Rethinking transformation – the potential of code generation

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

Hauke Brandes, parsQube GmbH
<hauke.brandes@parsqube.de>

Abstract

A code generator for document to document transformation is introduced. It reduces the development effort to editing a set of metadata items attached to a tree model of the target documents. Metadata values are XQuery expressions which are typically so simple that they do not require genuine programming skills. Nevertheless, expressions are more difficult to provide than static values, and therefore possibilities of further simplifying the development task are explored, striving to enable subject matter experts to define the transformation without writing XQuery expressions. This can be achieved by generating the expressions from assertions about alignments between source and target nodes, although specific requirements will often necessitate additional information. As alignments can be represented graphically by connecting lines, the approach amounts to a solid conceptual foundation for graphical mapping tools. Finally, the underlying model of code generation driven by target document structure is generalized into a conceptual framework which is not restricted to XML data sources. Its usefulness is demonstrated by a simple code generator for transforming RDF data into XML documents.

Table of Contents

Introduction 2
Rethinking document-to-document transformation 2
Source navigation based model 4
    Basic concepts 5
    Primitive operations 6
    Composing document-to-document transformation 6
Metadata model SNAT 7
    Metadata item model 8
    Advanced features 11
    Code generator SNAT 12
Source alignment based model 12
    What is an alignment? 13
    Representation versus information 13
    Annotated target tree (SAAT) 13
    Mapping alignments to navigation 14
    Infering context propagation 15
    Semantic versus structural relationships 15
    Shortest-path-principle 16
    Future work 17
Metadata model SAAT (Source Alignment Annotated Target tree) 17
    Minimal SAAT 17
    Alignment qualifiers 17
    Code generator SAAT 18
AT Map Machine 18
    Does SNAT presuppose XML? 18
    Does SAAT presuppose XML? 19
    Scheme for building SNAT-based code generators 19

Using SNAT as a meta model 20
Proof of concept: RDF-to-XML, SNAT-based 20
Metadata item model 21
Metadata value model 21
Code assembly model 22
Example SNAT and source code 22
Bibliography 24

Introduction

XML documents are trees of labeled information, and they may be associated with a document model (e.g. XSD), which prescribes a particular tree structure. Code for processing or constructing documents therefore often reflects the tree structure described by a document model. Such code might be called **document model driven code**.

While a document model implies the tree structure of its instance documents, this tree structures is usually implicit, rather than explicit, due to the use of references which "flatten" the representation [3]. However, a de-normalizing transformation (replacing references by a copy of the referenced component) can generate a **document model tree**, a tree-structured document model which mirrors the tree structure of its instance documents as closely as possible (see [3] for a detailed discussion). Augmenting the nodes of a document model tree with metadata, one obtains a **document metadata tree**. The metadata of a model node may convey hints how to process or create the instance document items modeled by the node. This implies a possibility to generate document model driven code from document metadata trees [3]. Typically (although not necessarily) the resulting code will mirror the model tree: starting at the root node of the model and descending from each model node to its children.

An important category of document model driven code is document construction from data sources, e.g. an XML or JSON document, relational database content or RDF data. This paper explores the generation of document constructors from metadata trees in depth. Major goals are to provide a sound conceptual base and to explore its application in practice. We focus on XML document to XML document transformation, but in the last part we generalize our findings, proposing a conceptual framework for building a family of code generators, each one dealing with a different source media type.

Note on terminology. This paper leans on the concept of a document model tree as defined in [3]. The term used in [3] to denote a node of a model tree is **location**. We use the terms model node and location as synonyms.

Rethinking document-to-document transformation

As a starting point, we introduce a simple example transformation.

Source document structure:

```
resources
. books
. . @lastUpdate?
. . book*
. . . @isbn
. . . title
. . . author*
. . . py?
```

Target document structure:

```
publications
```

```
. @updatedAt?
. publication*
. . @publicationYear?
. . isbn
. . title
. . creator*
. . . creatorRole
. . . creatorName
```

The implementation of a transformation can use a push approach or a pull approach:

- Push approach – the code follows the data structure of the source document, letting it trigger appropriate actions
- Pull approach – the code follows the data structure of the target document, letting it trigger appropriate actions

As a rule of thumb, the push approach is more appropriate when the structure of the target document strongly depends on the actual input data, which is typically the case with document oriented XML, mainly consisting of human readable text and structured according to its actual content. The pull approach, on the other hand, tends to be much more straightforward with data oriented XML, where the target document structure is mainly determined by the target document model, rather than unpredictable text structure.

