SPath: A Path Language for XML Schema

Erik Wilde (School of Information, UC Berkeley)
Felix Michel (ETH Zürich)

UCB iSchool Report 2007-001
February 2007

Available at http://dret.net/netdret/publications#wil07d

Abstract

While the information contained in XML documents can be accessed using numerous standards and technologies, accessing the information in an XML Schema currently is only possible using proprietary technologies. XML is increasingly being used as a typed data format, and therefore it becomes more important to gain access to the type system of an XML document class, which in many cases is an XML Schema. The *XML Schema Path Language (SPath)* presented in this paper provides access to XML Schema components by extending the well-known XPath language to also include the domain of XML Schemas. Using SPath, XML developers gain better access to XML Schemas and thus can more easily develop software which is type- or schema-aware, and thus more robust.

Contents

1 Introduction **2**

2 Problem **2**

3 Use Cases **3**

4 Navigating Schemas **4**

5 SPath Design **6**

6 Implementation Variants **11**

7 Implementation **12**

8 Related Work **13**

9 Conclusions **14**

1 Introduction

The *XML Path Language (XPath)* is one of the most successful specifications in the area of XML technologies. It defines an expression language for selecting parts of an XML document (XPath 1.0 [6]), and is currently being extended to a more powerful language (XPath 2.0 [1]). XPath 2.0 not only greatly extends the functionality of XPath, it also extends the underlying data model to not only be derived from a document, but rather from a document being validated and type-annotated by an *XML Schema* [3, 22]. Thus, XPath 2.0 becomes a *typed language* in the sense that it provides some functionality for working with the *typed content* of an XML document.

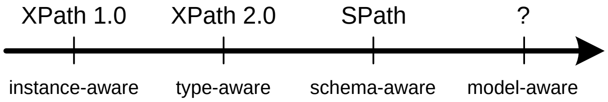

Figure 1: Scope of XPath Versions

Yet XPath 2.0 does not provide extensive functionality for accessing types in the context of the schema. While the structures of an XML document are represented by a tree of interconnected nodes (which can be navigated using one of XPath's most popular syntactic constructs, *location paths*), there is no such structure for types. Instead, types are identified by their *qualified names (QNames)*, and a rather small number of functions [18] is provided which work with these type identifiers. This makes it cumbersome to work with types with XPath 2.0, and makes XPath 2.0 type-aware, but not schema-aware (as shown in Figure 1[1]).

This paper introduces the *XML Schema Path Language (SPath)*, which builds on XPath 2.0 in several ways. It extends the data model to contain schema components as navigable structures, and it introduces new axes to navigate them, new node tests to work with them, and additional functions. The goal of SPath is to extend XPath to become a language which not only is well suited for working with XML documents, but also with XML Schemas. Our use cases (Section 3) suggest that this is useful when working with documents in a type-oriented environment, or working with schemas to perform tasks on schema structures.

In XPath 2.0, path expressions are /-separated sequences of expressions, where each expression other than the last must only return nodes. Each expression is repeatedly evaluated with each node of the resulting sequence of the previous expression as input. The results of all these invocations are then consolidated and passed as a sequence to the following step's expression. Because of this, function calls can be inserted into location paths, as long as they make use of the context, and generate node sequences. Our prototype (described in Section 7) is based on this by mapping SPath expressions to XPath expressions containing function calls.

2 Problem

The introduction of the *XML Information Set (Infoset)* [7] solved the problem which data model applications should use when working with XML documents. XPath 1.0, which is based on the Infoset, is one of the most successful technologies to provide access to XML structures. It is reused in various contexts, for example *XSL Transformations (XSLT)*, XML Schema, and the *Document Object Model (DOM)*. In all these cases, documents are considered to be a tree of document contents, and XPath provides access to that tree.

With the advent of XML Schema, the situation has become more complicated, because XML Schema turns XML documents into typed documents, with the type annotations being added by the validation

[1] The figure also includes a possible further development to make the underlying model available to XPath-based processing, but this issue it outside of the scope of this paper.

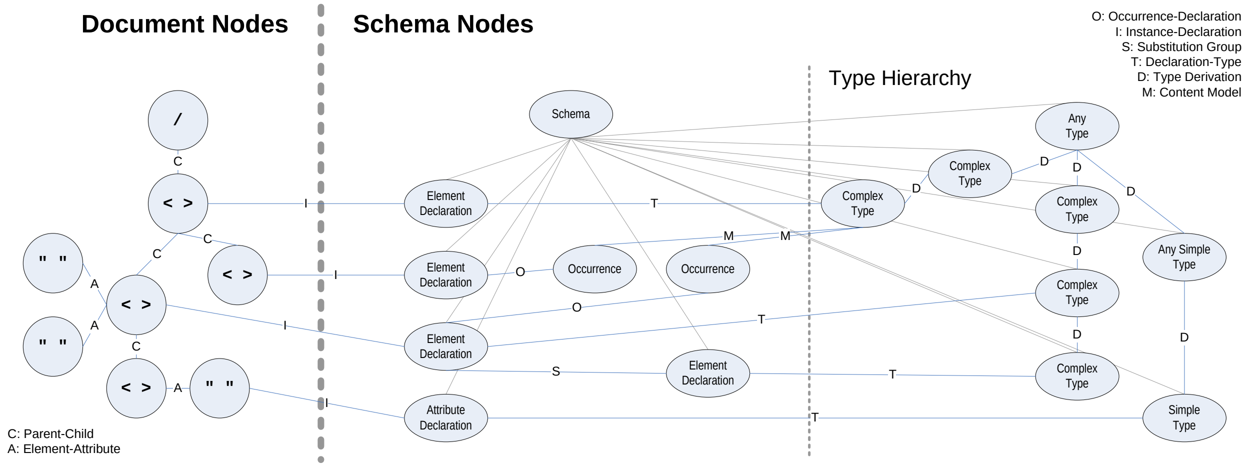

Figure 2: Document and Schema Abstract Models

process. XML Schema thus made XML more powerful and more complex, and XPath 1.0 was not sufficient anymore, because it is blind with respect to types. XPath 2.0 (which is developed in the larger context of the XSLT 2.0 and XQuery 1.0 activities) adds types to its data model, but only as an unstructured set of named items, and with little functionality to use them.

