

Derivation, tolerance and validity

a formal model of type definition in XML Schemas

Allen Brown Jr.

Microsoft, Redmond, USA

allenbr@microsoft.com

<http://msdn.microsoft.com/xml/>

Abstract:

We investigate the general mechanism of type derivations in XML schemas¹, of which the restrictions and extensions of the proposed schema standard (in and) are but special cases. The principal idea here is that the designer of a modified or derived type be able to declare the nature of the derivations made. Moreover, we would like to be able to substitute the derived type in places where the ancestral type was expected. But what if an application cannot tolerate such a substitution? In order to exploit fully such derived types, an application, in turn, needs to be able to declare what sorts of modifications it is willing to tolerate. Furthermore, the validator, upon hearing such advice from the application, needs to validate an instance according to the tolerances declared. In effect, we propose to make applications responsible for telling the validator what classes of type transformations lead to substitutable derivations from the application's perspective. The validator, then, is responsible for guaranteeing that the effective derivations fall within the application's tolerances.

Introduction

The principal idea here is that the designer of a modified or derived type be able to declare the nature of the modification made. In order to exploit fully such a declaration, an application needs to be able to declare what sorts of modifications it is willing to tolerate. Moreover, the validator, upon hearing such advice from the application, needs to validate an instance according to the tolerances declared. In effect we are proposing that we make applications responsible for telling the validator what classes of transformations lead to **substitutable** derivations from the **application's** perspective. The validator, then, is responsible for guaranteeing that the effective derivations fall within the application's tolerances.

We open by informally sketching a model of the declaration, the notion of tolerance, and the kinds of derivations with respect to which validations might be successfully conducted. Later, we will give a rigorous account of the concepts in question, from which one can design a concrete specification syntax, and implement a validator.

First, let's embellish the element declaration with a tolerance declaration:

```
<element type="mumble" tolerates="(attr-extension | elt-extension)*"/>
```

and then define a derived type:

```
<complexType name="grumble" base="mumble" derivedBy="attr-extension">  
... </complexType>
```

¹ An earlier version of this paper was submitted as a note to the W3C XML Schema Working Group. Some of the ideas in this paper are reflected by the current XML Schema candidate recommendation in the facilities provided by element equivalence, type derivation operators, and blocking of type derivation operators.

The former declares an element that permits ² the substitution of a type derived by any sequence of element or attribute extensions. The latter derivation defines a subtype by adding attributes to the base type. (See the section below on transformations.)

The example above illustrates how the original type designer might indicate the permissibility of the further elaboration of the type. But an application knowing only about the original type might not be as permissive as the designer. In general, an application should be able to declare to the validator that certain kinds of derivation are tolerated. One possibility is to make a global declaration of such tolerance. Another possibility is to make local declarations of the form "this application tolerates derivations of the **mumble** type by any sequence of transformations of the following kinds..."

Now there are two questions:

What kinds of derivation are there?

How might the validation of such derivations work?

In this informal account, we only consider derivations **vis à vis** content models, and "element only" content models at that. (It should be reasonably clear how to extend the same ideas to cover derivations with respect to attributes.) For the time being, types will be conflated with regular expressions, where a derived type is accessed through a new tag. ³ Finally, we assume that all content models are representable by regular expressions (in).

Validation: an informal account

First, let's consider validation without derivation. Validity is defined recursively roughly as follows: A node of an XML tree tagged with A , which has the type definition $A \rightarrow R$ (R being a regular expression in tags) is valid just in case the children of the node tagged with A (when taken in sequence) are in the regular set denoted by R , and each of the children is itself valid. An XML tree, by extension, is valid just in case all of its nodes are valid.

Without saying what can count as a derivation (other than to note that every type is a derivation of itself), we extend the definition of validity to accommodate derivation. A node of an XML tree tagged with A , is valid just in case A has some derivation B defined by $B \rightarrow Q$, such that each of the children of the node tagged with B (when taken in sequence) is in the regular set denoted by Q , and each of the children is valid. An XML tree, by extension, is valid just in case all of its nodes are valid.

