unsigned short DOCUMENT_TYPE_NODE          = 10;
  const unsigned short DOCUMENT_FRAGMENT_NODE      = 11;
  const unsigned short NOTATION_NODE               = 12;

  readonly attribute DOMString        nodeName;
           attribute DOMString        nodeValue;
```

Interfaces: NodeList

```
interface NodeList {
  Node                  item(in unsigned long index);
  readonly attribute unsigned long    length;
};
```

Interfaces NamedNodeMap

```
interface NamedNodeMap {
  Node   getNamedItem(in DOMString name);
  Node   setNamedItem(in Node arg) raises(DOMException);
  Node   removeNamedItem(in DOMString name) raises(DOMExceptio
  Node   item(in unsigned long index);
  readonly attribute unsigned long    length;
  // Introduced in DOM Level 2:
  Node   getNamedItemNS(in DOMString namespaceURI,
                        in DOMString localName);
  Node   setNamedItemNS(in Node arg) raises(DOMException);
  Node   removeNamedItemNS(in DOMString namespaceURI,
                           in DOMString localName) raises(DOME
};
```

Interfaces: CharacterData

```
interface CharacterData : Node {
  attribute DOMString        data;
  readonly attribute unsigned long    length;
  DOMString   substringData(in unsigned long offset,
                            in unsigned long count)
                            raises(DOMException);
  void        appendData(in DOMString arg)
                         raises(DOMException);
  void        insertData(in unsigned long offset,
                         in DOMString arg)
                         raises(DOMException);
  void        deleteData(in unsigned long offset,
                         in unsigned long count)
                         raises(DOMException);
  void        replaceData(in unsigned long offset,
                          in unsigned long count,
                          in DOMString arg)
                          raises(DOMException);
```

Interfaces: Attr

```
interface Attr : Node {
  readonly attribute DOMString        name;
  readonly attribute boolean          specified;
           attribute DOMString        value;
                                        // raises(DOMExceptior

  // Introduced in DOM Level 2:
  readonly attribute Element          ownerElement;
};
```

Interfaces: Element

```
interface Element : Node {
  readonly attribute DOMString  tagName;
  DOMString  getAttribute(in DOMString name);
  void       setAttribute(in DOMString name,
                          in DOMString value)
                                raises(DOMException);
  void       removeAttribute(in DOMString name)
                                raises(DOMException);
  Attr       getAttributeNode(in DOMString name);
  Attr       setAttributeNode(in Attr newAttr)
                                raises(DOMException);
  Attr       removeAttributeNode(in Attr oldAttr)
                                raises(DOMException);
  NodeList   getElementsByTagName(in DOMString name);
  // Introduced in DOM Level 2:
  DOMString  getAttributeNS(in DOMString namespaceURI,
                            in DOMString localName);
  void       setAttributeNS(in DOMString namespaceURI,
```

SAX

- Sequenz von Ereignissen
  startDocument, processingInstruction, startPrefixMapping, startElement, ignorableWhitespace, ...
- streng genommen muss nur aktuelles Ereignis im Hauptspeicher sein
- nur lesender Zugriff

SAX ContentHandler

```
interface ContentHandler {
  void setDocumentLocator(Locator locator);
  void startDocument();
  void endDocument();
  void startPrefixMapping(String prefix, String uri);
  void endPrefixMapping(String prefix);
  void startElement(String namespaceURI, String localName,
                    String qualifiedName, Attributes atts);
  void endElement(String namespaceURI, String localName,
                  String qualifiedName);
  void characters (char[] text, int start, int length);
  void ignorableWhitespace(char[] text, int start, int length)
  void processingInstruction(String target, String data);
  void skippedEntity(String name);
};
```

Extensible Stylesheet Language (XSL)

XSL Transformations (XSLT)

- Transformationssprache, um Transformationen von Dokumenten in Dokumente spezifizieren zu können.
- Eine Transformation wird in XSLT als ein Stylesheet spezifiziert.
- Ein Stylesheet besteht u.a. aus einer Sequenz von Regeln.
- Jede Regel bestehen aus einem Muster (Pattern), das Knoten im Urdokument bestimmt, und einem Template, das durch Instanziierung zu einem Teil des Ergebnisdokumentes wird.
- Wenn ein Template instanziiert wird, wird jede Instruktion des Templates ausgeführt. Deren Ergebnisse werden konkateniert und ergeben so einen Teil des Ergebnisdokuments.

Beispiel: DTD

```
<!ELEMENT world (country*)>
<!ATTLIST world id ID #REQUIRED>

<!ELEMENT country (city*)>
<!ATTLIST country
  id   ID #REQUIRED
  name CDATA #REQUIRED>

<!ELEMENT city EMPTY>
<!ATTLIST city
  id   ID #REQUIRED
  name CDATA #REQUIRED>
```

Beispiel: Dokument

```
<?xml version="1.0"?>
<!DOCTYPE world SYSTEM "world.dtd">
<world id="1">
  <country id="2" name="Germany">
    <city id="21" name="Berlin"/>
    <city id="22" name="Bonn"/>
  </country>
  <country id="3" name="France">
    <city id="31" name="Paris"/>
    <city id="32" name="Sanary"/>
  </country>
  <country id="4" name="Italy">
    <city id="41" name="Roma"/>
    <city id="42" name="Milano"/>
  </country>
</world>
```

Beispiel: Stylesheet

```
<?xml version="1.0"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform
  version="1.0">
```

```
<xsl:template match="/world">
  <world>
  <xsl:apply-templates select="country"/>
  </world>
</xsl:template>

<xsl:template match="country">
  <country>
  <name>
    <xsl:value-of select="@name"/>
  </name>
  <xsl:apply-templates select="city"/>
  </country>
</xsl:template>
```

```
<xsl:template match="city">
  <city>
    <xsl:value-of select="@name"/>
  </city>
</xsl:template>

</xsl:stylesheet>
```

Beispiel: Ergebnisdokument

```
<?xml version="1.0"?>
<world>
  <country>
    <name>Germany</name>
    <city>Berlin</city>
    <city>Bonn</city>
  </country>
  <country>
    <name>France</name>
    <city>Paris</city>
    <city>Sanary</city>
  </country>
  <country>
    <name>Italy</name>
    <city>Roma</city>
    <city>Milano</city>
  </country>
</world>
```

Prioritäten

1. Falls das Attribut xsl:priority einen Wert hat, so ist dies die Priorität der Regel.
2. Ein pattern der Form 'name' oder '@name' hat die Default-Priorität 0.
3. Ein Pattern, das ein Knotentest ist, also auf Element- oder Attributknoten testet, hat die Default-Priorität -0.5.
4. In allen anderen Fällen ist die Default-Priorität 0.5.

Default Regeln

```
<xsl:template match = "*|/">
      <xsl:apply-templates/>
</xsl:template>

<xsl:template match = "text()|@*">
      <xsl:value-of select = "."/>
```

Weitere Konstrukte für Templates

- xsl:if
- xsl:choose
- xsl:value-of
- xsl:forach
- xsl:apply-templates

Und vieles mehr.

Ausführungsmodell

- Die Ausführung findet immer unter einem bestimmten *current node* und einer *current node list* statt.
- Zu Anfang ist der Wurzelknoten der current node und die current node list besteht nur aus dem Wurzelknoten.
- Es wird immer eine Liste von Knoten abgearbeitet.
- Die Ergebnisse der Abarbeitungen aller Knoten in der Liste werden zum Gesamtergebnis konkateniert.
- Ein Template initiiert üblicherweise andere Regeln rekursiv.
- Mit xsl:import und xsl:include können andere Stylesheets benutzt werden.

Importieren und Inkludieren

- xsl:include wird durch den Inhalt des mittles des href-Attributes referenzierten Stylesheets ersetzt.
- xsl:import ist nur am Anfang eines Stylesheets erlaubt. Die importierten Regeln haben eine niedrigere Importpriorität als die im importierenden Stylesheet vorhandenen Regeln.
- Da inkludierte Stylesheets andere Stylesheets importieren können, wird festgelegt, dass die so importierten Regeln soweit nach oben geschoben werden, bis sie auf bereits importierte Regeln treffen. Die Ordnung der Regeln im so entstandenen Stylesheet beeinflußt die Regelauswahl im Fall von Konflikten.
- Da importierte Stylesheets wiederum andere Stylesheets importieren können, ergibt sich eine Importhierarchie. Eine Regel hat eine niedrigere Importpriorität als eine andere, falls sie bei einer Post-Order-Traversierung der Importhierarchie vorher besucht wird.

Wann eine Regel trifft

Sei ν der current node und λ die current node list. Sei P das Pattern des match-Attributs einer Regel.
XSLT spezifiziert, dass P auf ν und λ zutrifft (matches), falls es einen Knoten ν' innerhalb der Vorgängerknoten (ancestor-or-self) gibt, so dass ν im Ergebnis von P angewandt auf ν' als current node liegt.

Konfliktauflösung

Bestimme die Menge der zutreffenden Regeln T.

- Eliminiere alle Regeln in T, die eine niedrigere Importpriorität haben als eine andere Regel aus T.
- Eliminiere alle Regeln aus T, die eine niedrigere Priorität haben als andere Regeln aus T.

Patterns mit | werden als alternative Regeln, eine pro Alternative, aufgefaßt.
Falls mehr als eine Regel übrigbleibt, so ist dies ein Konflikt. Der XSLT-Prozessor hat jetzt zwei Möglichkeiten:

- Konflikt melden
- Konflikt auflösen und weitermachen

Bei letzterer Alternative muss er die Regel anwenden, die im Stylesheet als letztes vorkommt.

template rule sequence Ω

First, we define the *template rule sequence* Ω of a stylesheet such that

1. Ω contains all template rules of the stylesheet and the directly or indirectly included or imported stylesheets.
2. For any given document, current node, and current node list, the first template rule in Ω that matches is exactly the one the conflict resolution strategy of XSLT chooses (no matter whether there is a conflict or not).

Now, we can iterate over Ω until we find the first matching template rule.

Berechnung von Ω

1. For all templates with no priority given, add the default priority.
2. Determine the highest and the lowest priority (P_H and P_L) of all template rules in the stylesheet and all directly or indirectly included or imported stylesheets.
   Define $\delta_H = P_H + 1$ and $\delta_L = P_L - 1$.
3. Recursively replace `xsl:include` elements by the template rules in the included stylesheet.
4. Call `resolve-import` (stylesheet, 0, δ_H)
5. Add the default rules with priority δ_L.
6. Apply a stable sort to order template rules by ascending priorities.
7. Reverse the resulting sequence.

Resolve-Import

number resolve-import (stylesheet s , number w, number δ) $\{$

for each import element in document order

let s' be the referenced stylesheet

call $w = \texttt{resolve}(s', w, \delta)$

$w \mathrel{+}= \delta$;

for all templates T in s

add w to the priority of T.

replace all import elements in s by the sequence of

template rules found in the referenced stylesheet

return w

$\}$

Beispieltemplates

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transfor

<xsl:template match="Mondial">
  <countries>
    <xsl:apply-templates/>
  </countries>
</xsl:template>
```

```
<xsl:template match="country"/>

<xsl:template match="country[@population &lt; 1000000]">
  <country>
    <name><xsl:value-of select="@name"/></name>
    <eg><xsl:apply-templates select="./ethnicgroups/@name"/></
    <rel><xsl:apply-templates select="./religions/@name"/></re
  </country>
</xsl:template>

</xsl:stylesheet>
```

```
...
<xsl:template match="/">
  <xsl:for-each select = 'document("mondial.xml")//country'>
  <xsl:variable name = "ctry" select = "."/>
  <xsl:for-each select = 'document("mondial.xml")//mountain'>
  <xsl:variable name = "mtn"  select ="."/>
    <xsl:if test = "contains($mtn//@country , $ctry/@id)">
      <pair>
 <country><xsl:value-of select = "$ctry/@name"/></country>
 <mountain><xsl:value-of select = "$mtn/@name"/></mountain>
      </pair>
    </xsl:if>
  </xsl:for-each>
  </xsl:for-each>
</xsl:template>
...
```

```
<xsl:template match="/">
  <xsl:for-each select = 'document("mondial.xml")//mountain'>
  <xsl:sort select = "./@height" data-type = "number"
            order = "descending"/>
  <xsl:variable name = "mtn"  select ="."/>
  <xsl:for-each select = 'document("mondial.xml")//country'>
  <xsl:variable name = "ctry" select = "."/>
    <xsl:if test = "    ($mtn//@country = $ctry/@id)
                    and ($mtn//@height &gt; 1999)
                   ">
      <pair>
        <xsl:element name = "mountain">
          <xsl:attribute name="name">
            <xsl:value-of select = "$mtn/@name"/>
          </xsl:attribute>
          <xsl:attribute name="height">
            <xsl:value-of select ="$mtn/@height"/>
          </xsl:attribute>
        </xsl:element>
```

Data Model

- XML-Dokumente sind Texte
- Texte sind als Datenmodell ungeeignet
- Daher wurden verschiedene Datenmodelle für XML eingeführt
  - XPath 1.0 Datenmodell (vorher)
  - XML Information Set (jetzt)
  - Post-Schema-Validation-Infoset Model (PSVI)
  - XPath 2.0 und XQuery Datenmodell (später)
- Sie repräsentieren den Inhalt eines XML-Dokuments

XML Information Set (Infoset)

Document Information Items (entspricht Knotentypen)

- Document Information Item
- Element Information Item
- Attribute Information Item
- Processing Instruction Information Item
- Unexpanded Entity Reference Information Item
- Character Information Item
- Document Type Declaration Information Item
- Unparsed Entity Information Item
- Notation Information Item
- Namespace Information Item

XML Information Set (Infoset)

Jedes Information Item hat Properties. Hier die Properties für Element Information Item

- children
- parent
- attributes
- local name
- namespace name
- prefix

XML Information Set (Infoset)

Diskussion

- nur für (ganze) Dokumente definiert
- keine Typinformation

Post-Schema-Validation Infoset (PSVI)

Zusätzliche Properties für Elemente und Attribute

- validity: valid, invalid, notKnown
- validation attempted: full, none, partial
- schema normalized value: Wert
- type definition type: simple type oder complex type
- type definition anonymous: true/false
- type definition name: Typname
- identity constraint table

Post-Schema-Validation Infoset (PSVI)

Diskussion:

- nur für ganze Dokumente
- nur XML Schema Typen

XQuery 1.0 und XPath 2.0 Data Model

- sequence of Items (entspricht PSVI Information Item)
- singleton sequences werden mit enthaltenem Item identifiert
- neue Typen

XPath 2.0: new features

- use XQuery 1.0 and XPath 2.0 data model
- new operators: `intersect`, `except`
- larger set of functions
- sequence expressions
  - ',': comma: concatenation
  - `to`: construct sequence from numeric range
  - `some`, `every`: quantification
- `for...return`: iteration
- `if` (e1) `then` e2 `else` e3
- `cast`

XPath 2.0: element test

- `element()`, `element(*)`: matches any single element node
- `element(N)`: matches element named `N`
- `element(N,T)`: matches element named `N` of type `T` and derived types
- `element(*,T)`: matches element of type `T` and derived types

Analog für Attribute (`attribute`)

XPath 2.0: incompatibilities with XPath 1.0

- arithmetic operator's argument is sequence with more than one node: error
- inequality on strings: string comparison
- arithmetic on strings: error
- '=': on child-only elements: error

(alte semantik mit XPath 1.0 compabitility mode)

1 XQuery

- Anfragesprache für XML
- 2 Ausprägungen (die hier und eine XML-Syntax)
- noch nicht fertig
- Leider von Nicht-Datenbänklern zu sehr negativ beeinflußt (Mangelnde Deklarativität, mangelnde Optimierbarkeit, schlechte Syntax, mangelnde Expressivität, schlechtes Typsystem, etc.)

