Opening XML Schema's Data Model to XPath 2.0

Felix Michel
Department of Information Technology and Electrical Engineering
Swiss Federal Institute of Technology Zürich (ETHZ)

November 2006 (TIK Report 264)

Abstract

XML Schema is a very expressive grammar-based schema language that additionally supports advanced data modeling techniques, namely through its type concept, and allows for describing relationships between structural elements in an expressive and semantically meaningful way. Upcoming type-aware XML technologies like XSLT 2.0, XQuery, and XPath 2.0 increasingly strive to use this structural information, but its retrieval is difficult and only partially possible. Both a unified data model for XML Schema and a set of accessor functions are necessary for enabling new technologies to utilize XML Schema's full capabilities. We will present a function library relying on an XML-based representation of XML Schema's data model and demonstrate how this substantially enhances XPath 2.0 and XSLT 2.0 and how this proves highly beneficial to applications.

1. Motivation

XML Schema [1, 7], defined by the World Wide Web Consortium and woven into many others of their standards as a foundation, is the most important schema language for XML today. It is not only a very expressive grammar-based schema language, but also comprises a certain amount of support for advanced data modeling techniques such as reusable types, type derivation, type substitutions, substitution groups, and reusable element and attribute groups. Thus, XML Schema allows for describing relationships between structural elements in a more expressive and semantically more meaningful way. Technologies dealing with XML increasingly make use of this structural information, as type-aware technologies such as XSLT 2.0 [4], XQuery, and XPath 2.0 [2] become more popular. But much of the structural information that can be encoded using the powerful and expressive capabilities of XML schema is difficult to retrieve and only partially accessible to applications.

This is partly due to the still limited amount of type-related functionalities in languages like XPath or XSLT, but also to the particularities of the XML syntax of the normative XML representation for XML Schema, which makes it hard to reconstruct the underlying abstract data model in some cases. But simple and unified accessibility of the schema's data model, and the structures and their relationships described by this data model, are essential for utilizing the structural information.

Therefore we state the strong need for a unified data model for XML Schema, capturing the structures represented by the XML syntax while remaining principally independent from this XML representation. In addition, clearly defined ways of accessing this data model are needed.

In the following Section 2, we describe criteria, prerequisites and objectives of a unified abstract data model. In Section 3, we discuss three different approaches to attaining this goal. Finally, in Section 4, we present a function library which works on structures closer to the data model and gives access to more sophisticated model information described by the Schema. We demonstrate the benefits of such functionalities and conclude with an outlook in Section 5

2. 'Infoset' for Schema: A Unified Abstract Data Model

An abstract data model is to be distinguished from the data representation. For example in XML, the physical XML documents are the data representation, whereas the Infoset [13] is a possible abstract data model behind it. In the case of XML, agreement on a unified data model was complicated by the fact that XML was originally only designed by means of its syntax, and that the model behind it had to be abstracted later, when need for such a model arose. Therefore, multiple data models for XML have been proposed: Different perspectives, focuses, and contexts of intended use have led to different approaches, such as DOM, Infoset [13], the XPath 1.0 data model [11], and most recently XDM [3].

2.1. Present Data Models for XML Schema

For XML Schema, the W3C recommendation [1] lays out the abstract data model which consists of Schema Components first and carefully stresses its distinction from possible rep-

resentations, of which the normative XML representation is one example.[1] Yet the present situation seems to be similar to the one described above: Different competing data models, formulated from different perspectives and with different focuses, provide a set of overlapping functionality:

- Similarly to DOM in XML, there are programming APIs for XML Schema as well: Microsoft's XML Schema Object Model (SOM) [19], Eclipse's XML Schema Infoset Model (XSD) [17], the XML Schema Object Model (XSOM) [18] from java.net, and the Castor XML Schema Support [20]. Each of them includes a data model, which is very likely to be similar to the abstract data model in the Recommendation, but not necessarily identical because an actual implementation requires simplifications and adaptations to be made.[2]

- Level 3 of the Document Object Model (DOM), originally planned to formulate an abstract schema object model [16], that ambitiously intends to cover even multiple schema languages, namely DTD and XML Schema. Even though the abstract data model discussed here will be strictly focused on XML Schema, this project is worth being mentioned.

- There has been an attempt to describe XML Schema in a formal way [9]. Although this Formal Description did not aim at defining a data model different from the one in [1] at all, the different perspective and the fact, that it addressed only working drafts of XML Schema led to a potentially divergent data model. (For instance, Identity Constraints are not covered.)