² The XML Schema candidate recommendation supports a related construct in the block attribute associated with element declarations. While the *tolerates* attribute above is a permission, the block attribute is a prohibition.

³ From the perspective of the XML Schema candidate recommendation, this is tantamount to saying that every type is associated with exactly one element definition, and that the tag associated a newly derived type is in the equivalence class of the tag associated with the base type.

Derivation by transformation

Suppose $A_1 \rightarrow R_1$

..., $A_n \rightarrow R_n$

are type definitions, we will define a set of immediate transformations below yielding a type definition

$B \rightarrow Q$

such that B

is an immediate derivation from A_1 ,

..., A_n

by one such transformation. If T_1 ,

..., T_1

is a set of transformations, there is the obvious inductive definition of a derivation B

from A

by a sequence of immediate transformations taken from that set. So what might be interesting immediate transformations?

Consider the immediate derivation of B

from A_1 ,

..., A_n

by the "prepend" immediate transformation (or inheritance transformation) illustrated as follows:

$A_1 \rightarrow R_1$

$A_2 \rightarrow R_2$

⋮

$A_n \rightarrow R_n$

yields

$B \rightarrow R_1 R_2 \dots R_n Q$

where Q

is a new regular expression in tags. Similarly, one might define the "postpend" immediate transformation by

$B \rightarrow Q R_1 R_2 \dots R_n$

One can define a kind of "interleaving" immediate transformation by beginning with the type definitions

$$A_{1,1} \rightarrow R_{1,1}$$

⋮

$$A_{1,i} \rightarrow R_{1,i}$$

⋮

$$A_{m,1} \rightarrow R_{m,1}$$

⋮

$$A_{m,i_m} \rightarrow R_{m,i_m}$$

one "derives" them to an inherited type definition

$$B \rightarrow Q_0 R_{1,1} \dots R_{1,i_1} Q_1 \dots Q_{m-1} R_{m,1} \dots R_{m,i_m} Q_m$$

where the Q

's are all new regular expressions. The transformations described thus far are essentially additive. But there are lots of others.

Refining \bar{B}

from A

by the "insertion" immediate transformation:

$$A \rightarrow R_1 \dots R_{m-1} R_m \dots R_n$$

becomes

$$B \rightarrow R_1 \dots R_{m-1} Q R_m \dots R_n$$

Refining \bar{B}

from A

by the "deletion" immediate transformation:

$$A \rightarrow R_1 \dots R_{m-1} R_m R_{m+1} \dots R_n$$

becomes

$$A \rightarrow R_1 \dots R_{m-1} R_{m+1} \dots R_n$$

Deriving \bar{B}_i

from A
 by a "restriction on choices" immediate transformation::

$$A \rightarrow PQ'R$$

can be restricted to any of:

$$B_1 \rightarrow PQ^+R$$

$$B_2 \rightarrow PQ?R$$

$$B_3 \rightarrow PQR$$

Refining B

from A
 by the "cyclic permutation" immediate transformation.:

$$A \rightarrow R_1 \dots R_n \dots R_{n+1} \dots R_n$$

becomes

$$B \rightarrow R_1 \dots R_{n+1} \dots R_{n+1}R_n \dots R_n$$

No doubt you have noticed that we have not told you how the positions at which the transformations occur are to be identified. Also, the only transformations we have shown are on the immediate subexpressions of regular expressions. There are endless grammar-based processors that allow the identification of subexpressions and modification thereof, so we'll defer providing a solution.

By composing the transformations described, a wide variety of insertions, deletions and rearrangements can be accommodated. (Indeed, if we allow the transformations above to occur at any level, we get a universal editor.) While each immediate transformation is eminently understandable, their composition can rapidly become opaque, so one may want to declare the number of immediate transformations that are tolerable in refining B from A

. An application's robustness against transformations is more likely, however, to be determined by the nature of the transformations tolerated rather than their number.