In this discussion we focus on pull transformation, as we are interested in leveraging the information stored in the target document model. The following listing demonstrates the striking similarity between target document structure and code structure, which can be achieved using the pull approach.

Example 1. XQuery code implementing the example transformation as a pull transformation.

```
let $context := /* return

<publications>{
   let $value := $context/books/@lastUpdate return
   if (empty($value)) then () else
   attribute updatedAt {$value},
   for $context in $context/books/book return
   <publication>{
      let $v := $context/py return
      if (empty($v)) then () else
      attribute publicationYear {$v},
      <isbn>{$context/@isbn/string()}</isbn>,
      <title>{$context/title/string()}</title>,
      for $context in $context/author return
      <creator>{
         <creatorRole>{'Author'}</creatorRole>,
         <creatorName>{$context/string()}</creatorName>
      }</creator>
   }</publication>
}</publications>
```

Such code is constructed according to a simple pattern:

1. Instantiate the root element *(write start tag; content will follow)*
2. For each attribute model on this element model:
   a. Decide whether to construct this attribute
   b. If yes: set the value
3. For each child element model on this element model:

a. Decide how many instances to construct (possibly none)
b. For each instance:
   i. Instantiate element *(write start tag; content will follow)*
   ii. If element has simple content: set the value
   iii. If element has attributes/child elements: continue at (2)

The example illustrates that the code of a pull transformation can be regarded as a sum of two components: a scaffold which reflects the target document model and structures the code; and "nuggets" – small pieces of code pulling information from the source document. Note the frequent use of the variable $context, whose value is a shifting set of source nodes. It provides an appropriate starting point for navigating to the nodes of interest, for example:

```
$context/title
```

Here, the value is always bound to the "right" <book> element, which is the <book> currently processed. Such a context is essential for locating the appropriate source nodes.

The transformation can be regarded as a sequence of decisions, which is orchestrated by the target document structure. Main kinds of decision are:

A?	- Whether to construct an attribute	(2a)
#E?	- How many instances of an element to construct	(3a)
A=?	- Which value to use for an attribute	(2b)
E=?	- Which value to use for a simple content element	(3b.ii)

For following table compiles code snippets, stating the kind of decision they implement and identifying the source and target nodes involved.

Table 1. Code snippets taken from the example of a simple pull transformation.

Transformation code	Source nodes referenced by $context	Source nodes	Target nodes under construction	Decision kind
$context/books/@lastUpdate	<resources>	@lastUpdate	@updatedAt	A?
$context/books/@lastUpdate	<resources>	@lastUpdate	@updatedAt	A=?
$context/books/book	<resources>	<book>	<publication>	#E?
$context/py	<book>	<py>	@publicationYear	A?
$context/py	<book>	<py>	@publicationYear	A=?
$context/@isbn/string()	<book>	@isbn	<isbn>	E=?
$context/title/string()	<book>	<title>	<title>	E=?
$context/author	<book>	<author>	<creator>	#E?
'Author'	-	-	<creatorRole>	E=?
$context/string()	<author>	<author>	<creatorName>	E=?

The example suggests a great potential for code generation based on the target document model. This is best prepared by introducing a few abstractions.

Source navigation based model

Now we proceed to translate the informal picture of target document construction into a model which describes document construction as composed of three primitive operations orchestrated by simple rules of iteration, invocation and value passing. The crucial operations are mappings of an input sequence of source nodes to an output sequence of source nodes. As nodes can be perceived as places, such mappings may be viewed as navigation – hence we call the model a **source navigation based model.**

Basic concepts

As discussed in the previous section, document construction can be regarded as a sequence of basic decisions concerning the questions *how many instances to construct* and *which character sequence to use* (as simple element content or attribute value). Consider, for instance, the <creator> element. As its minimum and maximum cardinality differ (being 0 and unbounded), we must take a decision how many elements we will actually instantiate. However, as there may be more than one <publication> to be constructed, the answer clearly depends on which <publication> is considered – the question "how many instances?" cannot be answered globally, but only locally, concerning <creator> children of a particular <publication> element. In fact, we must ask and answer the question again and again for each <publication> appearing in the target document. Similarly, the string value to be used for @publicationYear must be determined repeatedly, once for each <publication> element in the transformation result.

