XML Schema

Tobias Maurer <tomaurer@studcs.uni-sb.de>

Seminar

Logische Aspekte von XML
nach einem Thema von
Prof. Gert Smolka
Naturwissenschaftlich-Technische Fakultät I
Fachrichtung 6.2 – Informatik
Universität des Saarlandes, Saarbrücken, 2003

Inhaltsverzeichnis

Kapitel 1

Einleitung

XML Schema ist eine Technologie zur Gültigkeitsprüfung von XML Dokumenten. Ein XML-Dokument (externe Darstellung) wird zusammen mit seinem XML Schema selbstbeschreibend (interne Darstellung). Man kann also aus einer externen Darstellung die dazu passende interne Darstellung herleiten.

1.1 Ein Beispiel aus der Praxis

Die Firma X besitzt Daten, die sie auf verschiedene Weise anbieten möchte, zum Beispiel online übers Internet, offline auf einer CD oder in gedruckter Form. Die Daten werden intern in einer XML-Datenbank gespeichert. Für jeden externen Dokumenttyp gibt es ein XML Schema, dass die Daten validiert, also dafür sorgt, dass nur gültige Dokumente gschrieben und gelesen werden.

1.2 Zum Umgang mit dieser Arbeit

Man sollte sich bei der Beschäftigung mit diesem Dokument immer vor Augen halten, dass XML Schema nicht das Ergebnis theoretischer Forschung ist, sondern lediglich eine Möglichkeit bietet, XML Dokumente zu beschreiben. Nachfolgende Beispiele zeigen eindeutige Schwachpunkte von XML Schema (und XML), allerdings werden diese in der Praxis, und XML ist nun mal ein praktisches Arbeitsmittel, keine oder nur eine zu vernachlässigende Rolle spielen.

XML Schema selbst ist ein großer und komplexer Standard, der mehr als 300 gedruckte Seiten umfasst.

1.3 Notation

Die in dieser Ausarbeitung verwendete Schreibweise entspricht größtenteils den Quellen, ist aber in einigen Punkten vereinfacht. Sie ist weitgehend selbsterklärend. An Stellen, wo sie von der intuitiven Bedeutung abweicht, wird dies gesondert hervorgehoben.

1.3.1 Beispiel zur Notation

- Vereinfachte XML Syntax

```
<xs:simpleType name=”feet”>
<xs:restriction base=”xs:integer”/>
</xs:simpleType>
<xs:element name=”height”
type=”feet”/>
```

- Unsere Schreibweise

define type $feet$ **restricts** $integer$
define element $height$ **of type** $feet$

1.4 Anforderungen an XML Schema

Neben der bereits erwähnten Selbstbeschreibung muss sichergestellt sein, dass beim Konvertieren aus der internen Darstellung in die externe und wieder zurück die neue interne Darstellung mit der alten identisch ist. Man spricht dabei vom *Roundtripping*, worauf später ausführlich eingegangen wird.

XML selbst besitzt, da zwischen *Integern* und *Strings* nicht unterschieden wird, keine der beiden Eigenschaften.

- Beispiel: Überführung in LISP Syntax

```
<foo>1 2 3</foo>
```
wird zu
(foo “1 2 3”) oder (foo 1 2 3)

```
<bar>1 two 3</bar>
```
wird zu
(bar 1 “two” 3) oder (bar 1 “two” “3”)

Kapitel 2

Benannte Typen und Strukturtypen

Beispiel:

type $feet = integer$
type $miles = integer$

In einer Sprache mit Namentypisierung werden dadurch zwei neue Typen definiert und man kann keinen Parameter vom Typ *feet* übergeben, wenn ein Parameter vom Typ *miles* erwartet wird. In einer Sprache mit strukturellen Typen ist dies jedoch ohne weiteres möglich, da *miles* und *feet* die gleiche Struktur haben. Es sind beides *Integer*.