For a model which would allow users to navigate not only a document, but also the rich structure of schemas, a data model is required which represents types (and other schema components) as a structured set of nodes, and provides ways to navigate these structures in a way similar to XPath's location paths for document structures. Figure 2 shows how documents and schemas could co-exist in a data model representing both, and the relationships between these structures (including relationships crossing the boundary between documents and schemas). Section 3 discusses some examples where this is useful and opens new possibilities for using XML and schemas.

SPath treats schema structures as nodes, so that they can be used in the same way as XPath's document-oriented node kinds, most notably by being used in location paths. This requires novel navigational facilities (i.e., axes), because XPath's axes are designed to operate on document structures. SPath therefore not only extends XPath's data model, it also introduces new language constructs (axes, node tests, functions) to operate on these nodes.

Section 5 presents SPath's design, but this does not imply that this is the only way the language could have been designed. There were a lot of design choices, and some of them could not be reconciled easily. One example for this is the question whether navigation between both universes (instances and schemas) should be done by using axes, or implicitly. We decided to have explicit axes, because otherwise we would have had to change the semantics of existing XPath axes (which then would have to accept schema nodes as well as document nodes as input). As a result, the current language design is a starting point from which further developments in this area can continue.

3 Use Cases

While the extension of navigable structures from instance data to schemas may seem like a natural evolution in the light of a more type-oriented view of XML, there still is the question of the use cases of such an extension. Many of today's applications seem to work well with the purely instance-based model of XPath

1.0, and the type-annotated model of XPath 2.0 provides type information to type-aware applications. Possible use cases of SPath fall into two categories: applications where in addition to instance processing, inspection of the schema opens up new possibilities (Section 3.1), and applications focusing on schemas instead of being centered around instance processing (Section 3.2).

3.1 Instance-Guided Schema Access

A language providing schema access becomes particularly useful when the *compile time schema* (the schema against which code has been written) is different from the *runtime schema* (the schema against which an instance has been validated). A typical example for this is the field of Web service evolution, where a Web service is implemented against a certain schema, but should be robust enough to be able to deal with updated versions of the schema. While a number of design guidelines for schemas as well as for software developers can help in making code more robust in such an environment [10], code that is able to access updated schemas at runtime and explore the differences to the schemas available at compile time is more likely to succeed in such an environment. In programming terms, this approach enables *late binding* of XML instances to existing code with the added ability for *type introspection* (whereas XPath 2.0 only provides *type reflection*).

3.2 Schema-Oriented Applications

In addition to instance-oriented applications, schema access is especially useful in schema-oriented applications. These are applications which mostly work on schemas, for example to create code which can then be used to process instances of the schema. One example for this is the generation of XForms out of schemas [11]. In this work, XForms-based GUIs are generated out of schemas. XSLT is used to implement that transformation. However, the implementation is restricted to certain classes of schemas, because the processing of arbitrary schema documents in XSLT is very hard. Better support of schema access in XPath would have made the work much easier, and would have allowed to combine the best tool for XForm generation, XSLT, with good support for accessing schema information.

Another example for schema-oriented applications is *schema mapping* [17,20], which, given a set of related schemas, tries to (semi-)automatically generate a mapping of instances of one schema to the other. Schema mapping is a very important area, because in many XML-based scenarios, peers would like to cooperate, but since they do not use the same schema, there must be mapping code which transforms between schemas.

Another possible class of schema-oriented applications is that of schema checking: Because of the complexity of XML Schema, many user groups employ *XML Schema design guidelines* to encourage or enforce certain design patterns for XML Schemas. However, because of a lack of appropriate tools, it is hard to automatically test schemas against these guidelines. Using SPath, it would be easily possible to create a Schematron-like [14] language for defining rule-based assertions for XML Schemas.

4 Navigating Schemas

The most important aspect of SPath is that it makes XML Schemas available as navigable structures in XPath 2.0. To make this possible, there must be a data model for the information of a schema which should be accessible for navigation (Section 4.1), and there should be ways to move between the two universes of documents and schemas, so that navigation becomes possible across the whole range of available information (Section 4.2).

4.1 Schema Data Models

In the same way as the Infoset does not expose certain aspects of the XML document syntax, XML Schema data models hide certain aspects of the XML syntax of XML Schema. And while the definition of the Infoset has been a vividly discussed process of abstraction from an existing syntax, one could expect that in the case of XML Schema, this had not been an issue, because XML Schema [22] defines an *abstract data model*, of which the normative XML syntax is one possible representation. In practice, however, it becomes apparent that the presence of an abstract data model does not rule out other data models with specific perspectives and scopes. In fact, despite of the existence of the abstract data model, different data models for XML Schema are proposed and used today.

Post Schema Validation Infoset (PSVI) contributions [22] include properties that are specified to be *isomorphic* to the corresponding Schema components. This vagueness and the missing description of the relationships between these properties leads to differing data models in implementations: The *Xerces Native Interface (XNI)* for example introduces an additional layer of abstraction in terms of Schema nodes that cover both simple and complex types.

Data models for XML Schema like Eclipse's *XML Schema Infoset Model (XSD)*, Microsoft's *XML Schema Object Model (SOM)*, or Castor's *XML Schema Support* all implicitly introduce a data model for XML Schema that is shaped by the respective needs and priorities.