To generalize, the basic decisions concerning the instantiation of a model node are taken in the *context of an individual parent node*. To put it differently: if the parent node has been identified, the decisions can be made, which means one subset of the model instances can be constructed, comprising all instances found under the given parent. It is important to remember that the complete set of instances of a given model node is but the union of the subsets obtained for all instances of the parent model node. This enables a decomposition of the highly complex operation of document construction into primitive operations - small building blocks of processing which implement the basic decisions. Defining a **local instantiation** of a model node as the subset of its instances sharing a particular parent node, we obtain the following picture:

- Document construction = instantiation of a document model
- Document model = a set of model nodes
- Instantiation of a model node = a set of local instantiations of a model node
- Local instantiation of a model node = result of a composite operation "local construction"
- Local construction = composition of primitive operations
- Primitive operation = implementation of a basic decision

This picture amounts to a decomposition of document construction into primitive operations. However, it does not give any hints how to implement those operations. Its practical value depends on an extension enabling their implementation. In particular, if the implementation can be derived from model and configuration data, code generation becomes feasible.

Focusing on document construction by transformation of a source document, we can translate the patterns observed in the previous section into formal concepts. Table 1, "Code snippets" showed how the basic decisions are driven by a navigation of the source document. This navigation was described as the mapping of a context of source nodes to another set of source nodes which enables the decision. A formal definition of primitive operations is prepared by defining input, output and semantics of these mappings. Besides, having realized the importance of parent node identity, we introduce the notion of a node path as a means for expressing node identity (see Table 2, "Some concepts facilitating the decomposition of document to document transformation into primitive operations.").

Table 2. Some concepts facilitating the decomposition of document to document transformation into primitive operations.

Name	Abbreviation	Property of …	Semantics
Node path	NP	Instance node	Identifies the identity of the node
Source context	SC	Node path	Given a node path, a set of source nodes enabling the construction of the target node identified by the node path
Propagated source context	$PSC_{parent\text{-}instance}$	Location	Given a location and the node path of an instance of the parent location: a set of source nodes enabling the construction of

Name	Abbreviation	Property of …	Semantics
			the location instances which are child of the node identified by the node path
Source context array	$SCA_{parent\text{-}instance}$	Location	Given a location and the node path of an instance of the parent location: an array with members being sets of source nodes; for each member, one location instance is constructed, with a source context equal to the value of the member

Primitive operations

Construction of the local instantiation of a model node is a composite operation which consumes as input the source context of the parent node. The operation can be implemented as a composition of three primitive operations described by the following table.

Table 3. Primitive operations serving as building blocks of the construction of the local instantiation of a model node.

Mapping name	Mapping input	Mapping output	Only applicable if …
context-propagator	parent-instance.SC	$Location.PSC_{parent\text{-}instance}$	-
context-distributor	$Location.PSC_{parent\text{-}instance}$	$Location.SCA_{parent\text{-}instance}$	-
context-atomizer	instance.SC	String (simple content)	L.has-simple-content=true

The **context-propagator** maps the source context of the parent instance to a sequence of source nodes called the *propagated context*, which is the "raw material" required for constructing the local instantiation of the current location. This raw material is used by the **context-distributor** to determine the number of location instances and provide each one of them with a *source context*. Each of these location instances can be constructed, as its source context is given and no other input is required: (a) if the location prescribes simple content, the text value is obtained by applying the **context-atomizer** to the source context; (b) if the model prescribes complex content, it is obtained by passing the source context to each attribute and child location, invoking its local instantiation.

It is interesting to note that the context-propagator is a pure navigation function: it maps an input sequence of nodes to an output sequence of nodes. Also the context-atomizer and the context-distributor consume a sequence of nodes. But while the context-atomizer produces a string, the output of the context-distributor is formally defined as an array of node sequences, with one member for each instance to be constructed. Note the special case of a location with cardinality 0..1, where the array is either empty or has a single member. In the most common case, such a context-distributor triggers an instance if the propagated context is non-empty, and it uses the propagated context as the source context of that instance.

The proposed definition of primitive operations provides a *bare minimum* of building blocks which can be composed into a complex document transformation. They should be regarded as primary building blocks which may be further decomposed into smaller units if this reduces overall complexity. For example, the mapping of the parent context to the propagated context may be defined as a sequence of two steps: an initial mapping, subsequently refined by an optional filter step. Such smaller units will be defined in the context of the metadata model, as they will correspond to distinct metadata items.