1985 fand ein Test der Strategic Defense Initiative statt, bei dem ein am Boden befindlicher Laser auf einen Spiegel an der Unterseite eines Spaceshuttels geschossen und von dort reflektiert werden sollte . Ein Astronaut gab dazu die Höhe des Lasers in den Bordcomputer des Spaceshuttles ein, welches sich daraufhin auf den Laser ausrichten sollte. Doch das Shuttle drehte sich um. Das Problem war, dass der Astronaut eine Laserhähe von 10023 Fuß über NN, was unter dem Shuttle liegt eingab. Die Software interpretierte dies jedoch als 100023 Meilen über NN, was eindeutig über dem Shuttle liegt. Um sich optimal auszurichten drehte sich das Shuttle also um.

Ein auf Namentypen basierendes Typsystem verhindert solche Missverständnisse.

Die Kluft zwischen benannten Typen und Strukturtypen ist jedoch nicht so groß, wie es zunächst scheint. Viele Sprachen machen von beiden Arten Gebrauch. In SML, zum Beispiel, sind Typen rein strukturell, aber mit *"datatype"* deklariert sind sie auch dann unterschiedlich, wenn sie dieselbe Struktur besitzen.

2.1 Werte

Wie oben bereits erwähnt verzichten wir auf die Angabe einer Abbildung *XML-Notation* → *unsere Notation*,da sie sich selbst erschließt. Weiter verzichten wir auf die Angabe der 19 primitiven Datentypen von XML und beschränken uns auf *String* und *Integer* als atomare Werte. Die Notation *xs:integer* wurde zu *Integer* bzw. *integer* vereinfacht, analog für *xs:string*.

$$
\begin{array}{lll}
Value & ::= & () \\
 & | & Item(,Item)* \\
Item & ::= & Element \\
 & | & Atom \\
Element & ::= & \textbf{element}\ ElementName\ OfType?\ \{Value\} \\
OfType & ::= & \textbf{of type}\ TypeName \\
Atom & ::= & String|Integer
\end{array}
$$

Ein Wert kann aus beliebig vielen *Items* bestehen. Ein *Item* ist entweder ein *Element* oder ein *Atom*. Ein Element wird durch einen Elementnamen, einen optionalen Typen und einen Wert dargestellt. *String* und *Integer* sind atomar.

Mit der obigen Definition lässt sich das folgende Element herleiten:

element $paper$ **of type** $paperType\{$
    element $title$ **of type** $string$ {“Data on the Web”},
    element $author$ **of type** $string$ {“Serge Abiteboul”}
}

Ein ungetypter Wert ist entweder ein Element ohne Typannotation oder ein *String*. Mit ungetypten Werten werden XML Dokumente vor der Validierung beschrieben. Jeder ungetypte Wert ist ein Wert. Die Typregel lässt sich für ungetypte Werte vereinfachen:

$$
\begin{array}{lll}
UntypedValue & ::= & () \\
 & | & UntypedItem(,UntypedItem)* \\
UntypedItem & ::= & \textbf{element}\ ElementName\ \{UntypedValue\} \\
 & | & String
\end{array}
$$

Unser Beispiel wird ohne Typannotation zu:

element $paper$ **of type** $paperType$ {
    element {"Data on the Web"},
    element $author$ {"Serge Abiteboul"}
}

Eine letzte Möglichkeit, Werte zu generieren, bieten die *Simple Values*. Sie bestehen aus beliebig vielen atomaren Werten. Jeder *Simple Value* ist wieder ein *Value*:

$$\begin{array}{lll} SimpleValue & ::= & () \\ & | & Atom(,Atom)* \end{array}$$

"Data on the Web" und 10023 sind *SimpleValues*.

