Orientation in Objects

Überblick: W3C XML Schema 1.1

Zeigt die neuen Features wie Assertions, bedingte Typisierung und mehr.

) Schulung)

) Beratung)

) Entwicklung)

) Artikel)

AUTOR

Matthias Born
Orientation in Objects GmbH

Veröffentlicht am: 8.2.2011

ABSTRACT

Das Ende 2004 veröffentlichte W3C XML Schema ist eine seit Jahren bewährte Methode, um XML Dokumente zu validieren. Der neue XML Schema 1.1 Working-Draft übernimmt u.a. die aus Schematron bekannten Assertions und fügt Features wie schemaweite Attribute, bedingte Typisierung und "openContent" dem Standard hinzu. Dieser Artikel bietet eine kurzen Überblick über die neuen Möglichkeiten.

Orientation in Objects GmbH

Weinheimer Str. 68
D-68309 Mannheim

Tel. +49 (0) 6 21 - 7 18 39 - 0
Fax +49 (0) 6 21 - 7 18 39 - 50

www.oio.de info@oio.de

EINLEITUNG

Das **W3C XML Schema**, kurz XSD, ist die umfangreichste und am weitesten verbreitete Technologie, um die Struktur von XML Dokumenten festzulegen und anschließend diese Dokumente gegen das Schema zu validieren. Der Standard wurde seit 2001 entwickelt und Ende 2004 erstmalig als Empfehlung veröffentlicht. Diese Version weißt jedoch einige wenige Lücken auf, die der aktuelle Working Draft "W3C XML Schema 1.1" zum Teil beheben soll. Am deutlichsten wird das, wenn man sich einmal Schematron betrachtet, einer XSLT basierten Sprache zur nachträglichen, im Unterschied zu XSD nicht abweisenden Validierung vom XML Dokumenten. Schematron ermöglicht es, über sogenannte Assertions (zu Deutsch: Zusicherungen) Bedingungen für verschiedene Werte zu formulieren und diese in Form von Berichten auszuwerten. Diese Assertions, die bedingte Typisierung und einige weitere Änderungen sollen in diesem Artikel vorgestellt werden.

DIE NEUERUNGEN IM ÜBERBLICK:

- Assertions in Form des `<assert>`-Elementes und einer `<assertion>`-Fassette
- Bedingte Typisierung - Alternative Datentypen in Abhängigkeit von Attributwerten
- Schemaweite Attribute
- Flexiblere Inhalte - `<openContent>` bzw. `<defaultOpenContent>`
- Wildcard Schema Components - `<any>` und `<anyAttribute>`
- "Versionsverwaltung"

ASSERTIONS — ZUSICHERUNGEN

Die wohl wichtigste Neuerung in XSD 1.1: Zu der bisher rein auf Grammatik basierenden Validierung kommt jetzt ein Feature, für bisher vor allem Schematron bekannt war: Die regelbasierte Validierung in Form von Assertions.

Die Assertions formulieren mittels XPath-Ausdrücken Bedingungen bzw. Regeln, die die Werte von Elementen oder Attributen erfüllen müssen, damit das XML Dokument valide ist. Wer sich ein bisschen mit XSLT/XPath auskennt, wird schnell erkennen: Die Möglichkeiten von XPath gehen weit über das hinaus, was bisher über Wertebereichsangaben oder Fassetten wie Auswahllisten, Mini-, Maximalwerte und Reguläre Ausdrücke zur Deklaration der eigentlichen Datentypen machbar war. Es ist sogar möglich, verschiedene Element- oder Attributwerte miteinander in Beziehung zu setzen und zu vergleichen. Genau diese Vergleiche waren es, die Schematron als zusätzliche Validierungsstufe bisher so verlockend erscheinen ließen.