1 Wie man sich selber sieht

XQuery is a

- functional language which allows various kinds of expressions to be nested with full generality.
- strongly-typed language.

(letzteres nicht im Sinne von streng typisierten Programmiersprachen)

1 XQuery: Bausteine

- Xquery 1.0
- XQuery 1.0 and XPath 2.0 Formal Semantics
- XQuery 1.0 and XPath 2.0 Data Model
- XQuery 1.0 and XPath 2.0 Functions and Operators
- XQuery 2.0 and XQuery 1.0 Serialization
- RFC 2119, RFC 2396, RFC2986, RFC3987, ISO/IEC 10646, Unicode
- XML 1.0, XML 1.1, XML Names, XML Names 1.1, XML ID, XML Schema
- XML Infoset

2 XQuery: Basics

- Grundbaustein von XQuery ist der Ausdruck
- Ergebnis der Auswertung eines Ausdrucks (Wert (value)) ist immer eine Sequenz von Werten (items)
- Items können atomare Werte sein oder Knoten.
- Ein atomarer Wert entstammt dem Wertebereich eines atomaren Typs aus XML Schema.
- Ein Knoten gehört zu einem von sieben Knotentypen (s. Data Model).
- Jeder Knoten hat eine eindeutige Identität, eine getypten Wert und einen string-Wert.
- Der getypte Wert eines Knotens ist eine Sequenz von atomaren Werten.
- Der string-Wert eines Knotens ist vom Typ xs:string.
- Eine Sequenz, die genau einen Wert hat, heißt Singleton-Sequenz.
- Ein Item ist identisch mit einer Singleton-Sequenz, die genau diesen Wert enthält.
- synonym: *data model instance* und *value*

2 Vordefinierte Namensräume

XQuery definiert folgende Namensraumpräfixe:

1. xml = http://www.w3.org/XML/1998/namespace
2. xs = http://www.w3.org/2001/XMLSchema
3. xsi = http://www.w3.org/2001/XMLSchema-instance
4. fn = http://www.w3.org/2005/xpath-functions
5. xdt = http://www.w3.org/2005/xpath-datatypes
6. local = http://www.w3.org/2005/xquery-local-functions

zusätzlich benutzt:

- err = http://www.w3.org/2005/xqt-errors

für Fehler

2.1 Kontext

Jede Anfrage wird in einem Kontext ausgewertet. Dieser besteht aus einem

- statischen Kontext und einem
- dynamischen Kontext.

Letzterer wird in XQuery evaluation context genannt.

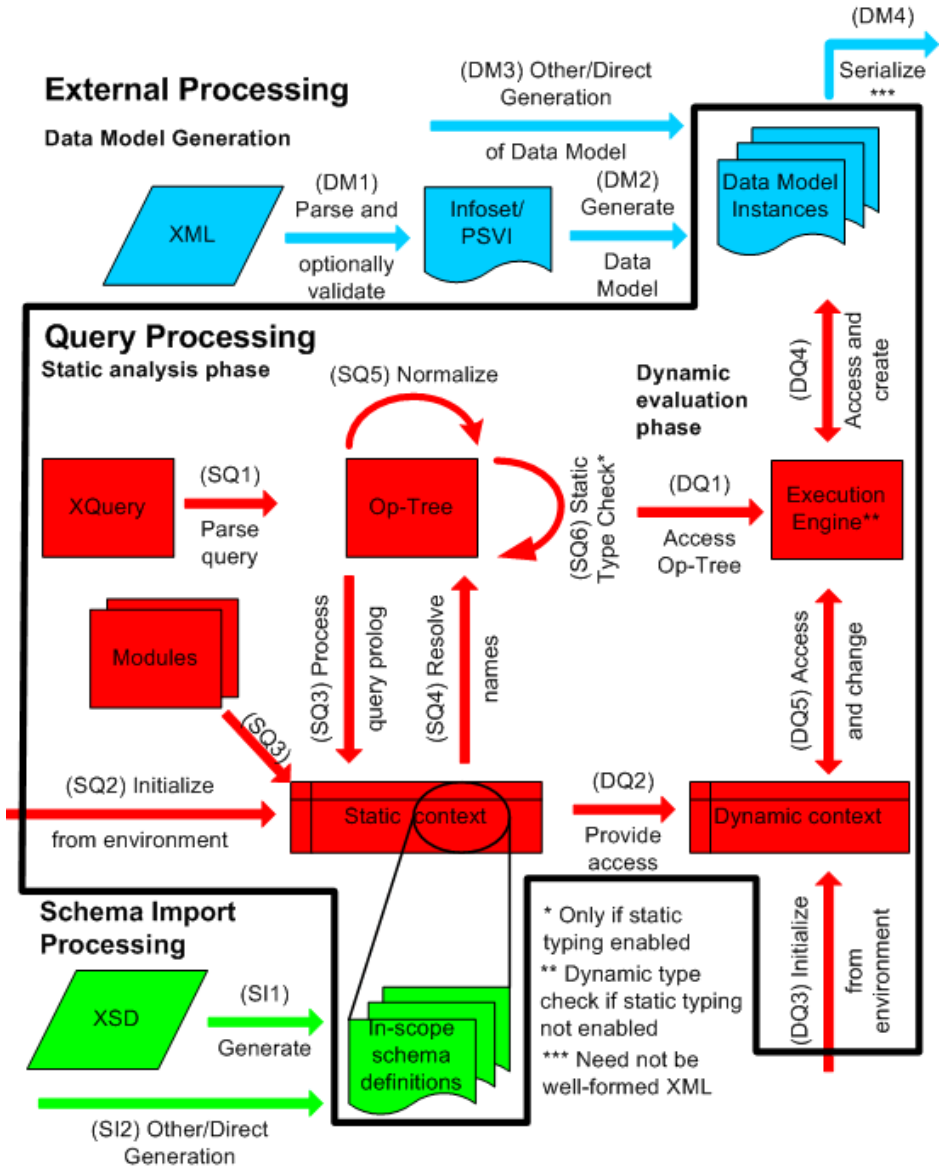
2.1.1 Statischer Kontext

- XPath 1.0 compatibility mode [false]
- Statically known namespaces
- Default element/type namespace
- Defaut function namespace
- In-Scope schema definitions: Menge von Paaren (QName, Typdeklaration)
- In-Scope variables: Menge von Paaren (QName, Typ)
- Context item static type
- Function signatures
- Statically known collations: Menge von Paaren (URI, collation)
- Default collation
- Construction mode (wird Typ bestehender Elemente beibehalten?)
- Ordering mode [ordered|unordered]
  beeinflußt Ergebnis von einigen XPath expressions, union, intersect, except, und FLWOR expressions ohne order by.
- Default order for empty sequences [greatest|least]
- Boundary-space policy
- Copy-namespace mode

2.1.2 Dynamischer Kontext

- focus: besteht aus context item, context position, context size
- Variable values (QName, value)
  dynamic value of a variable includes its dynamic type
- Function implementations
- Current dateTime
- Implicit timezone
- Available documents
- Available collections
- Default collection

2.2 Processing Model



2.2 XQuery/XPath Data Model (XDM)

- Document is represented as an XDM instance (value)
- Each element node and attribute node in an XDM instance has a **type annotation**. `type-name` property
- attribute whose type is unknown: `xdt:untypedAtomic`
- element whose type is unknown: `xdt:untyped`
  alle Nachfolger-Elemente: `xdt:untyped`
- partially valid element: `xs:anyType`
  Kinder können spezifischer getyped sein.

2.3 Error Handling

- static errors
- dynamic errors
- type error [either of the above]

latter: detection is implementation dependent

2.4.1 Document order

Document order and reverse document order are relevant for XQuery.
see [XQuery 1.0 and XPath 2.0 Data Model]

2.4.2 Atomisierung

- Atomisierung wird (implizit) angewendet, wenn eine Sequenz von atomaren Werten verlangt wird.
- Das Ergebnis ist entweder eine Sequenz von atomaren Werten oder ein Typfehler.
- Atomisierung ist definiert als das Ergebnis der Anwendung von `fn:data`:
  - Falls jedes Item in der Argumentsequenz entweder ein atomarer Wert ist oder ein Knoten dessen typisierter Wert eine Sequenz von atomaren Werten ist, so ist das Ergebnis die Konkatenation der Sequenzen der Items.
  - Ansonsten gibt es einen Typfehler.
- Atomisierung wird benutzt bei: arithmetischen Ausdrücken, Vergleichen, Funktionsargumenten und -ergebnissen, cast-Ausdrücken, Knotenkonstruktoren, `order by` Klausel

2.4.3 Effektiver boolscher Wert

Falls ein boolscher Wert verlangt wird, so wird der effektive boolsche Wert eines Ausdrucks mit Hilfe der Funktion fn:boolean bestimmt.

- leere Sequenzen: false
- Sequenz, deren erstes Element ein Knoten ist: true
- Singleton Sequenz mit Item vom Typ `xs:boolean`: ebendieser Wert
- Singleton Sequenz mit Item vom Typ `xs:string` oder xdt:untypedAtomic: false, falls die Länge 0 ist, sonst true.
- Singleton Sequenz mit Item numerisch: false, falls 0 oder NaN, sonst true.
- in allen anderen Faellen: type error

2.4.3 Effektiver boolscher Wert

The effective boolean value of a sequence that contains at least one node and at least one atomic value is nondeterministic in regions of a query where ordering mode is unordered.

2.4.3 Effektiver boolscher Wert

Der effektive boolsche Wert wird gebraucht bei:

- Logical expressions (`and`, `or`)
- The `fn:not` function
- The `where` clause of a FLWOR expression
- Certain types of predicates, such as a[b]
- Conditional expressions (`if`)
- Quantified expressions (`some`, `every`)

The definition of effective boolean value is not used when casting a value to the type `xs:boolean`, for example in a cast expression or when passing a value to a function whose expected parameter is of type `xs:boolean`.

2.4.4 Input Sources

Wie SQL Relationsnamen als Ankerpunkte braucht (oder OQL benannte Objekte und Extensionen), so hat XQuery Input-Funktionen:

- fn:doc gibt ein durch eine URI identifiziertes Dokument zurück. (genauer: Document Root Node)
- fn:collection gibt eine Kollektion (Sequenz) von Dokumenten zurück. Diese wird durch eine URI identifiziert. Ohne Argument wird die default collection zurückgegeben.

Types: sequence type

A **sequence type** is a type that can be expressed using the SequenceType syntax.

```
SequenceType ::= ("empty-sequence" "(" ")")
                 | (ItemType OccurrenceIndicator?)
```

Types: schema type

A **schema type** is a type that is (or could be) defined using the facilities of XML Schema (including the built-in types of XML Schema).
A schema type can be used as a type annotation on an

- element or
- attribute node

(unless it is a non-instantiable type such as xs:NOTATION or xdt:anyAtomicType, in which case its derived types can be so used).
Every schema type is either a complex type or a simple type.
Simple types are further subdivided into

- list types, union types, and atomic types

Types

Atomic types represent the intersection between the categories of sequence type and schema type.
An atomic type, such as xs:integer or my:hatsize, is both a sequence type and a schema type.

2.5.1 XPath Data Types (xdt)

xdt:untyped is used as the type annotation of an element node that has not been validated, or has been validated in skip mode.
No predefined schema types are derived from xdt:untyped.
xdt:untypedAtomic is an atomic type that is used to denote untyped atomic data, such as text that has not been assigned a more specific type.
An attribute that has been validated in skip mode is represented in the data model by an attribute node with the type annotation xdt:untypedAtomic.
No predefined schema types are derived from xdt:untypedAtomic.

