

Subtyping in XML Schema

Murali Mani
May 16, 2002
revised June 1, 2002

Introduction

This article covers the problems we have to answer when we consider subtyping for XML Schemas, and two different solutions that exist currently. This is written mainly with reference to the XML-Schema proposal from W3C, and two theoretical frameworks for subtyping given in [2], [3].

Why is typing in XML Schemas different and challenging?

In this article, we study the problem which we call the **type membership problem**. *The type-membership problem tries to answer whether a given object, o belongs to type A or not.* In traditional programming languages, this problem is answered as follows: *Suppose o is declared to be of type B , then o is of type A iff B is a subtype of A .* In XML, the type membership problem occurs in two different scenarios – during **type assignment**, and during **type checking**. Type assignment does the following – given a set of type definitions and a set of objects (subtrees), we try to assign types to each object. Type checking occurs during processing and does the following – given an expression where we expect an object of type Type1, can we use an object of type Type2?

Let us look at type assignment more closely. In traditional programming languages, type assignment is non-existent – here, type assignment for objects is done by the user. The user provides a set of type definitions, and whenever he creates an object, he explicitly declares the type to which the object belongs. However, it is different for XML Schemas. Here the user provides a set of type definitions, which define a tree language (set of trees) in an XML Schema. Given a document, the user does not specify the types to which the different objects (subtrees) belong; instead, an XML processor has to assign the types for the various subtrees in the document.

For example, the user will provide an XML Schema such as

```
Library -> library (Book*)  
Book -> book (Title, Author*)  
Title -> title (String)  
Author -> author (String)
```

Consider the following document, Doc1, which is specified to be valid against the above XML Schema.

```
<library>  
  <book><title>XYZ</title><author>ABC</author><author>DEF</author></book>  
</library>
```

Now, the types for the different subtrees are not specified. (The above document has five subtrees, whose roots are library, book, title, author and author.) However an XML processor can determine the types unambiguously by “comparing” the document against the schema (let us assume that the type assignment is unique).

As mentioned before, the type membership problem for XML Schemas comes in two different scenarios – (a) during type assignment, when the document has no types assigned to the objects (subtrees), and we have to assign types for the different subtrees, (b) during document processing, when the different subtrees have types assigned to them.

Let us first examine the type membership problem during type assignment (remember that this problem does not occur in traditional programming languages). For example, consider a subtype Book1 defined for Book as
Book1 extends Book -> book (Title, Author*, Publisher)

Now consider the document, Doc2

```
<library><book>  
  <title>XYZ</title><author>ABC</author><author>DEF</author>  
  <publisher>IEEE</publisher>  
</book></library>
```

Now, does the subtree rooted at library belong to (valid against) the type Library? The answer is the subtree is valid, provided the book subtree (or node) in the document is assigned type Book1, and we know that Book1 is a subtype for Book.

So what is the algorithm for checking whether an object belongs to a particular type or not? Suppose we are given a subtree T, and we have to check whether T is valid against type Type1. But T need not belong to $TL(Type1)$, where $TL(A)$ is the tree language generated by the tree type A. One way of doing validation is as follows: we consider all subtypes of Type1 recursively till there exists a Typek such that T belongs to $TL(Typek)$. In short, the definition of validity of a tree T against a type Typei is defined in this framework as follows:

T is valid against Typei iff there exists a type Typek, such that T belongs to $TL(Typek)$ and Typek is a subtype of Typei.

Now, T will be **assigned** type Typek. However the above definition for type membership is not sufficient in the XML scenario where the types for objects are not declared by the users – *the problem is that the above type assignment can become ambiguous* (because there is no fixed order defined for type hierarchy between siblings). For example, consider a base type Book defined as
Book -> book (Title, Author*) and two subtypes defined for Book as
Book1 extends Book -> book (Title, Author*, Publisher)
Book2 extends Book -> book (Title, Author*, Publisher*)

Now suppose we are validating the following subtree against the tree, Doc3

```
<book>
  <title>XYZ</title>
  <author>ABC</author><author>DEF</author>
  <publisher>IEEE</publisher>
</book>
```

Now the above tree can be assigned type Book1 or Book2, and this is ambiguous. Avoiding this ambiguity is a primary goal of several schema languages, such as XML-Schema. We will see in the next section how the unique type assignment is maintained in XML-Schema in the presence of subtypes, by allowing user to declare types for objects in a controlled manner.

