Location trees enable XSD based tool development

Hans-Jürgen Rennau, parsQube GmbH
<hans-juergen.rennau@parsqube.de>

Abstract

Conventional use of XSD documents is mostly limited to validation, documentation and the generation of data bindings. The possibility of additional uses is little considered. This is probably due to the difficulty of processing XSD, caused by its arcane graph structure. An effective solution might be a generic transformation of XSD documents into a tree-structured representation, capturing the model contents in a transformation-friendly way. Such a tree-structured schema derivative is offered by location trees, a format defined in this paper and generated by an open-source tool. The intended use of location trees is an intermediate to be transformed into interesting artifacts. Using a chemical image, location trees can play the role of a catalyst, dramatically lowering the activation energy required to transform XSD into valuable substances. Apart from this capability, location trees are composed of a novel kind of model components inviting the attachment of metadata. The resulting metadata trees enable innovative tools, including source code generators. A few examples illustrate the new possibilities, tentatively summarized as XSD based tool development.

Table of Contents

Introduction

Well-written XML schema documents (XSD) contain a wealth of information. Its potential value is by no means exhausted by conventional schema uses, which are validation, documentation and data binding (e.g. [3]). Why are there hardly any tools available for unlocking the treasure? A major reason is probably the sheer difficulty to write code which evaluates XSD documents reliably - coping with the complete range of the XSD language, rather than being limited to XSD documents written in a particular style.

The difficulty of processing XSD has a key reason: the semantics of XSD are graph-structured, not tree-structured like the instance documents which XSD describes. To realize this, think of the

many relationships between schema components, like uses-type, extends-type, restricts-type, uses-group, uses-attributegroup, memberof-substitutiongroup. A clear perception of the problem suggests a straightforward solution: we need a generic and freely accessible transformation of XSD into a tree-structured equivalent. This can be used as a *processing-friendly intermediate* which is readily transformed into useful artifacts. The new intermediate should play a role similar to a catalyst in chemical processes: it is neither input nor output, but it reduces the energy required to turn input into output. Availability of this catalyst might turn the development of XSD processing tools into a fascinating and rewarding challenge.

Departing from an analysis of the main problems besetting XSD processing, this paper proposes a solution based on a tree-structured representation of XSD contents. The new format is explained in detail, and the new possibilities of XSD processing are illustrated by several examples.

Problem definition

XSDs describe the structure and constrain the contents of XML documents ([5], [6], [7], [8]). These documents are tree-structured. The schema itself, however, is a peculiar mixture of tree and graph structure. It is a graph whose nodes are trees (e.g. type definitions) as well as nodes within trees (e.g. element declarations within type definitions). The edges of this graph connect trees (e.g. a type references its base type) as well as tree nodes and trees (e.g. an element references its type definition). The data model of XSD is justified by the conflicting needs to describe tree structure and to support reusable components. (If not for the second, XSD would probably consist of straightforward representations of the desired tree structures!) Nevertheless, the model as it is creates several problems which deserve attention. Ignoring the problems, we would limit ourselves to the services of mature and completed tools. Thus we would miss the chance to discover innovative uses of the precious information content locked up in XSD documents.

The problem of understanding

To understand a schema means, in the first place, to understand the tree structure of its instance documents. However, this is often difficult - or even virtually impossible - when studying the schema contents in their raw form.

Fortunately, several commercial IDEs offer graphical representation of schemas. At first sight, they give us everything we need in order to understand the schema: the graphical representation is very clear and intuitive. But when working with large schemas, a serious shortcoming becomes obvious. The graphical representation is very space consuming, rendering fast browsing of larger chunks of the schema impossible. The screen is completely filled by a small number of tree nodes; a *sliding* glance at more than a small piece of a large whole requires a "bumpy" navigation involving much scrolling mixed with a series of clicks for alternately collapsing and expanding nodes.

The query problem

A second problem is closely related to the problem of understanding. Managing a large schema requires more than clear pictures of pieces of tree structure: it requires *querying* the schema. Example questions: Which data paths will be affected if I change this element declaration? (There may be hundreds of them, but also none at all!) Do all `<Foo>` elements have the same type? What are the elements on which a @Bar attribute may appear? Compared to the previous schema version, what data paths have been added or removed? If the schema is large, the number of tree nodes is too large for finding the answers by visual inspection. The IDEs give us icon trees, but no data trees, no data sets for queries which give us the answers.

The transformation problem

Schemas describe the tree structure of instance documents, and in doing so they define crucial information about relevant domain entities. Questions arise: What are the key entities, what are their

properties, what are the data types of those properties, what are their cardinalities? We should have tools to reorganize such information into comprehensive and intuitively structured representations. But the required schema transformation is too difficult for a developer who is not a schema specialist.

The metadata problem

XML schema components can be associated with annotations. Special schema elements are available for this purpose (`xs:annotation`, `xs:documentation`, `xs:appinfo`). The schema author may also enhance any schema component with attributes which are not defined by the schema language. The names of these additional attributes are only limited to belong to a non-XSD namespace. This possibility to add arbitrary name/value pairs means language support for the addition of metadata.

When dealing with data-oriented XML, elements and attributes describe real-world entities and metadata can extend these descriptions, e.g. adding processing hints. Metadata of an element declaration might, for instance, specify the data source from where to retrieve the value of the element, as well as the data target where to store it:

```
<xs:element name="loyaltyNumber" type="xs:string"
    x:dataSource= "msgs.travelInfo#//Passenger/@loyalty"
    x:dataTarget= "dbs.someDatabase.someTable.someColumn"/>
```

Such metadata could be put to many uses, including code generation and system consistency checks (are the schema types of data source and data target compatible?). But the possibilites are much more limited than they appear at first sight, due to the *reuse* of schema components (like type definitions) in different contexts: such metadata would typically apply in one context, but not in the other. We come to realize that many kinds of interesting metadata cannot be attached to the schema components themselves: they should be attached to a novel kind of entity representing the use of a particular element or attribute declaration at a particular place within a particular document type. Such *component use* is an abstraction lying midway between a schema component and the items of an instance document. This paper attempts to capture component use conceptually, proposing the notion of an *info location*. It introduces the means to materialize component uses, turning them into the elements of a new kind of resource called a *location tree*.