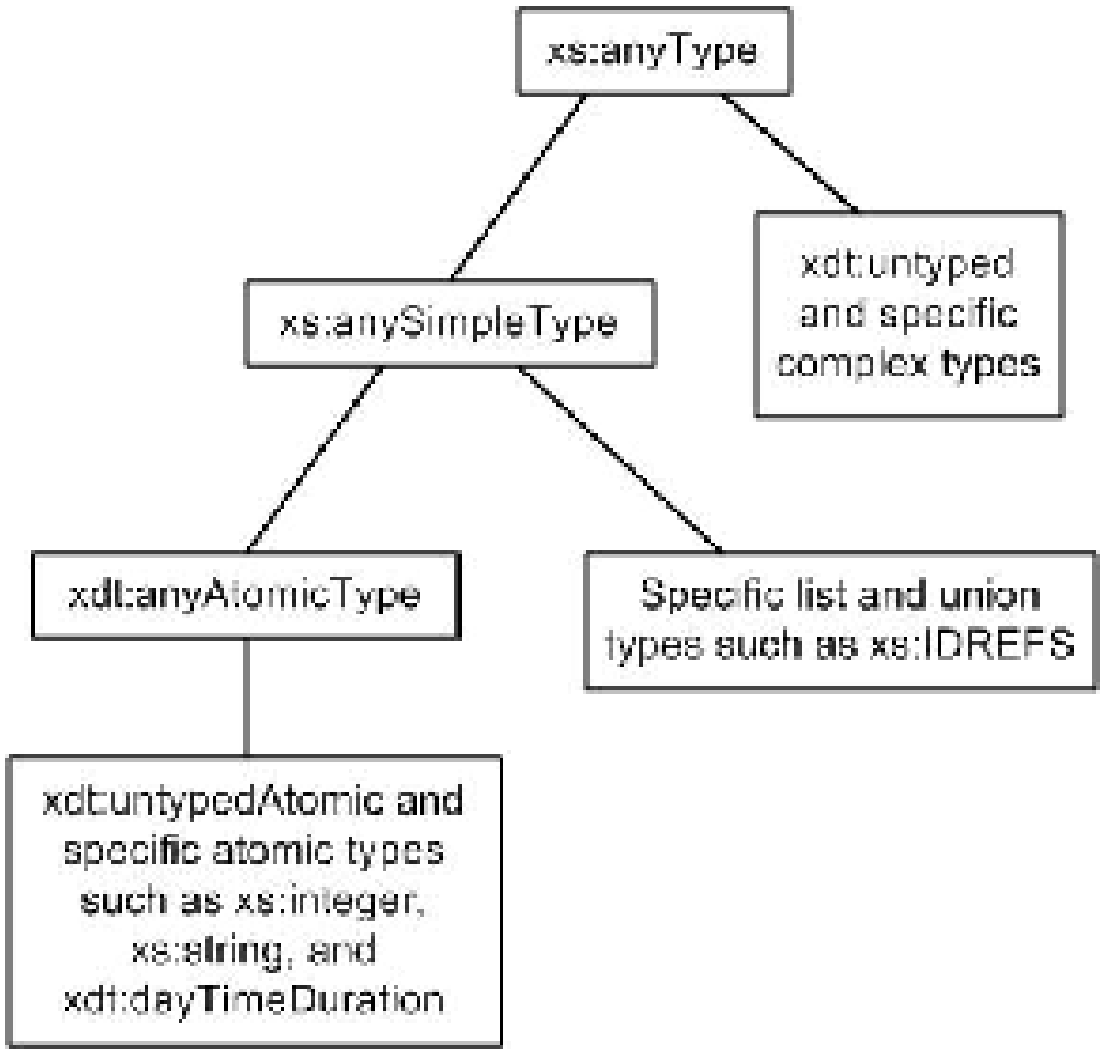
2.5.1 XPath Data Types (xdt)

xdt:dayTimeDuration is derived by restriction from xs:duration.
The lexical representation of xdt:dayTimeDuration is restricted to contain only day, hour, minute, and second components.
xdt:yearMonthDuration is derived by restriction from xs:duration.
The lexical representation of xdt:yearMonthDuration is restricted to contain only year and month components.

2.5.1 XPath Data Types (xdt)

xdt:anyAtomicType is an atomic type that includes all atomic values (and no values that are not atomic).
Its base type is xs:anySimpleType from which all simple types, including atomic, list, and union types, are derived.
All primitive atomic types, such as xs:integer, xs:string, and xdt:untypedAtomic, have xdt:anyAtomicType as their base type.
Note: `xdt:anyAtomicType` will not appear as the type of an actual value in an XDM instance.

2.5.1 Typhierarchie



2.5.2 Typed Value und String Value

- Dokumentknoten werden dynamisch (e.g. durch Validierung) mit Typ versehen.
- Ist dieser Typ ein anderer als xs:simpleType oder xs:anySimpleType, so ist der *typed value* anders als der *string value*. Es findet entsprechend eine Konversion statt.

Die entsprechenden Zugriffsfunktionen heißen fn:data und fn:string. Die Funktion fn:data kann Laufzeitfehler erzeugen.

2.5.2 Typed Value and String Value

- typed value: sequence of atomic values extracted by `fn:data`
- string value: string extracted by `fn:string`

The string value of a node must be a valid lexical representation of the typed value of the node.

2.5.2 Typed Value

The typed value of a node is never treated as an instance of a named list type.
Instead, if the type annotation of a node is a list type (such as xs:IDREFS), its typed value is treated as a sequence of the atomic type from which it is derived (such as xs:IDREF).

2.5.3 SequenceType Syntax

Jeder Typ in XQuery wird mit einem Sequenztyp beschrieben.

2.5.4 SequenceType Matching

Um Typgleichheit/-verträglichkeit festzustellen gibt es eine Abgleichdefinition.
Um die Zugehörigkeit eines Items zu einem Typ zu überprüfen gibt es ebenfalls eine Abgleichsdefinition.

Konvertierungen

- Promotion
- Untertypen
- implizite Konvertierung
- explizite Konvertierung
- Casting

Behandeln wir hier nicht in aller Breite, ist aber essentiell für das Verstehen der (präzisen) Semantik von XQuery.

3 Ausdrücke: Top Level

A **query** consists of one or more modules.
If a query is executable, one of its modules has a Query Body containing an expression whose value is the result of the query.

3 Expr

```
Expr ::= ExprSingle ("," ExprSingle)*
ExprSingle ::= FLWORExpr
               | QuantifiedExpr
               | TypeswitchExpr
               | IfExpr
               | OrExpr
```

3.1 Ausdrücke: Bottom Level

- Literale: 5, 5.5, 5E5, “5”
- Variablen: $x
- Context item expression: “.”
- Funktionsaufrufe: f(5)
- un-/ordered: `(un)ordered` “{” Expr “}”
- geklammerte Ausdrücke

3.2 Path Expressions

- Pfadausdrücke (XPath 2.0)

3.3 Sequence Expressions

3.3.1 Constructing Sequences:

- (10, 1, 3, 4)
- (10, (1, 2), (), (3, 4))
- ($x, $y)
- 10 to 10
- (10, 1 to 4)
- 5 to 10

3.3.2 Filter Expressions

```
\$book/(chapter | appendix)[fn:last()]
```

3.3.3 Kombinieren von Sequenzen

- `union`
- `intersect`
- `except`

Nur Knoten dürfen in Argumentsequenzen vorkommen. Anderenfalls gibt es einen dynamischen Fehler. Das Ergebnis ist immer Duplikatfrei und enthält die Knoten in Dokumentordnung.
(Systemabhängige stabile Ordnung auf Knoten verschiedener Dokumente.)

3.4 Arithmetische Ausdrücke

werden wie folgt ausgewertet:

1. Atomisierung wird auf die Operanden angewendet: **atomized operands**
2. Operand ist leere Sequenz: Ergebnis leere Sequenz
3. Länge Operand > 1: Fehler
4. Falls ein Operand den Typ xs:anySimpleType hat, wird der Operand nach xs:double umgewandelt. Falls die Umwandlung fehlschlägt, gibt es einen Fehler.
5. Die Operanden werden gemäß ihrer Typen und denen des Operators verarbeitet.

3.5 Vergleiche

- Wertvergleich: eq, ne, lt, le, gt, ge
- Allgemeiner Vergleich: =, !=, <, <=, >, >=
- Knotenidentitätsvergleich: is, isnot
- Knotenordnungsvergleich: <<, >>

3.5.1 Wertvergleich

1. Atomisierung wird auf die Operanden angewendet: **atomized operand**
2. Operand ist leere Sequenz: Ergebnis ist leere Sequenz
3. Länge Operand > 1: Fehler
4. Falls ein Operand den spezifischten Typ xs:untypedAtomic hat, so wird dieser Operand in `xs:string` umgewandelt.
5. Ergebnis entsprechend des Wertevergleichs gemäß des Operators.

3.5.1 Wertvergleich Beispiel

( ) eq 1 eq 2 ergibt: ( ).

3.5.2 Allgemeine Vergleiche

Es wird existenzielle Semantik zu den Wertevergleichsoperatoren hinzugefügt. Beispielsweise ergibt $(1, 2, 3) < 2$ `true`.
Allgemein: Seien A und B (atomisierte) Sequenzen. Dann:

$$A\,\theta\,B \;:\prec\succ\; \exists a \in A\; \exists b \in B\; a\theta' b$$

Wobei θ' ein Wertvergleich und θ der zugehörige allgemeine Vergleich ist.

3.5.2 Allgemeine Vergleiche

```
(1, 2) = (2, 3)
(2, 3) = (3, 4)
(1, 2) = (3, 4)
```

3.5.2 Allgemeine Vergleiche

```
(1, 2)  = (2, 3)
(1, 2) != (2, 3)
(1, 2)  < (2, 3)
(1, 2) <= (2, 3)
(1, 2)  > (2, 3)
(1, 2) >= (2, 3)
```

3.5.3 Knotenvergleiche

- Beide Operanden müssen entweder die leere Sequenz sein, oder eine Sequenz, die aus einem Knoten besteht. Sonst gibt es einen Fehler.
- Falls ein Operand die leere Sequenz ist, so ist das Ergebnis die leere Sequenz.
- Vergleich erfolgt dann aufgrund der Knotenidentität/Dokumentordnung.

3.5.3 Knotenvergleiche

```
<a>5</a> is <a>5</a>
```

Zwischenbetrachtung

- Viele Laufzeitfehler möglich.
- Sprache nicht streng typisiert.
- leere Sequenz ersetzt nicht vorhandenen NULL-Wert.
- Sehr viel Interpretation zur Laufzeit notwendig.

3.6 Logische Ausdrücke

and, **or**: erst effektiven boolschen Wert der Operanden ausrechnen, dann die übliche logische Verknüpfung

3.7 Konstruktoren

gibt es jede Menge.

3.7 Konstruktoren: Element

```
<book isbn="isbn-0060229357">
    <title>Harold and the Purple Crayon</title>
    <author>
        <first>Crockett</first>
        <last>Johnson</last>
    </author>
</book>
```

Achtung:

```
<book>
  5 <$x
</book>
```

ergibt Problem.

3.7 Eingebettete XQuery Ausdrücke

```
<example>
   <p> Here is a query. </p>
   <eg> $b//title </eg>
   <p> Here is the result of the above query. </p>
   <eg>{ $b//title }</eg>
</example>
```

“{{” ergibt dann ‘{’ in Text.

3.7 Eingebettete XQuery Ausdrücke

```
<shoe size="7"/>
<shoe size="{7}"/>
<shoe size="{()}"/>
<chapter ref="[{1, 5 to 7, 9}]"/>
<shoe size="As big as {$hat/@size}"/>
```

3.7 Berechnete Konstruktoren

```
element book {
   attribute isbn {"isbn-0060229357" },
   element title { "Harold and the Purple Crayon"},
   element author {
      element first { "Crockett" },
      element last {"Johnson" }
   }
}
```

3.7 Berechnete Konstruktoren

```
element
   {$dict/entry[@word=name($e)]/variant[@xml:lang="it"]}
   {$e/@*, $e/node()}
```

3.8 FLWOR-Ausdrücke

for	bindet Variablen iterativ an Elemente einer Sequenz
let	bindet Variable an einen Wert
where	Prädikat
order by	zum Sortieren der Variablenbindungen
return	erzeugt Ergebnis pro Variablenbindung

3.8 FLWOR-Ausdrücke

1. The **for** and **let** clauses in a FLWOR expression generate an *ordered sequence of tuples of bound variables*, called the **tuple stream**.
2. The optional **where** clause serves to filter the tuple stream, retaining some tuples and discarding others (effective boolean value).
3. The optional **order by** clause can be used to reorder the tuple stream.
4. The **return** clause constructs the result of the FLWOR expression.
   The **return** clause is evaluated once for every tuple in the tuple stream, after filtering by the **where** clause, using the variable bindings in the respective tuples.
   The result of the FLWOR expression is an ordered sequence containing the results of these evaluations, concatenated as if by the comma operator.

3.8 FLOWR-Ausdrücke

```
for $d in fn:doc("depts.xml")/depts/deptno
let $e := fn:doc("emps.xml")/emps/emp[deptno = $d]
where fn:count($e) >= 10
order by fn:avg($e/salary) descending
return
   <big-dept>
      {
      $d,
      <headcount>{fn:count($e)}</headcount>,
      <avgsal>{fn:avg($e/salary)}</avgsal>
      }
   </big-dept>
```

3.8.1 For and Let Clauses

```
[let|for] $s in (<one/>, <two/>, <three/>)
return <out>{$s}</out>
```

```
<out>
   <one/>
   <two/>
   <three/>
</out>
```

```
<out>
   <one/>
</out>
<out>
   <two/>
</out>
<out>
   <three/>
</out>
```

3.8.1 Ordering

```
for $i in (1, 2), $j in (3, 4)
```

produces tuple sequence (ordering mode = ordered):

```
($i = 1, $j = 3)
($i = 1, $j = 4)
($i = 2, $j = 3)
($i = 2, $j = 4)
```

3.8.1 Positional Variable

```
for $car at $i in ("Ford", "Chevy"),
    $pet at $j in ("Cat", "Dog")
```

ergibt

```
($i = 1, $car = "Ford", $j = 1, $pet = "Cat")
($i = 1, $car = "Ford", $j = 2, $pet = "Dog")
($i = 2, $car = "Chevy", $j = 1, $pet = "Cat")
($i = 2, $car = "Chevy", $j = 2, $pet = "Dog")
```

3.8.1 Type Declarations

Für jede Variable in **let** oder **for** ist eine Typdeklaration möglich:

```
let $salary as xs:decimal :=  "cat"
return $salary * 2
```

ergibt Fehler

3.8.2 Where Clause: Beispiel

```
fn:avg(for $x at $i in $inputvalues
    where $i mod 100 = 0
    return $x)
```

3.8.3 Order by Clause

```
OrderByClause ::= (("order" "by") | ("stable" "order" "by"))
                  OrderSpecList
OrderSpecList ::= ("," OrderSpec)*
OrderSpec     ::= ExprSingle OrderModifier
OrderModifier ::= ("ascending" | "descending")?
                  ("empty" ("greatest" | "least"))?
                  ("collation" URILiteral)?
```

3.8.3 Order by Clause

1. atomize result of the expressions in the **order by** clause
2. if the dynamic type of one of these items is `xdt:untypedAtomic` then cast to `xs:string`
3. convert to least common type with `gt` operator
   if not possible: type error
4. sortiere

3.8.4 Beispiel

```
for $b in $books/book[price < 100]
order by $b/title
return $b
```

3.8.4 Beispiel

```
for $b in $books/book
stable order by $b/title collation
                          "http://www.example.org/collations/fr-ca",
                $b/price descending empty least
return $b
```

3.8.3 Order by Clause

```
<?xml version = "1.0"?>
<persons>
 <person name="anton" age="two and a half"/>
 <person name="eva"   age="25"/>
 <person name="anna"  age="3"/>
</persons>

for $p in document("persons.xml")//person
order by $p/@age ascending
return $p/@name
```

3.8.3 Order by Clause

```
attribute name {"eva"},
attribute name {"anna"},
attribute name {"anton"}
```

3.8.4 Beispiel

List books published by Addison-Wesley after 1991, including their year and title

```
<bib>
 {
  for $b in document("http://www.bn.com/bib.xml")/bib/book
  where $b/publisher = "Addison-Wesley" and $b/@year > 1991
  return
    <book year="{ $b/@year }">
     { $b/title }
    </book>
 }
</bib>
```

3.8.4 Beispiel

Create a flat list of all the title-author pairs, with each pair enclosed in a "result" element.

```
<results>
  {
    for $b in document("http://www.bn.com/bib.xml")/bib/book,
        $t in $b/title,
        $a in $b/author
    return
        <result>
            { $t }
            { $a }
        </result>
  }
</results>
```

3.8.4 Beispiel

For each author in the bibliography, list the author's name and the titles of all books by that author, grouped inside a "result" element.

```
<results> {
    for $a in distinct-values(
    document("http://www.bn.com/bib.xml")//author)
    return
        <result>
            { $a },
            {
   for $b in document("http://www.bn.com/bib.xml")/bib/book
   where some $ba in $b/author satisfies deep-equal($ba,$a)
   return $b/title
            } </result>
  } </results>
```

3.9 Ordered and Unordered Expressions

```
OrderedExpr    ::= "ordered" "{" Expr "}"
UnorderedExpr  ::= "unordered" "{" Expr "}"
```

3.10 Conditional Expressions

```
if ($part/@discounted)
  then $part/wholesale
  else $part/retail
```

[else ist nicht optional]

3.11 Quantified Expressions

```
QuantifiedExpr  ::=  ("some" | "every")
                     "$" VarName TypeDeclaration? "in" ExprSingle
                     ("," "$" VarName TypeDeclaration? "in" ExprSing
                     "satisfies" ExprSingle
TypeDeclaration  ::=  "as" SequenceType
```

3.11 Quantified Expressions

```
some $x in (1, 2, 3), $y in (2, 3, 4)
     satisfies $x + $y = 4

every $x in (1, 2, 3), $y in (2, 3, 4)
     satisfies $x + $y = 4
```

[EBV (effective boolean value)]

3.12.1 Instance of

`Expr "instance" "of" SequenceType`

evaluates to true if value of `Expr` matches `SequenceType`

3.12.2 Typeswitch

```
typeswitch($customer/billing-address)
   case $a as element(*, USAddress) return $a/state
   case $a as element(*, CanadaAddress) return $a/province
   case $a as element(*, JapanAddress) return $a/prefecture
   default return "unknown"
```

Here: optional variable $a not needed.