Mögliche Anwendungsbeispiele:

- Wenn das Fahrzeug vom Typ "Motorrad" ist, sollte es nicht mehr als 3 Räder haben...
- Wenn eine Lieferung per Luftfracht erfolgen soll, dann kann das Transportmittel schlecht ein Auto oder Boot sein.
- Bestellung von Verbrauchsmaterial: Je nach Preis muss die Freigabe durch eine höhere Position erfolgen: Teamleiter: 0-99 Euro, Abteilungsleiter: 100- 999 Euro usw.
- Terminkalender: Die Startzeit eines Termins muss kleiner sein als dessen Ende.

Assertions lassen sich auf zwei Wegen realisieren:

- dem `<assert>`-Element für komplexe Datentypen und
- der `<assertion>`-Fassette für simple Datentypen.

DAS <ASSERT>-ELEMENT

Das `<assert>`-Element wird verwendet, um Regeln für komplexe Typen zu formulieren. Die Syntax ist denkbar einfach: `<assert test="`*xpath*`" />`. Der im `test`-Attribut formulierte XPath-Ausdruck wird während der Validierung zu true oder false ausgewertet. Liefert er false zurück, sind die geforderten Bedingungen nicht erfüllt und das Dokument demnach nicht valide. Dabei ist zu beachten: "Assertions always look down", d.h. die benutzten XPath-Ausdrücke dürfen sich nur auf die Attribute und Nachfahren des aktuell deklarierten Elementes beziehen. Ausschlaggebend für den Einsatz von Assertions ist demnach, den korrekten Ausgangspunkt (Kontext-Knoten) für die XPath-Ausdrücke zu ermitteln. Werden mehrere Assertions formuliert, werden diese "verundet" und alle müssen erfüllt werden, damit das Element valide ist. Die in XSD 1.1 verwendete XPath-Version ist XPath 2.0.

Beispiel Geheimhaltung: Unser XML Schema beschreibt ein XML Dokument, das aus beliebig vielen Absätzen zusammengesetzt wird. Alle Absätze sind nach einer Geheimhaltungsstufe klassifiziert. Aber auch das Root-Element und somit das Dokument selber. Folgende Bedingung muss erfüllt sein: Die Geheimhaltungsstufe der einzelnen Absätze darf die des gesamten Dokuments nicht überschreiten. Nur gleiche oder als weniger geheim eingestufte Informationen dürfen enthalten sein.

```
<Document classification="secret">
  <Para classification="unclassified">...</Para>
  <Para classification="secret">...</Para>
  <Para classification="unclassified">...</Para>
  <Para classification="secret">...</Para>
</Document>
```

Beispiel 1: Assertions, Beispiel Geheimhaltung, XML Dokument

```
<xs:element name="Document">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Para" type="ParaType" maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="classification" type="classificationLevels" use="required"/>
    <xs:assert test="if (@classification eq 'secret') then ... else … true()" />
  </xs:complexType>
</xs:element>
```

Beispiel 2: Assertions, Beispiel Geheimhaltung, XML Schema Dokument

```
if (@classification eq 'secret') then not(Para/@classification = 'top-secret')
else if (@classification eq 'confidential')
    then not(Para/@classification = 'top-secret')
     and not(Para/@classification = 'secret')
else if (@classification eq 'unclassified')
    then not(Para/@classification = 'top-secret')
     and not(Para/@classification = 'secret')
     and not(Para/@classification = 'confidential')
else true()
```

Beispiel 3: Vollständiger XPath Ausdruck

CONDITIONAL PRESENCE

Eine weitere Einsatzmöglichkeit des `<assert>`-Elements ist die "Conditional Presence", also verschiedene (Kind-) Elemente in Abhängigkeit von Attributwerten.