Location trees

An info location tree (**location tree**, for short) is an XML document which represents the tree structure of documents described by an XML schema. More precisely, a location tree captures the tree structure implied by a complex type definition or a group definition. As a rough approximation, think of location trees as XML documents capturing the information visualized by the familiar graphical views of schema documents.

A simple example

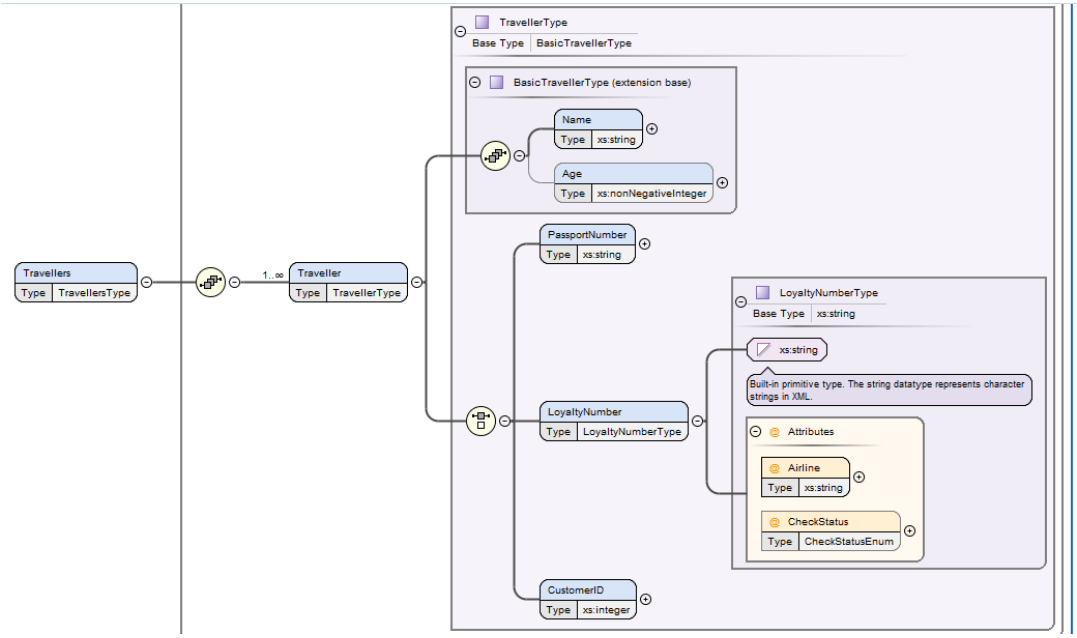
Let us look at a simple example. The following schema defines documents rooted in a `<Travellers>` element:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns="http://example.com/ns"
           targetNamespace="http://example.com/ns"
           elementFormDefault="qualified">
    <xs:element name="Travellers" type="TravellersType"/>
    <xs:complexType name="TravellersType">
        <xs:sequence>
            <xs:element name="Traveller" type="TravellerType"
                        maxOccurs="unbounded"/>
```

```
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="BasicTravellerType">
        <xs:sequence>
            <xs:element name="Name" type="xs:string"/>
            <xs:element name="Age" type="xs:nonNegativeInteger" minOccurs="0"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="TravellerType">
        <xs:complexContent>
            <xs:extension base="BasicTravellerType">
                <xs:choice>
                    <xs:element name="PassportNumber" type="xs:string"/>
                    <xs:element name="LoyaltyNumber" type="LoyaltyNumberType"/>
                    <xs:element name="CustomerID" type="xs:integer"/>
                </xs:choice>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>

    <xs:complexType name="LoyaltyNumberType">
        <xs:simpleContent>
            <xs:extension base="xs:string">
                <xs:attribute name="Airline" type="xs:string" use="required"/>
                <xs:attribute name="CheckStatus" type="CheckStatusEnum"/>
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>

    <xs:simpleType name="CheckStatusEnum">
        <xs:restriction base="xs:string">
            <xs:enumeration value="NoCheck"></xs:enumeration>
            <xs:enumeration value="Ok"></xs:enumeration>
            <xs:enumeration value="NotOk"></xs:enumeration>
            <xs:enumeration value="Unknown"></xs:enumeration>
        </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

Though this is a very small and simple schema, the tree structure of a `Travellers` document is not immediately obvious. Figure 1 offers a graphical representation of this tree structure, created using the XML IDE Oxygen 18.1.

Figure 1. Graphical XSD representation (Oxygen 18.1)



Graphical representation of the `Travellers` schema, created by Oxygen 18.1

Such a representation is tremendously helpful, but its use is restricted to human inspection, as it cannot be used as processing input. A processing-friendly alternative is offered by a location tree. The following listing shows a pruned version, leaving out items (including type information) for the sake of readibility:

```
<a:Travellers xmlns:a="http://example.com/ns">
   <a:Traveller z:occ="+">
      <a:Name/>
      <a:Age z:occ="?"/>
      <z:_choice_>
         <a:PassportNumber/>
         <a:LoyaltyNumber>
            <z:_attributes_>
               <Airline/>
               <CheckStatus occ="?">
            </z:_attributes_>
         </a:LoyaltyNumber>
         <a:CustomerID/>
      </z:_choice_>
   </a:Traveller>
</a:Travellers>
```

