XML Vorlesung FH Aargau, SS 2006

XML Schemasprachen

Erik Wilde

5.6.2006

http://dret.net/lectures/xml-fhnw-ss06/

Übersicht

- Nachteile von XML Schema
- Schemasprachen als Konzept
  - Grammatiken und Anderes
- zwei Beispiele für andere Schemasprachen
  - regelbasierte Validierung (Schematron)
  - Zeichen-orientierte Validierung (CRVX)
- Schemasprachen selbstgemacht

Wozu überhaupt Schemasprachen?

- Einschränkung der akzeptierten Dokumente
  - definiertes Standardvokabular für Einschränkungen
  - Verwendung von Standardsoftware zur Validierung
- Schemasprachen können nie alles
  - DTDs unterstützen kaum Datentypen
    - dass ein Attributwert eine positive Zahl sein muss
  - XML Schema unterstützt keine Co-Constraints
    - dass ein Attributwert kleiner sein muss als ein anderer
  - Schematron kann nicht mit externen Daten arbeiten
    - dass ein Attributwert in einer Datenbank existieren muss

Schemasprachen sparen Geld

- Schemas schränken XML deklarativ ein
  - werden von Software interpretiert und angewendet
  - häufig verwendete Software-Komponenten
    - tendenziell weit verbreitet und ausgereift
- keine Eigenentwicklung von Software nötig
  - falls Software für die Schemasprache existiert
  - reduziert Fehlermöglichkeiten
  - einfacher Update (neuen Validator installieren)
- besserer "Schutz" der Applikation
  - weniger Fehlerbehandlung in der Applikation

XML DTDs

- die XML Spezifikation definiert zwei Syntaxen
  - die Syntax von XML Dokumenten
  - die Syntax von XML DTDs
    - und wie sie den Aufbau von Dokumenten beeinflussen
- DTDs definieren Grammatiken
  - Aufbauregeln für XML Dokumente
  - definieren unendlich grosse Mengen an Dokumenten
    - mit wenigen (trivialen) Ausnahmen
- DTDs müssen von Parsern interpretiert werden
  - Entity-Definitionen sind in der DTD

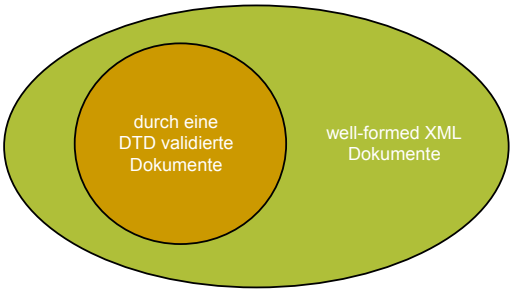
Inhalt einer DTD

- Elementtyp Definitionen
  - Element und sein Content Model
- Attributlisten Definitionen
  - Mengen von Attributen für Elemente
  - der Typ von Attributen (u.U. Aufzählungsliste)
  - diverse Qualifier für Attribute
    - optional, verlangt, Default, fixer Wert
- Entity Definitionen
  - Interpretation erzeugt neuen Text im Dokument

Nachteile der DTDs

- keine Beziehungen zwischen Elementtypen
  - keine Typ-Hierarchie der Elemente
  - zusammenhangsloses Nebeneinander
- keine Unterstützung von Wiederverwendung
  - verbreitetes Parameter Entity Design Pattern
- keine anwendungsorientierten Datentypen
- keine Unterstützung für XML Namespaces
  - "DTDs and Namespaces don't mix"
- keine XML Syntax
  - kann nicht mit XML Tools verarbeitet werden

Schemasprachen als Konzept



XML Schema

- Kompromiss aus vielen Vorschlägen
  - komplett neue Abstraktionsebene
  - umfangreiche Bibliothek an eingebauten Datentypen
- Modellierungshierarchien
  - Typenhierarchien für Simple und Complex Types
    - basierend auf einem konyeptionellen *Ur-Type*
  - unterschiedliche Ableitungsmethoden
    - Simple: Restriction, List und Union
    - Complex: Restriction und Extension
- Ziel war eine möglichst mächtige Sprache
  - nicht unbedingt der beste Ansatz (monolithisch)