- Section 2.4 of the "XQuery 1.0 and XPath 2.0 Formal Semantics" [5] formally describes the type system of XPath 2.0 and XSLT 2.0, which is based on XML Schemas type concept. Hence, an essential part of XML Schema's data model is remodeled here.

- The appendix D of the same document [5], unsuspiciously named "Importing Schemas", outlines a simplified data model of XML Schema, which comprises only

[1] More precisely, the Recommendation [1] says: "Just as [XML 1.0 (Second Edition)] and [XML-Namespaces] can be described in terms of information items, XML Schemas can be described in terms of an abstract data model. In defining XML Schemas in terms of an abstract data model, this specification rigorously specifies the information which must be available to a conforming XML Schema processor. The abstract model for schemas is conceptual only, and does not mandate any particular implementation or representation of this information. To facilitate interoperation and sharing of schema information, a normative XML interchange format for schemas is provided."

[2] For example, if the structure of the object model is a tree, a spanning tree has to be identified within the non-tree structure of Schema Components relationships.

parts considered relevant in the context of XQuery and XSLT 2.0. This process of simplification very much resembles to the way XML's Infoset emerged. In fact, the Recommendation [13] states in its Introduction somewhat vaguely:

> "It does not attempt to be exhaustive; the primary criterion for inclusion of an information item or property has been that of expected usefulness in future specifications."

- The "XQuery 1.0 and XPath 2.0 Data Model (XDM)" [3] again captures some parts of XML Schema's type concept, additionally introducing accessor functions to XML's data model. Some of these methods (e.g. `dm:type-name`) may be considered to be pointing into the XML Schema data model as well.

Because the XML Schema Recommendation defined an abstract data model beforehand, the actual data models described are perhaps only marginally divergent, but many of the differences reflect substantial desiderata:

- A "formal description", rather than a definition only by prose or via a representation [8] would be desirable.
- Some simplifications or abstractions may be considered: For example, the functionally equivalent replacement of the properties {`min / max occurs`} of the "particle" Schema Component by mere Kleene-operators, as it is done in the type system in [5].
- Although it is not necessarily part of a data model, unified and abstractly defined accessor functions to the data model are desirable.

It is neither the goal of this paper to judge the quality of the above data models, nor to propose better solutions in the respective areas. This enumeration should rather demonstrate that the design and definition of a unified data model is always a process of harmonizing differently focused and scoped approaches, as it was the case for the XML Infoset.

2.2. Prerequisites

This leads to a set of requirements for a unified abstract data model. The task of defining an abstract data model for XML Schema is multi-faceted and requires other issues to be addressed. The prerequisites and requirements we outline in the following are very similar to the ones for XML Infoset, although not all of them have been addressed in the final Recommendations of the Infoset and XDM.

1. We need to agree on rules for **canonicalization** of XML Schemas. This may not include *normalization* (e.g. as proposed by Embley's *XNF* [15]), but should at least provide rules analogous to the ones for canonical XML [14].

2. We need to have precise **designators** for addressing the components of the data model.[3] The W3C Working Draft "XML Schema: Component Designators" addressed this problem [10].

3. In order to be able to precisely describe, navigate and maybe visualize the structure of the data model, we need a **mathematical model** capable of capturing this structure. While XML instances can be described reasonably by rooted trees, the task is distinctly more complicated for XML Schemas. Perhaps a combination of a rooted tree (representing the type hierarchy), a tree grammar (describing the possible document structures), and a set of relations connecting the two might be an approach.

4. Additionally, we might want to negotiate ways of keeping track of **additional, external information**. The methods as well as the extent of such functionality are debatable. We give two brief examples where such features may be useful:

   - If the data model of a schema has not been created synthetically, but has been retrieved from schema documents, it might be desirable to store information about the original XML representation in order to enable loss-less round-tripping, schema file editing, and so on.
   - An XML Schema might contain references to a more abstract (e.g. conceptual) model from which it has been derived, thus allowing for retrieval of some conceptual information at a later time or if the schema is used in a context where the conceptual model itself is not available.

[3] This is not a trivial task for XML Schema. As Thompson points out in [22], expanded names in XML Namespaces [21] can be ambiguous. Furthermore, in [10], Holstege and Vedamuthu emphasize the fact that XML Schema does not require all component names to be unique: e.g., element declaration components and type definition components may use the same names. This resembles to the controversial concept of XML Namespace Partitions once part of a non-normative appendix to the XML Namespace Recommendation [23].

5. Finally, the definition of a class of **accessor functions** and their common syntax and use certainly would be beneficial, although this should be done separately from an abstract data model in a strong sense.