In a similar way as application interfaces have to make simplifications and adjustments, formal descriptions of a data model apply a particular perspective as well. For instance, *XML Schema Formal Semantics* [4] omit identity constraints, and *XDM Formal Semantics* [8], the formal description of the *XQuery 1.0 and XPath 2.0 Data Model (XDM)* [9], only considers the type system of XML Schema, because this is the most relevant part from the XDM perspective. The same document (in appendix D) defines how XML Schema components are to be imported into XDM. This basically is a definition of another data model of XML Schema from a certain perspective.

For utilizing a data model in practice, its components should be identifiable in order to be accessible. *XML Schema Component Designators* [12] propose a path syntax for identifying XML Schema components. Even though the objective is limited to identification and designation of components and the intended area of application therefore is different from SPath, integration of these designators into SPath will be considered depending on forthcoming draft versions.[2]

SPath also defines a data model which is a perspective of the abstract data model that is particularly suited for what we think are the essential areas of use of SPath. Yet the above examples demonstrate that there is no data model that fits all needs. Therefore we had to make design decisions connected to the design objectives and areas of application as well as to the syntax of SPath. Section 5.1 discusses our data model and these decisions. Section 7.1 presents the data model of the underlying function library of our prototype implementation that allows for different views on XML Schema's data model. The current disparity between SPath's data model and the function library's data model reflects the fact that, when developing SPath as a proposal which certainly will continue to evolve, we chose to introduce an implementation layer which makes it as easy as possible to evolve SPath without having to constantly rewrite the complete data model of SPath's implementation.

4.2 Integration of Schemas and Documents

One of the fundamental questions when introducing schema components as first-class citizens into XPath is how to integrate them with the existing data model, which is built around node kinds representing XML document structures. The SPath data model (described in Section 5.1) introduces new node kinds for schemas, which are then accessible for navigation by axes and functions. Figure 2 shows how this integration

[2] A function `get-by-scd()` could take such a designator and return the corresponding SPath node, which then could be used in SPath expressions.

makes documents as well as schemas accessible as nodes. Because our design assumes that a well-defined dividing line between these two classes of nodes is useful, we divide the SPath data model into two *universes*, one containing the established XPath node kinds, whereas the other contains the new node kinds of the SPath data model.

5 SPath Design

The primary goal of SPath's design is to remain as much within the limits of XPath's design principles and syntax as possible. This is easier said than done, because XPath is not a strict "design by rule" language, but has a lot of design decisions in it which instead are based on usability and utility. For example, the seemingly simple question, what an axis is, is not easy to answer. The most accurate answer probably is "anything that is likely to be used frequently as a way to explore relationships between nodes."

For SPath's design, the goal is to apply the design principles behind XPath to schemas, while still maintaining a dividing line between these two universes, so that they are perceived as separate, but interconnected. The following principles have been adopted from XPath, but have been extended to cover XML Schema structures as well:

- *Node Kinds:* XPath's data model supports seven node kinds which are based on the nodes found in an Infoset. Node kinds are important because they define which kind of structure can be represented and navigated as a graph. In XPath 2.0, PSVI type information is not represented as a node, it is represented as a QName.
- *Navigating Structures:* In XPath, structural information of XML documents is navigated using location paths, while additional information is accessible through functions. XPath 2.0 introduces types; however, these are only defined as QNames, and they are only accessible through functions. SPath turns schema structures into nodes and makes them navigable through location paths as well.
- *Axes:* Frequently used relationships between nodes should be navigable using axes, which provide a powerful and easily usable syntax for selecting nodes relevant to a given task.
- *Bidirectional Navigation:* Since navigation through axes provides access to relevant relationships between nodes, these relationships should be traversable bidirectionally. Like in XPath, this does not make sense for all axes, but for basic navigational tasks, bidirectional navigation should be possible.[3]
- *Node Tests as Predicate Shorthands:* Node tests are shorthand notations for predicates. They are useful because they allow the most frequently used predicates to be written in a more concise form. Node tests depend on the nodes that may be selected by an axis, so for the new node kinds introduced by SPath, there are new node tests as well.

Apart from these reused concepts, SPath introduces one new concept into XPath, which is the concept of *Axis Modifiers* (described in detail in Section 5.3). The origin of this concept is the fact that for certain kinds of navigational tasks, it is necessary to use the originating as well as the candidate node's context for selecting the nodes of an axis. Specifically, when limiting navigation of the type hierarchy to a specific derivation method, it is important to inspect the whole chain of type derivation between the context nodes of the step and candidate nodes. Thus, the derivation method must be part of the axis, because for node tests and predicates, the context of the step is no longer available and thus cannot be used to test for the

[3] Note, however, that this does not imply that bidirectional navigational directions always cancel out each other. For example, in XPath `parent::node()/child::node()` often is not equal to the context node itself, because it also selects all siblings.

derivation method. Any change to this would have fundamentally changed how XPath's process of evaluating steps is defined.

Axis modifiers apply to SPath axes only and do not conflict with the syntax of XPath, because they use hyphens to combine the axis name and the modifier. Thus, existing XPath syntax processing can be used for parsing SPath as well.

5.1 SPath Data Model

As described in Section 4.1, the question which data model to choose for a schema-aware technology is not an easy one. In the spectrum of possibilities, one end would be a data model isomorphic to the XML Schema components, possibly even extended to retain the structure of the schema documents. The other end is a highly customized model which is less faithful to XML Schema components, but more useful in terms of what users want to do with it.

SPath's data model is more on the side of the spectrum where full XML Schema isomorphism is not supported, with the idea being that some of the idiosyncrasies of XML Schema internals can then be hidden from users. Figure 4 shows the node kinds introduced by SPath, and these are added to XPath 2.0's existing set of node kinds (the seven node kinds derived from the Infoset's eleven information item types).