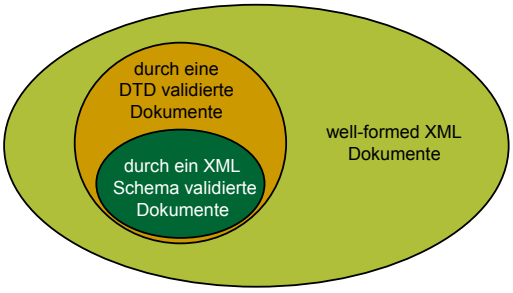
XML Schema Eigenheiten

- keine Unterstützung von Entities
  - falls benötigt, in einer minimal-DTD definiert
- nur Namespace-Compliant XML erlaubt
- Identity Constraints als `ID/IDREF++`
  - basieren auf Elementnamen (keine Typen!)
  - Bruch im Design von XML Schema
- ist definiert auf Infosets
  - Validierung ergänzt das Infoset
  - Zugriff nicht durch einen Standard definiert
    - Abhängigkeit von der spezifischen Implementierung

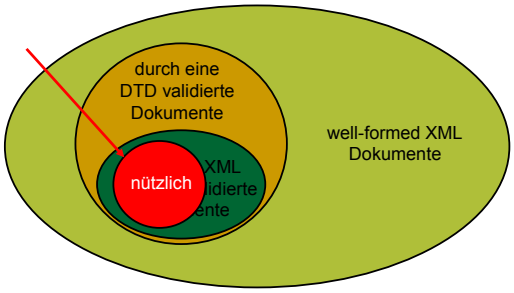
Nachteile von XML Schema

- hohe Komplexität
- momentan nur mangelhafte Implementierungen
  - zum Schreiben von XML Schema selber
    - einzig IBM's SQC implementiert XML Schema komplett
  - zum Validieren von XML Dokumenten
    - keine komplette Implementierung bekannt
- keine Co-Constraints
  - minimal unterstützt durch Identity Contraints
- kein explizites Root Element
- Zeichen in *mixed Content* haben keinen Typ
  - keine Einschränkungen für *mixed Content* möglich

XML Schema als DTD++



Schema-valid vs. Application-valid



Verbindung Instanz/Schema

- verschiedene Mechanismen sind denkbar
  - irgendwie verankert in der XML Syntax
- DTDs benutzen spezielle Konstrukte
  - Document Type Declaration: `<!DOCTYPE ...`
- XML Schema benutzt Namespaces
  - es gibt einen Instance Namespace
  - und dieser definiert ein entsprechendes Attribut
- RELAX NG definiert keine Verbindung
- XML ermöglicht weitere Varianten
  - Kommentare, Processing Instructions, Namespaces

RELAX NG

- alternative Grammatik-basierte Schemasprache
  - RELAX, entwickelt von Murata Makoto
  - TREX, entwickelt von James Clark
- basierend auf bekanntem Formalismus
  - DTDs benutzen einen ad hoc Formalismus
  - XML Schema benutzt einen ad hoc Formalismus
  - RELAX NG benutzt *Hedge Automata*
- keine Typen (⇒ keine Typhierarchien)
- keine eigene Typbibliothek
  - verwendet häufig die XML Schema Simple Types

RELAX NG Prinzipien

- das Dokument bleibt unverändert
  - keine Default-Werte im Schema
  - keine Entities
- nur Beziehungen Instanz-Schema
  - DTDs und XML Schema: Instanz-Instanz
    - `ID/IDREF(S)` bzw. Identity Constraints
  - XML Schema: Schema-Schema
    - Typhierarchie
- Gleichbehandlung von Attributen und Elementen
  - Content Model enthalten beide
  - ermöglicht viele nützliche Anwendungen

DTD Beispiel

```
<!ELEMENT document (heading, chapter)>

<!ELEMENT heading (#PCDATA)>

<!ELEMENT chapter (heading, para+)>

<!ELEMENT para (#PCDATA)>
```