3. Approaches

In the following sections, we discuss three possible approaches to obtaining a unified data model. They differ mainly in the way of how such a model can be achieved, and not in the underlying structure of the model. Although the approaches presented are on different logical levels, *Table 1* compares them from a more practical point of view.

3.1. Abstract Specification

An abstract specification would certainly be the preferred way of defining a unified abstract data model. This might be done in prose (akin to the Recommendation specifying the XML Infoset [13]), but a formal approach might be considered as well. In [6], Siméon and Wadler clearly demonstrate benefits of a formal description in the area of XQuery.

In this paper however, we will not strive for such abstract or even formal definitions. As the expression "*unified* data model" implies, this should be done by a competent committee. The following Sections 3.2 and 3.3 do not have the ambition of defining the ultimate abstract data model either, but they provide pragmatic ways for experimenting with data models, giving an idea of what and how such models can contribute in the context of the upcoming type-aware technologies, and letting us determine needs and requirements more precisely. We firmly believe that in order to prove useful in practice, an abstract definition must be preceded by cycles of experimental phases.

3.2. SCX: Schema Components XML Representation

As mentioned in Section 2.1, the W3C Recommendation specifying XML Schema [1] says:

> "The abstract model for schemas is conceptual only, and does not mandate any particular implementation or representation of this information. To facilitate interoperation and sharing of schema information, a normative XML interchange format for schemas is provided."

This phrase inspired us to think of ways of defining another syntax, maybe facilitating "interoperation and sharing of schema information" even more. While proposals like for instance [24] were aiming at introducing alternative non-XML syntaxes for XML Schema, which can be translated into XML Schema documents, we decided to create an XML Schema schema describing XML files that in turn can be used in order to describe XML Schemas, but more closely represent the structure of the Schema Components used in the

abstract data model of the specification. In the following, we refer to the former schema as the "XML Schema for SCX", and we refer to the XML files compliant to this schema as "SCX schemas". Obviously, SCX schemas cannot be used for validation of instance documents with current validators. The sole objective of SCX schemas is to grant facilitated access to the data model of XML Schemas. The structure of SCX schemas is — as far as possible — an XML representation of the Schema Components described in the specification, hence the name "Schema Component XML representation". An excerpt from the Schema for SCX shall illustrate this:

```
<xs:complexType name="complexTypeDefinitionType">
  <xs:sequence>
    <xs:element name="properties">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="name" type="xs:NCName" minOccurs="0" />
          <xs:element name="targetNamespace" type="scx:xmlnsType"
                      minOccurs="0" />
          <xs:element name="prohibitedSubstitutions"
                      type="scx:substitutionControlSet" />
          ...
    <xs:element name="pointers">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="baseTypeDefinition" type="scx:typeRef" />
          <xs:element name="attributeUses" type="scx:attributeUseRef" />
          ...
    <xs:element name="ports">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="baseTypeDefinition"
                      type="scx:complexTypeDefinitionRef"
                      minOccurs="0" maxOccurs="unbounded" />
          ...
```

Note that most of the relationships among Schema Components are not modeled using hierarchical nesting of elements,[4] but through ID/IDREF-like "pointers" described by Identity Constraints. If we think again in terms of the non-normative diagram form Appendix E of the Recommendation, the `scx:pointers` element contains outgoing, the `scx:ports` element ingoing arcs. Because of this, those arcs are bidirectionally navigable. This alleviates for example the retrieval of all derived types from a given base type. In practice though, navigation can't be done as conveniently as for nested structures, where XPath offers sophisticated axes. But using XSLT's `xsl:key` construct, it can be done in a reasonable way.

Because it is bound to and limited by the syntax of XML Schema, SCX might not be a preferable way for defining an abstract data model, but it provides a syntax facilitating access to its structure, its components, and their relationships. Representations of Schema Compo-

[4] In fact, only the containment relationship between the "Schema as a whole" element and the components is modeled as a conventional hierarchical nesting.

nents (or, speaking in analogy to XML Infoset: information items) in XML can be especially useful in the context of XSLT, e.g. the function library to be described in Section 4.

3.3. DMR³: Data Model with Redundantly Resolved References

For our work, we created a testbed which completely relies on portable, presently available standard technologies like XML Schema, Schematron, and XSLT. As a first approach, we define an extension to the XML representation of XML Schema that enables us to store the data model of a fully assembled XML Schema (i.e., the abstract Schema with all dependencies like `xs:include` and `xs:import` resolved). This augmented schema format also includes redundant information about the relationships among schema components. We call these preprocessed and enhanced Schemas DMR3 schemas (for "Data Model with Redundantly Resolved References") and use them as an intermediate format. This intermediate format facilitates accessing the abstract model and enables us to create the XSLT 2.0 function portrayed in more detail in Section 4.