The node kinds are a different view of the structures defined by an XML Schema, instead of being a subset of XML Schema's components. In part, one goal of the data model is to unify the worlds of elements and attributes (inspired by the unified approach of RELAX NG [13]). Instead of replicating XML Schema's strong separation of these two concepts, `declaration`s represent element as well as attribute declarations. The same can be said about `occurrence`s, which represent element as well as attribute usages in `type`s. If SPath users wish to make the distinction between elements and attributes, they can use node tests (Section 5.4) for doing so.

Apparently, SPath has no direct representation of XML Schema's model groups, even though it exposes this information through axes which provide information about potential neighbors of a node. This means that the grammar information of a schema is preserved in SPath. However, it is not available in terms of how the grammar is defined, but rather in terms of what the language defined by that grammar is. Specifically, occurrences have the properties `optional` and `unbounded` and refer to a declaration, and XML Schema's content models are mapped to this alternative representation of the grammar. This maps the hierarchical structure of possibly nested model groups to an expanded sequence of occurrences.

This design has been chosen to better support the instance-oriented applications described in Section 3.1, but it may not be the ideal solution for some of the purely schema-oriented applications described in Section 3.2. Again, as described in Section 4.1, data models are determined by assumptions about possible use cases, and since SPath focuses on integrating schema information with instance information, the design approach was to put the focus instance-oriented applications.

5.2 SPath Axes

SPath's main contribution is the introduction of new axes, which can be used to navigate the schema structures represented by SPath's new node kinds. In the same way as XPath axes never produce run time errors when being applied to a node, SPath axes also can be used starting from any node kind. This means that the "In" node kinds shown in Figure 3 are not the only node kinds for which an axis is permitted, but the only ones for which an axis can yield a non-empty result. Similarly, standard XPath axes being applied to SPath nodes will always return an empty sequence.

Some axes accept node kinds from both universes (for example the `type` axis, which returns the type of an element or attribute in an instance, or of a declaration or occurrence in a schema), but all of them always return node kinds only belonging to one of the universes. With one exception, they also always return only one node kind from the schema universe. The exception is the `substituted-by`/`substitutes` axis pair,

Axis Name	In	Out	Semantics
`type`	e a d o s	t	Returns the nodes' associated type. For document nodes, the type annotation is used, for schema nodes the declared type.
`declaration`	e a o c s t	d	Returns the declarations to which the node refers.
`instance`	d o t	e a	Returns all nodes in type-annotated documents which are of that type or use that declaration (or the declaration used by an occurrence).
`occurrence`	e a d s	o	Returns the occurrences of the given context.
`basetype`	e a d o t	t	Returns the direct base type of a type, or of a type inferred from a non-type node (supported node kinds are elements and attributes in instances, and declarations and occurrences in schemas). Returns `empty()` for untyped elements or attributes.
`supertype`	e a d o t	t	Recursive version of `basetype`, returning the types in reverse derivation order.
`supertype-or-self`	e a d o t	t	Same axis as `supertype`, but also includes the context node.
`derivedtype`	e a d o t	t	Returns derived type(s) of a type, or of a type inferred from a non-type node (supported node kinds are elements and attributes in instances, and declarations and occurrences in schemas). Returns `empty()` for untyped elements or attributes.
`subtype`	e a d o t	t	Recursive version of `derivedtype`, returning the types in derivation order.
`subtype-or-self`	e a d o t	t	Same axis as `subtype`, but also includes the context node.
`contains`	t	o	Returns a set of all occurrences (elements and attributes) within types.
`followed-by`	e o	o	Returns a set of element occurrences, indicating which elements are possible `following-sibling`s of the context (which can be a document or a schema node).
`preceded-by`	e o	o	Returns a set of element occurrences, indicating which elements are possible `preceding-sibling`s of the context (which can be a document or a schema node).
`substituted-by`	e d o t	d t	Based on document or schema nodes, this axis returns the elements and/or types which can be substituted by the given elements or types.
`substitutes`	e d o t	d t	Based on document or schema nodes, this axis returns the elements and/or types which can substitute the given elements or types.
`constraint`	e d o s	c	Selects `constraint` nodes and selects all constraints which are defined for the supplied context.
`constrained-by`	e a o	c	Selects the constraints which select the node through their selector.
`refer`	c	c	Selects the keys used by key references.
`referred-by`	c	c	Selects the key references referred to by keys.
`schema`	e a d t c	s	For schema universe nodes, this axis returns the schema node of the containing schema. For instance universe nodes, it returns the schema node of the schema which has validated this node.

Figure 3: SPath Axes ("In" and "Out" use shorthand notations for the supported node kinds: element, attribute, declaration, occurrence, type, constraint, schema; see Section 5.1 for a description.)

`schema`	Represents the schema as a whole, in particular it does not represent schema documents, but the whole schema assembled from potentially multiple schema documents.
`type`	Represents types, which in XML Schema can be simple or complex types.
`declaration`	Represents declarations, which in XML Schema can be element or attribute declarations.
`occurrence`	Represents occurrences of declarations in types, summarizes XML Schema's concepts of *particles* (referring to element declarations) and *attribute uses*.
`constraint`	Represents an identity constraint, which is always associated with an element declaration.

Figure 4: SPath Node Kinds

which returns both declarations and types. The explanation for this is that the type substitution mechanism itself mixes the two universes by allowing essentially the same type-perspective effect through the different mechanisms of substitution groups (substituting elements) and type substitution (substituting types).