The meaning of this representation is easy to grasp intuitively. Instance documents have a `<Travellers>` root element which contains one or more `<Traveller>` elements. Note the @z:occ attribute on `<Traveller>` which indicates that the `<Traveller>` element of the location tree represents a sequence of one or more elements in the instance document. We understand that the first child of `<Traveller>` is a `<Name>` element, which is followed by an optional `<Age>` element. The last element in `<Traveller>` is one of three possibilities: either a `<PassportNumber>` element, or a `<LoyaltyNumber>` element, or a `<CustomerID>` element. Note that the `<z:_choice_>` element does not represent any node in an instance document; rather, it is an auxilliary element indicating that at this point of the structure the contents of instance documents is given by exactly one of several possibilities. Finally we note that `<LoyaltyNumber>` elements have a couple of attributes, @Airline (required) and @CheckStatus (optional).

Roughly speaking, each location tree element which is not in the z-namespace represents a set of XML elements or attributes - a set comprising all items found within instance documents "at the same location". How to define the location? All items found at a particular location have the same data path (e.g. `/Travellers/Traveller/Age`). Note that the inverse is not always true (though it is always true in the example): two elements may have the same data path but occupy different locations. This would for example be the case if an element's content model were the sequence of child elements `<a>`, `<b>`, `<c>`, `<a>`. All `<a>` elements would have the same datapath, but the first and last siblings among them would occupy two distinct locations. This matches the intuitive expectation of a location tree or a graphical schema view: the siblings `<a>`, `<b>`, `<c>`, `<a>` should certainly be represented by four location nodes or graph icons, rather than three.

Info locations

In spite of the obvious intuitive meaning of location tree nodes, a formal definition of their semantics is not trivial. The key abstraction is an **info location**, which is akin to an RDF class whose members are XML nodes. An info location is a class of XML nodes belonging to a document type's instance documents, where class membership depends on the node's validation path. The validation path is the sequence of schema components used to validate the item itself, its ancestors and their preceding siblings, when validated in document order. Two items are said to occupy the same location if their validation paths are equal according to an algorithm of comparison. (Note. The comparison of validation paths must be performed according to an algorithm, rather than as a straightforward comparison, in order to ignore the irrelevant variability introduced by optional and multiple occurrences.)

The following listing shows the complete location tree of `<Travellers>` elements. **Element locations** are represented by elements not in the z-namespace, excluding child elements of `<_attributes_>` elements. **Attribute locations** are represented by child elements of `<z:_attributes_>` elements. The properties of the info locations are exposed by attributes.

```
<?xml version="1.0" encoding="UTF-8"?>
<z:locationTrees xmlns:xs="http://www.w3.org/2001/XMLSchema"
                 xmlns:a="http://example.com/ns"
                 xmlns:z="http://www.xsdplus.org/ns/structure"
                 count="1">
  <z:locationTree compKind="elem"
                  z:name="a:Travellers"
                  z:loc="element(a:Travellers)"
                  z:type="a:TravellersType"
                  z:typeLoc="complexType(a:TravellersType)">
      <z:nsMap>
         <z:ns prefix="a" uri="http://example.com/ns"/>
         <z:ns prefix="xs" uri="http://www.w3.org/2001/XMLSchema"/>
         <z:ns prefix="z" uri="http://www.xsdplus.org/ns/structure"/>
      </z:nsMap>
      <a:Travellers
                  z:name="a:Travellers"
                  z:type="a:TravellersType"
                  z:typeVariant="cc"
                  z:typeLoc="complexType(a:TravellersType)">
         <a:Traveller
                  z:name="a:Traveller"
                  z:occ="+"
                  z:type="a:TravellerType"
                  z:typeVariant="cc"
                  z:baseType="a:BasicTravellerType"
                  z:derivationKind="extension"
                  z:typeLoc="complexType(a:TravellerType)"
                  z:loc="complexType(a:TravellersType)/
```

```
                    xs:sequence/a:Traveller"
            name="Traveller"
            type="TravellerType"
            maxOccurs="unbounded">
    <a:Name
            z:name="a:Name"
            z:type="xs:string"
            z:typeVariant="sb"
            z:typeDef="string"
            z:loc="complexType(a:BasicTravellerType)/
                    xs:sequence/a:Name"
            name="Name"
            type="xs:string"/>
    <a:Age  z:name="a:Age"
            z:occ="?"
            z:type="xs:nonNegativeInteger"
            z:typeVariant="sb"
            z:typeDef="nonNegativeInteger"
            z:loc="complexType(a:BasicTravellerType)/
                    xs:sequence/a:Age"
            name="Age"
            type="xs:nonNegativeInteger"
            minOccurs="0"/>
    <z:_choice_>
       <a:PassportNumber
            z:name="a:PassportNumber"
            z:type="xs:string"
            z:typeVariant="sb"
            z:typeDef="string"
            z:loc="complexType(a:TravellerType)/
                    xs:choice/a:PassportNumber"
            name="PassportNumber"
            type="xs:string"/>
       <a:LoyaltyNumber
            z:name="a:LoyaltyNumber"
            z:type="a:LoyaltyNumberType"
            z:typeVariant="cs"
            z:baseType="xs:string"
            z:derivationKind="extension"
            z:builtinBaseType="xs:string"
            z:contentType="xs:string"
            z:contentTypeVariant="sb"
            z:contentTypeDef="string"
            z:typeLoc="complexType(a:LoyaltyNumberType)"
            z:loc="complexType(a:TravellerType)/
                    xs:choice/a:LoyaltyNumber"
            name="LoyaltyNumber"
            type="LoyaltyNumberType">
          <z:_attributes_>
             <Airline
                z:name="Airline"
                z:type="xs:string"
                z:typeVariant="sb"
                z:typeDef="string"
                z:loc="complexType(a:LoyaltyNumberType)/
                        @Airline"
                name="Airline"
                type="xs:string"
```

```
                        use="required"/>
                    <CheckStatus
                        z:name="CheckStatus"
                        z:occ="?"
                        z:type="a:CheckStatusEnum"
                        z:typeVariant="sa"
                        z:typeDef="string: enum=(NoCheck|NotOk|Ok|Unknown)"
                        z:baseType="xs:string"
                        z:derivationKind="restriction"
                        z:builtinBaseType="xs:string"
                        z:typeLoc="simpleType(a:CheckStatusEnum)"
                        z:loc="complexType(a:LoyaltyNumberType)/
                               @CheckStatus"
                        name="CheckStatus"
                        type="CheckStatusEnum">
                        <z:_stypeInfo_ z:name="a:CheckStatusEnum">
                            <z:_builtinType_ z:name="xs:string"/>
                            <z:_restriction_ z:name="a:CheckStatusEnum">
                                <z:_enumeration_ value="NoCheck"/>
                                <z:_enumeration_ value="Ok"/>
                                <z:_enumeration_ value="NotOk"/>
                                <z:_enumeration_ value="Unknown"/>
                            </z:_restriction_>
                        </z:_stypeInfo_>
                    </CheckStatus>
                </z:_attributes_>
            </a:LoyaltyNumber>
            <a:CustomerID
                z:name="a:CustomerID"
                z:type="xs:integer"
                z:typeVariant="sb"
                z:loc="complexType(a:TravellerType)/
                       xs:choice/a:CustomerID"
                name="CustomerID"
                type="xs:integer"/>
        </z:_choice_>
    </a:Traveller>
</a:Travellers>
</z:locationTree>
</z:locationTrees>
```