3.12.3 Cast

```
Expr cast as AtomicType "?"?
```

3.12.3 Cast

1. atomize `Expr`
2. result contains more than one item: type error
3. zero item: if ("?" not specified) type error else empty sequence
4. evaluation: cast value or raise error if not castable

3.12.4 Castable

```
Expr castable as AtomicType "?"?
```

castable: falls cast keinen Fehler erzeugt gibt castable true zurueck

3.12.4 Castable

Usage:

```
if ($x castable as hatsize)
   then $x cast as hatsize
   else if ($x castable as IQ)
   then $x cast as IQ
   else $x cast as xs:string
```

3.12.5 Constructor Functions

There exists a constructor function for every instantiable `xdt:anyAtomicType`

```
xs:date("2000-01-01")
xs:decimal($floatvalue * 0.2E-5)
xdt:dayTimeDuration("P21D")
```

The semantics of the constructor function Type($arg) are defined to be equivalent to the expression ($arg cast as T?).

3.12.6 Treat

```
TreatExpr  ::=  CastableExpr ( "treat" "as" SequenceType )?
```

- static type of TreatExpr is SequenceType
- at runtime (dynamically) it is checked whether CastableExpr really results in a value of SequenceType. If not, an error is raised.

Was es sonst noch gibt

- `validate`
- Extension Expressions (herstellerabhängige Erweiterungen)
- Modules and Prologs

Was in XQuery noch fehlt

- explizite Gruppierung
- false aggregates

Beispiel für implizite Gruppierung

```
for $pn in distinct-values(document("catalog.xml")//partno)
let $i := document("catalog.xml")//item[partno = $pn]
where count($i) >= 3
order by $pn
return
   <well-supplied-item>
      {$pn}
      <avgprice> {avg($i/price)} </avgprice>
   </well-supplied-item>
```

Beurteilung

- Viele Laufzeitfehler möglich (resultiert auch in Indeterminismus)
- Sprache nicht streng typisiert
- leere Sequenz ersetzt nicht vorhandenen NULL-Wert
- Sehr viel Interpretation zur Laufzeit notwendig
- Semantik der gleichen Anfrage bei gleichem Dokument unterschiedlich, je nachdem, ob das Dokument validiert wurde oder nicht
- Scheußliche Syntax
- Einige einfache Anfragen nur kompliziert formulier- und optimierbar
- Effizienz oft nur möglich, falls der Benutzer explizit `unordered` hinschreibt.
- Aber: gegenüber älteren Versionen bereits erhebliche(!) Verbesserungen

XQuery Update Facility

XQuery Update ist eine Sprache, in der Änderungen an XML-Dokumenten vorgenommen werden können.
Dabei gibt es die folgenden Operatinonen:

- insert: Einfügen eines neuen Knotens
- delete: Löschen eines Knotens
- modify: Modifizieren eines Knotens
- copy: Erzeugt eine Kopie eines Knotens mit neuer Identität

XQuery Update: Neue Ausdrücke

XQuery Update erweitert XQuery:

- fünf neue Ausdrücke (insert, delete, replace, rename, transform)
- Klassifikation von XQuery-Ausdrücken:
  - basic updating expression: insert, delete, replace, rename, oder ein Aufruf an eine updating function.
  - updating expression: Ausdruck, der einen basic updating expression enthält, der nicht innerhalb der `modify`-Klausel eines transform-Ausdrucks steht. aber keiner ist.
  - non-updating expression: alle anderen XQuery-Ausdrücke
- spezifiziert die Stellen, an denen die (basic) updating expressions in XQuery-Ausdrücken vorkommen dürfen

XQuery Update: Processing Model

- Jeder XQuery (mit Update) Audruck liefert entweder eine XDM-Instanz zurück oder eine pending update list, die ebenfalls im erweiterten XDM repräsentiert werden kann.
- pending update list: liste von noch auszuführenden primitiven Änderungen (update primitive) deren erste Komponente der zu ändernde Knoten ist (target node).
- XQuery Update spezifiziert Änderungsoperationen, die aber nicht dem Benutzer zur Verfügung stehen.
- Falls der äußerste Ausdruck eines XQuery (Update) Ausdrucks eine pending update list zurückgibt, so wird nach dessen Auswertung auf die resultierende pending update list implizit die Funktion `upd:applyUpdates` auf diese angwendet.
  [snapshot semantics]

XQuery Update: `insert`

Syntax:

```
InsertExpr  ::=  "do" "insert" SourceExpr
                     ( (("as" ("first" | "last"))? "into")
                       | "after"
                       | "before" )
                  TargetExpr
SourceExpr  ::=   ExprSingle
TargetExpr  ::=   ExprSingle
```

SourceExpr, TargetExpr: kein updating expression.

XQuery Update: `insert`

- as first: einfügen als erste(s) Kind(er)
- as last: einfügen als letzte(s) Kind(er)
- after: einfügen als direkt nachfolgende(n) Geschwisterknoten
- before: einfügen als direkt vorangehende(n) Geschwisterknoten
- into ohne first, last: irgendwo als Kinderknoten (implementation dependent), aber Einfügepositionen stehen nicht in Konflikt mit anderen inserts mit first/last im selben Snapshot.

XQuery Update: `insert`

Beispiel 1:

```
do insert <year>2005</year>
    after fn:doc("bib.xml")/books/book[1]/publisher
```

Beispiel 2:

```
do insert $new-police-report
   as last into fn:doc("insurance.xml")/policies
      /policy[id = $pid]
      /driver[license = $license]
      /accident[date = $accdate]
      /police-reports
```

XQuery Update: `insert`

Auswertung:

1. SourceXpr auswerten $\rightsquigarrow$ sequence of nodes called **insertion sequence**
2. Aufteilen der insertion sequence:
   - $alist: Sequenz der Attributknoten
   - $clist: Sequenz der anderen Knoten
3. TargetExpr auswerten
   - `into`: Resultat muss einzelner Element- oder Document-Knoten sein.
   - `before/after`: Resultat muss einzelner Elementknoten mit nicht-leerer Parent-Property sein.
   - sei $target der target node.
4. Baue pending update list (nächste Folie)

XQuery Update: `insert`

Die pending update list besteht aus folgenden Einträgen:

- falls `first` spezifiziert:
  - falls $alist nicht leer: `upd:insertAttributes($target, $alist)`
  - falls $clist nicht leer: `upd:insertIntoAsFirst($target, $clist)`
- falls `last` spezifiziert:
  - falls $alist nicht leer: `upd:insertAttributes($target, $alist)`
  - falls $clist nicht leer: `upd:insertIntoAsLast($target, $clist)`
- falls `into` aber nicht `first`/`last` spezifiziert:
  - falls $alist nicht leer: `upd:insertAttributes($target, $alist)`
  - falls $clist nicht leer: `upd:insertInto($target, $clist)`
- falls `before` spezifiziert: $parent sei parent von $target:
  - falls $alist nicht leer: `upd:insertAttributes($parent, $alist)`
  - falls $clist nicht leer: `upd:insertBefore($target, $clist)`
- falls `after` spezifiziert: $parent sei parent von $target:
  - falls $alist nicht leer: `upd:insertAttributes($parent, $alist)`
  - falls $clist nicht leer: `upd:insertAfter($target, $clist)`

XQuery Update: `delete`

Syntax:

```
DeleteExpr      ::=     "do" "delete" TargetExpr
TargetExpr      ::=     ExprSingle
```

TargetExpr: kein updating expression

XQuery Update: `delete`

Beispiel 1:

```
do delete fn:doc("bib.xml")/books/book[1]/author[last()]
```

Beispiel 2:

```
do delete /email/message
     [fn:currentDate() - date > xs:dayTimeDuration("P365D")]
```

XQuery Update: `delete`

Auswertung:

1. sei $tlist das Ergebnis der Auswertung des TargetExpr.
2. falls $tlist keine Sequenz von Knoten ist (kann auch leer sein): Fehler.
3. es wird eine neue pending udpate list erzeugt.
4. für jeden Knoten $tnode in $tlist hänge

   ```
   upd:delete($tnode)
   ```

   an die pending update list.

XQuery Update: `replace`

Syntax:

```
ReplaceExpr ::=  "do" "replace"
                 ("value" "of")?
                 TargetExpr "with" WithExpr
TargetExpr  ::=  ExprSingle
WithExpr    ::=  ExprSingle
```

TargetExpr, WithExpr dürfen kein updating expression sein.

XQuery Update: `replace`

Zwei Semantiken, je nachdem, ob `value of` spezifiziert wurde:

- Ersetzen eines Knotens
- Ersetzen eines Knotenwerts

XQuery Update: `replace`: ohne `value of`

Ersetzen eines Knotens (`value of` nicht spezifiziert)
Ein Knoten wird durch eine Sequenz von Knoten ersetzt.
Beispiel:

```
do replace fn:doc("bib.xml")/books/book[1]/publisher
with fn:doc("bib.xml")/books/book[2]/publisher
```

XQuery Update: `replace:` ohne `value of`

Auswertung:

1. Sei $rlist das Ergebnis der Auswertung des WithExpr (muss Knotenliste sein).
2. Werte TargetExpr aus. Das Ergebnis muss ein einzelner Knoten mit nicht-leerer Parent-Property sein. Nenne diesen Knoten $target und seinen Parent $parent.
3. Falls $target ein element, text, comment, oder processing instruction Knoten ist, so darf $rlist nur aus element, text, comment, oder processing instruction Knoten bestehen. Falls $target ein Attributknoten ist, so darf $rlist nur aus Attributknoten bestehen.
4. Die pending action list besteht aus

```
upd:replaceNode($target, $rlist)
```

XQuery Update: `replace:` mit `value of`

Ersetzen eines Knotenwertes (`value of` spezifiziert)
Der Wert eines Knotens wird unter Beibehaltung seiner Identität geändert.
Beispiel:

```
do replace value of fn:doc("bib.xml")/books/book[1]/price
with fn:doc("bib.xml")/books/book[1]/price * 1.1
```

XQuery Update: `replace: mit value of`

Auswertung:

1. WithExpr wird ausgewertet und das Ergebnis in einen Textknoten $text gepackt.
2. TargetExpr wird ausgewertet und das Ergebnis muss ein einzelner Knoten $target sein.
3. Die pending update list enthält:
   - falls $target ein Elementknoten ist:
     ```
     upd:replaceElementContent($target, $text)
     ```
   - falls $target ein Attribut-, Text-, Kommentar- oder PI-knoten ist:
     ```
     upd:replaceValue($target, $string)
     ```
     wobei $string der string-value von $text ist.
   - error sonst

XQuery Update: `rename`

Syntax:

```
RenameExpr  ::=  "do" "rename" TargetExpr "as" NewNameExpr
TargetExpr  ::=  ExprSingle
NewNameExpr ::=  ExprSingle
```

TargetExpr, NewNameExpr dürfen keine updating expression sein.

XQuery Update: `rename`

Beispiel 1:

```
do rename fn:doc("bib.xml")/books/book[1]/author[1]
as "principal-author"
```

Beispiel 2:

```
do rename fn:doc("bib.xml")/books/book[1]/author[1]
as $newname
```

XQuery Update: `rename`

Auswertung:

1. TargetExpr wird ausgewertet. Das Ergebnis muss Element-, Attribute- oder PI-Knoten sein. Sei $target dieser Knoten.
2. NewNameExpr wird ausgewertet und muss zu einem QName $QName evaluieren.
3. Die Pending update list enthält dann

```
upd:rename($target, $QName)
```

XQuery Update: `transform`

- erstellt eine Kopie eines existierende Knotens in einer XDM-Instanz
- hierbei werden neue Ids erzeugt
- dieser kann modifiziert werden

Transform-Ausdruck ist non-updating expression, da er keinen existierenden Wert modifiziert.

XQuery Update: `transform`

Syntax:

```
TransformExpr ::=  "transform"
                   "copy" "$" VarName ":=" SourceExpr
                          ("," "$" VarName ":=" SourceExpr)*
                   "modify" ModifyExpr
                   "return" ReturnExpr
SourceExpr ::= ExprSingle
ModifyExpr ::= ExprSingle
ReturnExpr ::= ExprSingle
```

SourceExpr (lhs of $:=$) kein updating expression; ModifyExpr ist updating expression; ReturnExpr kein updating expression;

XQuery Update: `transform`

Beispiel:

```
for $e in //employee[skill = "Java"]
return
   transform
      copy $je := $e
      modify do delete $je/salary
      return $je
```

XQuery Update: `transform`

Auswertung

1. Die SourceExpr werden ausgewertet und eine Kopie der resultierenden Knotensequenz(!) an die Variablen gebunden.
2. Die pending update list $pul wird aus dem ModifyExpr erzeugt
3. Aufruf von `upd:applyUpdates($pul)`
4. ReturnExpr wird ausgewertet und das Ergebnis zurückgegeben

XQuery Update: Konsistenzbedingungen

- kein Knoten darf zweimal umbenannt werden
- kein Knoten darf zweimal ersetzt werden
- kein Knotenwert darf zweimal ersetzt werden
- Achtung bei gleichzweitiger Wertänderung von Knoten und Modifikationen von Nachfolgerknoten: letztere gehen verloren! Beispiel:

```
do replace $A/B with <C>Hello</C>,
do replace value of $A with <D>Goodbye</D>
```

$A enthält nach Auswertung nur noch einen Textknoten mit Inhalt "Goodbye".