As in XPath, axes navigating through hierarchical structures (i.e., the type hierarchy) have various variations, providing functionality for single step navigation (`basetype`/`derivedtype`), recursive navigation (`supertype`/`subtype`), and recursive navigation including the context node (`supertype-or-self`/`subtype-or-self`).

5.3 SPath Axis Modifiers

Axis modifiers provide more specific navigational facilities than the pure axes. Their syntax is defined to not violate the basic XPath syntax. This means that axis modifiers are appended to axis names, using a hyphen as separator. It is possible to append several modifiers to an axis. Axis modifiers are only applicable to the axes navigating the type hierarchy (`basetype`, `derivedtype`, `supertype`, `subtype`, `supertype-or-self`, and `subtype-or-self`).

The available axis modifiers are `restriction`, `extension`, `list`, and `union`, and they are used to limit the navigation to all types being derived using the specified mechanism. If no axis modifier is specified, the default is to not limit the navigation to a specific derivation mechanism. For example, starting from a type, the SPath `subtype-restriction::*` will select all types being directly or indirectly derived by restricting the context type, limiting the search through the type hierarchy to this derivation method only. Specifically, the SPath will *not* select all subtypes which are derived by restriction from their base type (this is supported by a node test), it will only select those which are derived by *restriction only* from the context node of the SPath.

5.4 SPath Node Tests

Node tests in XPath can either be *name tests* or *kind tests*. SPath follows this principle. As in XPath, name tests in SPath can be either a colon-separated combination of a prefix or wildcard and a local-name or wildcard, or a single wildcard (`*`). Since the data model of XML Schema (and thus the data model of SPath) contains *unnamed* nodes, the semantics of the wildcard have been extended to select unnamed nodes as well.

XPath defines *kind tests* which cover all *node kinds* encountered in XPath. Likewise, SPath provides a set of *kind tests* for each of the SPath node kinds that are described in Section 5.1. And as it is the case for some of the kind tests in XPath, all of SPath's kind tests have the appearance of functions accepting a varying number of arguments for narrowing the set of nodes matched.

First of all, all kind tests can occur without any arguments, testing only the kind of nodes without restricting the result set: `type()`, `declaration()`, `occurrence()`, `constraint()`, and `schema()`.

The kind tests for nodes that can be *named* (i.e., `type()`, `declaration()`, and `constraint()`) accept a first argument that can either be a QName or a wildcard (`*`). If the wildcard is specified, *anonymous* nodes are returned as well.

Additionally, the kind test accept the following optional arguments that can be one of the strings that are shown in the list:

- `type(name-or-wildcard, category, derived-by)`

`category`	`'simple'`, `'simple-atomic'`, `'simple-union'`, `'simple-list'`, `'complex'`
`derived-by`	`'extension'`, `'restriction'`

- `declaration(name-or-wildcard, category, scope)`

`category`	`'element'`, `'attribute'`
`scope`	`'local'`, `'global'`

- `constraint(name-or-wildcard, category)`

`category`	`'key'`, `'keyref'`, `'unique')`

- `occurrence(category)`

`category`	`'element'`, `'attribute'`

- `schema(namespace)`

`namespace`	The schema's target namespace

The properties of all nodes can be further examined using the functions described in Section 5.6.

5.5 SPath Predicates

SPath does not change anything about the semantics of XPath predicates, which means that they are evaluated in the usual way: for each item in the sequence produced by the expression preceding the predicate list, each predicate is evaluated with this item as the context, and only if all predicates evaluate to true, the item remains in the final result sequence of the step.

This means that predicates in SPath expressions can use the full set of XPath expressions as predicates. An important task of predicates, however, is the filtering of nodes based on certain criteria which in many cases are specific to the node kind. Because SPath defines a number of new node kinds, it also defines a number of functions which allow the filtering of these node kinds in predicates. Section 5.6 describes the functions available in SPath.

5.6 SPath Functions

A wide range of the functionality required for working with SPath is provided by the SPath axes and their modifiers described in Sections 5.2 and 5.3, and by the node tests introduced in Section 5.4. The extent to which such functionality should be covered by functions or rather by dedicated syntax constructs like axes, is a question of language design. Rather than enumerating all functions defined by SPath, we explain our rationale and show two different *categories* of SPath functions.

SPath follows the principle of defining functions only for information that is either a literal property (rather than a structural) or where the function requires more or different arguments other than the context node. Examples for the latter case are the functions shown in Figure 5, which could not be expressed as axes without changing XPath's axis syntax. Examples for the former case are functions returning node properties like *name*, *namespace URI*, *derivation control*, or *selector* and *field* for `constraint()` nodes. It is obvious

`constrains(constraint as constraint(), context as occurrence()) as occurrence()*`	Returns all occurrences that are constrained by `constraint` when evaluated in the context of `occurrence`.[4]
`constrains(constraint as constraint(), context as node()) as node()*`	The same as above, but in the instance universe.

Figure 5: SPath Functions for Constraints

from this enumeration that such functions most often are in direct correspondence to Schema components and thus (speaking in terms of our prototype implementation) to the second category of functions described in Section 7.2.

Since many functions like `name()` or `namespace-uri()` are semantically equivalent to the corresponding XPath functions, the respective XPath functions are extended to polymorphic functions accepting nodes from both universes.

6 Implementation Variants

SPath is a proposal for how XPath could evolve from a type-aware to a schema-aware language. The approach for implementing the language for experimentation took into account the fact that the language will evolve for some time before settling into a mature (and maybe even standardized) language. The following implementation variants were considered, and in order to keep the language as open for changes and interested users as possible, the prototype implementation described in Section 7 uses the implementation variant described in Section 6.3.

6.1 Extending the Language

The most efficient and most elegant solution would be to integrate SPath into the language itself. This way, SPath would become an integral part of the language and be available in every processor. However, this is not realistic in the short term, and also is not a good strategy for experimenting with SPath, because it makes it too hard to change the language.