2.2 Typen

$$\begin{array}{lll} Type & ::= & () \\ & | & ItemType \\ & | & Type, Type \\ & | & Type|Type \\ & | & Type? \\ & | & Type+ \\ & | & Type* \end{array}$$

Ein Typ ist eine leere Sequenz, ein *ItemType*, eine Reihe von Typen, eine Alternative von zwei Typen oder Mehrfachforkommen. "?" bedeutet dabei optional, "+" steht für ein oder mehrfach und "*" für beliebig oft.

$$\begin{array}{lll} ItemType & ::= & (integer|string) \\ & | & ElementType \\ ElementType & ::= & \textbf{element}\ ElementName?\ OfType? \end{array}$$

ItemTypes sind entweder atomare Typen oder bereits definierte Elementtypen. Ein Elementname ohne Typ verweist auf eine globale (*element author*), ein

Elementname mit Typ auf eine lokale Deklaration. Steht der Typname alleine, passt er zu jedem Elment, dessen Typ von ihm abgeleitet ist (*element author of type publication type*). Das Wort *element* alleine passt zu jedem Element.

Definition	::=	**define element** $ElementName\ OfType$
	|	**define type** $TypeName\ TypeDerivation$

Elemente und Typen werden durch einen Namen und einen Typen definiert.

Die folgenden einfachen Typen lassen sich durch Anwenden der oben definierten Regeln herleiten:

define element $title$ **of type** $string$
define element $author$ **of type** $string$
define element $year$ **of type** $string$
define element $abstract$ **of type** $string$

Daraus lässt sich der folgende komplexe Typ entwickeln (Elementdeklarationen können dabei global oder innerhalb einer Typdeklaration vorkommen):

define element $paper$ **of type** $paperType$
define type $paperType$ {
    element $title$,
    element $author*$,
    element $abstract?$,
    element $year$
}

Hier sind *title, author, abstract, year* lokal und *paper* global definiert. Auch eine äquivalente abstrakte Typdeklaration ist möglich:

define type $publicationType$ {
    element $title?$,
    element $author*$,
    element $abstract?$,
    element $year$
}

Eine Typableitung kann einen gegebenen Typen entweder einschränken oder erweitern (ähnlich der Klassenvererbung bei Java):

$TypeDerivation$	::=	restricts $AtomicTypeName$
	|	restricts $TypeName$ $\{Type\}$
	|	extends $TypeName$ $\{Type\}$

Der *AtomicTypeName* kann nicht erweitert werden, da er bereits atomar ist. Durch Einschränkung des bereits definierten, komplexen Typs *publicationType* lässt sich ein neuer Typ *bookType* wie folgt deklarieren:

define type $bookType$ **restricts** $publicationType$ $\{$
element $author*$,
element $title$,
element $year$
$\}$

Eine Erweiterung von *publicationType* kann wie folgt aussehen:

define type $ISBNbookType$ **extends** $bookType$ $\{$
element $ISBN$ **of type** $integer$
$\}$

Mit den neuen Typen kann man dann ein neues Element *book* definieren:

define element $book$ **of type** $bookType$

Als weiteres Beispiel soll die Definition des in XML Schema integrierten Typs *anyType* angegeben werden. *anyType* ist der Typ, von dem jeder andere Typ abgeleitet werden kann.

define type $anyType$ **restricts** $anyType$ $\{$
    element $(integer|string|element)*$
$\}$

Kapitel 3

Validierung

Wir haben jetzt eine Grammatik, mit der sich Typen, getypte Werte und ungetypte Werte aufbauen lassen. Auch wissen wir, dass ungetypte Werte die externe Repräsentation und getypte Werte die interne Datenrepräsentation sind. Doch wie lässt sich das überprüfen?

Wir definieren dazu eine Relation (*validate as*), die erfüllt ist, wenn der ungetypte Wert unter dem angegebenen Typ gültig ist und der validierte Wert das Ergebnis ist. Da es zu einem gegebenenen Typen für jeden ungetypten Wert mindestens einen Wert gibt, für den die Relation gilt, ist die Validierung eine partielle Funktion.