Imagine that an application can declare that it will tolerate derivations B from A

by any sequence of transformations in a set of immediate transformations $\{T_1, \dots, T_1\}$

. Thus, in the validation definition above, we amend it to look for not just derivations, but "tolerated" derivations. The validator, of course, should leave a record of what derivations were used at each node during the validation process. Moreover, the application can have access to the transformational relationship between the type associated with a node's actual tag and the type that led to the validation of the node's children.

The formalities

Now we are ready to provide mathematical models of schemas, instances and derivation wherein:

Schemas embed context-free grammars the bodies of whose productions are regular expressions.

The heads of those productions are the names of types.

The types are associated with attribute definitions.

There is a partial order on types that maximizes refinable content models.

Instances are ordered, labeled, attributed trees.

Basic validity is defined in terms of types that are maximal in the partial order.

Validity with derivation is defined relative to lesser types in the partial order.

The partial order on types is segmented in such a way as to mimic various classes of derivation.

Validity is further sharpened to allow for tolerance of instances according to whether they fall into certain classes of derivation or not.

Schemas

A **schema** ⁴ S is a structure $\langle E, T, A, V, \triangleleft, f, g, h, e_0 \rangle$
where

E

is a finite set of **element** names

T

is a finite set of **type** names

A

is a finite set of **attribute** names

V

is a finite set of **values**

$e_0 \in E$

is the distinguished **root** element

$\mathcal{R}(E)$

is the set of regular expressions over E

$f : T \rightarrow \mathcal{R}(E) \cup \mathcal{P}(V)$

is the **content modeling** function ⁵

⁴ The schemas defined in this section and the instances defined in the next correspond more accurately to the XML schema and the XML instance information set contributions detailed in the XML Schema candidate recommendation.

⁵ $\mathcal{R}(E) \cap \mathcal{P}(V) = \emptyset$

.If $t, t' \in T$

$$g : T \rightarrow \mathcal{P}(V)^A$$

is the **attribution** function ⁶

$$h : E \rightarrow T$$

is the **type declaration** function

immediate derivation \triangleleft

, is a partial order where $\triangleleft \subseteq (\mathcal{R}(E)^2 \cup (\mathcal{P}(V)^A)^2 \cup T^2)$

, and whenever $t' \triangleleft t$

$$f(t') \triangleleft f(t)$$

$$g(t') \triangleleft g(t)$$

if $(g(t))(a) \neq \emptyset$

then $(g(t'))(a) \triangleleft (g(t))(a)$

derivation is the transitive closure of immediate derivation

A type $t \in T$

is **basic** if there does not exist a distinct type t'

such that $t \triangleleft t'$

. ⁷ Also, we will sometimes call $f(t)$
the **signature** of t .

and $t' \triangleleft t$

then both $h(t)$

and $h(t')$

are in $\mathcal{R}(E)$

or they are both in $\mathcal{R}(E)$

⁶ For $a \in A$

such that $g(a) \neq \emptyset$

we will say that the type T

has the attribute a

⁷ We will take as simple those types in T

Instances

An **instance** I of a schema S is a structure where

$$\langle N, <, <<, \ell, n_0 \rangle$$

N

is a finite set of **nodes**

n_0

is the distinguished **root** node

$<$

and $<<$

are discrete partial orders⁸ over N

such that if $b < a$

and $c < a$

then b

and c

are children of a

(and siblings of one another), and either $b << c$

(b

is the elder sibling of c

), or $c << b$

(c

that are mapped by ℓ to elements of \mathcal{R} ,
i.e. strings, integers, dates, etc.

⁸ A partial order $<$

is discrete if whenever $n < n'$

there does not exist n''

such that $n < n'' < n'$

for distinct n, n', n'' .

is the elder sibling of b
)

for every node $n \in N, n \prec^* n_0$

for every pair of siblings $n, n' \in N$

there is no $n'' \in N$

such that $n'' \prec^* n$

and $n'' \prec^* n'$

for nodes n', n''

, $n' \prec n''$

only if there exists a node n

such that $n' \prec n$

and $n' \prec n$

a **labeling** is a function $\ell: N \rightarrow (E \times \prod_{A \in \mathcal{A}} V^X \times T(\perp) \times V(\perp))$

where ⁹ $E \subseteq E', T \subseteq T', \mathcal{A} \subseteq \mathcal{A}', V \subseteq V'$

A labeling, being an ordered 4-tuple, has a first component called its **tag** ¹⁰, a second component, called its **attribution**, a third component called its **type**; and a fourth component called its **value**. When the type is \perp , it is said to be **implicit**, and otherwise **explicit**. The only nodes of an instance with a value component of \perp

are the non-minimal nodes of the instance. The partial order \prec

induces a tree on the nodes of I rooted at n_0 .