6.2 Extending the Processor

An easier way would be to extend a processor so that it supports SPath evaluation, for example by using a special extension function of the processor. This would be better for prototyping than the previously described solution, but would limit the language to a single processor and thus make it harder to get as many prototype users as possible.

6.3 XSLT-Based Extension Functions

Since we believe that SPath is a useful prototype, but not the language that eventually may become a widely accepted standard, we built our software to be as easily usable as possible. Our prototype implementation is not limited to a special processor. Our approach pre-processes a stylesheet and substitutes SPath expressions with function calls to a library. This library is written in XSLT. Thus, after pre-processing a stylesheet using SPath, it can be run on any XSLT 2.0 processor.

[4] The second argument is necessary, because in XML Schema, the *selector* XPath is evaluated relative to a given context.

7 Implementation

The prototype implementation relies on a function library written in plain XSLT 2.0 which has been developed from a basic function library [19]. The function library does neither require a *Schema-aware* XSLT processor, nor any extensions or modifications to be made to the processor. It retrieves the Schema information from the *XML Schema for XML Schemas* and from the XML Schema documents representing the XML Schema and that has been written in the normative XML syntax. It then represents the information of the *assembled* XML Schema using a data model that consists of what can be called *XSLT objects*. These objects represent a subset of the Schema components from XML Schema's abstract data model [22] as well as the node kinds from SPath's data model as it is introduced in Section 5.1, and they make the properties and relationships easily accessible through XPath.

We consider our function library to be a useful tool kit for experimenting with SPath expressions, for developing its syntax, and for identifying use cases for schema-aware applications. Furthermore, the functions available in the library are more modular and fine-grained than the expressions and functions of SPath, and can be used for any application aiming for easy access to schema information. In fact, the expressions found in SPath are only shorthand notations for common combinations of function calls, additionally providing a more powerful syntax. Bearing in mind the considerations from Section 6, our main objectives are high portability and use of open standards, instead of utmost performance or robustness.

7.1 Data Model Entities

In order to process the information from XML Schemas, first of all a representation of the entities of which the data model consists has to be provided. The Infoset calls these entities *Information Items*, the abstract data model of XML Schema is composed of *Schema Components*, and SPath makes these entities accessible and navigable as *nodes*. As described in Section 4.1, our function library provides a set of structures that cover both a large subset of the XML schema components, as well as the node kinds of SPath as described in Section 5.1. Since the library is written in XSLT, and since XML is a format for structured data, it is an obvious choice to use XML for representing these structures. Therefore the library contains XSLT functions that return temporary XML trees that contain the information of the respective data model entities. The contents of these XML structures can easily be accessed through XPath, and with the type support of XSLT / XPath 2.0, they can be used as structured types, and functions accepting only those types can be written. Alluding to terms known from *Object Orientation*, one may want to refer to these structures as *objects*, to the sub-elements as *properties*, to the former functions as *constructors*, and to the latter as *methods*.

7.2 Functions

Our library provides a wide set of XSLT functions accessible in a namespace that usually is mapped to the prefix `scf` (for *schema component functions*). Instead of enumerating all functions, the following list describes five categories into which the available functions can be divided:

1. *Constructor functions* returning the XSLT objects described above.

2. Functions representing *properties of the data model nodes*. The property {`type definition`} of the *Element Declaration* Schema component is represented by a function `scf:type-definition()` that takes an element declaration object and returns a type definition object. The polymorphism described in Section 5.6 is implemented using wrapper functions that disambiguate the kind of node and perform the appropriate action.

3. *Axis-supporting functions*: For example, the `scf:get-supertypes()` function backs the `supertype` axis. However, the functions generally are more modular than the SPath expressions, letting the

design of SPath control the level of granularity and degree of implicitness (e.g., whether a type axis should resolve all substitutions or not).

4. Functions that provide for *operators and language constructs* that are needed for comparing and testing the XSLT objects. For example, `scf:type-equal()` tests equality of two `scf:type` objects, and `scf:instance-of()` is required in order to extend XPath's `instance of` construct to cover complex types as well.

5. *Internal functions* that are not accessible to applications. Encapsulation of these functions is achieved by declaring internal functions in a different namespace. Internal functions include functions concerned with particularities of XML Schema's XML syntax, or helper functions (e.g., easing conversion of QNames).

7.3 SPath Mapping

As described in Section 6.3, the current prototype is based on the approach to textually substitute SPath expressions in an XSLT stylesheet with function calls to an XSLT library implementing SPath. Instead of parsing all XPath expressions into a parse tree, we use XPath's and XSLT's regular expression processing facilities, which are sufficient to implement some pattern-based search and replace heuristics, but have inherent limitations. For small experiments, the current implementation is sufficient, but it does not work for all possible SPath/XPath expressions. For a more robust implementation, we are currently considering either an XSLT-based parser, or a non-XSLT implementation of the mapping process using available parser generation tools such as JavaCC.

7.4 Example Mapping

Figure 6 (on page 16) shows an XSLT stylesheet that uses SPath, the intermediate stylesheet after preprocessing, and the output of the stylesheet when run with a simple document and schema (shown in Figure 7 on page 17) as inputs. The parts of the initial stylesheet which are relevant for SPath as well as the corresponding mapped counterparts in the generated XSLT are highlighted.

As one example of the mapping, the `basetype-extension::*` step of the SPath expression is mapped to `scf:base-type-modified(., 'extension')` in the resulting XPath. The following rules have driven that mapping:

- *Axes* are mapped to the functions which are backing these axes (as described in Section 7.2) in the function library. The `scf:base-type-modified()` function is backing any `basetype` axis step using an axis modifier. The modifier(s) of the axis step are mapped to an argument of the function (all axis functions have the current context as first argument).