Now let us consider the type membership problem that occurs during processing. Typical scenario where this occurs is as follows. Suppose we have defined a function, Fn as
Fn: P1, P2, P3, ..., Pn -> R

Fn takes n parameters of types P1, P2, P3, ..., Pn as input and produces an output object of type R. Now the questions are: can we pass an object of type Qi in stead of an object of type Pi? - the answer is yes, provided Qi is a subtype of Pi. The second question is can we use the output of Fn in an expression that requires an object of type S. The answer is yes provided R is a subtype of S. (Note that the same questions arise in traditional programming languages also, and they have identical answers. However, for XML schemas, there are two different ways of defining when a type is subtype of another. The two different solutions should be clear to the reader from this article).

Subtyping in W3C's XML-Schema

XML-Schema defines subtyping using six main constructs: *extension*, *restriction*, *xsi:type*, *final*, *block* and *substitution groups* (constructs such as abstract types are not significant to our discussion). We base our discussion on “tree types”, that is types that produce trees. XML-Schema actually defines subtyping over “hedge types” – types that produce hedges (hedge is an ordered list of trees). Using tree types does **not** affect the conclusions, and it is more convenient. One difficulty in using tree types is that our discussion actually becomes slightly complicated when we consider substitution groups, as we will see later.

The definitions of the different constructs used by XML Schema are as follows:

extension: Suppose we have defined a type as Type1 -> root (RE1), we can define Type2 as an extension of Type1 and define it as Type2 -> root (RE1, RE2).

The example of defining Book1 as a subtype of Book (above) is an example of extension.

Book -> book (Title, Author*)

Book1 extends Book -> book (Title, Author*, Publisher)

There are “no” constraints on what types should be present in RE2. (Actually there are constraints, but these constraints are not significant to our discussion here). For example, we can define Book3 as

Book3 extends Book -> (Title, Author*, Title)

The above extension cannot be captured by the subsumption framework defined in [2], as we will see later.

restriction: This is a way of defining subtypes as follows: Suppose we have defined a type as Type1 -> root (RE1), and we define Type2 restricts Type1 -> root (RE2), that is Type2 is a subtype obtained by restriction from Type1, then L (RE2) is a subset of L (RE1). Here L (RE) refers to the regular language denoted by the regular expression RE. For example, we define a subtype Book4 of Book by restriction below:

Book4 restricts Book -> book (Title, Author, Author)

It is easy to observe that if type A is declared as a subtype of type B by restriction, then TL (A) is a subset of TL (B). However, the converse need not be true. This is different from [3] where type A is a subtype of type B **iff** TL (A) is a subset of TL (B). For example, consider Book5 -> book (Title, Author1*), where TL (Author1) is a subset of TL (Author). Now Book5 is a subtype of Book in [3], whereas Book5 cannot be declared a subtype of Book in XML-Schema.

Also, in XML-Schema, subtypes should be declared explicitly, whereas such explicit declaration is not needed in [3]. For example, consider a type Book6 defined as Book6 -> book (Title, Author, Author). Even though L (Title, Author, Author) is a subset of L (Title, Author*), Book6 is not a subtype of Book in XML-Schema. However, in [3], Book6 is a subtype of Book.

xsi:type: This construct allows users to declare types for objects in a controlled manner. *This construct is essential to ensure that the type assignment is unambiguous in XML-Schema.* For example, consider the type definitions, Book, Book2, Book4 as

Book -> book (Title, Author*)

Book2 extends Book -> book (Title, Author*, Publisher*)

Book4 restricts Book -> book (Title, Author, Author)

Consider the documents Doc4, Doc5 and Doc6 as:

Doc4:

```
<book><title>XYZ</title><author>ABC</author><author>DEF</author></book>
```

Doc5:

```
<book xsi:type="Book2">
  <title>XYZ</title><author>ABC</author><author>DEF</author>
</book>
```

Doc6:

```
<book xsi:type="Book4">
  <title>XYZ</title><author>ABC</author><author>DEF</author>
</book>
```

The documents are the same, except that the type assignments for each document will be different, the tree rooted at book will be assigned type Book for Doc4, Book2 for Doc5, and Book4 for Doc6. Also note that all the book objects belong to type Book, because of the subtyping relationships.

Type membership of a tree against a type in XML-Schema is defined as given below. We consider two cases:

- a) *Suppose T has no xsi:type defined:* T is valid against Type1 iff T belongs to TL (Type1)
- b) *Suppose T has defined xsi:type to be Typek:* T is valid against Type1 iff T belongs to TL (Typek) and Typek is declared to a subtype of Type1.

Type assignment of a tree, T in a valid document by the XML processor is as follows:

- a) *Suppose T has defined xsi:type to be Typek:* T is assigned type Typek.
- b) *Suppose T has no xsi:type defined:* T is assigned the type as verified by the XML processor during validation.