⁹ Here we accommodate element and attribute "openness". Openness could be disabled by requiring that $E = E'$, etc.

¹⁰ For node n

$$\tau(n)$$

is the tag component of $\ell(n)$.

Basic validity

An instance I is $S_{=}$ -**valid** (read "S-basic valid") with respect to a schema S iff all of its nodes have implicit

types, τ_0

's tag is e_0

and each node $n \in N$

is $S_{=}$ -valid with respect to a schema S . A node $n \in N$

is $S_{=}$ -valid with respect to a schema S iff either $\tau(n) \notin E$

, or $\tau(n) \in E$

and

1. for every a
 - in the domain of n
 - 's attribution p
 - , either $a \notin A$
 - , or $a \in A$
 - with $p(a) \in (g(h(\tau(n))))(a)$
 - ; and
2. either
 1. $f(h(\tau(n))) \in \mathcal{R}(V)$
 - ;
 2. the value component of $\ell(n)$
 - is in $f(h(\tau(n)))$
 - ¹¹ ; and
 3. n
 - has no children
3. or
 1. $f(h(\tau(n))) \in \mathcal{R}(E)$
 - ;

¹¹ This corresponds to an element with non-element content contained in a specified subset of the set of values V

2. the value component of $\ell(n)$ is \perp ;
3. the sequence $\tau(n_1), \tau(n_2), \dots, \tau(n_k)$, where n_1, n_2, \dots, n_k are **all** the children of n taken in $<<$ order whose tags are in E , is in the regular set defined by $f(\lambda(\tau(n)))$; and
4. each of the n_1, n_2, \dots, n_k is S_- -valid with respect to S .

Validity with derivation

The intent of derivation is to control the circumstances under which a tag might occur at a node where another kind of tag is expected.¹² Our task here is to define the circumstances under which such a **substitution** might be valid. Regular expressions offer us an obvious mechanism by which we might achieve

this—namely, **alternation**. If $e, e' \in E$

, $\lambda(e') \triangleleft \lambda(e)$,

, then e'

is a \triangleleft

-**alternative** of e

. Let e_1, e_2, \dots, e_k

be **all** of the \triangleleft

-alternatives of e

, then $e \mid e_1 \mid e_2 \mid \dots \mid e_k$

is the \triangleleft

-**closure** of e

. By extension, for $r \in R$

the \triangleleft

¹² This can occur either through the derivation of one type definition from others, or the equivalencing among tags.

-closure of \bar{r}
 (denoted \bar{r}
) is obtained from r

by replacing each element in the latter with its
 -closure. So any occurrence of the

above would be replaced with $(e | e_1 | e_2 | \dots | e_k)$

.¹³ Finally, if $t, t' \in \bar{I}$

, $t' \triangleleft^* t$

, then t'

is a \triangleleft

-alternative of t

. Let t_1, t_2, \dots, t_m

be **all** of the \triangleleft

-alternatives t

, then the

-effective content model of t
 is

either $(\overline{f(t)} | \overline{f(t_1)} | \dots | \overline{f(t_m)})$

when $f(t) \in \mathcal{R}(E)$

;

or $f(t) \cup f(t_1) \cup \dots \cup f(t_m)$

when $f(t) \in \mathcal{R}(V)$

An instance I is

-valid (read "

-derivation valid") with respect to a schema S iff

¹³ In considering the

-closure of a tag we incorporate the equivalence among elements, sometimes called the variation of tags.