Because the format only adds information in a standard-compliant way (using XML Schema's `xs:appinfo` element), the DMR³ documents are still perfectly usable for validation of XML instances. The validity of the additional parts is ensured by both an XML Schema schema (`DMR3.xsd`) and a Schematron schema (`DMR3-constraints.sch`). The former describes the syntactical structure of the DMR³ elements, the latter checks additional constraints like uniqueness of IDs and consistency of Schema Component relationships. The use of XML Schema describing parts of XML Schemas leads to the unusual appearance of `xsi:schemaLocation` in a DMR³-annotated XML Schema:

```
<xs:schema
  targetNamespace="http://people.ee.ethz.ch/~femichel/namespaces/example-1"
  xmlns:dmr="http://people.ee.ethz.ch/~femichel/namespaces/data-model-r3"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://people.ee.ethz.ch/.../data-model-r3 DMR3.xsd"
...
```

The purpose of DMR³ is clearly *not* to be a proposal for an abstract data model. Its syntax is of a very doubtful beauty and undergoes frequent changes. But it proves useful as a sandbox for testing functions and for evaluating more thoroughly the value of different functionalities and properties. It helps determining a possible set of useful and necessary properties, structures and accessor functions that might become part of a unified data model. An example excerpt from a DMR³ schema looks as follows:

```
<xs:element name="nested" dmr:uid="eld-2"
  dmr:tns="http://people.ee.ethz.ch/~femichel/namespaces/example-1">
  <xs:complexType dmr:uid="ctd-4"
    dmr:tns="http://people.ee.ethz.ch/~femichel/namespaces/example-1">
    <xs:sequence>
      <xs:annotation><xs:appinfo>
        <dmr:flatSet>
          <dmr:element ref="eld-3" type="ctd-1" by="declaration" />
```

```
            <dmr:element ref="eld-3" type="ctd-2" by="typesubstitution" />
    ...
```

Note that we addressed some of the issues described in Section 2.2, and some of the problematic discrepancies between XML Schema Components and their XML representation in a very pragmatic way:

- We identify the components by simply introducing unique `dmr:uid` identifiers.
- We label anonymous type definitions as well and treat them in the same way as global, named type definition. This approach is also taken by [5, 6].
- We add the target namespace information — delegated to the `xs:schema` element in the normative XML syntax — to each particular definition and declaration. This is a re-approximation to XML Schema's abstract data model, where the respective Schema Components have a property `{target namespace}`.
- The `xs:sequence` model group shows an `xs:annotation` element used for holding DMR[3] information. Here it contains what we call a *flat set*: the expanded set of all (element, type) pairs that can appear as particles of this model group in an instance.

3.4. Comparison

As mentioned in the beginning of this Section 3, the three approaches presented are on different logical levels. Even though it is worth to compare them using the criteria of Use, Accessibility, and their respective Level of Abstraction.

	DMR3	**SCX**	**Abstract**	
			Prose	**Formal Description**
Level of abstraction	Low. Large parts of the present XML representation of XML Schema are adopted.	Fair. But still a representation with possible limitations imposed by the syntax.	High. The effective level of abstraction depends on linguistic subtleties.	High. Additionally, the formal notation enables abstraction from semantics, thus yielding higher generality.

Accessibility	High. Readily accessible by existing standard technologies. Still a valid Schema, can be used for validation.	Fair. Accessible with standard technologies. Cannot be used for validation with conventional validators.	Poor. No machine-readable representation.	Potentially machine-readable, but not for use with presently deployed technologies.
Use	Testbed for use and development of the Schema Component Functions described in Section 4	Supporting aid for reasoning about and test object while developing of an abstract data model. Partially useable for SCF objects described in Section 4.3.	Subject for discussions, specification.	Specification, verification.

Table 1: Comparison of approaches

4. The Schema Components Function Library

One of the elements touted as primary advantage of XPath 2.0 and the technologies built on top of it, i.e. XSLT 2.0 and XQuery, is XPath 2.0's *type support*. The types used in XPath 2.0 are based on the type system of XML Schema. While basic XSLT processors only support a subset of XML Schema's built-in atomic types, Schema-aware XSLT processors even allow for user-defined types to be used. Even though the idea of using XML Schema types as a foundation of XPath's type system may seem striking, there are still some open issues, if one has a closer look at XPath 2.0. The reasons are various:

- First of all, the expressions "type support" and "type-aware" are potentially misleading. Section 4.1 tries to clarify this.
- Furthermore, many of the more advanced type-related functionalities are only available using Schema-aware processors. It has to be seen how popular such processors, and hence how widely useable the related functionalities will become.
- The support for Schema types provided in XPath 2.0 covers — perhaps surprisingly — only parts of the accessor functions of the XPath 2.0 data model [3].