Hier am o.g. Transportbeispiel: In Abhängigkeit vom Attribut Transport-`mode`, werden nur bestimmte Kind-Elemente zugelassen:

```
<xs:element name="Transportation">
  <xs:complexType>
    <xs:choice>
      <xs:element name="airplane" type="xs:string" />
      <xs:element name="boat" type="xs:string" />
      <xs:element name="car" type="xs:string" />
    </xs:choice>
    <xs:attribute name="mode" type="modeType" use="required"/>
    <xs:assert test="
       if (@mode eq 'air') then airplane
       else if (@mode eq 'water') then boat
       else if (@mode eq 'ground') then car
       else false()"/>
  </xs:complexType>
</xs:element>
```

Beispiel 4: Kind-Elemente mittels <assert> prüfen

Da die Validierung mittels Assertions sich nur auf die Nachfahren eines Elementes bezieht, ist es leider nicht möglich, gegen externe Dokumente zu validieren:

```
<xs:element name="Standort">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Land" type="xs:string" />
    </sequence>
    <xs:assert test="country = doc('countries.xml')//country" />
  </xs:complexType>
</xs:element>
```

Beispiel 5: Validierung gegen externe Dokumente nicht möglich

<ASSERTION>-FASSETTE

Die `<assertion>`-Fassette wird verwendet, um simple Datentypen mittels XPath-Ausdrücken noch weiter einzuschränken, als das über die gängigen Wertebereiche bzw. Fassetten möglich ist. Zum Beispiel die zusätzliche Einschränkung eines beliebigen Wertebereiches auf ausschließlich gerade Zahlen. Diese neue Fassette kann für jeden beliebigen Datentyp eingesetzt werden. Ebenfalls sind wieder mehrere Assertions möglich. Außerdem gibt es eine Build-In-Variable **$value**, die den Wert des Datentyps enthält und auf die innerhalb des XPath-Ausdrucks zugegriffen werden kann.

```
<xs:simpleType name="gerade>
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="0" />
    <xs:maxInclusive value="100" />
    <xs:assertion test="$value mod 2 = 0" />
  </xs:restriction>
</xs:simpleType>
```

Beispiel 6: Assertion-Fassette für gerade Zahlen

Die Einschränkungen, dass die XPath-Ausdrücke nur Zugriff auf die Nachfahren eines Elementes haben, sind der "streaming validation" großer XML Dokumente geschuldet. Nur so ist gewährlistet, das jederzeit auf alle benötigten Informationen zugegriffen werden kann (SAX-Validierung). Leider hat das zur Folge, dass die Assertions in XSD 1.1 niemals an den Funktionsumfang des DOM/XSLT/XPath basierten Schematron heran reichen werden.

Ein möglicher Ausweg aus diesem Dilemma könnte die Vererbung von Attributen sein. Dabei werden die Attribute eines Elementes, das ein Vorfahr sein muss aber trotzdem außerhalb des aktuellen Kontextes steht, als `inheritable="true"` gekennzeichnet und an alle Kind-Elemente weiter vererbt. Diese Attribute können von hier ab verwendet werden, ohne dass sie im XML Dokument erscheinen müssen. Dadurch können die XPath-Ausdrücke der Assertion auf Informationen zugreifen, die normalerweise nicht in ihrer Reichweite liegen. Leider funktioniert diese Vorgehensweise nur bei Attributen, so dass man schon fast gezwungen wird, mehr Informationen in Attribute auszulagern, obwohl sie dort vielleicht gar nicht so gut aufgehoben sind. Diese Möglichkeit ist also eher als Notlösung denn als Königsweg zu verstehen, wenn Strukturen geändert werden müssen, um Validierung zu ermöglichen.

```
<xs:element name="Bibliothek">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Buch" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <!-- -->
          </xs:sequence>
          <xs:assert test="if (@xml:lang='en') then Verlag eq 'Wrox Press'
                      else if (@xml:lang='fr') then Verlag eq 'Bayard Presse'
                      else false()" />
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute ref="xml:lang" inheritable="true" />
  </xs:complexType>
</xs:element>
```

Beispiel 7: Vererbtes Attribute "xml:lang" und Zugriff via Assertion

CONDITIONAL TYPE ALTERNATIVES (CTA) — BEDINGTE TYPISIERUNG

In XSD 1.1 wird es möglich sein, unterschiedliche Datentypen für ein Element auszuwählen. Dies "Datentyp-Alternativen" werden in Abhängigkeit von Attribut-Werten mittels XPath-Ausdrücken selektiert. Im einfachsten Fall ist die Syntax wie folgt: `<alternative test="`*xpath*`" type="`*type*`" />`. Dabei gilt: Im `test`-Attribut wird die Bedingung formuliert und das `type`-Attribut wird der zu verwendete Datentyp festgelegt.