's tag is either e_0
or a Δ

-alternative of e_0
, and each node $n \in N$
is e_0

-valid with respect to a schema S . A node $n \in N$
is e_0

-valid with respect to a schema S iff either $\tau(n) \notin E$
, or $\tau(n) \in E$
and

1. if the type component of n

is $t = \ell(n) \neq \perp$

, then $t \triangleleft^* h(\tau(n))$

and the base type for validation is t

2. if the type component of n

is $\ell(n) = \perp$

, then the base type for validation is $t = h(\tau(n))$

3. for every a

in the domain of n

's attribution p

, either $a \notin A$

, or if $a \in A$

and $(g(t))(a) \neq \emptyset$

then $p(a) \in (g(t))(a)$

; and

4. either

1. $f(\tau(n)) \in \mathcal{R}(V)$
- ;
 2. the value component of $\ell(n)$ is in the \mathcal{S} -effective content model of \mathcal{I} ; and
 3. n has no children
5. or
 1. $f(\tau(n)) \in \mathcal{R}(E)$
 - ;
 2. the value component of $\ell(n)$ is \perp ;
 3. the sequence $\tau(n_1), \tau(n_2), \dots, \tau(n_k)$, where n_1, n_2, \dots, n_k are **all** the children of n taken in \ll order whose tags are in E , is in the regular set defined by the \mathcal{S} -effective content model of \mathcal{I} ; and
 4. each of the n_1, n_2, \dots, n_k is S -valid with respect to S .

Validity with derivation and tolerance

Recalling that \triangleleft

is a subset of $\mathcal{R}(E)^2 \cup (\mathcal{R}(V)^A)^2 \cup T^2$, consider a segmentation of \triangleleft

into a finite collection of segments $\langle_1, \langle_2, \dots, \langle_k$

such that $\langle = \langle_1 \cup \langle_2 \cup \dots \cup \langle_k$

. Since we are free to construct \langle

and partition it any way we like, we could think of each of the segments as corresponding to a particular atomic derivation transformation on types. (In the next section we'll examine some particular kinds of atomic transformations.) Furthermore, we can abuse the notation of juxtaposition and consider composing

the atomic transformations as relations, e.g. $\langle_1 \langle_2 \dots \langle_k$

. Indeed, we can consider regular expressions over atomic transformations, and note that in the usual notation

for regular expressions $\langle = \langle_1 | \langle_2 | \dots | \langle_k$

Let S and I be as above. Suppose \langle

is segmented into a finite collection of segments $\langle_1, \langle_2, \dots, \langle_k$

such that $\langle = \langle_1 \cup \langle_2 \cup \dots \cup \langle_k$

. Let $R(\langle)$

be a regular expression over the segments of \langle

. Putting $R(\langle)$

in the place of \langle

, we can define the $R(\langle)$

-alternatives, the $\cup_{R(\langle)}$

-closure, and the $\cup_{R(\langle)}$

-effective signature by analogy with the \langle

-alternatives, the $\cup_{R(\langle)}$

-closure, and the $\cup_{R(\langle)}$

-effective signature above. An instance I is $\cup_{R(\langle)}$

-**valid** (read "S -derivation valid tolerating transformations in the regular set $R(\langle)$

") with respect to a schema S iff τ_0

's tag is either ϵ_0

or an $R(\langle)$

-alternative of \mathcal{E}_0
, and each node $n \in N$
is $\mathcal{E}_0(n)$
-valid with respect to a schema S . A node $n \in N$
is $\mathcal{E}_0(n)$
-valid with respect to a schema S iff either $\tau(n) \notin \bar{E}$
, or $\tau(n) \in \bar{E}$
and

1. if the type component of n
is $t = \ell(n) \neq \perp$
, then $(R(\triangleleft))^* h(\tau(n))$
and the base type for validation is t
2. if the type component of n
is $\ell(n) = \perp$
, then the base type for validation is $t = h(\tau(n))$
3. for every a
in the domain of n
's attribution \mathcal{P}
, either $a \notin A$
, or if $a \in A$
and $(g(t))(a) \neq \emptyset$
then $\mathcal{P}(a) \in (g(t))(a)$
; and
4. either
 1. $f(h(\tau(n))) \in \mathcal{R}(V)$
;