The usage of xsi:type along with other constraints imposed by XML-Schema (not discussed here) ensure that the type assignment is unambiguous, and that validation can be done in linear time.

final: This is used to prevent subtypes from being derived for a type. For example, suppose Book was defined with a final value as extension, we cannot define subtypes Book1 and Book2, which are extensions of Book. A type can have a value of final as restriction.

block: This is a feature used to specify that an object does not belong to a specific type. For example, consider an object T declared to be of Type1, where Type1 is a restriction of Type2. Now T belongs to Type2 under normal circumstances. However, if we defined that Type2 has a block value of restriction, then T does not belong to Type2. The same thing applies to extension also.

Having restriction as the value for block is a contradiction from [3]. If Type1 is a subtype of Type2 by restriction in XML-Schema, then Type1 will necessarily be a subtype of Type2 in [3]. Hence any object of Type1 is also an object of Type2.

Substitution group: The subtyping which we have seen so far requires that the root element of the subtype be the same as the root element of the parent type. However, this need not always be the case. XML Schema allows subtyping even where the root elements are different using substitution groups. This works as follows:

We can declare any tree type A to be a subtype of tree type B. There is a constraint however, which we will explain using hedge types. Let tree type A be defined as A -> a (AA), and tree type B be defined as B -> b (BB). Now we can define A as a subtype of B

only if AA is a derived type of BB (through extension or restriction). Now in the document where an object of B is supposed to occur, we can substitute an object of A.

An example of substitution group is given below. Consider the types

Library -> library (Book*)

Book -> book (Title, Author*)

Book2 extends Book -> book (Title, Author*, Publisher)

Article -> article (Book2)

Note we write the rule for Article as Article -> article (Book2), which represents that the root is article, and the content is equal to the content of Book2. Now we can declare Article is a subtype of Book. (The content of Article is equal to the content of Book2, which is a subtype of Book).

Now consider the document, Doc7. Doc 7 is valid against the type Library.

```
<library>
  <article>
    <title>XYZ</title><author>ABC</author><author>DEF</author>
    <publisher>IEEE</publisher>
  </article>
</library>
```

During type assignment, the type membership is answered by XML-Schema as follows:

An object o is said to belong to type $Type1$ iff there exists a type $Type2$, such that o belongs to $TL (Type2)$, and $Type2$ is declared to be a subtype of $Type1$. The object o is assigned type $Type2$ here. There is a constraint that $Type1$ should not have defined block such that an object of $Type2$ cannot be considered as an object of $Type1$.

After type assignment, the objects (subtrees) have all been assigned types. These objects can now be processed. The type membership problem during processing is answered as follows. Though XML-Schema does not explicitly mention how block will be used here, we assume a similar usage of block as during type assignment.

An object o that belongs to type $Type2$ is said to belong to type $Type1$ iff $Type2$ is declared as a subtype of $Type1$. There is a constraint that $Type1$ should not have defined block such that an object of $Type2$ cannot be considered as an object of $Type1$.

Subtyping using the Subsumption Framework

The subsumption framework defines subtyping as follows: A type A can be a subtype of another type B, only if we can define a **subsumption mapping** from the content of A to the content of B. Subsumption mapping from A to B is a mapping from the tree types of A to the tree types of B. An example is given below.

Title -> title (String)

Author -> author (Son | Daughter)
Author1 -> author (Son)
Book1 -> book (Title, Author*)
Book2 -> book (Title, Author1*)

Now we can declare Book2 as a subtype of Book1, the subsumption mapping, f is defined as follows:

$f(\text{Title}) = \text{Title}; f(\text{Author1}) = \text{Author}$

Now Author is a valid subsumption mapping from Author1 because, the roots of Author1 is a subset of the roots of Author, and the regular language defined by the content of Author1 is a subset of the regular language defined by the content of Author.

Observation: *There exists a subsumption mapping corresponding to any restriction.*

The above is straightforward – the identity mapping will serve as the necessary subsumption mapping. For example, consider

Book -> book (Title, Author*)
Book4 restricts Book -> book (Title, Author, Author)

The subsumption mapping from Book4 to Book, f , is given by
 $f(\text{Title}) = \text{Title}, f(\text{Author}) = \text{Author}$

Capturing extension using the subsumption framework is not easy. To capture extension, [2] defines a new type corresponding to every type definition as follows. Suppose A be a type given by A -> root (RE), now let us define a type corresponding to it called A', defined by A' -> root (RE, Any*), where Any can match any tree.