XQuery Update: Return-Klausel

Die Return-Klausel eines FLWOR-Ausdrucks kann updating expression enthalten.
Beispiel:

```
for $p in /inventory/part
let $deltap := $changes/part[partno eq $p/partno]
return do replace value of $p/quantity
      with $p/quantity + $deltap/quantity
```

XQuery Update: Return-Klausel

Auswertung:

1. Die for, let, where, order-by Klauseln werden wie gehabt ausgewertet und resultieren in einer Sequenz von Mengen von Variablenbindungen (Tupel).
2. Für jedes Tupel wird die pending update list berechnet.
3. Die pending update lists werden gemischt (successiver Aufruf von `upd:mergeUpdates`.
4. Die gemischte pending update list wird ausgeführt.

XQuery Update: Andere Ausdrücke

updating expressions sind auch in anderen Ausdrücken erlaubt:

1. typeswitch expression
2. conditional expression
3. comma expression
4. parenthesized expression
5. function declarations

Überall sonst führt das Auftauchen eines updating expressions zu einem Fehler.

XML wozu? (die Versprechungen)

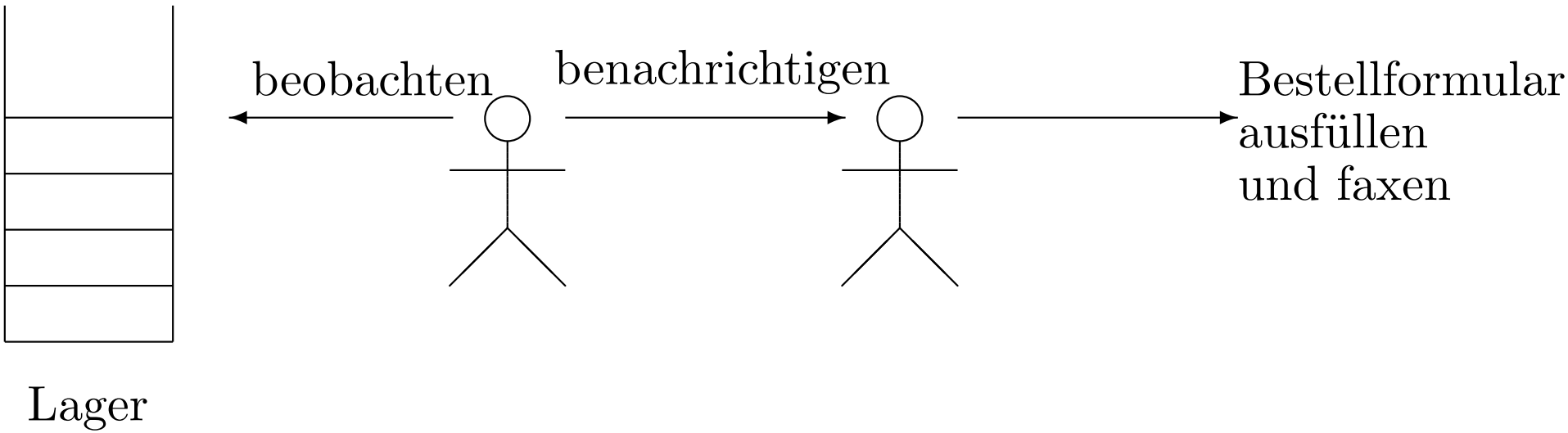
Integration
von
Daten und Anwendungen
innerhalb von Organisationen
und
Organisationsübergreifend

Datenintegration wozu?

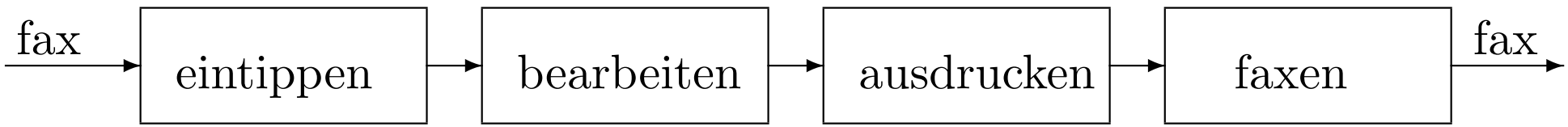
Motivation ist immer Erkenntnisgewinn

- Wetterdaten/Erdbebendaten
- Gendaten, Expressionsdaten, medizinische Daten

Anwendungsintegration wozu?



Anwendungsintegration wozu?



Ohne Integration: Nachteile

- viel Handarbeit
- dauert lange (Ketten durchlaufen)
- fehleranfällig
- Korrelationen von Daten schwierig

Bisherige Ansätze zur Datenintegration

- Datenbanken
- verteilte Datenbanken
- föderierte Datenbanken, Multidatabase Systems
- objekt-orientierte Datenbanken
- Data Warehouse
- Wrapper-/Mediator-Architekturen

Alles nicht sonderlich erfolgreich, da auf proprietären Protokollen und Datenrepräsentationen arbeitend. Es bleibt fraglich, ob mittels XML Integration erreicht werden kann.

Einsatzgebiete XML

- Datenrepräsentation
- Dokumentrepräsentation
- Webserver (statisch, dynamisch)
- Nachrichtenaustausch (SOAP)
- Application Server
- Datenintegration (Hoffnung)
- Anwendungsintegration (Hoffnung)
- Web Services (eine Ausprägung der Anwendungsintegration)

Hoffnung: mehr oder weniger berechtigt. Aufwand bleibt!

www.xml.org: registry (Stand 20.01.03)

Accounting (7)
Automotive (4)
Business Services (1)
Customer Relation (2)
ERP (1)
Food Services (2)
Industrial Control (1)
News (3)
Robotics/AI (2)
Software (10)
Travel (1)

Aerospace (1)
Banking (1)
Catalogs (1)
Databases (1)
Education (9)
Geography (2)
Internet/Web (7)
Other Industry (5)
Science (7)
Supply Chain (5)
Weather (1)

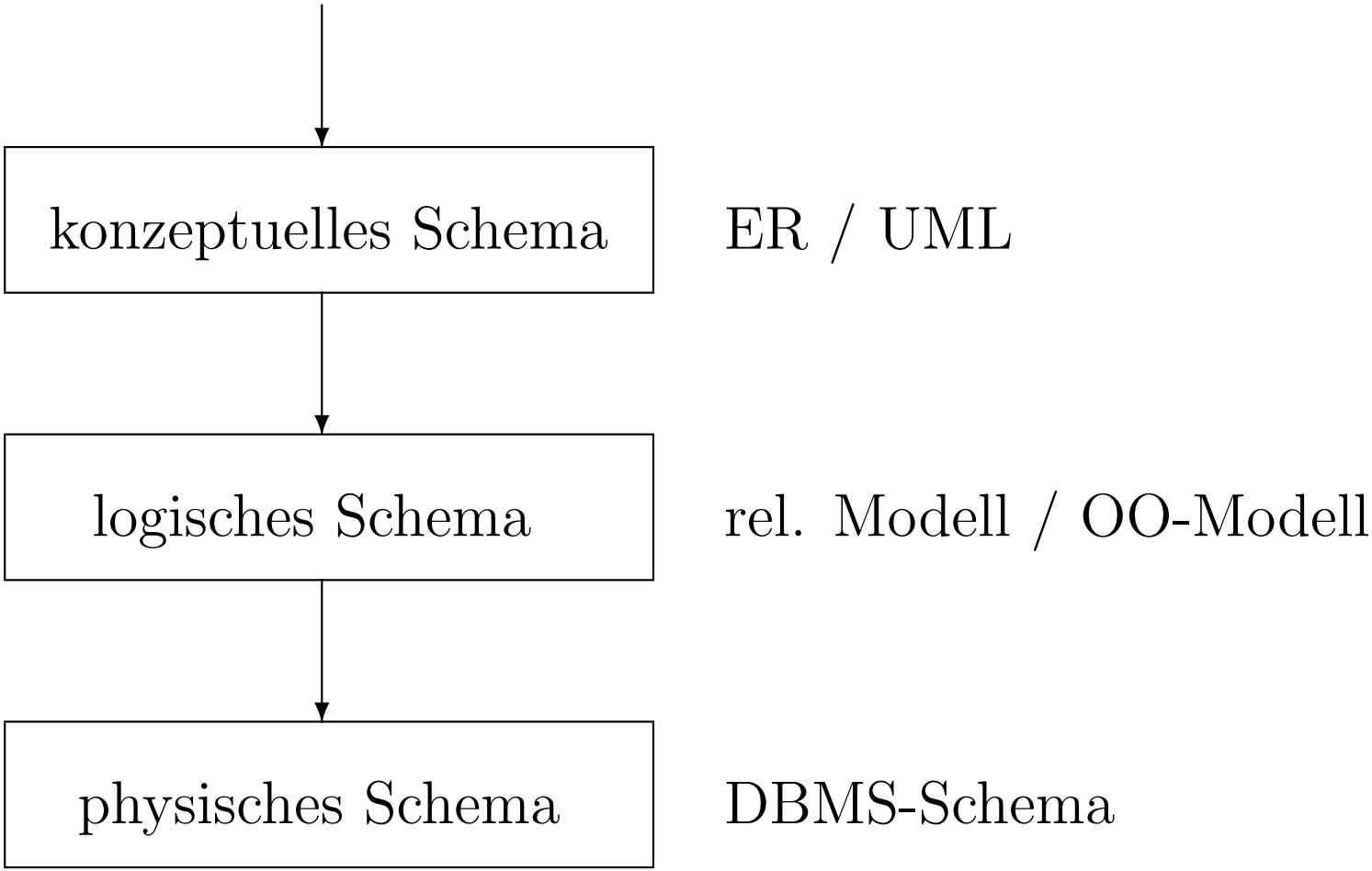
Arts/Entertainment (6)
Biology (4)
Chemistry (3)
E-Commerce (9)
Energy/Utilities (23)
Healthcare (6)
Manufacturing (4)
Publishing/Print (3)
Security (1)
Telecommunications (5)
XML Technologies (19)

Astronom
Business
Construct
EDI (2)
Financial
Human R
Multimed
Real Esta
Social Sci
Transport

XML Hauptziel

Erfassung von mehr Semantik

Datenbankentwurfszyklus



XML im Datenbankentwurfszyklus

XML ist

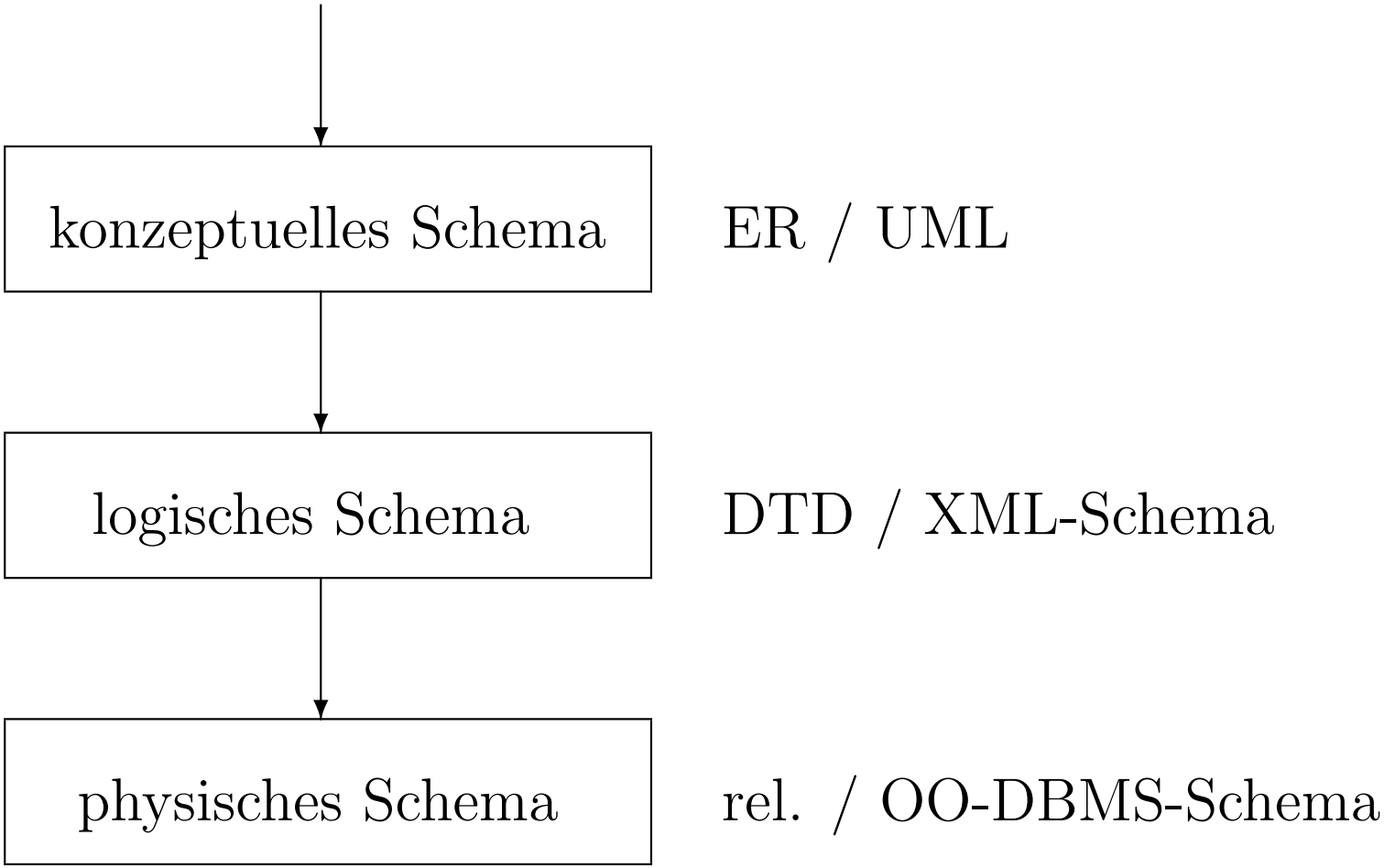
konzeptuelles Modell bestimmt nicht: Aggregation, Assoziation, Generalisierung/Spezialisierung, nicht graphisch

logisches Modell jein. pro: genügend Modellierungsstärke. contra: benötigt selbst logisches Modell (Bäume)