Folgende Bedingungen müssen erfüllt sein:

- Die alternativen Typen müssen vom Default-Element-Typ abgeleitet sein.
- "CTA can't look up, can't look down": Es können ausschließlich am Element deklarierte Attribute zur Auswahl des Datentyps herangezogen werden.

Werden mehrere CTA formuliert, wird diejenige ausgewählt, deren `test`-Attribut zuerst zu **true** ausgewertet wird.

Und: Der alternative Datentyp muss nicht global, sondern kann auch inline im `<alternative>`-Element deklariert werden.

Als Beispiel verschiedene Veröffentlichungen: Deren Aufbau ist abhängig vom Attribut "typ". Für Buch bzw. Artikel werden spezielle Typen verwendet, für alle anderen der `publicationType`.

```
<Veroeffentlichung typ="Buch|Artikel|Blogpost">
  <!-- Vom Attribut "typ" abhängiger Aufbau -->
</Veroeffentlichung>
```

Beispiel 8: Bedingungen bestimmen den verwendeten Datentyp - XML

```
<xs:element name="Veroeffentlichung" type="publicationType">
  <xs:alternative test="@typ eq 'Buch'" type="bookType" />
  <xs:alternative test="@typ eq 'Artikel'" type="articleType" />
</xs:element>
```

Beispiel 9: Bedingungen bestimmen den verwendeten Datentyp - XSD

Für den Fall, dass ein fremdes Schema importiere wurde, aber die zulässigen Werte nur auf Buch und Artikel beschränkt werden sollen, könnte noch eine dritte Alternative mit dem ebenfalls neuen `error`-Datentyp formuliert und alle anderen Datentypen somit verboten werden:

```
<xs:alternative test="(@typ ne 'Buch') and (@typ ne 'Artikel')" type="xs:error" />
```

Beispiel 10: Alternative mit Datentyp `error`

Der `error`-Datentyp dient dabei dem expliziten Abbruch der Validierung, wenn die durch die XPath-Ausdrücke beschriebenen Bedingungen erfüllt sind.

BEDINGTE TYPISIERUNG MITTELS ASSERTIONS

Alternativ können CTAs auch über Assertions abgebildet werden. Dabei werden jedoch keine vordefinierten Datentypen ausgewählt, sondern mittels komplexer XPath-Ausdrücke verschiedene Nachfahren zugelassen oder verboten. Der Vorteil: Es können mehr Daten als nur die Attribute des Elternelements zur Auswahl des "Datentyps" berücksichtigt werden:

```
<xs:assert test="
 if (@kind eq 'book') then Title and Date and ISBN and Publisher
    and empty(* except (Title[1], Date[1], Author, ISBN[1], Publisher[1]))
 else if (@kind eq 'magazine') then Title and Date
    and empty(* except (Title[1], Date[1]))
 else Title and Date
    and empty(* except (Title[1], Date[1], Author))" />
```

Beispiel 11: Assertion statt CTA

SCHEMAWEITE ATTRIBUTE — DEFAULT ATTRIBUTES

In Fällen, in denen ein oder mehrere Attribute an nahezu allen Elementen erscheinen dürfen, gab es in XSD 1.0 zwei Möglichkeiten:

1. Es wird eine Attributgruppe erstellt und diese zu allen komplexen Typen hinzugefügt.
2. Alle verwendeten Datentypen erben von einem komplexen abstrakten Grunddatentyp, der nichts weiter deklariert als die benötigten Attribute.