Composing document-to-document transformation

The following listing provides pseudo-code of a generic function `get-local-instantiation`, which constructs a local instantiation: consuming a model node and the source context of a parent instance node, returning the instances of the model node which are child nodes of that parent node.

Note the representation of the mapping functions as properties of the model node, using dot notation, for instance:

```
$location.context-propagator(…)
```

This reflects our intent to provide the primitive operations as metadata of the model node.

Figure 1. Pseudo code of `get-local-instantiation`, a function constructing the nodes of a local instantiation. An auxiliary function `new-node` is assumed to be available, which constructs a node, given the node kind, the node name and the node contents.

```
get-local-instantiation($location       as location,
                        $parent-context as node()*)
as node()* {
   $propagated-context = $location.context-propagator
                                    (parent-context = $parent-context)
   $context-array      = $location.context-distributor
                                  (propagated-context = $propagated-context)
   for $context in $context-array:
      node-for-context(location = $location,
                       context = $context)

node-for-context($location as location,
                 $context  as node()*)
as node() {
   $content =
      if $location.is-leaf:
         $location.context-atomizer(context = $context)
      else:
         for $child-location in $location.child-locations:
            get-local-instantiation(      location = $child-location,
                                    parent-context = $context)
   return
      new-node(   kind = $location.node-kind,
                  name = $location.name,
               content = $content)
}
```

Definition of a function `get-local-instantiation` amounts to a complete definition of document-to-document transformation: the transformation is achieved by applying `get-local-instantiation` to the root location of the target document. In this initial invocation, the input value of a parent context is axiomatically set to the root node of the source document.

While the primitive operations can be regarded as properties of the model nodes, the metadata expected by a code generator need not necessarily represent them one-to-one, as long as they can be *derived* from the metadata. The following section introduces a metadata model which is used by an actual code generator described later.

Metadata model SNAT (Source Navigation Annotated Target tree)

The source navigation based model associates each location with primitive operations, and it assumes that an implementation of these operations can be derived from metadata of the location. The next step towards building a code generator is therefore the design of a **metadata model**, which defines the names and semantics of metadata items. This section introduces a model which is intended to minimize

the effort required for implementing a transformation. Metadata items are represented by attributes of a location tree ([3]), so that a transformation is specified by editing a location tree augmented by metadata attributes.

Metadata item model

The following table describes the most important metadata attributes in terms of a name and semantics. It also indicates the scope of the attributes, which is the set of locations where this attribute may appear. The scope is defined by constraints imposed on the location properties. The @src item, for example, appears only on locations with simple content, and @for-each, @group-by and @sort-by appear only on locations with a maximum cardinality greater 1. By default, all attribute values are interpreted as XQuery expressions. Exceptions are enabled by special syntax rules not described here.

Table 4. The metadata model of a SNAT (Source Navigation Annotated Target tree). Each kind of metadata is modeled in terms of an attribute name, value semantics and attribute scope, which restricts the use of this attribute to locations with certain properties. The column Gen? indicates whether the attribute is generated as part of the initial SNAT tree generated from the target XSD (Y) or added to the generated SNAT tree by hand if required (N).

Attribute name	Location properties	Meaning	Example	Gen?
alt	Location of an element or attribute which is optional	XPath expression, evaluated if @src or @ctxt yields the empty sequence; the value of @alt is used as if yielded by @src or @ctxt, respectively	"#UNKNOWN"	Y
atom	Location of a simple content element or attribute	XPath expression, evaluated in a context binding variable $v to the value of an accompanying @src; if specified, the value of @atom, rather than the concatenated string values of @src, is used as element content or attribute value	substring($v, 1, 3)	N
case	Child of a choice group descriptor (`z:_choice_`)	XPath expression; the selected choice branch is the first child of `z:_choice_` whose @case has a true effective Boolean value	@Success eq "false"	Y
ctxt	Location of a complex element with maximum cardinality equal 1	XPath expression; its value is used as the propagated source context	PubInfo/Address	Y
dflt	Location of a simple content element or attribute which is mandatory	XPath expression, evaluated if @src yields the empty sequence; the value of @dflt is used as if yielded by @src	"?"	Y
for-each	Location of an element with maximum cardinality >1	XPath expression; its value is used as the propagated source context; if not accompanied by @group-by, one target instance per value item is constructed, otherwise one target instance per group of value items	books/book	Y
group-by	Location of an element	XPath expression, evaluated in the context of each value item yielded by	../author/surName	N