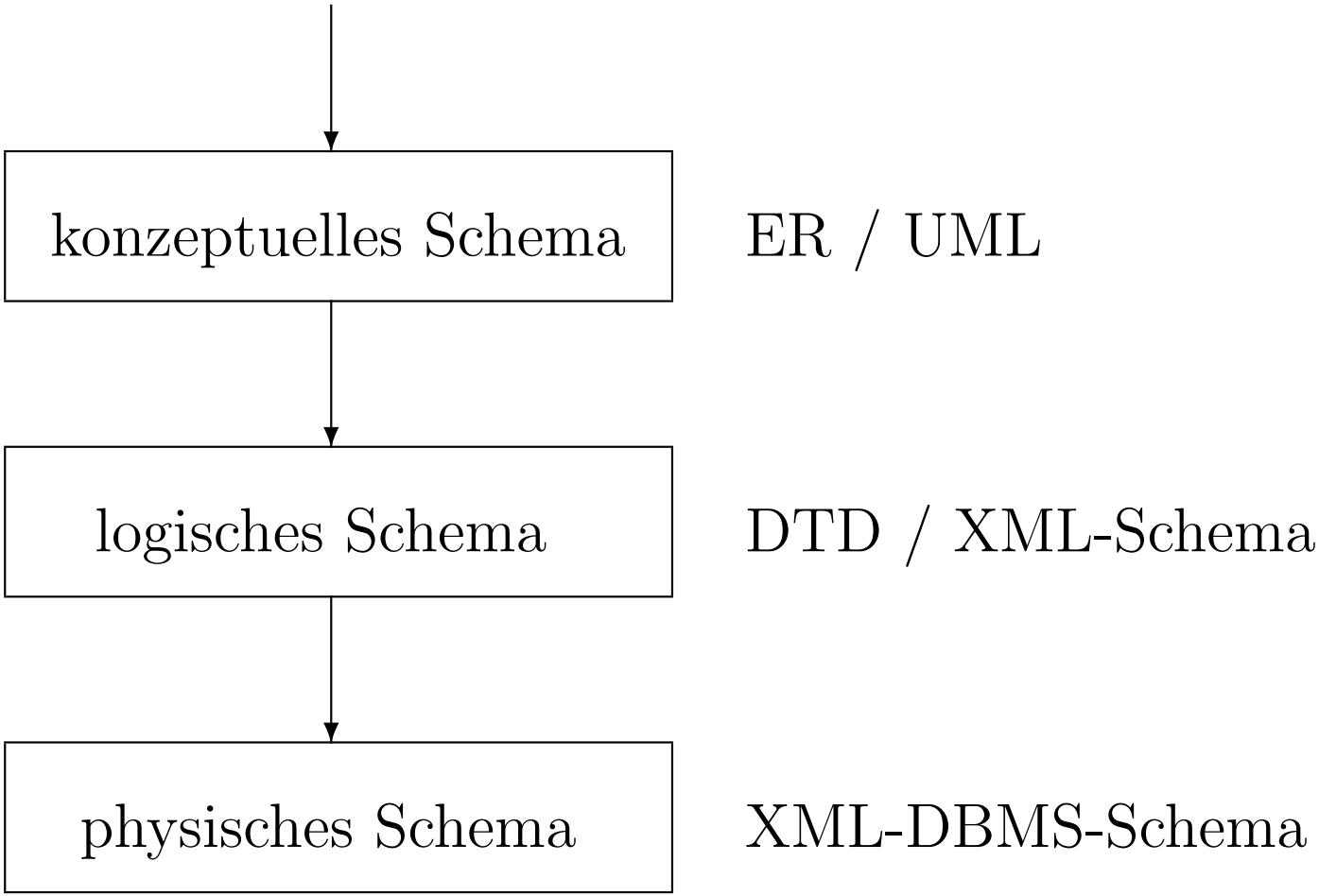
physisches Modell jein. pro: textuelle repräsentation ist physisch. contra: Indizes plus andere Maßnahmen zur Beeinflussung der Laufzeit

Entwurf auf der grünen Wiese: Wie sehen es andere?

Grüne Wiese nach [1]

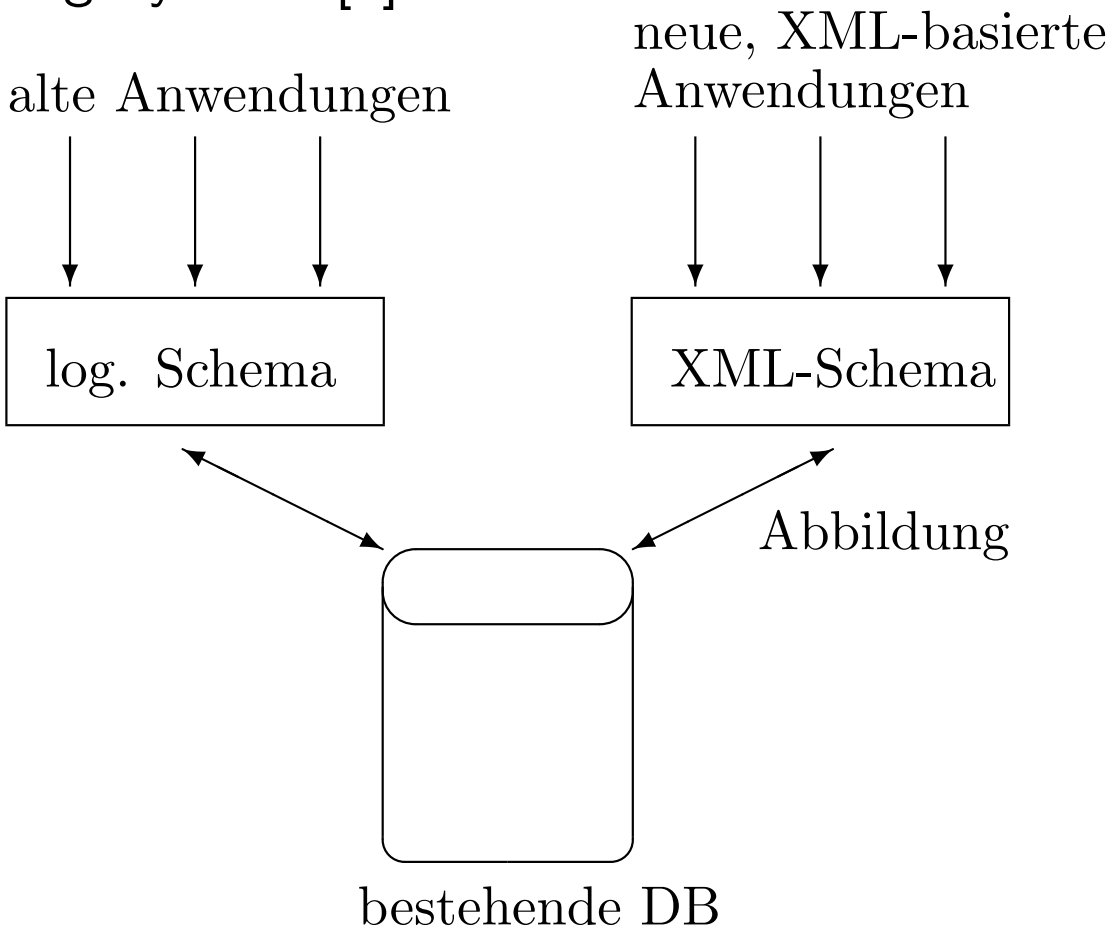


Modellierung unter Verwendung von XBMS

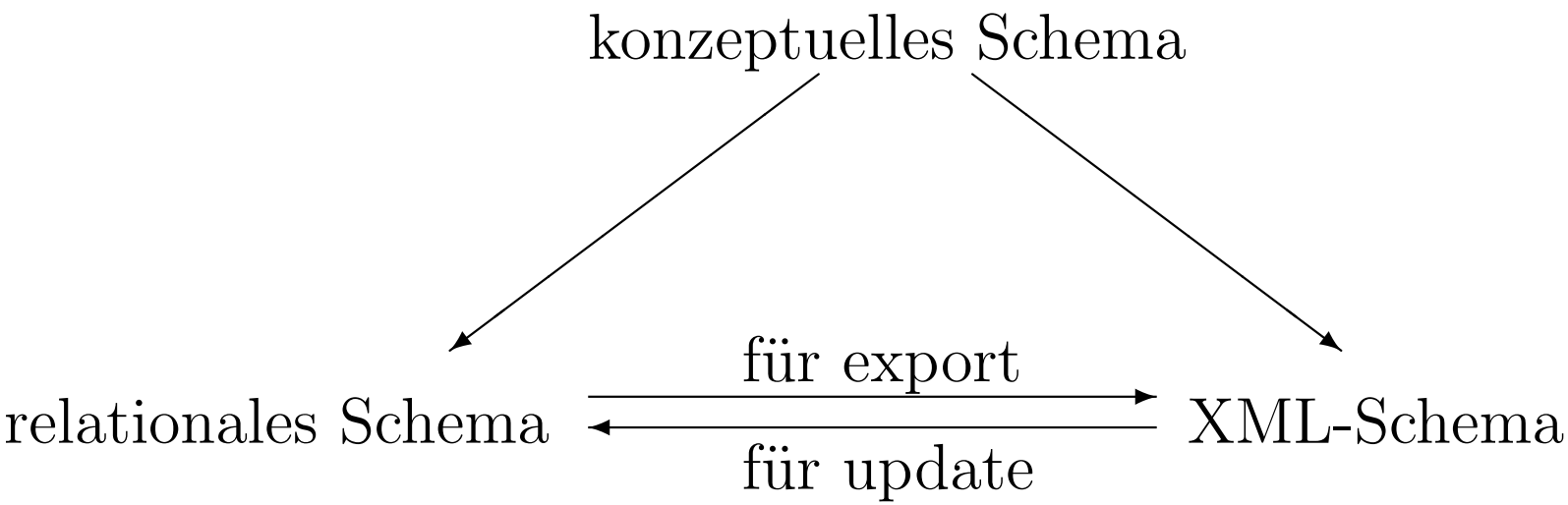


Entwurf unter Legacy: Wie sehen es andere?

Legacy nach [1]



Entwurf unter Legacy



Abbildungen

- ER/UML $\longrightarrow$ DTD/XML-Schema
- DTD/XML-Schema $\longrightarrow$ relationales Schema
- relationales Schema $\longrightarrow$ DTD/XML-Schema

Wir behandeln nur ER $\longrightarrow$ DTD/XML-Schema. Die Abbildung von UML $\longrightarrow$ DTD/XML-Schema erfolgt analog.
Die letzte Abbildung ist relativ trivial und wird hier ebenfalls nicht behandelt. Auch behandeln wir nicht die zugehörige Konversion der Instanzdaten.

Allgemeine Anmerkungen

- Element vs. Attribut
- Element vs. Typ
- Globale vs. lokale Deklaration
- Elemente mit variablem Inhalt
- Beziehungen

Attribut vs. Element

Geschmacksache?

- Intention ist eher: Attribute enthalten nicht für Leser bestimmte Informationen
- Verwendung von Attributen ist kompakter
- Mehrwertige Attribute: nur mittels Listentypen.
- Nullwerte: optionales Attribut, optionales Element
- Komplexe Werte: in Attributen nur atomare Typen erlaubt.

Empfehlung: Elemente bevorzugen

Elemente vs. Typen

Sollte für Element zunächst ein neuer Typ definiert werden, oder soll die Unterstruktur direkt in der Elementdeklaration enthalten sein?

- Abstrakte Elemente: immer Typdeklaration
- Verwendung von Typdeklarationen erhöht Wiederverwendbarkeit.

Empfehlung: Typdeklarationen bevorzugen.

Globale vs. lokale Deklaration

Möglichkeiten:

Russian doll Alle Elemente werden lokal deklariert
Behindert Wiederverwendbarkeit, minimiert Seiteneffekte bei Änderungen, behindert Änderungen aufgrund redundanter Typdeklarationen

salami slice Alle Elemente werden global deklariert und in jedem Kontext nur einmal referenziert.
Damit sind alle Elemente wiederverwendbar und im Instanzdokument kann jedes Element als Wurzelelement dienen.

venetian blind Typdeklarationen global, Elementdeklarationen jedoch für nicht Wurzelelemente lokal

Elemente mit variablem Inhalt

Bsp: Sowohl Personen als auch Unternehmen können Kunden sein

- abstrakter Obertyp
- Verwendung von `choice`

Letztere Möglichkeit bei sich stark unterscheidenden Elementen die einfachere Möglichkeit.

Beziehungen

Umsetzung durch

Hierarchiebeziehung Hier wird die Beziehung durch Schachtelung ausgedrückt. Funktioniert sehr gut für Komposition.

Fremdschlüssel Klassische Lösung. in XML:

- ID/IDREF(S) hat keine Typüberprüfung, nur innerhalb eines Dokumentes
- key/keyref
- Eigenständiges Beziehungselement (für n-m-Beziehungen oft günstiger, Beziehungen mit Attributen)

XLink keine Typüberprüfung möglich

Umsetzung von EER nach DTD/XML-Schema

- Entitytypen und Attribute
- Beziehungen
- Generalisierung

Entitytypen und Attribute

Entitäten werden in XML Schema durch komplexe Typdeklarationen und Elementdeklarationen umgesetzt. Dabei ist zu unterscheiden,

- ob die Typdeklaration global und benannt oder als anonymer Typ erfolgen kann und
- ob die Elementdeklaration global oder lokal erfolgen soll.