Um diese Sonderfälle zu vereinfachen, werden in XSD 1.1 schemaweite Attribute eingeführt. Ein Anwendungsbeispiel könnten die HTML-Standard-Attribute `style`, `class` und `id`
Dazu wird eine Attributgruppe erstellt und diese über das `<schema>`-Attribute `default-attributes` als "schema-wide" kennzeichnen. Sie gehört dann automatisch an jedes Element dieses Schemas.

```
<xs:schema ... default-attributes="defaultAttributeGroup">
  <xs:attribute-group name="defaultAttributeGroup">
    <!-- some attribute declarations -->
  </xs:attribute-group>
  <!-- All other declarations -->
</xs:schema>
```

Beispiel 12: Default-Attribute deklarieren

Dabei ist allerdings zu beachten, dass die Default-Attribute nur in dem Schema gültig sind, in dem sie deklariert wurden. Sollte es ein Element geben, an dem diese Attribute nicht verwendet werden sollen, kann man diese über das Attribut `defaultAttributesApply="false"` deaktivieren.

FLEXIBILITÄT MITTELS <OPENCONTENT> BZW. <DEFAULTOPENCONTENT>

Ein Problem der aktuellen XML Schema Spezifikation: Sie bietet kaum Möglichkeiten, flexible Schema zu erstellen. Nutzer eines Schemas haben i.d.R. keine Möglichkeit, auf geänderte Anforderungen ohne die Hilfe der Schema-Autoren zu reagieren. Und ehe ein weit verbreitetes Schema geändert wird... Über die "openContent"-Elemente kann man mögliche Änderungen in den Anwendungsfällen von vorn herein einplanen und dann die XML-Dokumente sofort anpassen und das Schema später aktualisieren.

Im Normalfall wird man dazu den Inhalt spezieller Elemente als `<openContent mode="interleave|suffix|none">` deklarieren. Zum Beispiel ein Buch. Bisher bekannte Informationen wie Titel, Autor, Datum, ISBN und der Herausgeber werden weiterhin in der Sequenz deklariert. Zusätzlich wird jetzt noch am Anfang der complexType-Deklaration das Buch als offen für neue Elemente gekennzeichnet. Das könnten Informationen wie die Anzahl der Seite, Art der Bindung o.ä. sein. Interleave bedeutet in diesem Zusammenhang, dass neue Elemente irgendwo zwischen den angegebenen Elementen benutzt werden dürfen. Das gilt allerdings nur für die Kind-Elemente von Buch. Für weiter entfernte Nachfahren muss diese Offenheit an deren Elternelement erneut angegeben werden.

```
<xs:element name="Buch" maxOccurs="unbounded">
  <xs:complexType>
    <xs:openContent mode="interleave">
      <xs:any processContents="lax" />
    </xs:openContent>
    <xs:sequence>
      <xs:element name="Titel" type="xs:string"/>
      <xs:element name="Autor" type="xs:string" />
      <!-- ... -->
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Beispiel 13: Element Buch als Open Content

Innerhalb von `<openContent>` wird angegeben, welche Elemente an dieser Stelle zulässig sind. Einschränkungen auf einen bestimmten Namensraum wären hier denkbar.

DAS GESAMTE SCHEMA ALS "OFFEN" DEKLARIEREN

Ganz am Anfang des Schemas kann man aber auch das Element `<defaultOpenContent>` als globale Komponente angeben.

```
<xs:defaultOpenContent mode="interleave" appliesToEmpty="false">
  <xs:any />
</xs:defaultOpenContent>
```

Beispiel 14: Das gesamte Schema als "offen" deklarieren

Dadurch wird das gesamte Schema als **offen** für beliebige andere Elemente gekennzeichnet. Über das Attribut `appliesToEmpty` lässt sich bestimmen, ob zusätzliche Elemente auch in leeren Elementen zulässig sind oder nicht.

CUSTOMIZE DATA — OVERRIDE AND ERROR

Ein wichtiges Thema ist auch das Anpassen bestehender Spezifikationen oder fremder XML Schema an die eigenen Anforderungen. Idealerweise natürlich, ohne die fremde Spezifikation modifizieren, um sie jederzeit durch eine aktuellere Version ersetzen zu können. Bisher wurde dazu das Element `<redefine>` aus XML Schema 1.0 verwendet. Dieses gilt ab XSD 1.1 als veraltet und wird durch `<override>` ersetzt. Im eigenen Schema verwendet dient es dazu, global deklarierte Items (Elemente, Attribute, Datentypen) aus einem anderen Schema zu überschreiben oder gar zu ersetzen.