2. the value component of $\ell(n)$ is in the $\mathcal{S}_{\mathcal{A}(\mathcal{A})}$ -effective content model of t ; and
 3. n has no children
5. or
1. $f(n(\tau(n))) \in \mathcal{R}(E)$;
 2. the value component of $\ell(n)$ is \perp ;
 3. the sequence $\tau(n_1), \tau(n_2), \dots, \tau(n_k)$, where n_1, n_2, \dots, n_k are **all** the children of n taken in \ll order whose tags are in E , is in the regular set defined by the $\mathcal{S}_{\mathcal{A}(\mathcal{A})}$ -effective content model of t ; and
 4. each of the n_1, n_2, \dots, n_k is $\mathcal{S}_{\mathcal{A}(\mathcal{A})}$ -valid with respect to S .

Modeling particular derivation transformations

We now explore how some of the forms of modification described above fit into the formal model. Generally speaking, we proceed by taking a schema S and defining a new schema S' where the components of the latter structure are defined by making incremental changes to the components of the former, but leaving the root element the same in both structures. We will subscript the components of S and S' with their respective

schema names, e.g. \mathcal{F}_S

and \triangleleft_S .

Adding a new tag

From schema S with type t

we create a new schema S' with a new element e'
:

$$\begin{aligned} S' &= \langle E_S, T_S, A_S, V_S, \triangleleft_S, f_S, g_S, h_S, e_0 \rangle \\ E_{S'} &= E_S \cup \{e'\} \\ h_{S'}(e) &= \begin{cases} h_S(e) & \text{if } e \in E_S \\ t & \text{if } e = e' \end{cases} \end{aligned}$$

and \triangleleft_S .

may now order types with content models that are regular expressions over $\frac{E_S}{\triangleleft_S}$.

Creating a new type by adding an attribute

From schema S with type t''

we create a new schema S' having a new type t'

with an additional attribute a'

assigned from the subspace $V' \subseteq V$

:

$$\begin{aligned} S' &= \langle E_S, T_{S'}, A_S, V_S, \triangleleft_{S'}, f_{S'}, g_{S'}, h_S, e_0 \rangle \\ T_{S'} &= T_S \cup \{t'\} \\ \triangleleft_{S'} &= \triangleleft_S \cup \{ \langle g_{S'}(t'), g_{S'}(t'') \rangle, \langle t, t'' \rangle \} \\ f_{S'}(t) &= \begin{cases} f_S(t) & \text{if } t \in T_S \\ f_S(t') & \text{if } t = t' \end{cases} \\ g_{S'}(t) &= g_S(t) \text{ for } t \in T \\ (g_{S'}(t))(a) &= \begin{cases} (g_S(t''))(a) & \text{if } a \neq a' \\ V' & \text{if } a = a' \end{cases} \end{aligned}$$

Creating a new type by restricting choices

From schema S with type t''

we create a new schema S' having a new type t'

such that $f_{S'}(t') = r'$

is derived from $f_S(t'')$

by restriction of choices (see the example in the **derivation by transformation section**):

$$\begin{aligned}
 S' &= \langle E_{S'}, T_{S'}, A_{S'}, V_{S'}, \triangleleft_{S'}, f_{S'}, g_{S'}, h_{S'}, e_0 \rangle \\
 T_{S'} &= T_S \cup \{t'\} \\
 \triangleleft_{S'} &= \triangleleft_S \cup \{ \langle r', f_{S'}(t'') \rangle, \langle t, t'' \rangle \} \\
 f_{S'}(t) &= \begin{cases} f_S(t) & \text{if } t \in T_S \\ r' & \text{if } t = t' \end{cases} \\
 g_{S'}(t) &= \begin{cases} g_S(t) & \text{if } t \in T_S \\ g_S(t') & \text{if } t = t' \end{cases}
 \end{aligned}$$