- Finally, the type information retrievable in XPath 2.0 is only very shallow. This is substantially due to the lack of a unified data model for Schema Component information in general and for type information in particular.

This last point, of course, is the connecting link to the discussion of unified data models for XML Schema above. Although the XML Schema Recommendation comprises an abstract data model, there is no appropriate, unified data model and no defined ways of accessing it that technologies like XPath could rely on. As Kay points out in [12], there is a data model in actual implementations of XSLT processors, but this data model is opaque, implementation-dependent, and therefore not of further use:

> "Any real schema-aware XPath processor will need to have some kind of access to schema information both at compile time and at runtime, but the W3C specifications have not tried to model exactly what this should look like. In practice, the type annotation on a node is likely to be implemented as some kind of pointer into the metadata representing the cached schema information."

A *unified* data model exactly would enable utilizing this "metadata" and "schema information". Section 4.2 discusses potential accessor methods to this information, Section 4.3 shows how the data model might look for XML Schema types, and Section 4.4 demonstrates, how availability of this information will be beneficial to applications.

4.1. Types in XPath 2.0

In Computer Science, types usually exhibit the following properties: They define a set of permissible values, and they define which operations are possible on a value of that type. In a more general sense, types can be seen as an abstraction describing a set of instances. In XML schema languages, the aspect of operations often is only subordinate, and thus the more general notion seems to be more appropriate. In XML Schema, the complex types are examples of types in the latter sense, while simple types are also types in the former sense. A heterogeneity of type concepts is observable in XPath as well.

The notion of "types" is applied in at least three different senses in XPath 2.0:

1. XPath 2.0 is a *typed programming language*, and its type system is based on *strong typing*. This is a noteworthy advantage over the type system of XPath 1.0, and often, when praising the type support of XPath 2.0, one merely refers to this fact.

2. There are different *kinds of nodes* in XPath, both 1.0 and 2.0. XPath distinguishes seven kinds of nodes: Element nodes, attribute nodes, text nodes,

namespace node, and so on. These kinds of nodes sometimes are called " types" as well (in fact, they are types as well), but using the expression " kinds" helps minimizing the confusion.

3. With the upcoming XPath 2.0, *XML Schema types* can be used in XPath, XSLT and XQuery. It is far beyond the scope of this paper to discuss the type concept of XML Schema, let us only recall that there are two categories of types: *simple* and *complex* types. This third facet of types in XPath 2.0 is not independent from the two above: The strong typing of XPath uses atomic types from XML Schema, and in the `KindTest` constructs occurring in sequence type descriptors, names of elements or even complex types from a Schema might be used, allowing in fact for testing the "type" of nodes in the sense of XML Schema.[5]

Furthermore, according to Kay [12], XPath distinguishes "types of the values that XPath can manipulate from the types that can appear as annotations to nodes." Here again, the atomic XML Schema types can appear in both categories. Instead of discussing the type system of XPath 2.0 in more depth, we recall the principal advantages typing can provide:

- Typed values allow for error-checking both at compile time and at runtime.
- Performance optimizations can be made, because the restricted value sets of the types allows for predictions.
- Typed programming languages allow for polymorphism.

By using our library that provides functions enforcing the use of types in XPath and XSLT, we endeavor to expand these fundamental advantages of typing. Section 4.4 demonstrates how the function library can utilize typed information in order to make applications more effective and robust.

4.2. Functions and Path Languages

Our functions for accessing the Schema Component information of an XML Schema are written as standard XSL functions using `xsl:function`. This ensures maximum portability and does not require any additions or modifications to be made to available XSLT processors. In our current implementation, the functionality is focused on type information, and the implementation is based on DMR[3]-annotated schemas (see Section 3.3). However, this is no severe restriction, as these schemas can be generated by sole use of XSLT.