```
<xs:override schemaLocation="office-calendar.xsd">
  <xs:element name="meeting">
    <!-- -->
  <xs:element>
<xs:override>
```

Beispiel 15: Element "meeting" mittels <override> überschreiben

Der Unterschied? `<redefine>` dient der erneuten Deklaration eines globalen Typs aus einem anderen Schema durch Erweiterung oder Einschränkung. `<override>` hingegen erlaubt es, jedes globales Item aus dem anderen Schema zu ändern, egal ob Element, Attribut, simpleType, complexType, Element- oder Attribute-Gruppe. Dabei ist die Änderung nicht durch Vererbung beschränkt. Das Ersetzen ganzer Elemente oder das Zusammenstellen neuer Gruppen ist jetzt möglich.

Der neue Datentyp `error` wird verwendet, wenn man Teile des importierten Schemas von der weiteren Verwendung ausschließen will: Für das angepasste Buch-Schema sind die nur für den internen Gebrauch bestimmten Reviews nicht mehr zulässig:

```
<xs:override schemaLocation="buch.xsd">
  <xs:element name="Review" type="xs:error" />
</xs:override>
```

Beispiel 16: Element "Review" verbieten

Sobald der Parser irgendwo im XML Dokument auf ein `<review>`-Element trifft, wir die Validierung wie bei jedem anderen Strukturfehler abgebrochen.

SONSTIGES

BENUTZERFREUNDLICHE ANORDNUNG VON INFORMATIONEN — <ALL>

Das bereits aus XSD 1.0 bekannte Element `<all>` erfährt eine kleine aber wirkungsvolle Änderung. Die innerhalb von `<all>` deklarierten Elemente durften bisher lediglich optional sein. Eine durch `maxOccurs` festgelegte Häufigkeit > 1 war nicht zulässig. Das erzwang in den meisten Fällen die Verwendung der Kind-Elemente in einer von `<sequence>` festgelegten Reihenfolge und `<all>` wurde zu einer Randerscheinung.
Jetzt darf jedes innerhalb von `<all>` deklarierte Element beliebig oft erscheinen und `<all>` wird `<sequence>` somit ebenbürtig. Dadurch ist es möglich, mit `<all>` dem Nutzer alle Freiheiten zu lassen, in welcher Reihenfolge er die Elemente auch immer angeben möchte.

WILDCARD SCHEMA COMPONENTS — <ANY> UND <ANYATTRIBUTE>

Auch die aus XSD 1.0 bekannten Wildcard-Schema-Komponenten `<any namespace="..." />`, `<anyAttribute namespace="..." />` wurden einer kleinen Verbesserung unterzogen, was ihre Attribute anbelangt. Demnächst kann angegeben werden, welchen Namespace (**notNamespace**) und welche Elemente (**notQName**) man an dieser Stelle besser **nicht** sehen will.

CONDITIONAL INCLUSION — "VERSIONS KONTROLLE"

Sobald es mehrere Versionen einer Spezifikation gibt, und ganz sicher wird XSD 1.1 nicht die letzte sein, stellt sich die Frage: Welche Implementierung unterstützt welche Version? Dazu wird in XSD 1.1 der Version Control Namespace `xmlns:vc="http://www.w3.org/2007/XMLSchema-versioning"` eingeführt und in einem Schema können verschiedene Parser-Implementierungen berücksichtig werden. Alten XSD 1.0 Parser ist dieses Namensraum vollkommen unbekannt und sie ignorieren ihn. Für alle "neuen" Perser gilt: Über Attribute wie `vc:minVersion` und `vc:maxVersion` erkennt der Parser, mit welcher Schema-Version das Dokument validiert werden soll. Auch können vendor (also Parser-Hersteller) spezifische Fassetten oder Datentypen in der Unterscheidung berücksichtig werden. Dazu dienen Attribute wie `vc:typeAvailable` und `vc:typeUnavailable`. Es können also einfach mehrere Item-Deklarationen angeboten werden und je nach Parser/Validator sucht der sich die für ihn passendste heraus.