Now we can try to define subsumption mappings for extensions as follows: Suppose we have A -> root (RE1) and B extends A -> root (RE1, RE2), we can define subsumption mappings from B' to A' where there will be identity mappings from every type in RE1, and the mapping for every type in RE2 will be to Any.

Observation: *There does not exist a subsumption mapping corresponding to every extension.*

We cannot define an subsumption mapping for the following valid extension in XML Schema.

Book -> (Title, Author*)
Book3 extends Book -> (Title, Author*, Title)

To summarize, subsumption framework can capture restriction, however it cannot capture extension in XML schema. The type membership problem during type assignment is answered by the subsumption framework as follows. (Note that this is not defined precisely in [2], but we believe that this is how they would have defined).

An object o is said to belong to type $Type1$ iff there exists a type $Type2$, such that o belongs to $TL (Type2)$ and a subsumption mapping is declared from $Type2'$ to $Type1'$.

The type membership problem after type assignment is answered as follows:

An object o that belongs to type $Type2$ belongs to type $Type1$ iff a subsumption mapping is declared from $Type2'$ to $Type1'$.

Also one more thing to observe is that if we can define a subsumption mapping from $Type2'$ to $Type1'$, then $TL (Type2')$ is a subset of $TL (Type1')$.

Implicit Subtyping for XML using Regular Tree Language Inclusion

This is the framework defined in XDuce [3]. Here the user does not explicitly declare any subtyping. In stead subtyping is implicit – a tree type A is a subtype of a tree type B , iff $TL (A)$ is a subset of $TL (B)$.

In this framework, the type membership problem during type assignment is answered as follows:

An object o is said to belong to type $Type1$ iff o belongs to $TL (Type1)$.

The type membership during processing is answered as follows:

An object o that belongs to type $Type2$ belongs to type $Type1$ iff $TL (Type2)$ is a subset of $TL (Type1)$.

Another construct provided in [3] for subtyping is subtagging. However this is not significant to our discussion.

Discussion

From the above discussions and frameworks, it should be clear that there are two different frameworks for defining subtyping in XML. One is *implicit subtyping based on inclusion* as in XDuce [3], and the other is *explicit subtyping* as in XML-Schema and [2]. The two frameworks are incompatible, and hence it is difficult to combine the two. As an example, let us consider the following four types:

Book -> book (Title, Author*)

Book1 -> book (Title, Author*, Publisher)

Book5 -> book (Title, Author1*), where $TL (Author1)$ is a subset of $TL (Author)$

Book6 -> book (Title, Author, Author)

The table below illustrates which of the three types (Book1, Book5 and Book6) can be declared as subtypes of Book in XML-Schema and [3].

Type	Can be Subtype in XML-Schema?	Can be subtype in XDUce?
Book1	Yes	No
Book5	No	Yes (always a subtype)
Book6	Yes	Yes (always a subtype)

The above differences manifest during type checking as follows: Suppose we have a function, $F1: T \rightarrow \text{null}$ (takes a parameter of type T , and returns nothing), can we pass to $F1$ another function $F2$ which returns type $T1$. In [3], the answer is yes provided $TL(T1)$ is a subset of $TL(T)$. In [1], the answer is yes provided $T1$ is declared a subtype of T .

Another question that we consider important is how different subtrees will be selected in the first place, before we start processing them. In other words, will we start from the root of the document always and select subtrees, or is there a way by which we can arbitrarily select a set of subtrees? The approaches used in XQuery etc seem to indicate that the former seems to be the way things will be done.

The above question is an attempt to examine applications for type assignment, and any advantages unambiguous type assignment might provide. Discussions in the xml-dev mailing list (in May 2002) show one application of type assignment – there are some XQuery constructs such as *o instanceOf T* that try to examine whether an object o is an instance of type T or not. Similarly another application of type assignment where unique type assignment might have some advantages is in design of XML document editors that allow us to update XML documents – we can develop efficient algorithms for validating updates if we know the types assigned to the different elements in the original document before the updates.

Acknowledgements:

Acknowledgements are due to the xml-dev mailing list, where I could get answers to several of my questions answered. Several members of the list assisted me, of which I would like to especially thank Dare Obsanjo.

References:

[1]: **XML Schema, W3C**

[2]: **Subsumption for XML**, Gabriel Kuper, Jerome Simeon, Int'l Conference on Database Theory, 2001

[3]: **Regular Expression Types for XML**, H. Hosoya, J. Vouillon, B. Pierce, Int'l Conference on Functional Programming, 2000

[4]: **Foundations of Object Oriented Programming Languages: Types and Semantics**, K. B. Bruce, <http://www.cs.williams.edu/~kim/FOOLbook.html>