- Entscheidungskriterium lokal/global: Wiederverwendbarkeit
- Schwacher Entitytyp: lokal zum Element des zugehörigen starken Entitytyps
- lokal auch für solche Beziehungen, die als Hierarchiebeziehung abgebildet werden (s.u.)
- alle anderen Fälle: globale Variante empfohlen

Attribute

- Attribute mit atomarer Domäne können auch auf XML Attribute abgebildet werden
- ansonsten empfiehlt sich immer die Verwendung von Unterelementen

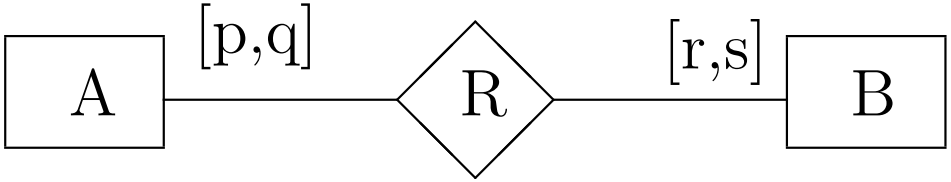
Beziehungen

- Umsetzung als Hierarchiebeziehung
- Umsetzung mittels ID/IDREF(S) ohne Beziehungselement
- Umsetzung mittels key/keyref
- Umsetzung mit Beziehungselement
- Zweiseitige Referenzen
- Umsetzung mit XLink
- Attributierte Beziehungen
- Rekursive Beziehungen
- Mehrstellige Beziehungen

Umsetzung als Hierarchiebeziehung

Einschränkungen:

- Lebensdauer gekoppelt
- nur 1-1 und 1-n Beziehungen



<A>
  <B> ... </B>
  ...
</A>

Umsetzung als Hierarchiebeziehung (DTD)

p	**q**	**Deklaration für A**
0	*	<!ELEMENT A (B*) >
1	*	<!ELEMENT A (B+) >
1	1	<!ELEMENT A (B*) >
0	1	<!ELEMENT A (B?) >
2	5	<!ELEMENT A (B, B, (B, (B, B?)?)? >

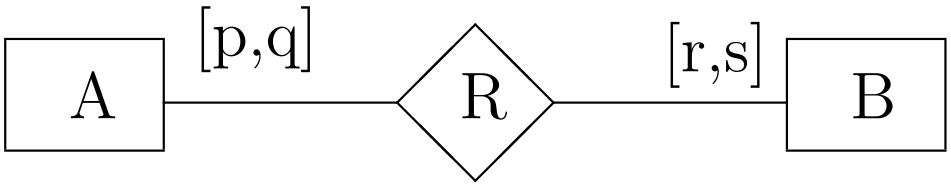
Umsetzung als Hierarchiebeziehung (XSchema)

```
<complexType name = "A">
    <sequence>
        <element name = "B" type = "B"
                 minOccurs = "p" maxOccurs = "q" />
    </sequence>
</complexType>
```

Umsetzung als Hierarchieebene

- Wo immer dies mit der Anwendungssematik übereinstimmt, sollte die Umsetzung mittels Hierarchieebenene vorrang haben. Gründe: Typsicherheit und Kardinalitäten.
- Schwache Entitytypen immer mittels Hierarchien umsetzen.

Umsetzung mittels ID/IDREF(S) ohne Beziehungselement



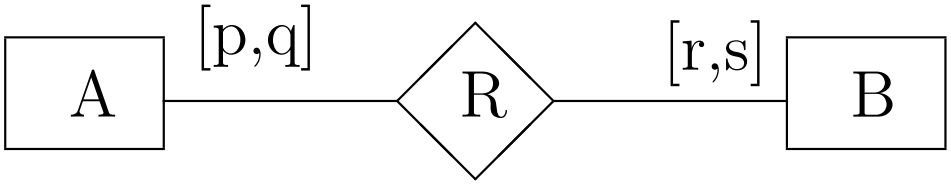
<!ELEMENT A ... >
<!ELEMENT B ... >
<!ATTLIST A id ID # REQUIRED>

r	**s**	Deklaration des verweisenden Attributs für A
0	1	<!ATTLIST B ref IDREF #IMPLIED >
0	*	<!ATTLIST B ref IDREFS #IMPLIED >
1	1	<!ATTLIST B ref IDREF #REQUIRED >
1	*	<!ATTLIST B ref IDREFS #REQUIRED >

Umsetzung mittels ID/IDREF(S) ohne Beziehungselement

- mangelnde Ausdrucksmächtigkeit für Kardinalitäten
- mangelnde Typsicherheit
- stillschweigende Übereinkunft, dass IDREFS keine Duplikate enthält.
- nur innerhalb eines Dokuments
- bei XML Schema statt ID/IDREF(S) key/keyref einsetzen

Umsetzung mittels key/keyref



Annahme: A hat Schlüsselbestandteile *ka1* und *ka2*.
Typdefinition von A:

```
<complexType name = "A">
    <sequence>
        <element name = "ka1" />
        <element name = "ka2" />
        ...
    </sequence>
</complexType>
```

Elementdeklaration für A:

```
<element name = "a" type = "A" />
```

Schlüsseldefinition für A:

```
<key name = "schluesselA">
    <selector xpath = "a" />
    <field xpath = "ka1" />
    <field xpath = "ka2" />
</key>
```

Typdefinition von B

```
<complexType name = "B">
  <sequence>
    ...
    <element name = "ref" minOccurs = "r" maxOccurs = "s">
      <complexType>
        <sequence>
          <element name = "ka1" />
          <element name = "ka2" />
        </sequence>
      </complexType>
    </element>
  </sequence>
</complexType>
```

Elementdeklaration für B

```
<element name = "b" type = "B" />
```

Fremdschlüsseldeklaration für B

```
<keyref name = "fremdschluessel" ref = "schluesselA">
    <selector xpath = "b/ref" />
    <field xpath = "ka1" />
    <field xpath = "ka2" />
</keyref>
```

Umsetzung mit Beziehungselementen

- Analog zur Umsetzung in relationales Schema
- Kardinalitätsinformation geht verloren
- ansonsten Umsetzung mittels ID/IDREF(S) oder key/keyref

Zweiseitige Referenzen

- zweimal ID/IDREF(S) bzw. key/keyref
- Nachteil: Konsistenzsicherung

Umsetzung mit XLink

- Umsetzung als *extended link* mittels Beziehungselementen
- Beziehungselement wird zum *extended link*
- für jedes Beziehungselement muss es ein *locator*-Element geben, das dieses identifiziert
- Sinnvoll: Elemente mit gleichem Typ und demselben *label* versehen
- Zusätzlich muss man noch *arc*-Elemente für die Beziehungen auf Instanzebene einführen
- pro Richtung ein solches Element

Muster für die Verwendung von XLink

Typdefinition für einen XLink-Verweis

```
<complexType name = "XLinkVerweis">
    <attribute ref = "xlink:type" fixed = "locator" />
    <attribute ref = "xlink:href" use = "required" />
    <attribute ref = "xlink:label" use = "required" />
</complexType>
```

Beziehungselement und Beziehungsattribute

```
<complexType name = "R">
    <attribute ref = "xlink:type" fixed = "extended" />
    <sequence>
        <element name = "beziehungsattribute">
            <complexType>
                <attribute ref = "xlink:type" fixed = "resource" />
                <sequence>
                <!-- Unterstruktur für Attribute von R -->
                </sequence>
            </complexType>
```

```
<!-- Verweise auf Instanzen von A: -->
<element name = "rolleA">
<!-- hier evtl. noch Kardinalitäten angeben -->
    <simpleContent>
        <restriction base = "XLinkVerweis">
            <attribute ref = "xlink:label" fixed = "A" />
        </restriction>
    </simpleContent>
</element>
```

```
<!-- Verweise auf Instanzen von B: -->
<element name = "rolleB">
<!-- hier evtl. noch Kardinalitäten angeben -->
    <simpleContent>
        <restriction base = "XLinkVerweis">
            <attribute ref = "xlink:label" fixed = "B" />
        </restriction>
    </simpleContent>
</element>
```

```
<!-- Beziehungsarten (Kanten) von A nach B -->
<element name = "verknuepfungAnachB"
                minOccurs = "0" maxOccurs = "1" >
    <complexType>
        <attribute ref = "xlink:type" fixed = "arc" />
        <attribute ref = "xlink:from" fixed = "A" />
        <attribute ref = "xlink:to" fixed = "B" />
    </complexType>
</element>
```

```
        <!-- ... und B nach A -->
        <element name = "verknuepfungBnachA"
                        minOccurs = "0" maxOccurs = "1" >
            <complexType>
                <attribute ref = "xlink:type" fixed = "arc" />
                <attribute ref = "xlink:from" fixed = "B" />
                <attribute ref = "xlink:to" fixed = "A" />
            </complexType>
        </element>
    </sequence>
</complexType>
```

Schematische Verwendung in Instanzdokument

```
<r>
    <rolleA xlink:href = "#xpointer(...)" />
    <rolleA xlink:href = "#xpointer(...)" />
    <rolleB xlink:href = "#xpointer(...)" />
    <verknuepfungAnachB />
    <verknuepfungBnachA />
</r>
```

Umsetzung mit XLink

- dokumentübergreifend
- keine Typsicherheit
- Kardinalitäten schwierig umsetzbar
- Anwendung nicht intuitiv

Attributierte Beziehungen

- Umsetzung als Hierarchieebene: untergeordnete Elemente für Beziehungsattribute
- Fremdschlüssel: Zuordnung der Attribute zur Fremdschlüsselinstanz
- Beziehungselement: offensichtlich

Rekursive Beziehungen

- Umsetzung als Hierarchieebene: problematisch
- Fremdschlüsselbeziehungen: unproblematisch
- XLink: unproblematisch

Mehrstellige Beziehungen

- am einfachsten durch Beziehungselement: allerdings gehen Kardinalitäten verloren
- u.U. ohne Beziehungselement, falls dieses sich direkt einem Element zuordnen läßt. (bei Funktionalitäten dort, wo die eins steht)

Spezialisierung/Generalisierung

- Generalisierung mit DTDs
- Generalisierung mit XSchema

Generalisierung mit DTDs

Nur Wirkung der Generalisierung nachbildbar

- Vererbung durch Parameterentities durch
  - Parameterentities
  - Einbetten der Oberklassen repräsentierenden Elemente

Beispiel für Nutzung von Parameterentities

```
<!ENTITY % katalogelement "(artikel|dienstleistung)">
<!ENTITY % katalogelement.attr "(katalogelementID, name,
                                 beschreibung?)">
<!ELEMENT katalog (%katalogelement;+)>
<!ELEMENT artikel (%katalogelement.attr;,
                   hersteller, lieferzeit?, bild?, stueckpreis
<!ELEMENT dienstleistung (%katalogelement.attr;,
                          stundensatz)>
```

Generalisierung mit XML Schema

Generalisierung mittels XSchema in drei Varianten:

1. Extensionsableitung von Typen nutzen
2. Umsetzung mit Substitutionsgruppen
3. Falls Untertypen keine weiteren Attribute haben: Obertyp wird Typ, Instanzen des Untertyps sind Instanzen des Typs. (evtl. typ-enum)

1 ist beste Möglichkeit

SQL/XML

Constituents:

- XML Publishing Functions
- XML Data Type
- XQuery Functions
- Type/Value Mappings

SQL/XML: XML Publishing Functions

Konstruiere XML (z.B. aus relationalen Daten)

- XMLELEMENT, XMLATTRIBUTES
- XMLAGG
- XMLFOREST
- XMLCONCAT

SQL/XML: XML Publishing Functions

```
SELECT XMLELEMENT(NAME "title", title) AS movietitles
FROM   movies
```

XMLELEMENT erzeugt leeres Element für Null-Wert.

SQL/XML: XML Publishing Functions

```
SELECT
  XMLELEMENT(NAME "title",
    XMLATTRIBUTES(runningtime AS "RunningTime", -- 1. Attribut
                  (SELECT familyName
                   FROM   directors as d
                   WHERE  d.id=m.director)
                  AS "Director"),                -- 2. Attribut
    'This movie title is ' || m.title) AS movietitles
FROM movies AS m
```

SQL/XML: XML Aggregate Function

- nur XMLELEMENT: ein Tuple mit einem Attribut, das ein Element für jedes Tupel, das durch FROM, WHERE erzeugt wird, enthält.
- XMLAGG erzeugt eine Sequenz von Knoten; ein Knoten für jedes Tupel, das durch FROM, WHERE erzeugt wird. (wie normale Aggregatfunktion)

SQL/XML: XML Aggregate Function

```
SELECT
  XMLELEMENT(NAME "all-titles",
    XMLELEMENT(NAME "title", title)) as movietitles
FROM MOVIES
```

Ergebnis:

```
<all-titles><title>Blow Up</title></all-titles>
<all-titles><title>Themroc</title></all-titles>
```

SQL/XML: XML Aggregate Function

```
SELECT
  XMLELEMENT(NAME "all-titles",
    XMLAGG(XMLELEMENT(NAME "title, title)))
    as movietitles
FROM movies
```

Ergebnis:

```
<all-titles>
  <title>Blow Up</title>
  <title>Themroc</title>
</all-titles>
```

SQL/XML: XMLForest

- kürzt mehrere Aufrufe von XMLELEMENT ab
- ohne Angabe eines Elementnamens erzeugt XMLFOREST für jedes Attribut ein Element mit dem Namen des Attributs
- XMLFOREST ignoriert Null-Werte
- es können keine Attribute zu dem erzeugten Elementen hinzugefügt werden

SQL/XML: XMLForest

```
SELECT
  XMLELEMENT(NAME "movie-details",
    XMLELEMENT(NAME "title", title),
    XMLELEMENT(NAME "yearReleased", yearreleased),
    XMLELEMENT(NAME "runningTime", runningtime))
FROM movies
```

kann abgekürzt werden zu

SQL/XML: XMLForest

```
SELECT
  XMLELEMENT(NAME "movie-details",
    XMLFOREST(title as "title",
              yearReleased as "yearreleased",
              runningtime  as "runningTime"))
FROM movies
```

oder, ohne Attributnamen:

SQL/XML: XMLForest

```
SELECT
  XMLELEMENT(NAME "movie-details",
    XMLFOREST(title, yearReleased, runningtime))
FROM movies
```

SQL/XML: XMLCONCAT

XMLCONCAT combines a list of individual XML values to create a single value containing an XML sequence

```
SELECT XMLCONCAT(
         XMLELEMENT("givenName", givenname),
         XMLELEMENT("familyName", familyName))
FROM directors
```

SQL/XML: Others

- XMLROOT constructs a document root
- XMLCOMMENT constructs a comment node
- XMLPI constructs a processing instruction node

SQL/XML: noch ein Beispiel

```
select xmlelement(name "customer",
         xmlattributes(c.id as id),
         xmlforest(c.name, c.city),
         xmlelement(name "project",
                    (select xmlagg(xmlelement(name "project",
                                              xmlattributes(p.id),
                                              xmlforest(p.name)))
                     from   Projects p
                     where  p.custid = c.id)))
from   Customer c
```

SQL/XML: XML Data Type

Typename: `XML`, Values:

- `NULL`
- XML-Inhalt (XML-Element oder Sequenz von Knoten, Werten)
- XML-Dokument

Unterstützt Zuweisung, aber nicht Vergleich.

XML Type (Beispiel)

```
create table Angestellte (
  id         int,
  gehalt     decimal(10,2),
  bewerbung xml
);
```

SQL/XML: XML Data Type: type modifier

Typehierarchie: hier von oben nach unten.