Neu sind auch verschiedene Datentypen wie `precisionDecimal` (Exponentialdarstellung erlaubt), `dateTimeStamp`, `anyAtomicType`, `yearMonthDuration` und `dayTimeDuration` und dazu Fassetten wie `minScale` und `maxScale` für die arithmetische Genauigkeit oder `<explicitTimezone>` um die Angabe der Zeitzone zu erzwingen. Dazu kommen noch Änderungen, die Umsetzung von Ableitungen und Ersetzungsgruppen vereinfachen. Last but not least wurden die beiden Teile Part 1 (Structures) und Part 2 (Datatypes) textuell überarbeitet, dass deren Inhalte schneller und vor allem leichter verständlicher werden.

FAZIT

Auch wenn es der minimale Versionssprung nicht vermuten lässt, der Mehrwert der Änderungen ist nicht zu unterschätzen. Bedingte Typisierung und vor allem die Assertions verbessern die Möglichkeiten von XML Schema deutlich. Im Idealfall können sie sogar eine nachgeschaltete Validierung durch Schematron erübrigen. Der XSLT-Prozessor Saxon und der quasi Standard-Parser Xerces haben diese Änderungen bereits zum Großteil implementiert, so dass heute schon mit deren Einsatz begonnen werden kann.

Wer sich weiter und ausführlicher informieren will, dem seien die Foliensätze von Roger Castello ans Herz gelegt. In Zusammenarbeit mit Michael Kay (SAXON-XSLT-Prozessor) ist ihm wieder gelungen, eine anschauliche und umfangreiche Sammlung an Informationen und Beispielen zusammenzutragen. Der größte Teil der hier wiedergegebenen Beispiele und Informationen entstammen diesen Foliensätzen.

REFERENZEN

Roger Castello, XFront.com

- PPT: Überblick für Manager
  (http://www.xfront.com/xml-schema-1-1/xml-schema-1-1-for-managers.ppt?v=10)
- PPT: Vollständige Version
  (http://www.xfront.com/xml-schema-1-1/xml-schema-1-1.ppt?v=10)
- ZIP mit den beiden Foliensätzen und einer Bespielsammlung
  (http://www.xfront.com/xml-schema-1-1/xml-schema-1-1.zip?v=10)

IBM

- XML Schema 1.1, Part 1: An introduction to XML Schema 1.1
  (http://www.ibm.com/developerworks/xml/library/x-xml11pt1/)
- XML Schema 1.1, Part 2: An introduction to XML Schema 1.1: Co-occurence constraints using XPath 2.0
  (http://www.ibm.com/developerworks/xml/library/x-xml11pt2/)
- XML Schema 1.1, Part 3: An introduction to XML Schema 1.1: Evolve your schema with powerful wildcard support
  (http://www.ibm.com/developerworks/xml/library/x-xml11pt3/)

W3C

- W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures
  (http://www.w3.org/TR/xmlschema11-1/)
- W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes
  (http://www.w3.org/TR/xmlschema11-2/)
- Guide to Versioning XML Languages using new XML Schema 1.1 features
  (http://www.w3.org/TR/xmlschema-guide2versioning/)
- Schema für Part 1: Structures
  (http://www.w3.org/TR/xmlschema11-1/XMLSchema.xsd)
- Schema für Part 2: Datatypes
  (http://www.w3.org/TR/xmlschema11-2/XMLSchema.xsd)

Parser

- SAXON - XSLT, XQuery, and XML Schema processor
  (http://www.saxonica.com/welcome/welcome.xml)
- The Apache Xerces Project
  (http://xerces.apache.org/)

Other

- XSD 1.1 is a Candidate Recommendation
  (http://cmsmcq.com/mib/?p=494)