Attribute name	Location properties	Meaning	Example	Gen?
	with maximum cardinality > 1	@for-each; for each group of items with equal @group-by, one target instance is instantiated		
if	Location of a complex element which is optional	XPath expression, evaluated in the context of the value of an accompanying @ctxt; if specified, the element is only instantiated if the effective Boolean value of @if is true	.//(au, ed, py)	N
sort-by, sort-by2, sort-by3	Location of an element with maximum cardinality > 1	XPath expressions, optionally followed by the string "DESCENDING", used to control the order of value items received from @for-each, or the order of value item groups received from @for-each with @group-by	author/surName	N
src	Location of a simple content element or attribute	XPath expression; selects the nodes providing the information used as element content or attribute value	author/surName	Y

An example SNAT and the transformer code generated from it are shown by the following two listings.

Example 2. SNAT document, defining the transformation described as an introductory example.

```
<z:snats xmlns:z="http://www.xsdplus.org/ns/structure">
  <z:prolog/>
  <z:snat>
    <publications ctxt="books">
      <z:_attributes_>
        <updatedAt src="@lastUpdate" alt=""/>
      </z:_attributes_>
      <publication for-each="book">
        <z:_attributes_>
          <publicationYear src="py" alt=""/>
        </z:_attributes_>
        <isbn src="@isbn" dflt="'#MISSING'"/>
        <title src="title" dflt=""/>
        <creator for-each="author">
          <creatorRole src="'Author'" dflt=""/>
          <creatorName src="." dflt=""/>
        </creator>
      </publication>
    </publications>
  </z:snat>
</z:snats>
```

Example 3. XQuery code, generated from the SNAT document shown in the preceding listing.

```
let $c := *
```

```
let $c := $c/books
return
<publications>{
   let $v := $c/@lastUpdate
   return
      if (empty($v)) then () else
      attribute updatedAt {$v},
   for $c in $c/book
   return
      <publication>{
         let $v := $c/py
         return
            if (empty($v)) then () else
            attribute publicationYear {$v/string()},
         <isbn>{
            let $v := $c/@isbn
            return
               if (exists($v)) then $v/string() else '#MISSING'
         }</isbn>,
         <title>{$c/title/string()}</title>,
         for $c in $c/author
         return
            <creator>{
               <creatorRole>{'Author'}</creatorRole>,
               <creatorName>{$c/string()}</creatorName>
            }</creator>
      }</publication>
}</publications>
```

The code may be viewed as assembled from the primitive operations reflecting the metadata values. All data values, for example, are obtained by evaluating the expressions found in attributes @src, @alt and @dflt. Similarly, the propagation and distribution of the source context is guided by attributes @ctxt and @for-each. The following listing summarizes the rules how to derive the implementations of primitive operations from metadata item values.

Table 5. The primitive operations context-propagator, context-distributor and context-atomizer as implied by metadata values. Notation: v@foo is the value of the expression supplied by attribute @foo, “if @foo” means “if attribute @foo exists”, "if v@foo" means "if v@foo is not the empty sequence. The context-distributor is described informally by equating subsets of the propagated context (which is given by v@for-each or v@ctxt) with the source context (SC) of a distinct target item (TI).

Context-propagator		
if @for-each:	v@for-each	
else if (@ctxt and @alt):	if (v@ctxt) then v@ctxt else v@alt	
else if @ctxt:	v@ctxt	
else if (@src and @alt):	if (v@src) then v@src else v@alt	
else:	v@src	
Context-distributor		
if (@for-each and @group-by):	each group of items in v@for-each:	= SC of one distinct TI
else if @for-each:	each item in v@for-each:	= SC of one distinct TI
else:	all items in v@ctxt:	= SC of the only TI

Context-atomizer	
if (@atom):	v@atom
else:	string-join($source-context, " ")

By now we have decomposed document transformation into three primitive operations, and we have set up a model how to derive their implementation from a small set of metadata. In principle, the expressiveness of this metadata language is sufficient for describing arbitrary transformation. However, the benefits of the approach get quickly lost if the expressions supplied as metadata become very complex, or if the decomposition into independent expressions entails blunt repetition of non-trivial expressions. Therefore we extend the model by a few advanced features addressing these issues.