XML Schema Beispiel

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="document">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="heading"/>
    <xs:element ref="chapter"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
 <xs:element name="heading" type="xs:string"/>
 <xs:element name="chapter">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="heading"/>
    <xs:element name="para" type="xs:string" maxOccurs="unbounded"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

RELAX NG Beispiel

```
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
 <start><ref name="document"/></start>
 <define name="document">
  <element name="document">
   <ref name="heading"/>
   <ref name="chapter"/>
  </element>
 </define>
 <define name="heading">
  <element name="heading"><text/></element>
 </define>
 <define name="chapter">
  <element name="chapter">
   <ref name="heading"/>
   <oneOrMore>
    <element name="para"><text/></element>
   </oneOrMore>
  </element>
 </define>
</grammar>
```


RELAX NG Beispiel (Compact Syntax)

```
start = document

document = element document { heading, chapter }

heading = element heading { text }

chapter = element chapter {
    heading,
    element para { text }+
  }
```


XML Schema Beispiel (XSCS)

```
element document {
   ( heading, chapter )
}

element heading { xs:string }

element chapter {
   ( heading, paragraph { xs:string }+ )
}
```


RELAX NG Eigenheiten

- keine eigenbauten Simple Types
  - oft werden die XML Schema Simple Types verwendet
- keine eigenständige Typenebene
  - Patterns definieren die Grammatik
- nicht-deterministische "Content Models"
  - umstritten und problematisch in den Interpretation
- Gleichbehandlung von Attributen und Elementen
  - Attribute werden Teil des "Content Model"
- keine Einschränkung der `xs:all` Gruppe
  - freie Kombination mit anderen Gruppen

Grenzen von Schemasprachen

- was kennen Schemasprachen?
  - das Schema
  - die Instanz
  - u.U. in die Schemasprache eingebaute Dinge
    - Typbibliotheken, Funktionen, ...
- was kennen sie nicht?
  - applikationsspezifische Anforderungen
  - Daten ausserhalb von Schema oder Instanz
- Schemasprachen können sehr einfach sein
  - `(/order/@shipdate > today() + 2) & (/order/@cid = dblookup(customer))`

Applikationsanforderungen

- Schemasprachen sind nie komplett
  - Code ist immer notwendig
  - das Ziel ist so wenig Code wie möglich
- zwei prinzipielle Wege sind möglich
  - Schemasprachen kombinieren
    - modularer Ansatz
    - Validierung als Kombination verschiedener Tools
  - Schemasprachen erweitern oder definieren
    - sollte auch wieder auf XML Tools basieren
  - beide Wege können kombiniert werden
- gute Planung spart Implementierungsaufwand

Klassifizierung von Schemasprachen

- Grammatik-basierte Sprachen
  - sehr weit verbreitet in der XML-Welt (DTD)
  - sinnvolles Konzept für XML
  - DTD, XML Schema, RELAX NG
- regelbasierte Sprachen
  - definieren Regeln, die eingehalten werden müssen
  - eher isolierte Bedingungen
  - oftmals gute Ergänzung zu Grammatiken
- andere Sprachen
  - momentan noch stark in Entwicklung

Schematron

- benutzt *Tree Patterns*
  - Identifikation von Teilen (mittels XPath)
  - Tests basierend auf dem selektierten Context
- sehr einfache Abbildung auf XSLT
  - beliebige XPath Ausdrücke als Assertions
  - keine spezielle Software notwending
- zwei Arten von Constraints
  - Assertions müssen gelten (`true` ergeben)
    - anderfalls wird eine Aktion ausgelöst
  - Reports informieren über Tatsachen
    - werden ausgegeben, falls sie `false` ergeben

Schematron Beispiel

```
<house>
 <wall/><wall/><wall/><wall/>
 <window/><window/>
 <roof/>
</house>
```

```
<rule context="house">
 <assert test="count(wall) = 4">A house should have
  four walls</assert>
 <report test="not(roof)">This house does not have a
  roof</report>
 <report test="window">This house has
  windows</report>
</rule>
```