- The *node test* of the axis is mapped to a predicate, but since the node test does not define any filter, no such predicate is generated in the example mapping.

The sample schema defines a small type hierarchy using type extension and restrictions, and the output shows the result of the `demo:type-hierarchy` function being called with the sample instance's document element as input.

8 Related Work

Work related to the path-based navigation of XML Schemas can be found in different areas. The fundamental question of XML Schema data models as discussed in Section 4.1 is one of these areas. The (to our knowledge abandoned) attempt to make XML Schema accessible through the DOM [5], and the interfaces of specific parsers [16] is another area which is related to the work presented in this paper. While all of this work is relevant to SPath, all of it only has overlaps with SPath, leaving out some of the issues relevant to our work, while including others which are not significant for our work. To our knowledge, so far there have been no attempts to expand XPath to cover full navigational access to XML Schemas in a way directly comparable to SPath.

9 Conclusions

The main contribution of SPath to the evolving landscape of XML technologies is the integration of schemas into the data model of XPath. Additionally, SPath's expressive syntax allows the easy navigation of the complex structure defined by an XML Schema. SPath supports applications which process XML data in a type- and schema-aware way, rather than treating XML data as untyped character data. Using SPath's navigational features, applications can explore schemas at runtime and thus be programmed in a way which better supports loose coupling scenarios of XML-oriented software components.

Recent surveys of XML Schema usage [2,15] suggest that currently most schemas are simple in structure and thus processing them does not require a language as expressive as SPath. We believe, however, that the growing stack of XML technologies and its ongoing maturing into an integrated part of software design and development makes more robust ways of handling XML inevitable, and that typed XML and well-designed type systems (i.e., XML Schemas) will thus become more important.

In particular, for loosely coupled system design such as publicized by the *Service-Oriented Architectures (SOA)* approach, it will be become necessary to focus more on independent versioning of individual components of such an architecture. Such a view will require approaches comparable to the late binding principle of object-oriented design, and type reflection and introspection will be important foundations of such a development.

References

[1] Anders Berglund, Scott Boag, Donald D. Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML Path Language (XPath) 2.0. World Wide Web Consortium, Recommendation REC-xpath20-20070123, January 2007.

[2] Geert Jan Bex, Wim Martens, Frank Neven, and Thomas Schwentick. Expressiveness of XSDs: From Practice to Theory, There and Back Again. In *Proceedings of the 14th International World Wide Web Conference*, pages 712–721, Chiba, Japan, May 2005. ACM Press.

[3] Paul V. Biron and Ashok Malhotra. XML Schema Part 2: Datatypes Second Edition. World Wide Web Consortium, Recommendation REC-xmlschema-2-20041028, October 2004.

[4] 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.

[5] Ben Chang, Elena Litani, Joe Kesselman, and Rezaur Rahman. Document Object Model (DOM) Level 3 Abstract Schemas Specification. World Wide Web Consortium, Note NOTE-DOM-Level-3-AS-20020725, July 2002.

[6] James Clark and Steven J. DeRose. XML Path Language (XPath) Version 1.0. World Wide Web Consortium, Recommendation REC-xpath-19991116, November 1999.

[7] John Cowan and Richard Tobin. XML Information Set (Second Edition). World Wide Web Consortium, Recommendation REC-xml-infoset-20040204, February 2004.

[8] Denise Draper, Peter Fankhauser, Mary F. Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. World Wide Web Consortium, Recommendation REC-xquery-semantics-20070123, January 2007.

[9] Mary F. Fernández, Ashok Malhotra, Jonathan Marsh, Marton Nagy, and Norman Walsh. XQuery 1.0 and XPath 2.0 Data Model (XDM). World Wide Web Consortium, Recommendation REC-xpath-datamodel-20070123, January 2007.

[10] Adam FitzGerald. Best Practices for XML Schema Evolution in Application Development. In Proceedings of XML 2005 [21].

[11] Patrick Garvey and Bill French. Generating User Interfaces from Composite Schemas. In *Proceedings of XML 2003*, Philadelphia, Pennsylvania, December 2003.

[12] Mary Holstege and Asir S. Vedamuthu. XML Schema: Component Designators. World Wide Web Consortium, Working Draft WD-xmlschema-ref-20050329, March 2005.

[13] International Organization for Standardization. Information Technology — Document Schema Definition Languages (DSDL) — Part 2: Grammar-based Validation — RELAX NG. ISO/IEC 19757-2, November 2003.

[14] International Organization for Standardization. Information Technology — Document Schema Definition Languages (DSDL) — Part 3: Rule-based Validation — Schematron. ISO/IEC 19757-3, April 2006.

[15] Ralf Lämmel, Stan Kitsis, and Dave Remy. Analysis of XML Schema Usage. In Proceedings of XML 2005 [21].

[16] Elena Litani and Lisa Martin. An API to Query XML Schema Components and the PSVI. In *Proceedings of XML Europe 2004*, Amsterdam, Netherlands, April 2004.

[17] Jayant Madhavan, Philip A. Bernstein, and Erhard Rahm. Generic Schema Matching with Cupid. Technical report, Microsoft Corporation, Redmond, Washington, August 2001.

[18] Ashok Malhotra, Jim Melton, and Norman Walsh. XQuery 1.0 and XPath 2.0 Functions and Operators. World Wide Web Consortium, Recommendation REC-xpath-functions-20070123, January 2007.

[19] Felix Michel. Opening XML Schema's Data Model to XPath 2.0. Technical Report TIK Report No. 264, Computer Engineering and Networks Laboratory, ETH Zürich, Zürich, Switzerland, November 2006.