Advanced features

Several features extend the expressive power of the basic metadata model. These are:

- Value mappings
- User defined functions
- User-defined variables

Value mappings

A common requirement is the mapping of strings found in the source document to different strings to be used in the target document – for example the mapping of codes to names, or a translation between different code systems. While it would be possible to describe a value mapping by filling an @atom attribute with a switch expression, mappings are much easier to define and maintain when specified by dedicated mapping tables. Example:

```
<z:valueMap name="OperationalStatus">
  <z:entry from="true" to="Open"/>
  <z:entry from="false" to="Closed"/>
  <z:entry to="Closed"/>
</z:valueMap>
```

A SNAT document contains a prolog section where such value maps can be placed. An @atom item can invoke a particular mapping using the syntax `%map-$mapName`, where $mapName must be replaced by the name of the appropriate value map (e.g. `atom="%map-OperationalStatus"`).

User-defined functions

User-defined functions can be stored in the prolog of a SNAT document and invoked by expressions used in metadata attributes.

User-defined variables

Any location can be augmented by a binding of user-defined variables to expressions. The expression values are available when processing the descendants of the assigning node. In the following example,

```
<a:Images
   vars="group=string(. idiv 50 * 50) ; folder=concat('f', $group)"
   ...> ...
</a:Images>
```

two variables are introduced, `group` and `folder`, whose values are available when constructing the contents of an `<a:Images>` element:

```
<a:Section src="$folder"/>
```

Code generator SNAT

A SNAT-based code generator is available at github ([5]). It generates the code of document transformations defined by a SNAT document. The SNAT is prepared by hand-editing an initial version generated from an XSD description of the target documents. After changes of the XSD, the SNAT can be upgraded automatically. The upgrade ensures that the SNAT is kept in sync with the XSD. Usually, the upgrade must be finalized by a limited amount of hand-editing, for example adding manual settings for any new items.

A typical workflow is illustrated by an example in which we assume the use of BaseX ([2]) as XQuery processor. Let target documents of the desired transformation code have a root element `<publications>` and be described by an XSD `publications.xsd`. The first step is the generation of an initial version of a SNAT document (`publications.snat.xml`):

```
basex –b "request=snat?xsd=/a/b/c/publications.xsd,
                   ename=publications"
      –o publications.snat.xml
$HOME_XSDPLUS/xsdplus.xq
```

After editing the SNAT document, the transformation code (`publications.xq`) is generated:

```
basex –b "request=snat2xq?snat=publications.snat.xml"
      –o publications.xq
$HOME_XSDPLUS/xsdplus.xq
```

Given an input document (e.g. `books.xml`), the transformer can be tested, creating a target document `publications.xml`:

```
basex –i books.xml –o publications.xml publications.xq
```

After changes of the XSD, the SNAT document must be upgraded. After renaming the current SNAT document (e.g. to `publications.snat.v100.xml`), this is achieved by the following command:

```
basex –b "request=snat?xsd=/a/b/c/publications.xsd,
                   ename=publications,
                   upgrade=publications.snat.v100.xml"
      –o publications.snat.xml
$HOME_XSDPLUS/xsdplus.xq
```

As a new XSD version typically contains new elements and attributes, the automatic upgrade must be followed by hand-editing which supplies the necessary settings not yet made.

Source alignment based model

An important use case of document transformation is the consumption and construction of web service messages. Such messages often contain a variety of information items with subtle semantics, best understood by subject matter experts. These experts play a key role in the correct implementation of transformations. How can their contribution be made as efficient as possible?

Creating a SNAT document does usually not require genuine coding skills. Most entries are simple path expressions identifying source nodes by a node name or a path which is a sequence of node names. Nevertheless, often also a few more complex expressions are required, which would be difficult to supply for a person who is not a software developer.

The question arises if SNAT documents might themselves be *generated* from simpler input, which can be provided by subject matter experts without an IT background. The core of their expertise is a thorough understanding of item semantics on both sides, source and target. The expert recognizes

the alignments between source items and target items, and we are going to explore possibilities of leveraging this competence in an optimized way. The idea is to obtain SNAT documents from an identification of alignments and a minimum of additional information.

What is an alignment?

We use the term *alignment* to mean a directed relationship of dependency: a target location is aligned with a source location when the construction of target nodes which are target location instances requires information about the presence or content of source nodes which are source location instances. There may be a semantic equivalence between the aligned locations, but this is not necessarily the case. In our introductory example of a transformation, the target location `creator` which is child of a `publication` location may be regarded as semantically equivalent to an `author` location found under a `book` location. As a counter example, consider a target location `airline` whose nodes are constructed by extracting information from flight numbers provided by `flightNumber` elements: the target location `airline` depends on the source location `flightNumber`, but there is obviously no semantic equivalence.