5 For a more detailed description, see chapter 9 in [12].

The function library uses two namespaces, mapped to the prefixes `scf` and `scfi` respectively. The latter is used for internal functions only intended to be used by the function library itself. Use of split namespaces introduces a certain amount of encapsulation known from Object-Oriented programming. A few example functions demonstrate what functions our library contains:

`scf:get-type( element() ) as element(scf:type)`

This basic function returns the type information of a particular instance node as an object of type `scf:type`. There is a function `scf:get-type-definition()` as well, which returns the node representing the type definition element from the Schema document, but this is deprecated and now only used for debugging purposes. This exactly reflects the difference between a unified data model access and this "some kind of pointer into the metadata", as it is expressed in Michael Kay's quote above. A mere pointer into the dark of a particular Schema-representation provides no abstraction from the (possibly changing) syntax or from its deficiencies. Even worse, the Schema data may be stored in a format inaccessible to XPath or it may be not at hand at all. A unified data model in contrast offers a neat and portable interface to what we might want to call a "type information item" in allusion to the XML Infoset.

`scf:derived-from(element( scf:type ), xs:QName) as xs:boolean`

This tests whether a type represented by a `scf:type` object is — directly or indirectly — derived from the global type with the qualified name specified in the second argument. The first argument is an object of type `scf:type`, because it may be an unknown type obtained by applying `scf:get-type` to any element from an actual instance.

`scf:instance-of(node(), xs:QName) as xs:boolean`

This is a generalization of XPath's `instance of` construct, covering user-defined complex types. The first argument is a node from an XML instance.

`scf:get-parent-type( element(scf:type) ) as element( scf:type)`

The function returns the type information item of the parent type in the type hierarchy.

`scfi:get-model-group( element(xs:complexType) ) as element() *`

This function is marked as an internal function because in our current implementation it is still dealing with type definition elements from the DMR[3]-annotated XML Schema. But it should give an impression of the intended functionality: It returns a

sequence of all elements that potentially can occur as particles of this complex type's model groups. This set of particles does not take into account the cardinalities and order constraints of the model groups anymore, but resolves all derivations. For this function to be as neat as `scf:get-type()`, we need to define an element information item analogously to the type information item.

Having such functions at hand, it is only a matter of syntax to turn them into path expression:

```
scf:derived-from($a, $b) ↔ $a/derived-from($b)
scf:get-parent-type($type) ↔ $type/parent-type::*[1]
```

In fact, the project context where we are developing the function library presented here is the creation of a path language to be used on XML Schema Components. However, this would require changes of the XPath syntax to be made. The function-style syntax in contrast can be used with standard technology.

4.3. The `scf:type` Object

The object returned by `scf:get-type()` referred to as type information item is what can be called an "XSLT object". The combination of `xsl:element`, `xsl:function` and XPath allows for application of Object-Oriented concepts. The following function for example can be seen as "constructor" of the `scf:type` "object":

```
<xsl:function name="scf:type" as="element(scf:type)">
  <xsl:param name="def" as="element(xs:complexType)" />
  <scf:type>
    <xsl:choose>
      <xsl:when test="$def/@name">
        <scf:name>
          <xsl:value-of select="$def/@name" />
        </scf:name>
      </xsl:when>
      <xsl:otherwise>
        <scf:scope>
          <xsl:sequence select="scfi:get-element($def/..)" />
        </scf:scope>
      </xsl:otherwise>
    </xsl:choose>
    <scf:tns>
      <xsl:value-of select="$def/@dmr:tns" />
    </scf:tns>
  ...
```

Called with a type definition element from a DMR[3]-annotated Schema, it returns an object representing the type information item. While this resembles to the idea of representing Schema Components in an XML syntax depicted in Section 3.2, the relationships to other information items can be expressed much more elegant by using `xsl:sequence` than using

ID/IDREF constructs. Using the former, non-tree graphs navigable using XPath can be constructed.

The task of designing these constructor function in fact is the task of defining a unified data model and its accessor functions. This clearly shows how and why building the function library and defining a unified data model are related and why the latter is a prerequisite for the former. The following Section shows an example, where use and access of the `scf:type` object will become more clear.

4.4. Use Cases

A brief example shall illustrate the use and benefits of even very basic functionalities. We assume a Schema with a target namespace mapped to the prefix `ex`) that defines a global named type `ex:baseType`, whose model group contains an element called `ex:nested`. A second type `ex:extType` is derived from it by extension, adding a second element called `ex:additional` to the model group. An element `ex:base` is declared with `type="ex:baseType"`. A sample instance may look like that:

```
<ex:root xmlns:ex="http://people.ee.ethz.ch/~femichel/namespaces/example-1">
  <ex:base>
    <ex:nested>ex:nested text</ex:nested>
  </ex:base>
</ex:root>
```

The value of effective block value of the element `ex:base` is the empty set, therefore type substitution can occur in instances ("at runtime"):

```
  <ex:base xsi:type="ex:extType">
    <ex:nested>ex:nested</ex:nested>
    <ex:additional>
      <ex:first>additional text</xe:first>
      <ex:second/>
    </ex:additional>
  </ex:base>
```

Assume a stylesheet processing this instance. Using the Schema Components Functions library it easily can determine the actual type of element `ex:base` at runtime and act accordingly. In this basic example, a child element of the extended derived type is only accessed if the instance actually uses this type. Furthermore, use of the function `scf:derived-from()` is demonstrated.