Schematron Eigenschaften

- Rules definieren den Context von Assertions
  - Abbildung auf XSLT Templates
  - enthalten Assertions und Reports
  - Abstract Rules ermöglichen Modularität
    - Gruppierung einer Menge von Tests
    - Referenzierung aus anderen Rules
- Patterns fassen Rules zusammen
- Phases referenzieren Patterns
  - Anwendung des Schemas zu verschiedenen Zeiten
  - Parametrisierung der Validierung

Schematron Assertions

- Bedingungen, die gelten müssen
  - Assertions müssen gelten (sonst Fehlermeldung)
  - Reports dürfen nicht gelten (sonst Fehlermeldung)
- Assertions enthalten eine Meldung
  - `<name/>` gibt den Kontext aus
  - `<emph>` für Hervorhebungen ist erlaubt

```
<assert test="count(topicref[@TID = ../../@TID])">
  description must mention the described topic</assert>
```

- offene Frage: Kontext der Auswertung?

Schematron Rules

- Kontext für Assertions
  - definiert mit einem XSLT Pattern
    - Untermenge von XPath
    - Elemente, Attribute, Prädikate, Pfade, Unions
  - Knoten, für die Assertions ausgewertet werden

```
<rule context="desc">
    <assert ... </assert>
</rule>
```

- offene Frage: Gruppierung von Rules?

Schematron Patterns

- Patterns sind Gruppen von Rules
  - inhaltliche Zusammenfassung von Rules

```
<pattern name="description checking">
   <rule ... </rule>
</pattern>
```

- Patterns sind Kinder des Schemas selber
  - identifiziert über den Schematron Namespace Name

```
<schema xmlns="http://www.ascc.net/xml/schematron">
    <pattern ... </pattern>
</schema>
```


Schematron Phases

- Gruppierung von Patterns in Phasen
  - Ebenfalls Kinder des `schema` Elements
  - Referenzierung der Patterns durch den Namen
- Verwendung für verschiedene Passes
  - einfacher Check als erste Phase
  - kompletter Check als zweite Phase
- Phasen können ausgewählt werden
  - per Default sind alle Phasen aktiv
  - beim Aufruf können Phasen ausgewählt werden

Schematron Design Guidelines

- Zugriff auf externe Dokumente ist möglich
  - mit Hilfe der `document()` Funktion aus XSLT
- Abwägung zwischen Rules und Assertions
  - komplizierter Kontext und einfache Assertions
  - einfacher Kontext und komplizierte Assertions
- kompletter Ersatz von DTD oder XML Schema
  - meistens sehr aufwendig
  - Änderungen sind kompliziert und fehleranfällig
- Ergänzung bestehender Grammatiken
  - um Co-Constraints und andere Mechanismen

XML Character Validation

- viele Software hat Zeichensatz Limiten
  - Legacy Software ist häufig nur 8-bit fähig
- XML basiert auf Unicode
  - damit ca. 90'000 erlaubte Zeichen
  - die spezifische Codierung ist nicht festgelegt
    - UTF-8 und UTF-16 müssen unterstützt werden
- XML kennt einige Einschränkungen
  - XML Namen (Element und Attribute)
- an anderen Stellen volle Freiheit
  - `#PCDATA` oder `xs:string` erlaubt alles
  - XML Schema und *mixed Content* problematisch

XML und Unicode

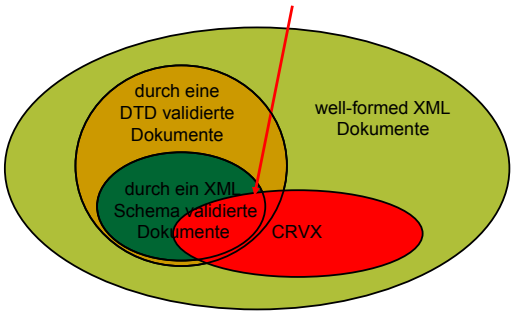
- XML erlaubt "beliebige" Unicode Zeichen
  - in Inhalten und sogar in XML Namen

```
<文書 改訂日付="１９９９年３月１日">
  <題>サンプル</題>
  <段落>これはサンプル文書です。</段落>
  <!-- コメント -->
  <段落>&会社名;</段落>
  <図面 図面実体名="サンプル"/>
</文書>
```