The next two sections give an overview of the elements and attributes used by location trees.

Location tree structure

Now we take a closer look at the structure of location trees. The following table summarizes the elements of which a location tree is composed. (Descendants of `<z:_stypeInfo>` elements are omitted.)

Table 1. Location tree elements

Location tree element	Meaning	XSD element
Any element not in the z-namespace	Represents an element or attribute location	xs:element, xs:attribute
z:_attributes_	Wraps the representations of all attribute locations belonging to an element location	-

Location tree element	Meaning	XSD element
z:_sequence_	Represents a `sequence` compositor	xs:sequence
z:_choice_	Represents a `choice` compositor	xs:choice
z:_all_	Represents an `all` compositor	xs:all
z:_any_	Represents an element wildcard	xs:any
z:_anyAttribute_	Represents an attribute wildcard	xs:anyAttribute
z:_sgroup_	Contains the element locations corresponding to a substitution group	-
z:_groupContent_	Signals a group reference which *remains unresolved* as it constitutes a cyclic definition (the reference occurs in the content of a member of the referenced group or in its descendant)	xs:group (with a @ref attribute)
z:_stypeInfo_	Structured representation of a simple type definition	xs:simpleType
z:_annotation_	Reports a schema annotation	xs:annotation
z:_documentation_	Reports a schema documentation	xs:documentation
z:_appinfo_	Reports a schema appinfo	xs:appinfo

Note that group definitions (`<xs:group>`) are not represented by distinct location tree elements, as a group reference is represented by the content of the referenced group. Similarly, attribute group references are always resolved and attribute group definitions (`<xs:attributeGroup>` do not appear as distinct entities.

An important aspect of location tree structure is the representation of element composition (sequence, choice and all group). Representation rules aim at simplification: the number of group compositor elements is reduced, nested grouping is replaced by flattened grouping if possible, and irrelevant variability of XSD content is removed by normalized representation. Simplification is achieved by the definition of default composition and rules of group normalization.

Sequence is the **default composition**: any sequence of items (element locations and or compositors) which are children of an element location represents a sequence of child elements. The location tree uses only such `<z:_sequence_>` elements as cannot be left out without loss of information. This is the case when the sequence has a minimum or maximum occurrence unequal 1, or when the sequence is child of a choice. **Group normalization** is achieved by content rewriting. It replaces the use of compositors as found in the XSDs by a simpler model which is equivalent with respect to the instance documents described. Normalization rules effect the removal of "pseudo groups" containing a single item, the flattening of nested choices and the flattening of nested sequences. Note that the removal of a compositor element may change the occurrence constraints of its child elements. As an example, consider a pseudo group occurring one or more times and containing an optional element (occurrence "zero or one"). After unwrapping this element, its occurrence will be "zero or more".

An important aspect of location tree contents is the handling of type derivation. Remember the general goal to capture the structure of instance documents by a model structure which is as similar and straightforward as possible. Element content defined by a derived complex type is always represented in resolved form, expressing the integration of base type content and derived type content as defined by the XML schema specification. Consider a type extending another type with complex content. The derived type is treated like a non-derived type with equivalent content: its attributes comprise all

attributes of derived type definition and the (recursively resolved) base type definition; and its content is a sequence containing the contents of the (recursively resolved) base type, followed by the explicit content model of the derived type. Subsequent group normalization ensures that derivation does not introduce any structural complexity.

Location tree attributes

The elements of a location tree (element and attribute locations, group compositors, wildcards and substitution groups) represent model components – the building blocks of which the model is composed. The **properties** of these components are represented by info attributes. Example properties are the element or attribute name, type name and cardinality constraints. The following table compiles the most important attributes used to express component properties.