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:ex="http://people.ee.ethz.ch/~femichel/namespaces/example-1"
  xmlns:scf="http://people.ee.ethz.ch/.../schema-component-functions"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  <xsl:import href="SClib.xsl" />
  <xsl:variable name="DMR3" select="document('example-1.xsd')" />
  <xsl:template match="/">

    <!-- determine runtime type of ex:root/ex:base -->
```

```
      <xsl:variable name="runtimetype" select="scf:get-type(ex:root/ex:base)"
                    as="element(scf:type)" />
      <xsl:if test="$runtimetype/scf:name='extType' and
              $runtimetype/scf:tns='http://people.ee.ethz.ch/.../example-1'">

        <!-- print out content of ex:root/ex:base/ex:additional/ex:first -->
        <xsl:value-of select="ex:root/ex:base/ex:additional/ex:first" />
      </xsl:if>

      <!-- test derived-from(ex:extType, ex:baseType) -->
      <xsl:variable name="testtype" select="scf:get-type(ex:root/ex:base)"
                    as="element(scf:type)" />
      <xsl:variable name="basetype"
           select="QName('http://people.ee.ethz.ch/.../example-1','baseType')"
           as="xs:QName" />

      <xsl:value-of select="scf:derived-from($testtype, $basetype)" />
...
```

Note the simple access to the type's target namespace in the expression `$runtimetype/scf:tn`.[6] The fact that we used XML when modeling the `scf:type` object lets the access to its properties smoothly be integrated with the conventional XPath syntax. The document `'example-1.xsd'` read into the variable `$DMR3` is one of the annotated Schemas introduced in Section 3.3.

Already in this very limited example, the benefits are clearly visible. If we think of fields of application, where even less predictions can be made at compile-time, the advantages both in terms of performance and robustness become even more evident. Examples for such cases of use are *Web Services*, where XSLT-based technologies are heavily used and where versioning and maybe even dynamic generation of XML Schemas increasingly will necessitate Schema-inspection at runtime.

5. Conclusions

We clearly stated the need for a unified data model for XML Schema and a set of accessor functions to the data model. The present situation shows that the abstract data model defined in the Recommendation does not obviate such a definition. We discussed possible approaches and two XML syntaxes capturing XML Schema's data model. The objective was neither to redesign the abstract data model underlying to XML Schema, nor to design an ultimate syntax describing the data model. Our intermediate syntaxes are used as a testbed for developing our function library, and demonstrate opportunities and advantages for future technologies which might make stronger use of the rich possibilities of XML Schema's abstract data model.

---

6 Obtaining the target namespace can be cumbersome otherwise, especially if one is dealing with tree-fragments where `root()` does not provide the desired information from the `xs:schema` element anymore. Because the data model of SCF is closer to the Schema Components, the target namespace property is attached to each `scf:type` object rather than delegated to the Schema's root element.

We presented our function library that makes information from XML Schema's data model accessible in a neat way and that may serve as a foundation for a more sophisticated path language for XML Schema Components.

We advocate that both a unified abstract model meeting the requirements listed in Section 2.2, a set of powerful functions exhibiting the data model's structural information akin to the functions presented in Section 4.2, and a path language based on these two technologies will become indispensable with the spread of type-aware technologies like XSLT 2.0.

6. References

1. HENRY S. THOMPSON, DAVID BEECH, MURRAY MALONEY and NOAH MENDELSOHN: XML Schema Part 1: Structures, Second Edition. World Wide Web Consortium, Recommendation REC-xmlschema-1-20041028, October 2004