[20] René J. Miller, Mauricio A. Hernández, Laura M. Haas, Lingling Yan, C. T. Howard Ho, Ronald Fagin, and Lucian Popa. The Clio Project: Managing Heterogeneity. *ACM SIGMOD Record*, 30(1):78–83, March 2001.

[21] *Proceedings of XML 2005*, Atlanta, Georgia, November 2005.

[22] 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.

XSLT 2.0 using SPath:

```
<xsl:stylesheet version="2.0" xmlns:xsl="..." xmlns:demo="...">
 <xsl:import-schema namespace="http://dret.net/www2007example" schema-location="sample.xsd"/>
 <xsl:function name="demo:type-hierarchy">
  <xsl:param name="element" as="element()"/>
  <xsl:value-of select="name($element)"/>
  <xsl:text> &#x2208; </xsl:text>
  <xsl:value-of select="demo:type-and-decls($element/type::*)"/>
 </xsl:function>
 <xsl:function name="demo:type-and-decls">
  <xsl:param name="type" as="type()"/>
  <xsl:value-of select="name($type)"/>
  <xsl:if test="$type ne anyType()">
   <xsl:variable name="decls" select="$type/declaration::*" as="declaration()*"/>
   <xsl:value-of select="if ($decls) then
                         string-join((’(’, for $i in $decls return name($i), ’)’), ’ ’) else ()"/>
   <xsl:text>&#xa; </xsl:text>
   <xsl:value-of select="if ($type/basetype-extension::*) then ’ extends ’ else ’ restricts ’"/>
   <xsl:value-of select="demo:type-and-decls($type/basetype::*)"/>
  </xsl:if>
 </xsl:function>
</xsl:stylesheet>
```

SPath Expressions translated to Function Library:

```
<xsl:stylesheet version="2.0" xmlns:xsl="..." xmlns:xs="..." xmlns:demo="..." xmlns:scf="...">
 <xsl:import href="SClib.xsl"/>
 <xsl:variable name="SCFS" select="doc(’sample.xsd’)"/>
 <xsl:function name="demo:type-hierarchy">
  <xsl:param name="element" as="element()"/>
  <xsl:value-of select="scf:name($element)"/>
  <xsl:text> &#x2208; </xsl:text>
  <xsl:value-of select="demo:type-and-decls($element/scf:get-type(.))"/>
 </xsl:function>
 <xsl:function name="demo:type-and-decls">
  <xsl:param name="type" as="element(scf:type)"/>
  <xsl:value-of select="scf:name($type)"/>
  <xsl:if test="scf:type-not-equal($type, scf:anyType())">
   <xsl:variable name="decls" select="$type/scf:type-definition-inverse(.)" as="element(scf:element)*"/>
   <xsl:value-of select="if ($decls) then
                         string-join((’ (’, for $i in $decls return scf:name($i), ’)’), ’ ’) else ()"/>
   <xsl:text>&#xa; </xsl:text>
   <xsl:value-of select="if ($type/scf:base-type-modified(., ’extension’))
                         then ’ extends ’ else ’ restricts ’"/>
   <xsl:value-of select="demo:type-and-decls($type/scf:base-type-definition(.))"/>
  </xsl:if>
 </xsl:function>
</xsl:stylesheet>
```

Processing Results:

```
USwestCoastBusinessAddress ∈ USwestCoastBusinessAddressType ( USwestCoastBusinessAddress )
 restricts USbusinessAddressType ( USbusinessAddress )
 extends USaddressType ( USaddress)
 extends addressType ( address europeanAddress )
 restricts anyType
```

Figure 6: Sample XSLT, Mapping, and Results

Sample XML Schema:

```
<xs:schema xmlns:xs="..." targetNamespace="http://dret.net/www2007example" xmlns="...">
 <xs:element name="address" type="addressType"/>
 <xs:element name="europeanAddress" type="addressType"/>
 <xs:element name="USaddress" type="USaddressType"/>
 <xs:element name="USbusinessAddress" type="USbusinessAddressType"/>
 <xs:element name="USwestCoastBusinessAddress" type="USwestCoastBusinessAddressType"/>
 <xs:complexType name="addressType">
  <xs:sequence>
   <xs:element name="name" type="xs:string"/>
  </xs:sequence>
 </xs:complexType>
 <xs:complexType name="USaddressType">
  <xs:complexContent>
   <xs:extension base="addressType">
    <xs:sequence>
     <xs:element name="state" type="stateType"/>
    </xs:sequence>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>
 <xs:complexType name="USbusinessAddressType">
  <xs:complexContent>
   <xs:extension base="USaddressType">
    <xs:sequence>
     <xs:element name="companyname" type="xs:string"/>
    </xs:sequence>
   </xs:extension>
  </xs:complexContent>
 </xs:complexType>
 <xs:complexType name="USwestCoastBusinessAddressType">
  <xs:complexContent>
   <xs:restriction base="USbusinessAddressType">
    <xs:sequence>
     <xs:element name="name" type="xs:string"/>
     <xs:element name="state" type="westCoastStateType"/>
     <xs:element name="companyname" type="xs:string"/>
    </xs:sequence>
   </xs:restriction>
  </xs:complexContent>
 </xs:complexType>
 <xs:simpleType name="stateType">
  <xs:restriction base="xs:token">
   <xs:length value="2"/>
  </xs:restriction>
 </xs:simpleType>
 <xs:simpleType name="westCoastStateType">
  <xs:restriction base="stateType"/>
 </xs:simpleType>
</xs:schema>
```

Sample XML Document:

```
<USwestCoastBusinessAddress xmlns="...">
    <name>Erik Wilde</name>
    <state>CA</state>
    <companyname>UC Berkeley</companyname>
</USwestCoastBusinessAddress>
```

Figure 7: Sample Schema and Instance