Table 2. Property attributes

Location tree attribute	Property	Remarks
z:name	Element name, attribute name	As normalized qualified name
z:occ	Cardinality	Examples: *?, *, +, 1-4, 2*
z:type	Type name	As normalized qualified name
z:typeVariant	Type variant	sb: builtin type
		sa: simple type, atomic
		sl: simple type, list
		su: simple type, union
		cs: complex type, simple content
		cc: complex type, complex content
		cc: complex type, empty content
z:typeDef	A human-readible string representation of a simple type definition	Examples: *string:enum=(NotOk|Ok)* *string:maxLen=64*
z:baseType	Base type name	As normalized qualified name
z:builtinBaseType	Name of the builtin base type (if type variant = sa)	As normalized qualified name
z:contentType	Name of the content type (if type variant = cs)	As normalized qualified name
z:contentTypeVariant	Content type variant (if type variant = cs)	See z:typeVariant; value is one of: sb, sa, sl, su
z:contentTypeDef	A human-readible string representation of the content type (if type variant = cs)	Examples: *decimal:range=[0.901,1.101]* *string:pattern=#[A-Z]{2}#*
z:itemType	Name of the list item type (if type variant = sl)	As normalized qualified name
z:itemTypeVariant	Item type variant (if type variant = sl)	See z:typeVariant; value is one of: sb, sa, su
z:typeLoc	XSD type location	If item has a user-defined type: identifies the location of the type definition within the XSDs

Location tree attribute	Property	Remarks
z:loc	XSD component location	Identifies the location of the component within the XSDs
any name without namespace	XSD component property (e.g. {min occurs})	The attribute is a copy of the XSD attribute found on the XSD element represented by the location tree element (e.g. @minOccurs)
any name in a namespace which is not the z-namespace	XSD annotation property (e.g. {myorg:lastUpdate})	The attribute is a copy of the annotation attribute found on the XSD element represented by the location tree element (e.g. @myorg:lastUpate)

Normalized qualified names use prefixes which are derived from the target namespaces (and possibly also annotation namespaces) encountered in the set of XSDs currently considered. The mapping of namespace URI to prefix is achieved by assigning to the *sorted* set of namespace URIs the prefixes a – y, a2 – y2 etc. The prefix z is reserved for the URI http://www.xsdplus.de/ns/structure, and the prefix xs is used for the XSD namespace. All URI to prefix mappings are documented by a `<z:nsMap>` element which is child of the `<z:locationTree>` element.

The location attributes (@z:loc, @z:typeLoc) identify an individual XSD component and thus document the alignment between location tree and XSD components. The XSD component is identified by a simple path syntax consisting of a first step identifying a top-level component (e.g. `complexType(a:TravellersType)`) followed by an optional logical path (e.g. `xs:choice/a:LoyaltyNumber`) drilling down into the top-level component. The @z:loc attribute of an element location (attribute location) identifies the element declaration (attribute declaration) aligned with the location. Similarly, @z:typeLoc identifies the type definition referenced by an element or attribute location.

Two further attributes are used in order to flag an element location or a group reference as "recursion point":

Table 3. Recursion point attributes

Location tree attribute	Meaning	Value
z:recursionType	The type of the parent element location is equal to the type of an ancestor element location.	Type name, as normalized qualified name
z:recursionGroup	Flags a group reference found within the contents of an element which is a member of the referenced group, or is a descendant of such a member	Group name, as normalized qualified name

Open source tool for creating location trees

An open source tool for transforming arbitrary XSD into location trees is available [10]. The tool is written in the XQuery language [9]. It can be used as a command-line tool, or as an XQuery library ready for import by XQuery programs. Execution requires an XQuery processor supporting XQuery version 3.1 or higher. See [10] for a documentation of tool usage. Command-line invocation requires two mandatory parameters:

- A FOXpath expression [2] selecting one or more XSD documents
- A name pattern selecting the schema components for which to construct the location trees; the name pattern can alternatively be used to select element declarations, complex type definitions or group definitions

Tool output is an XML document with a `<z:locationTrees>` root element containing `<z:locationTree>` elements representing location trees obtained for the selected schema components. Here comes an example call, using the XQuery processor BaseX [1]:

```
basex –b request=ltrees?xsd=/projects/ota//*.xsd,enames=*RQ xsdp.xq
```

It creates location trees for each element declaration whose name ends with RQ.

XSD based tool development

Location trees are ordinary XML documents. The semantic graph structure of XSD documents has been replaced by semantic tree structure, so that processing location trees is a straightforward operation. Initial transformation of XSD into location trees makes valuable information, hitherto virtually out of reach from a developer's point of view, readily accessible. Location trees thus encourage the development of innovative XSD processing tools. This section illustrates the new possibilites by several examples ranging from very simple to fairly complex tools.

Getting your feet wet - first schema queries

The simplest way of using location trees consists in translating questions about a schema into queries of its location trees. As an example, consider this question about a schema: *at which places (expressed as data paths) do instance documents with a `<Travellers>` root contain elements with the name "CustomerID"?*

Our starting point is the location tree obtained for the top-level element declaration with name "Travellers". The following XQuery query, which expects as context node a `z:locationTrees` document, gives the answer:

```
declare namespace z="http://www.xsdplus.org/ns/structure";
declare namespace f="http://www.xsdplus.org/ns/xquery-functions";

declare function f:path($e as element()) as xs:string {
  '/'||string-join((
       $e/ancestor::*[not(self::z:*)]/local-name(),
       $e/concat(parent::z:_attributes_/'@', local-name())), '/')
};

//*:CustomerID/f:path(.)
```

Extending this adhoc query into a veritable tool is easy: we introduce two external variable which expect name patterns for the document root element and the items of interest:

```
declare namespace z="http://www.xsdplus.org/ns/structure";
declare namespace f="http://www.xsdplus.org/ns/xquery-functions";
declare variable $root external := '*';
declare variable $item external := '*';

declare function f:path($e as element()) as xs:string {
  '/'||string-join((
       $e/ancestor::*[not(self::z:*)]/local-name(),
       $e/concat(parent::z:_attributes_/'@', local-name())), '/')
};

declare function f:regex($name as xs:string) as xs:string {
  concat('^', replace(replace($name, '\*', '.*'), '\?', '.'), '$')};

/*/z:locationTree/(* except z:*)[matches(local-name(.), f:regex($root), 'i')]
//(* except z:*)[matches(local-name(), f:regex($item), 'i')]/f:path(.)
```