CRVX

- Character Repertoire Validation for XML
  - einfache Schemasprache für Zeichensatz-Validierung
  - `http://dret.net/projects/crvx/`
- verschiedene Einschränkungen möglich
  - verschiedene syntaktische Strukturen
    - z.B. Element-Namen oder Attribut-Werte
  - verschiedene Kontexte eines Dokuments
    - verwendet XPath zur Identifikation von Kontexten
  - verschiedene Sichten auf XML
    - pure XML: keine Namespaces
    - Namespace-compliant XML: Prefixes und NCNames

CRVX als Special Purpose Schema



CRVX (I)

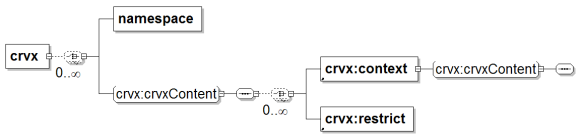
- Character Repertoire Validation for XML
  - einfache Schemasprache für Zeichensatz-Validierung
  - `http://dret.net/projects/crvx/`
- eine Variante für DSDL
- Character Handling ist ein einfaches Problem
  - aber sehr häufig und wichtig zu beachten

CRVX (II)

- verschiedene Einschränkungen möglich
  - verschiedene syntaktische Strukturen
    - z.B. Element-Namen oder Attribut-Werte
    - berücksichtigt auch *mixed Content* (XML Schema nicht!)
  - verschiedene Kontexte eines Dokuments
    - verwendet XPath zur Identifikation von Kontexten
  - verschiedene Sichten auf XML
    - pure XML: keine Namespaces
    - Namespace-compliant XML: Prefixes und NCNames

CRVX Schema

- Schema für eine Schemasprache
  - definiert mit XML Schema
  - einfachster Weg für eine Definition
  - kann mit XML Tools verarbeitet werden



Schemasprachen selbstgemacht

- wechselnde Anforderungen einer Applikation
  - z.B. verschiedene Anwendungskontexte
  - z.B. Anbindung an unterschiedliche Infrastruktur
- Schemasprachen können einfach sein
  - einfaches Schema (für die Schemasprache)
  - Implementierung mit XML Tools
  - Abbildung auf bestehende Software-Komponenten
- Aufwand vs. Ertrag
  - einfache Parametrisierung einer XML-Anwendung
  - bessere Wiederverwendbarkeit
  - Selbstschutz der Entwickler (Encapsulation)

Validation Pipelines

- Spezialfall allgemeiner Processing Pipelines
  - u.U. stärker eingeschränkter Satz an Tools
    - Parser, XML Schema Processor, Schematron, CRVX, ...
- Business Rule Validation ist kompliziert
  - u.U. Anbindung an externe Datenbestände
  - oder allgemein Anbindung an externe Applikation
    - Pipeline Komponente mit einem Validation API
    - kann an beliebigen Code gebunden werden
    - aber nur ein Validation API benutzen (z.B. read-only)

Sequence Matters

- Validation Pipelines kombinieren Schemas
  - sequentielle Abarbeitung der Pipeline
- die Reihenfolge kann wichtig sein
  - z.B. CRVX und Default-Werte aus einem Schema
- u.U. mehrfache Ausführung von Komponenten
  - Validation als Graph mit Ablaufsteuerung
    - Pipeline-Sprachen werden u.U. komplex
    - siehe *Web Services Flow Language (WSFL)*

Zusammenfassung

- Schemasprachen als Programmierung
  - deklarativ vs. prozedural
  - u.U. eigene Schemasprachen benutzen
- wachsende Zahl an Schemasprachen
  - XML wird weiterhin komplexer
  - Toolset an klar fokussierten Tools
  - modularer Aufbau einer Anwendung
- Validierung als Pipeline
  - Kombination verschiedener Schemasprachen