Representation versus information

Alignments can be represented graphically: by lines connecting the symbol of a target location with the symbols of one or more source locations. This possibility enables graphical mapping tools as offered by various products (e.g. [1]). But while graphical representation is very helpful for human understanding and can facilitate human decisions, it is only a representation and cannot be equated with the information content provided by the alignments. Technically speaking, it is a frontend used for collecting information, to be distinguished from the information itself.

Annotated target tree (SAAT)

We propose to model alignments as metadata describing target locations. The alignments of a given target location consist of source location identifiers. A readible form of these are name paths like `/resources/books/book`. A straightforward representation of the alignments describing a document-to-document transformation is a location tree of the target documents, augmented by additional attributes on the location nodes which convey the name paths of the aligned source locations. Such a document we call a SAAT – Source Alignment Annotated Target tree. Here comes an example, in which the attributes supplying alignments are named `al`:

```
<z:saats xmlns:z="http://www.xsdplus.org/ns/structure">
  <z:saat>
    <publications al="/resources/books">
      <z:_attributes_>
        <updatedAt al="/resources/books/@lastUpdate"/>
      </z:_attributes_>
      <publication al="/resources/books/book">
        <z:_attributes_>
          <publicationYear al="/resources/books/book/py" />
        </z:_attributes_>
        <isbn al="/resources/books/book/isbn"/>
        <title al="/resources/books/book/title"/>
        <creator al="/resources/books/book/author">
          <creatorRole/>
          <creatorName al="/resources/books/book/author"/>
        </creator>
      </publication>
    </publications>
  </saat>
<saats>
```

Compare this to the SNAT tree specifying the transformation precisely, in which all attributes describing navigation are highlighted:

```
<z:snats xmlns:z="http://www.xsdplus.org/ns/structure">
  <z:snat>
    <publications ctxt="books">
      <z:_attributes_>
        <updatedAt src="@lastUpdate" alt=""/>
      </z:_attributes_>
      <publication for-each="book">
        <z:_attributes_>
          <publicationYear src="py" alt=""/>
        </z:_attributes_>
        <isbn src="@isbn" dflt="'#MISSING'"/>
        <title src="title" dflt=""/>
        <creator for-each="author">
          <creatorRole src="=Author" dflt=""/>
          <creatorName src="." dflt="'?'"/>
        </creator>
      </publication>
    </publications>
  </snat>
<snats>
```

In a SNAT tree, navigation is described by **expressions**, to be evaluated at runtime in a current context implied by the preceding steps of navigation. Alignments, on the other hand, are static **identifiers**, not expressions (in spite of the apprearance). Nevertheless, the example exhibits a striking correspondence between the name paths describing alignments and navigation expressions.

Table 6. A comparison between navigation steps and alignments.

Target location	Alignment	Navigation	Navigation attribute
`publications`	**/resources/books**	/resources/books	ctxt
`. @updatedAt`	/resources/books/**@lastUpdate**	@lastUpdate	src
`. publication`	/resources/books/**book**	book	for-each
`. . @publicationYear`	/resources/books/book/**py**	py	src
`. . isbn`	/resources/books/book/**isbn**	isbn	src
`. . title`	/resources/books/book/**title**	title	src
`. . creator`	/resources/books/book/**author**	author	for-each
`. . . creatorRole`	-	-	-
`. . . creatorName`	/resources/books/book/**author**	.	src

Corresponding values of alignment and navigation are marked by a subset relationship: the navigation leads to a set of nodes which is a subset of the nodes represented by the corresponding alignment. So, for example, navigation `author` (which is short for `child::author`) is evaluated in the context of an individual `book` element and selects all `author` nodes found under that `book` node. This is a subset of the nodes represented by the alignment `/resources/books/book/author`, which is the set of *all* `author` nodes found under any `book` node.

Mapping alignments to navigation

A SNAT document describes an intended transformation precisely, whereas alignment is less precise. Consider, for example, the relationship between `publication`, `isbn` and `title` nodes. The SNAT

document guarantees that each source `book` is mapped to a distinct `publication` and that the `isbn` and `title` children of a `publication` element refer to the same `book` source element. This expectation is very intuitive, and therefore it may go unnoticed that the alignments shown above do not make these commitments: the alignments would not be contradicted by a transformation where a `publication` contains a `title` taken from one `book` and an `isbn` taken from another `book`, or (if we ignore cardinality constraints) a `publication` containing all `title` and all `isbn` values found in any book.