After storing this code in a file `locationPaths.xq` and writing the location trees of all top-level elements of the Niem 3.0 XSDs (downloaded from [4] into a file `ltrees-niem30.xml`:

```
basex -b "request=ltree?xsd=/xsdbase/niem-3.0//*.xsd,enames=*,global"
   /tt/xsdp/xsdp.xq > ltrees-niem30.xml
```

the following invocation

```
basex -i ltrees-niem30.xml -b item=*xml* locationPath.xq
```

yields this output:

```
/EDXLDistribution/contentObject/nonXMLContent
/EDXLDistribution/contentObject/xmlContent
/EDXLDistribution/contentObject/xmlContent/keyXMLContent
/EDXLDistribution/contentObject/xmlContent/embeddedXMLContent
```

These four data paths leading to items with a name containing "xml" have been selected from 90763 data paths which can be obtained from the location trees in a straightforward way (see function `f:path` above). The example shows how simple queries applied to location trees can give valuable insight which would be very difficult to obtain by processing the XSD documents themselves. The next section discusses more complex uses of location trees.

Schema reporting - treesheets

Location trees are not meant for human inspection – they are intermediates serving as input to the query or transformation producing an artifact of interest. An important category of such artifacts are various representations of the schema-defined tree structures and information associated with their main building blocks, the elements and attributes.

A **treesheet** is a text document which combines an indentation-based representation of the tree structure (on the left-hand side) with information about the info locations of which the tree is composed (on the right-hand side). The following treesheet, for example

```
Element: Travellers;  Type: TravellersType
=========================================

Travellers     ~    ~    ctype: a:TravellersType
. Traveller+   ~    ~    ctype: a:TravellerType
. . Name  ...  ...  ...  type:  string
. . Age?  ...  ...  ...  type:  nonNegativeInteger
. . choice{
. . 1 PassportNumber  .  type:  string
. . 2 LoyaltyNumber  ..  type:  string
. . 2   @Airline .  ...  type:  string
. . 2   @CheckStatus? .  type:  string: enum=(NoCheck|NotOk|Ok|Unknown)
. . 3 CustomerID    ...  type:  integer
. . }
```

displays type information, whereas the next one

```
Element: Travellers;  Type: TravellersType
=========================================

Travellers     ~    ~    anno: A travel party.
. Traveller+   ~    ~    anno: Represents an individual traveller.
. . Name  ...  ...  ...  anno: The surname, as used in the passport.
. . Age?  ...  ...  ...  anno: The age at checkin date.
. . choice{
. . 1 PassportNumber  .  anno: The passport number in latin letters.
```

```
. . 2 LoyaltyNumber  ..  anno: The Loyalty number, assigned by the airline.
. . 2   @Airline .   ...  anno: Airline, identified by IATA code.
. . 2   @CheckStatus? .  anno: Specifies if checked and the check result.
. . 3 CustomerID     ...  anno: The customer ID, assigned by the back office.
. . }
```

displays the contents of schema annotations, retrieved from `<xs:documentation>` elements of the schema. These examples align the tree items with information originating from the XSD (type information and schema annotation). The information might however also be metadata externally provided, or facts based on instance document data. (See the following sections for more about facts and metadata.) Each type of treesheet can be viewed as a different facet of the schema-defined tree. Opening different treesheets of the same document type in an editor and switching between their tabs can offer an interesting experience of contemplating complex information trees from different sides. The open source tool [10] provides a `treesheet` operation transforming XSD components into treesheets of various kinds and controlled by various representational options.

Fact trees

A location tree can be regarded as a model of item use: given a document type, *where* (data path) to use *what* (element/attribute name) and *how* (number of occurrences, data type or content structure). Any set of instance documents, on the other hand, generates facts about how items are actually used. Example facts about an info location are the frequency of documents containing items in this location, as well as the value strings used in these items. Interesting reports can be created by merging these facts into a pruned version of the location tree. Recipe:

- Starting point: location tree
- Remove all elements which do not represent compositors, elements or attributes
- Remove all or a selection of attributes
- Add attributes conveying facts

A simple example uses several attributes telling us how info locations are actually used:

- @dcount – the number of documents observed (attribute at the root element only)
- @dfreq – the relative frequency of documents containing an item at this location
- @ifreq – the mean/minimum/maximum number of items at this location, ignoring documents without items in this location (attribute omitted if the schema does not allow for multiple items)

The resulting fact tree may reveal interesting details, like choice branches never used, or optional elements never or virtually always used:

```
<a:Travellers xmlns:a="http://example.com/ns" dcount="78925">
      <a:Traveller z:occ="+" dfreq="1.00" ifreq="3.01 (1-8)">
        <a:Name dfreq="1.00"/>
        <a:Age z:occ="?" dfreq="0.99"/>
        <z:_choice_>
          <a:PassportNumber dfreq="0.64"/>
          <a:LoyaltyNumber dfreq="0.36">
            <z:_attributes_>
              <Airline dfreq="0.35"/>
              <CheckStatus dfreq="0.09"/>
            </z:_attributes_>
          </a:LoyaltyNumber>
          <a:CustomerID dfreq="0"/>
        </z:_choice_>
      </a:Traveller>
    </a:Travellers>
```

If the fact tree has been obtained for a set of request messages belonging to test suites, attributes like @dfreq can express test coverage in a very interesting way.