2. ANDERS BERGLUND, SCOTT BOAG, DON CHAMBERLIN, MARY F. FERNÁNDEZ, MICHAEL KAY, JONATHAN ROBIE and JÉRÔME SIMÉON: XML Path Language (XPath) 2.0. World Wide Web Consortium, Candidate Recommendation CR-xpath20-20060608, June 2006
3. MARY FERNÁNDEZ, ASHOK MALHOTRA, JONATHAN MARSH, MARTON NAGY and NORMAN WALSH: XQuery 1.0 and XPath 2.0 Data Model (XDM). World Wide Web Consortium, Candidate Recommendation CR-xpath-datamodel-20060711, July 2006
4. MICHAEL KAY: XSL Transformations (XSLT) Version 2.0. World Wide Web Consortium Candidate Recommendation CR-xslt20-20060608, June 2006
5. DENISE DRAPER, PETER FANKHAUSER, MARY FERNÁNDEZ, ASHOK MALHOTRA, KRISTOFFER ROS, MICHAEL RYS, JÉRÔME SIMÉON and PHILIP WADLER: XQuery 1.0 and XPath 2.0 Formal Semantics. World Wide Web Consortium, Candidate Recommendation CR-xquery-semantics-20060608, June 2006
6. JÉRÔME SIMÉON and PHILIP WADLER: The essence of XML. *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1-13, New Orleans, Louisiana, USA, 2003
7. PAUL V. BIRON and ASHOK MALHOTRA: XML Schema Part 2: Datatypes Second Edition, World Wide Web Consortium, Recommendation REC-xmlschema-2-20041028, October 2004
8. XML Schema for XML Schemas. http://www.w3.org/2001/XMLSchema.xsd, (last visited 2006-10-29)

9. ALLEN BROWN, MATTHEW FUCHS, JONATHAN ROBIE and PHILIP WADLER: XML Schema: Formal Description. World Wide Web Consortium, Working Draft WD-xmlschema-formal-20010925, September 2001

10. MARY HOLSTEGE and ASIR S. VEDAMUTHU: XML Schema: Component Designators. World Wide Web Consortium, Working Draft WD-xmlschema-ref-20050329, March 2005

11. JAMES CLARK and STEVE DEROSE: XML Path Language (XPath), Version 1.0. World Wide Web Consortium, Recommendation REC-xpath-1999111, November 1999

12. MICHAEL KAY: XPath 2.0 Programmers Reference. Wiley Publishing, Indianapolis, 2004

13. JOHN COWAN and RICHARD TOBIN: XML Information Set (Second Edition). World Wide Web Consortium, Recommendation REC-xml-infoset-20040204, February 2004

14. JOHN BOYER: Canonical XML Version 1.0. World Wide Web Consortium, Recommendation REC-xml-c14n-20010315, March 2001

15. DAVID W. EMBLEY and WAI Y. MOK: Developing XML Documents with Guaranteed " Good" Properties. *Proceedings of the 20th International Conference on Conceptual Modeling*, volume 2224 of *Lecture Notes in Computer Science*, 426-441, Yokohama, Japan, November, 2001. Springer Verlag.

16. BEN CHANG, ELENA LITANI, JOE KESSELMAN and REZAUR RAHMAN: Document Object Model (DOM) Level 3 Abstract Schemas Specification Version 1.0. World Wide Web Consortium Note NOTE-DOM-Level-3-AS-20020725, July 2002

17. XML Schema Infoset Model. http://www.eclipse.org/xsd/, (The XSD Javadoc includes UML diagrams of the data model), (last visited 2006-10-28)

18. XSOM. https://xsom.dev.java.net/, (last visited 2006-10-28)

19. XML Schema Object Model (SOM). http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconXSDSchemaObjectModelSOM.asp, (last visited 2006-10-28)

20. Castor XML Schema Support. http://www.castor.org/xmlschema.html, (last visited 2006-10-28)

21. TIM BRAY, DAVE HOLLANDER, ANDREW LAYMAN, and RICHARD TOBIN. Namespaces in XML 1.1. World Wide Web Consortium, Recommendation REC-xml-names11-20040204, February 2004.

22. HENRY S. THOMPSON: Names, Namespaces, XML Languages and XML Definition Languages. *Proceedings of the XML 2005 Conference*, Atlanta, Georgia, USA, November 2005. http://www.idealliance.org/proceedings/xml05/ship/82/XML_2005_82.HTML, (last visited 2006-10-29)

23. TIM BRAY, DAVE HOLLANDER and ANDREW LAYMAN: Appendix A.2 XML Namespace Partitions (non-normative; deleted in the current version [21]) of Namespaces in XML. World Wide Web Consortium Recommendation REC-xml-names-19990114, January 1999

24. KILIAN STILLHARD and ERIK WILDE: XML Schema Compact Syntax (XSCS) Version 1.0. *TIK Report No. 166*, Computer Engineering and Networks Laboratory, ETH Zürich, Zürich, Switzerland, March 2003