Validierung wird im Folgenden mit Hilfe von *matching* und *erasure* beschrieben. *matching* ist erfüllt, wenn ein gegebener Wert zu einem ebenfalls gegebenen Typen passt. *erasure* löscht die Typannotationen. Auf beide Relationen wird später noch detaillierter eingegangen.

3.1 Gültigkeitstheorem

validate as $Type\ \{\ UntypedValue\ \} \Longrightarrow Value$

genau dann, wenn

$Value$ matches $Type$,
$Value$ erases to $UntypedValue$

Das Theorem lässt sich durch Induktion über die Herleitungen beweisen. Sehen wir uns nun die Relationen *matches* und *erases to* genauer an.

3.2 matches

$$Value \text{ matches } Type$$

soll dann gelten, wenn der gegebene Wert zum ebenfalls gegebenen Typen passt.

3.2.1 Beispiel

element $author$ **of type** $string$ {"Robert Harper"},
element $author$ **of type** $string$ {"John Mitchell"},

matches

element $author$ **of type** $string+$

Eine Liste mit zwei *author*-Elementen vom Typ *string* passt auf eine Elmentliste mit mindestens einem *author*-Element vom Typ *string*.

3.2.2 Beispiel

element $configuration$ **of type** $configurationType\{$
    element $shuttle$ **of type** $miles$ $\{120\}$,
    element $laser$ **of type** $miles$ $\{1023\}$
$\}$

matches

element $configuration$ **of type** $configurationType$

Die Definition von *configuration* fordert, dass *laser* vom Typ *feet* ist. Obwohl rein strukturell kein Unterschied zwischen *miles* und *feet* besteht, akzeptiert *matches* an einer Stelle, an der es ein Element vom Typ *feet* erwartet, kein Element vom Typ *miles*. Es wird auf Namenebene unterschieden. Die *matches*-Relation ist daher **nicht** erfüllt.

3.2.3 Beispiel

element $configuration$ {
element $shuttle$ {“120“},
element $laser$ {“10023“},
}

matches

element $configuration$ **of type** $configurationType$

Shuttle und *laser* sind hier ungetypt. Dies hat ebenfalls zur Folge, dass die *matches*-Relation unerfüllt ist.

3.3 erases to

$Value$ **erases to** $UntypedValue$

ist erfüllt, wenn der gegebene Wert zum ungetypten Wert überführt werden kann.

3.3.1 Beispiel

element $fact$ **of type** $intOrStr${“I” , ”saw” , 8 , ”cats”}
erases to
element $fact${“I saw 8 cats”}

erases to überführt die interne Repräsentation in eine externe. Dazu wandelt *erases to* alle atomaren Werte in Strings um und fügt sie zu einem String zusammen, wobei die Teilstrings voneinander durch Leerzeichen getrennt werden.

Kapitel 4

Vereinfachung

Im Folgenden wird ausgeführt, wie man die *matching*-Relation optimieren kann, so dass sie entscheidende Vorteile gegenüber der entsprechenden Funktion anderer XML Typsysteme bietet.

4.1 Eindeutigkeit

Ein Typ *(Type)* heißt eindeutig im Sinne der Validierun, wenn es für jeden ungetypten Wert *(UntypedValue)* höchstens einen Wert *(Value)* gibt, so dass gilt:

$$\textbf{validate as } Type\{UntypedValue\} \Rightarrow Value$$

4.2 Roundtripping

Wenn wir einen internen Wert in einen externen umwandeln (durch *erases to*) und wieder zurück in einen internen (durch Validierung gegen den Typen) und der resultierende, interne Wert der gleiche ist wie der Ausgangswert, reden wir von *Roundtripping*. Voraussetzung dafür ist, dass der Typ eindeutig ist.

Bei mehrdeutigen Typen, wie *intOrStr* (siehe obiges Beispiel) ist *Roundtripping* daher nicht möglich.