A fact tree can be transformed into a treesheet:

```
Element: Travellers;  Type: TravellersType
==========================================

Travellers     ~    ~    count: 78925
. Traveller+   ~    ~    dfreq: ********** (1.00, ifreq=3.01 (1-8)
. . Name  ...  ...  ...  dfreq: ********** (1.00)
. . Age?  ...  ...  ...  dfreq: ********** (0.99)
. . choice{
. . 1 PassportNumber  .  dfreq: ******     (0.64)
. . 2 LoyaltyNumber  ..  dfreq: ****       (0.36)
. . 2   @Airline .  ...  dfreq: ****       (0.35)
. . 2   @CheckStatus ..  dfreq: *          (0.09)
. . 3 CustomerID    ...  dfreq:            (0)
. . }
```

Note that the concept of "facts about locations" is not limited to usage statistics. The next section discusses the enhancement of locations by metadata. Such metadata may imply new kinds of facts which can be represented by specialized fact trees. Assume, for example, location metadata which specify from where to retrieve item data. If those "places" can also be expressed as info locations (e.g. belonging to the location tree of some web service message), *derived facts* of data consistency emerge. If the data source of an enumerated string is an unconstrained string, a data type inconsistency emerges as a fact.

The effort required to transform location trees into treesheets and fact trees is moderate, although it is substantially greater than the simple queries shown earlier. The final section about the use of location trees presents a complex application performing code generation. It is based on metadata trees which are created by a semi-automatic procedure: automated transformation of location trees into an initial version of a metadata tree, followed by manual editing which replaces the generated dummy values of metadata attributes with real values.

Metadata trees and code generation

Location trees are composed of nodes representing the elements and attributes which can be used by instance documents. Adding to the nodes of a location tree metadata, we obtain a metadata tree. Recipe:

- Starting point: location tree
- Remove all elements which do not represent compositors, elements or attributes
- Remove all or a selection of attributes
- Add attributes specifying metadata

Like a location tree, a metadata tree captures the structure of instance documents. Unlike location tree nodes, however, metadata tree nodes describe the items of instance documents *beyond XSD-defined information*. If metadata provide information about how to process the items, the metadata tree may be transformed into source code which implements the processing.

This section discusses an example of source code generation based on metadata trees. Our goal is a generator of program code which implements the transformation of source documents with a particular document type (e.g. "TravelInfo") into target documents with a different document type (e.g. "Travellers"). The solution involves the following main steps:

1. Design a metadata model
2. Create a metadata tree generator
3. Create a metadata tree transformator

Design a metadata model

Our approach is to use a metadata tree derived from the location tree of the *target* document type. The metadata model must define a set of metadata which suffices to determine every detail of the

code to be generated. Note that the required types of metadata may be different for different nodes of the location tree: they will depend on node properties like the content type variant (simple versus complex) or occurrence constraints. Each type of metadata is represented by attributes with a particular name. Metadata values are XPath expressions to be evaluated in the context of source document nodes. The following table provides an overview of the more important metadata types and how their usage depends on node properties.

Table 4. Metadata model for doc2doc transformation

Attribute name	Node condition	Meaning	Example
alt	Location of a simple content element or attribute, which is optional	XPath expression; evaluated if @src yields the empty sequence; if @alt yields a non-empty value, a target node is created and the value is used	`'#UNKNOWN'`
case	Child of a `<z:_choice_>`	XPath expression; the selected choice branch is the first child of the `choice` compositor whose @case has a `true` effective boolean value	`@Success eq "false"`
ctxt	Complex element location	XPath expression; its value is used as the new data source context	`Services/Hotel`
default	Location of a simple content element or attribute, which is mandatory	XPath expression; its value is used if @src yields the empty sequence	`'?'`
for-each	Element location (or compositor) with maxOccurs > 1	XPath expression; for each item of the expression value an element node is created (or compositor contents are evaluated)	`Contacts/Contact`
if	Complex element location (or compositor) which is optional	XPath expression; if its effective boolean value is `true`, an element node is created (or compositor contents are evaluated)	`Contacts/Contact`
src	Location of a simple content element or attribute	XPath expression; its value is used as element content or attribute value	`@Currency`

Create a metadata tree generator

Writing the metadata tree generator is a fairly straightforward task: the generator transforms location trees into a modified copy in which most location tree attributes are removed and attributes representing the metadata are added. The values of metadata attributes are placeholders, to be replaced by hand-editing. For each location tree node an appropriate selection of metadata attributes is made, according to the "Node condition" column in table Table 4, “Metadata model for doc2doc

transformation". Conditions are checked by evaluating the attributes of the location tree node. The condition "Location of a simple content element or attribute", for instance, is true if the location tree attribute @z:typeVariant has a value starting with "s" or equal to "cs". If the condition also requires that the location is optional, the @z:occ attribute is also checked – and so forth.

Applying our metadata tree generator to the `Travellers` schema, we obtain this initial metadata tree:

```
<a:Travellers xmlns:a="http://example.com/ns"
                        ctxt="###">
   <a:Traveller         for-each="###">
      <a:Name           src="###"
                        default=""/>
      <a:Age            src="###"
                        alt=""/>
      <z:_choice_>
         <a:PassportNumber
                        case="###"
                        src="###"
                        default=""/>
         <a:LoyaltyNumber case="###"
                        ctxt="###"
                        src="###"
                        default="">
            <z:_attributes_>
               <Airline
                        src="###"
                        default=""/>
               <CheckStatus
                        src="###"
                        alt=""/>
            </z:_attributes_>
         </a:LoyaltyNumber>
         <a:CustomerID  case="###"
                        src="###"
                        default=""/>
      </z:_choice_>
   </a:Traveller>
</a:Travellers>
```

Create a metadata tree transformator

A metadata tree transformator transforms a metadata tree into a useful artifact. In our present context, the metadata tree transformator is a code generator. It transforms the metadata tree into code implementing the transformation of a source document into a target document. If code in several programming languages is required, we can create several transformators, one for each required language. Here, we limit ourselves to writing a code generator creating XQuery code. The task is neither trivial, nor exceedingly difficult. See [10], module `xmap2xq.xqm` for a solution.