1. XML(SEQUENCE) irgendeine Sequenz von Knoten und/oder Werten
2. XML(CONTENT) Sequenz von wohlgeformten XML-Fragmenten
3. XML(DOCUMENT) muss wohlgeformtes Dokument sein

SQL/XML: XML Data Type: type modifier

SQL/XML erlaubt noch einen zweiten Typmodifikator:

- UNTYPED untypisiert
  (xdt:untypedAtomic für Elemente, xdt:untypedAtomic für Attribute)

```
XML(DOCUMENT(UNTYPED))
```

- XMLSCHEMA
  Gültigkeit bzgl. Schema wird verlangt

```
XML( DOCUMENT( XMLSCHEMA ID "movies.xsd" ) )
```

- ANY
  wahlweise mit Schema oder ohne

SQL/XML: String $\Longleftrightarrow$ XML

- XMLPARSE
- XMLSERIALIZE

SQL/XML: XQuery Functions

- XMLQUERY wertet XQuery-Ausdruck aus
- XMLTABLE spezifiziert eine berechnete Relation (derived table)
- XMLEXISTS evaluiert einen XQuery-Ausdruck und testet, ob das Ergebnis leer ist

SQL/XML: XQuery Functions: Beispieldaten

```
ID | MOVIE_XML
--------------
42 | <movie>
   |   <title>...</title>
   |   <yearReleased>...</yearReleased>
   | </movie>
43 | <movie>
   |   ...
   | </movie>
```

SQL/XML: XQuery Functions: XMLQUERY

Syntax:

```
XMLQUERY(
  <xquery-expr>
  [ PASSING BY { REF | VALUE } {<var> as <expr>, ','}* ]
  [ RETURNING {CONTENT | SEQUENCE} [BY {REF | VALUE}]  ]
  {NULL | EMPTY} ON EMPTY
)
```

SQL/XML: XQuery Functions: XMLQUERY

- PASSING: binden von Variablen, die in <xquery-expr> benutzt werden
- RETURNING CONTENT: Ergebnis wird serialisiert und per Wert an die SQL-Engine geliefert
- RETURNING ... REF: Rückgabe via Referenz, erlaubt auch Navigieren zu Parent
- NULL ON EMPTY: Falls <xquery-expr> zu leerer Sequenz evaluiert ist der Rückgabewert NULL; EMPTY on EMPTY: leere Sequenz wird zurückgegeben

SQL/XML: XQuery Functions: XMLQUERY

```
SELECT
  XMLQUERY('
    for $m in $col/movie
    return $m/title'
    PASSING movie AS "col"
    RETURNING CONTENT
    NULL ON EMPTY
  ) as result
FROM MOVIES_XML
```

SQL/XML: XQuery Functions: XMLQUERY

```
SELECT
  XMLQUERY('
    for $m in doc("movies.xml")/movies/movie
    return $m/title'
    RETURNING CONTENT
    NULL ON EMPTY
  ) as result
FROM DUAL -- in Oracle eingebaute Relation mit einem Tupel
```

SQL/XML: XQuery Functions: XMLQUERY

```
SELECT
  AVG(
    XMLCAST(
      XMLQUERY('
        for $m in $col/movie
        return $m/runningTime/text()'
        PASSING movie as "col"
        RETURNING CONTENT
        NULL ON EMPTY)
      AS decimal(8,1))
  ) AS "avgRunningTime"
FROM MOVIES_XML
```

SQL/XML: XQuery Functions: XMLQUERY

```
SELECT XMLQUERY('
  for $m in $col/movie
  let $producers := $m/producer
  where $m/yearReleased > 1950
  return <output> {$m/title}
    {for $p in $producers
     return <prodFullName>
              {concat($p/givenName, " ", $p/familyName)}
            </prodFullName>
    }</output>
  PASSING movie as "col" RETURNING CONTENT NULL ON EMPTY
  ) as "Results"
FROM MOVIES_XML
```

SQL/XML: XQuery Functions: XMLTABLE

XMLTABLE erzeugt eine Relation. Syntax:

```
XMLTABLE (
  [ namespace-declaration, ]
  xquery-expr,
  [ PASSING argument-list ]
  COLUMNS column-definitions
```

where `column-definitions` may contain

```
column-name FOR ORDINALITY
```

and/or list of

```
column-name data-type
[ BY REF | BY VALUE ]
[ default-clause ]
[ PATH xquery-expr ]  (: the column pattern :)
```

SQL/XML: XQuery Functions: XMLTABLE

column definitions:

- `column-name AS ORDINALITY`: durchnumerieren, in Document Order
- `PATH xquery-expr`: xquery-expr kann beliebiger XQuery-Ausdruck sein, ist aber vermutlich in den meisten Fällen ein XPath-Ausdruck
- `default-clause`: default-value, falls `PATH xquery-expr` leere Menge ergibt

SQL/XML: XQuery Functions: XMLTABLE

```
SELECT result.*
FROM
  movies_xml,
  XMLTABLE('
    for $m in $col/movie
    return $m/title'
    PASSING movies_xml.movie AS "col") AS result
```

SQL/XML: XQuery Functions: XMLTABLE

```
SELECT result.*
FROM movies_xml,
     XMLTABLE('
       for $m in $col/movie
       return $m'
       PASSING movies_xml.movie AS "col"
       COLUMNS
        "title"         VARCHAR(80)  PATH 'title',
        "runningTime"   INTEGER      PATH 'runningTime',
        "yearReleased" INTEGER       PATH 'yearReleased'
     ) AS result
```

SQL/XML: XQuery Functions: XMLTABLE

Attributname ist Default-Path:

```
SELECT result.*
FROM movies_xml,
     XMLTABLE('
       for $m in $col/movie
       return $m'
       PASSING movies_xml.movie AS "col"
       COLUMNS
         "title" VARCHAR(80),
         "runningTime" INTEGER,
         "producer[1]/familyName", VARCHAR(20)
     ) AS result(TITLE, RUNNINGTIME, YEARRELEASED, PRODUCER)
```

SQL/XML: XQuery Functions: XMLTABLE

Mehrfachwerte

- Denormalisierung (mehrere Attribute)
- XML
- Denormalisierung (mehrere Tupel), Detailrelation

SQL/XML: XQuery Functions: XMLTABLE: Denormalisierung

```
SELECT result.*
FROM movies_xml,
  XMLTABLE('for $m in $col/movie return $m'
    PASSING movies_xml.movie AS "col"
    COLUMNS
      "title" VARCHAR(80) PATH 'title',
      "producer1" VARCHAR(12) PATH 'producer[1]/familyName',
      "producer2" VARCHAR(12) PATH 'producer[2]/familyName'
                  DEFAULT 'none',
      "producer3" VARCHAR(12) PATH 'producer[3]/familyName'
                  DEFAULT 'none'
  ) AS result
```

SQL/XML: XQuery Functions: XMLTABLE: XML

```
SELECT result.*
FROM movies_xml,
  XMLTABLE('for $m in $col/movie return $m'
    PASSING movies_xml.movie AS "col"
    COLUMNS
      "title" VARCHAR(80) PATH 'title',
      "producers" XML PATH 'producer'
  ) AS result
```

SQL/XML: XQuery Functions: XMLTABLE: Detail Table

```
SELECT result."title", result2.*
FROM movies_xml,
  XMLTABLE('for $m in $col/movie return $m'
    PASSING movies_xml.movie AS "col"
    COLUMNS
      "title" VARCHAR(80) PATH 'title',
      "producers" XML PATH 'producer'
  ) AS result,
  XMLTABLE('for $prod in $p/producer return $prod'
    PASSING result."producers" AS "p"
    COLUMNS
      "ord" FOR ORDINALITY,
      "familyName" VARCHAR(20) PATH 'familyName'
  ) AS result2
```

Ergebnis

```
title                              ord familyName
-------------------------------- --- ----------
An American Werewolf in London       1  Folsey
An American Werewolf in London       2  Guber
An American Werewolf in London       3  Peters
Animal House                         1  Simmons
Animal House                         2  Reitman
```

SQL/XML: Mappings

- Mapping SQL character sets to Unicode
- Mapping SQL identifiers to XML Names
- Mapping SQL data types to XML Schema data types
- Mapping values of SQL data types to values of XML Schema data types
- Mapping an SQL table an XML document and an XML Schema document
- Mapping an SQL schema to an XML document and an XML Schema document
- Mapping an SQL catalog to an XML document and an XML Schema document
- Mapping Unicode to SQL character sets
- Mapping XML Names to SQL identifiers

XML Support in kommerziellen RBMS

1. IBM DB2 (unterstützt SQL/XML)
2. Oracle (unterstützt SQL/XML)
3. Microsoft SQL Server (SQLXML)

SQL Server

1. XML Publishing
2. XML Storage and Querying

SQL Server: XML Publishing

- FOR XML Zusatz zu SQL Anfragen
- Syntax: `FOR XML` xml-modus [,XMLDATA] [,ELEMENTS] [,BINARY BASE64]
- Drei Modi: RAW, AUTO, EXPLICIT
- XMLDATA: generiert XML Data Schema
- ELEMENTS: Ergebnis benutzt geschachtelte Elemente statt Attribute
- BINARY BASE64: Binärdaten werden BASE64 codiert

SQL Server: XML Publishing: Modi

- RAW: Jedes Tupel als row-Element mit XML-Attributen für Ergebnisattribute
- AUTO: Schachtelt gemäß der Einträge in der `FROM` Klausel:
  - erster Eintrag gibt äußeres Element
  - andere werden darin geschachtelt.
- EXPLICIT: Erweiterung der `SELECT` Klausel um spezielle Syntax um Ergebnisaufbau zu spezifizieren

SQL Server: XML Publishing: Beispielanfrage

```
SELECT Artikel.katalogelementId,
       Artikel.name,
       Bestellung.beschreibung
FROM Artikel INNER JOIN Bestellung ON
  Artikel.katalogelementId = Bestellung.katalogelementId
WHERE Artikel.katalogelementId = 407
```

SQL Server: XML Publishing: RAW Modus

```
<row katalogelementId = "407"
     name = "10 kg Sack Zement"
     beschreibung = "Auch zur Anwendung im Freien geeignet"/>
```

SQL Server: XML Publishing: AUTO Modus

```
<artikel katalogelementId = "407"
         name = "10 kg Sack Zement">
         <bestellung beschreibung = "Nachlieferung"/>
</artikel>
```

SQL Server: XML Publishing: AUTO Modus mit ELEMENTS Direktive

```
<artikel>
  <katalogelementID>407</katalogelementId>
  <name>10 kg Sack Zement</name>
  <bestellung>
    <beschreibung>Nachlieferung</beschreibung>
  </bestellung>
</artikel>
```

SQL Server: XML Publishing: EXPLICIT Modus

Im EXPLICIT Modus werden einige zusätzliche Konstrukte eingesetzt:

- Spalten-Aliase erhalten eine neue Syntax, über die die Strukturierung des Ziel-XML-Dokuments gesteuert wird.
- Über zwei Metadaten-Spalten am Anfang einer jeden SELECT-Anweisung werden zusätzliche Steuerungsbefehle für die XML-Generierung kodiert.
- Direktiven steuern das Ausgabeformat der einzelnen Spaltenelemente.

SQL Server: XML Publishing: Erweiterung von AS

```
SELECT <Spalte>
       AS [ElementName!TagNumber!PropertyName!Directive]
```

Hierbei ist

- `ElementName` der Name des Elements, in dem die Daten im XML-Dokument ausgegeben werden,
- `TagNumber` eine eindeutige Nummer für jedes Element,
- `PropertyName` der Name des Elements oder Attributs, in das der Wert der Spalte geschrieben wird,
- `Directive` die Form, in der die Daten in das XML-Dokument eingefügt werden sollen. Zur Auswahl stehen neben `element` auch noch Anweisungen wie `id`, `xmltext` oder `cdata`. Gibt man die Direktive nicht an, wird ein Attribut Mapping verwendet, d.h. der `PropertyName` ist ein Attributname.

SQL Server: XML Publishing: Beispiel: Zieldokument

Wir gehen rückwärts vor um die Dinge zu veranschaulichen.

```
 <katalog name="Moebel">
      <artikel name="Regal"/>
      <artikel name="Tisch"/>
</katalog>
<katalog name="Farben">
      <artikel name="Blau"/>
      <artikel name="Gruen"/>
</katalog>
```

SQL Server: XML Publishing: Ausgangsergebnisrelation

Tag	Parent	katalog!1!name	artikel!2!name
1	NULL	Möbel	NULL
2	1	Möbel	Regal
2	1	Möbel	Tisch
1	NULL	Farben	NULL
2	1	Farben	Blau
2	1	Farben	Grün

Falsche SQL Anfrage im Buch zur Erzeugung dieser Relation

```
SELECT 1 as TAG,
       NULL as PARENT,
       artikel.katalogelementId as [artikel!1!ID],
       artikel.name as [artikel!1!name!element],
       bestellung.beschreibung as
              [artikel!1!beschreibung!element]
FROM artikel INNER JOIN Bestellung ON
  artikel.katalogelementId = bestellung.katalogelementId
WHERE artikel.katalogelementId = 407
FOR XML EXPLICIT
```

richtige: Übung

SQL Server: XQuery

- um auf SQL-Variablen und Attribute in XQuery-Ausdruck zugreifen zu können:

  ```
  sql:variable(<sqlvar-name>)
  sql:column(<column-name>)
  ```

- Funktionen, die XQuery-Ausdruck als Parameter bekommen:

  ```
  query, nodes, exist, value
  ```

  "Receiver" wird Kontextknoten
- XQuery: sehr eingeschränkt:
  - kein `let`
  - viele Achsen fehlen
  - viele Funktionen fehlen

SQL Server: XQuery

Mit table00(id int, doc XML):

```
SELECT doc.query('for $author in distinct-values(//author)
                  return $author') as RESULT
FROM table00
WHERE id = 1
```

SQL Server: XQuery

```
SELECT id,
       doc.QUERY('
         for $paper in /dblp/inproceedings
         return
          <paper>
            { $paper/title[1] }
            { $paper/year[1] }
          </paper>
       ')
FROM table00
WHERE id = 2
```

SQL Server: XQuery

```
SELECT id,
       paper.VALUE('title[1]', 'varchar(100)') as title,
       paper.VALUE('year[1]', 'int') as year
FROM table00 CROSS APPLY
     doc.NODES('/dblp/inproceedings') as papers(paper)
WHERE id = 2
```


W. Kazakos, A. Schmidt, and P. Tomczyk.
Datenbanken und XML.
Springer, 2002.


J. Melton and S. Buxton.
Querying XML.
Morgan Kaufman, 2006.


P. Walmsley.
Definitive XML Schema.
Prentice Hall, 2002.