4.2.1 Korrolar

$$\frac{\begin{array}{c}Value \textbf{ matches } Type \\ Value \textbf{ erases to } UntypedValue \\ \textbf{validate as } Type\{UntypedValue\} \Rightarrow Value' \\ Type \text{ ist eindeutig (i.S.d. Validierung)}\end{array}}{Value = Value'}$$

4.2.2 Beweis

Durch das Validierungstheorem folgt, dass die ersten beiden Annahmen äquivalent zu "**validate as** $Type\{UntypedValue\} \Rightarrow Value$" sind. Kombiniert man das mit der dritten Annahme und der Tatsache, dass *validate* bei einem eindeutigen Typ eine partielle Funktion ist, folgt daraus direkt die Konklusion.

Da XML Schema keine mehrdeutigen, komplexen Typen erlaubt, beinhalten die einzigen Gegenbeispiele zu *Roundtripping* Listen oder Vereinigungen einfacher Typen.

Die Autoren des Papers *"The Essence of XML"* kommentieren das so: *"Users will stub their toes on this rarely, but when they do it will hurt!"*.

Umgekehrtes Roundtripping funktioniert analog zu *Roundtripping*, jedoch wird dabei von einem externen Wert ausgegangen der mit dem resultierenden externen Wert verglichen wird. Die einzigen Fälle, in denen *Umgekehrtem Roundtripping* nicht möglich ist, beinhalten führende Nullen oder Fälle in denen Grundtypen mehrere Repräsentationen haben. Im Gegensatz zu *Roundtripping* ist *Umgekehrtes Roundtripping* daher kein ernsthaftes Problem.

4.3 Sinnvolle Werte

In anderen Zusammenhängen werden *sinnvolle Werte* wohlgetypt genannt. Ein Wert, der kein typannotiertes Element enthält ist sinnvoll. Enthält ein Wert ein Element mit Typannotation, muss der Wert auf den Typ in der Annotation *matchen*. Im Folgenden wird gezeigt, dass durch Validierung erhaltene Werte immer sinnvoll sind. Diese Eigenschaf kann man sich später zu Nutzen machen um die *matching*-Relation zu optimieren.

4.3.1 Subtype

Ähnlich zu *Xduce* definieren wir auch hier eine *Subtype*-Beziehung, zwischen zwei Typen *Typ*$_1$ und *Typ*$_2$, die gilt, falls *Typ*$_1$ durch Einschränkung (*restricts*

to) von *Typ*$_2$ definiert ist. Subtype ist nicht durch strukturelle Inferenzregeln sondern durch eine logische Äquivalenz definiert:

$$Type_1 \textbf{ subtype } Type_2$$

$$\Leftrightarrow$$

$$(\forall Value : Value \textbf{ matches } Type_1 \Rightarrow Value \textbf{ matches } Type_2)$$

$Type_1$ **subtype** $Type_2$ gilt, falls jeder Wert, der zum ersten Typen passt auch zum Zweiten passt.

Beispiel

$$\textbf{element of type } feet \quad \textbf{\textit{subtype}} \quad \textbf{element of type } integer$$

ist gültig, da *feet* eine Einschränkung von *integer* ist.

Bemerkung

Subtyping kann durch bekannte Algorithmen, die die Sprachen von zwei regulären Ausdrücken vergleichen getestet werden, da die pure Namentypisierung es uns ermöglicht, die *matching* Relation durch einfaches Überprüfen des Elementnamens zu implementieren, ohne dass die Struktur dabei betrachtet werden muss. Andere Typsysteme für XML benötigen Subtyping Algorithmen, die auf regulären Baumausdrücken und endlichen Baumautomaten beruhen, was wesentlich teurer ist.