Using the code generator

Now we are ready to use our code generator. Let the transformation source be documents similar to this:

```
<travelInfo xmlns="http://example2.com/msgs">
   <time startDate="2017-07-31"
         endDate="2017-08-08"/>
   <players>
      <passengers>
         <passenger surName="Boateng"
```

```
                    firstName="Rachel"
                    bookingAge="32"
                    loyalty="KLM:1234567"
                    loyaltyCheck="Ok"/>
         <passenger surName="Boateng"
                    firstName="Belinda"
                    documentNr="9293949596"/>
      </passengers>
   </players>
</travelInfo>
```

We edit the metadata tree appropriately:

```
<a:Travellers xmlns:a="http://example.com/ns"
                        ctxt="/s:travelInfo/s:players/s:passengers">
   <a:Traveller         for-each="s:passenger">
      <a:Name           src="@surName"
                        default=""/>
      <a:Age            src="@bookingAge"
                        alt=""/>
      <z:_choice_>
         <a:PassportNumber
                        case="@documentNr"
                        src="@documentNr"
                        default=""/>
         <a:LoyaltyNumber case="@loyalty"
                        ctxt="."
                        src="@loyalty/substring-after(., ':')"
                        default="'0'">
            <z:_attributes_>
               <Airline src="@loyalty/substring-before(., ':')"
                        default="'#UNKNOWN-AIRLINE'"/>
               <CheckStatus
                        src="@loyaltyCheck"
                        alt="'Unknown'"/>
            </z:_attributes_>
          </a:LoyaltyNumber>
          <a:CustomerID case="@custId"
                        src="@custId"
                        default=""/>
       </z:_choice_>
    </a:Traveller>
 </a:Travellers>
```

Passing this metadata tree to the code generator, we obtain the following *generated* XQuery program:

```
let $c := /s:travelInfo/s:players/s:passengers
return if (empty($c)) then () else

<a:Travellers xmlns:a="http://example.com/ns">{
   for $c in $c/s:passenger
   return
      <a:Traveller>{
         let $v := $c/@surName
         return
            <a:Name>{string($v)}</a:Name>,
         let $v := $c/@bookingAge
         return
            if (empty($v)) then ()
```

```
            else <a:Age>{string($v)}</a:Age>,
         if ($c/@documentNr) then
            let $v := $c/@documentNr
            return
                <a:PassportNumber>{string($v)}</a:PassportNumber>
         else if ($c/@loyalty) then
            let $contentV := $c/@loyalty/substring-after(., ':')
            let $contentV := if ($contentV) then $contentV else '0'
            return
              <a:LoyaltyNumber>{
                 let $v := $c/@loyalty/substring-before(., ':')
                 let $v := if ($v) then $v else '#UNKNOWN-AIRLINE'
                 return
                    attribute Airline {$v},
                 let $v := $c/@loyaltyCheck
                 let $v := if ($v) then $v else 'Unknown'
                 return
                    attribute CheckStatus {$v},
                 $contentV
              }</a:LoyaltyNumber>
         else if ($c/@custId) then
            let $v := $c/@custId
            return
               <a:CustomerID>{string($v)}</a:CustomerID>
         else ()
      }</a:Traveller>
   }</a:Travellers>
```

Discussion

The main building block of a location tree constitutes an interesting new concept. A location is obviously a *model entity*, as it depends on schema contents, never on instance document contents. It encapsulates model information about a well-defined class of real world data: the items occurring in a particular document type at a particular place. The limitation and precision of the modelling target make a location particularly well-suited for novel kinds of metadata, as well as a suitable entity for defining and collecting related facts. Locations, formally defined and readily available in materialized form, might influence the way how one thinks and speaks about structured information.

Appreciation of location trees entails additional appreciation for XSD, as location trees are derived from XSD. Location trees unite "the best of two worlds", XSD's advanced features of model design and component reuse, and a clear and straightforward relationship between model entity and real world data, which enables intense use of the model beyond validation and documentation.

Bibliography

[1] *BaseX - an open source XML database*. BaseX. 2017. http://basex.org.

[2] *FOXpath navigation of physical, virtual and literal file systems*. Hans-Juergen Rennau. 2017. http://archive.xmlprague.cz/2017/files/xmlprague-2017-proceedings.pdf.

[3] *JAXB – reference implementation and specification*. java.net. https://jaxb.java.net/.

[4] *NIEM 3.0*. Georgia Tech Research Institute, Inc. (US). https://release.niem.gov/niem/3.0/.

[5] *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. World Wide Web Consortium (W3C). 2014. https://www.w3.org/TR/xmlschema11-1/.

[6] *W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes*. World Wide Web Consortium (W3C). 2014. https://www.w3.org/TR/xmlschema11-2/.

[7] *XML Schema Part 1: Structures Second Edition*. World Wide Web Consortium (W3C). 2004. https://www.w3.org/TR/xmlschema-1/.

[8] *XML Schema Part 2: Datatypes Second Edition*. World Wide Web Consortium (W3C). 2004. https://www.w3.org/TR/xmlschema-2/.

[9] *XQuery 3.1: An XML Query Language*. World Wide Web Consortium (W3C). 2017. https://www.w3.org/TR/xquery-31/.

[10] *xsdplus - a toolkit for XSD based tool development*. Hans-Juergen Rennau. 2017. https://github.com/hrennau/xsdplus.