Creating a new type by cyclic permutation

From schema S with type t''

we create a new schema S' having a new type t'

such that $f_{S'}(t') = r'$

is derived from $f_S(t'')$

by cyclic permutation (see the example in the **derivation by transformation section**):

$$\begin{aligned}
 S' &= \langle E_{S'}, T_{S'}, A_{S'}, V_{S'}, \triangleleft_{S'}, f_{S'}, g_{S'}, h_{S'}, e_0 \rangle \\
 T_{S'} &= T_S \cup \{t'\} \\
 \triangleleft_{S'} &= \triangleleft_S \cup \{ \langle r', f_{S'}(t'') \rangle, \langle t, t'' \rangle \} \\
 f_{S'}(t) &= \begin{cases} f_S(t) & \text{if } t \in T_S \\ r' & \text{if } t = t' \end{cases} \\
 g_{S'}(t) &= \begin{cases} g_S(t) & \text{if } t \in T_S \\ g_S(t') & \text{if } t = t' \end{cases}
 \end{aligned}$$

Conclusions

In the foregoing we have abstracted

- types of elements
- types of attributes
- openness
- type derivation
- element equivalence
- tolerance
- schema validation both with and without tolerance

In doing so we assume only that attributes have simple values, and that elements are either simple values or structures characterized by regular expressions. Indeed the model of types (in) that we employ is essentially the extensional one traditionally studied by logicians. On the surface it would appear that this is not quite the same as the typical computer science model of types where two named types having the same type signature represent distinct types. It happens that we can encode the name of a type both in a regular expression and in an attribution, producing, in effect, distinct copies of the same structure. Using this device together with imposing various restrictions on the derivation partial order, we can reproduce the validation semantics of the XML Schema candidate recommendation. More importantly, however, we have a much richer notion of type derivation and validation.

While implementation is outside the scope of this paper, we should observe that validation, as presented, depends only on a set of regular expressions that can be statically constructed from the definition of a schema. Adding derivation and/or tolerance simply increases the size and/or number of the regular expressions. It is easy to imagine a compiler that undertakes such a construction. Validation at a node reduces to checking if a sequence of tokens associated with each of the child nodes is in the regular set (in) defined by a particular regular expression.

Bibliography

- [Regex] Alfred V. Aho. Algorithms for finding patterns in strings. In Jan van Leeuwen, editor, Handbook of Theoretical Computer Science. MIT Press Elsevier, Cambridge, Massachusetts, 1990.
- [Types] Solomon Feferman. Theories of finite type related to mathematical practice. In Jon Barwise, editor, Handbook of Mathematical Logic. Elsevier North Holland, Amsterdam, 1977.
- [XSD1] Paul V. Biron and Ashok Malhotra, editors. XML Schema Part 2: Datatypes. World Wide Web Consortium, Cambridge, Massachusetts, 2000.
- [XSD2] Paul V. Biron and Ashok Malhotra, editors. XML Schema Part 2: Datatypes. World Wide Web Consortium, Cambridge, Massachusetts, 2000.

Author

Allen Brown Jr.

Senior Program Manager

Microsoft

Postal Address:

One Microsoft Way
98052-6399 Redmond
Washington
USA

Telephone: +1 425 705-3290

Fax: +1 425 536-7329

E-mail: allenbr@microsoft.com

Web: msdn.microsoft.com/xml/

Allen L. Brown, Jr., Ph.D. - Dr. Brown is Senior Program Manager in Microsoft's Web Data Access group where he shepherds Microsoft's efforts in data access related XML standards. Prior to joining Microsoft, Dr. Brown had a lengthy career at the Xerox Corporation, where held various positions including both Research Fellow and VP and CTO of Xerox's XSoft Division. His product development experience includes the management of large teams of engineers, as well as lead architectural roles in both Xerox's landmark Star system, and its Astoria object-oriented document repository. Dr. Brown has held positions in the corporate research centers of IBM, GE and Xerox. He is the author of numerous published technical papers and the holder of several U.S. patents. Dr. Brown's university and national service record includes professorships at the George Washington University and Syracuse University, and membership on the National Research Council NIST Panel on Information Technology.