4.4 Definition: Sinnvolle Typen

Wenn eine Elment- oder Typdefinition sinnvoll ist, dann gilt:

Definition **ok**

Eine Elementdefinition ist immer sinnvoll:

Define element ElementName OfType **ok**

Eine Einschränkung eines atomaren Typen ist ebenfalls immer sinnvoll:

Define type TypeName restricts AtomicTypeName **ok**

Eine Einschränkung auf einen gegebenen Typ ist dann sinnvoll, wenn der Typ ein Untertyp des Grundtyps ist:

$$\frac{\begin{array}{c}BaseTypeName \textbf{ resolves to } BaseType \\ Type \textbf{ subtype } BaseType\end{array}}{\textbf{Define type } TypeName \textbf{ restricts } BaseTypeName\{Type\} \quad \textbf{ok}}$$

Eine Typerweiterung ist immer sinnvoll:

$$\frac{}{\textbf{Define Type}TypeName \textbf{ extends } BaseTypeName\{Type\} \quad \textbf{ok}}$$

4.5 Definition: Sinnvolle Werte

Ist ein Wert sinnvoll, dann gilt:

$$Value \textbf{ ok}$$

Die leere Sequenz ist stets sinnvoll:

$$\frac{}{()\textbf{ok}}$$

Wenn zwei Werte sinnvoll sind, so ist es auch ihre Sequenz:

$$\frac{\begin{array}{c}Value_1 \quad \textbf{ok} \\ Value_2 \quad \textbf{ok}\end{array}}{Value_1, Value_2 \quad \textbf{ok}}$$

Ein Element ist sinnvoll, wenn der Wert sinnvoll ist und er zu dem annotierten Typen matcht.

$$\frac{\begin{array}{c}TypeName \textbf{ resolves to} Type \\ Value \textbf{ ok} \\ Value \textbf{ matches } Type\end{array}}{\textbf{Element } ElementName \textbf{ of type } TypeName\{Value\} \quad \textbf{ok}}$$

Ein atomarer Wert ist sinnvoll.

$$\frac{}{Atom \quad \textbf{ok}}$$

4.6 Optimiertes Matching

Optimiertes *Matching* basiert auf der Annahme, dass jeder durch Validierung erlangte Wert sinnvoll ist.

$$\frac{\textbf{validate as } Type\{UntypedValue\} \Longrightarrow Value}{Value \textbf{ ok}}$$

Beweis durch strukturelle Induktion □.

In der *Matching*-Regel muss die Prämisse "$Value$ **matches** $Type$" daher nicht mehr getestet werden, wenn das Element durch Validierung erzeugt wurde.

Optimiertes Matching ist durch diese Vereinfachung leichter ausführ- und implementierbar. Die Vereinfachung verdanken wir der Namentypisierung, die sicherstellt, dass jedes validierte Element einen Typen besitzt, der seinen inhalt eindeutig beschreibt.

4.7 Abschlussbemerkung

Das hier verwandte, ausschließlich auf Namentypen basierende Typsystem erlaubt es uns zu testen, ob ein Typ und ein Element *matchen*, indem wir lediglich die Elementnamen (aber nicht die Inhalte) betrachten. Die einzige Voraussetzung, die wir benötigen, sind eindeutige Typen.

Das dadurch erreichte Verfahren zur Umsetzung der Matching Relation hat einen entscheidenden Vorteil zu Verfahren anderer Typsysteme, wie *Xduce* oder *YATL*: Es ist im Gegensatz zu anderen Systemen, die dazu reguläre Baumausdrücke und Baumautomaten benutzen, auf endliche Automaten oder durch reguläre Ausdrücke beschreibbar.

Literaturverzeichnis

[ABI00] ABITEBOUL, Serge; BUNEMAN, Peter; SUCIU, Dan: *Data on the Web* . 2000. – Morgan Kaufmann Publishers San Francisco, California

[POPL03] WADLER, Philip; SIMEON, Jerome: *The Essence of XML*. 2003. – http://www.research.avayalabs.com/user/wadler/topics/xml.html Avaya Labs