The core of this problem lies in **semantic relationships** pertaining to structural relationships in general, and parent-child relationships in particular. The node name `title`, for instance, indicates that node contents represent a title, but it does not hint at a semantic relationship between nodes, which constrains the child `title` to represent a property of the parent `publication`. (Were the element name `hasTitle`, this would be slightly different.) The SNAT model of a transformation preserves the intended semantic relationships implicitly: via navigation. Navigation to a *particular* `title` as well as to a particular `isbn`, executed while constructing an individual target `publication`, ensures that the target document expresses the intended semantics.

The question arises if we cannot set up rules how to map alignments to navigation steps. Table 6, “A comparison between navigation steps and alignments.” clearly suggests such a possibility. It should be remembered that the SNAT model is essentially a decomposition of transformation into navigation steps, or, the other way around, an integration of navigation steps into a complete picture of a transformation. If we can infer navigation steps from pairs of alignments (the alignments of parent and child locations), we have come very close to solving the problem of mapping alignments to a complete picture of the transformation. The task of generating transformation code from alignments can be redefined as the task of mapping a SAAT tree to a SNAT tree.

Infering context propagation

As discussed in previous sections, the SNAT model of a transformation defines three primitive operations – context-propagator, context-distributor, context-atomizer. Among these, it is the context-propagator which represents navigation: the mapping of a set of source nodes to another set of source nodes. The input of this mapping is the source context of an individual target node; the output of this mapping is the (collected) source context of the target nodes which are instances of a particular child location of the target node's location.

Semantic versus structural relationships

Consider the construction of a `publication` node, which has a single `book` node as source context. The context-propagator of the `title` child location should *not* select all `title` nodes found in the source document, but a *particular* `title` – the one which is child of the particular `book` providing the source context of the `publication`. This requirement captures a **semantic relationship** between the `publication` and `title` nodes under construction. If target node P and target node C have a certain semantic relationship R (here: C gives the title of P), then the source context of P and the source context of C should have the same semantic relationship R (the source context of C should give the title of the source context of P). Mapping alignments to navigations (thus: SAAT to SNAT) means **filtering the alignment**: selecting those instances of an alignment which have a particular semantic relationship with the source context of the parent instance. The semantic relationship is implicit - implied by a structural relationship, the parent-child relationship defined as part of the target document model.

As the semantic relationship guiding navigation is implicit, implied by a structural relationship between target nodes (parent-child), it must also be derived from a *structural relationship* between sources nodes. We have arrived at the general question: **how is the semantic relationship pertaining to a given parent-child pair in the target document model translated into a structural relationship connecting source nodes?** More precisely, we are looking for a rule mapping the source context of the parent node to the relevant subset of the source alignment of the child location. (Note that the source context of the parent node is in turn a subset of the source alignment of the parent node.)

We start by considering a particular case for which an intuitive solution is easily found. Let P and C denote two target locations and C be a child location of P, that is, C represents child nodes of P.

(Example: P = `publication`, C = `title`). Assume that the alignment of C is a child location of the alignment of P. (Example: alignment of P: `book`, alignment of C: `title`.) Then we may expect the parent-child relationship defined by the target document model to have similar semantics as the parent-child relationship between their alignments, as defined by the source document model. (Example: the child gives the title of the parent.) So we set up a first, preliminary rule: “if the alignments of parent and child locations are themselves parent and child locations, then set the context-propagator of the target child location to select those instances of the aligned location which are child nodes of the source context of the parent node.” In our example, this rule would yield the desired result: the source context of the target `title` would be the child node of the source node serving as the source context of the target `publication` node.

Parent-child relationships are often modified by the insertion of intermediate nodes providing some sort of grouping. Therefore we might generalize our preliminary rule to select all instances of the target child locations’s alignment which are *descendants* of the target parent node’s souce context. While this will cover many cases, this is a far cry from a general solution, as hierarchical relationships need not be preserved by the transformation. As an example, consider target documents focusing on authors and describing publications as child elements of the node representing an author

Shortest-path-principle

A general approach to the mapping of alignments to navigation must be based on a generalized way of mapping semantic relationships assumed in the target model to structural relationships observed in the source model. Given a parent and a child location from the target model, we assume a semantic relationship between these locations and we also assume a similar semantic relationship to exist between the alignment of the target parent location and the alignment of the target child location. The cases where the alignments of a parent-child pair are themselves a parent-child pair or an ancestor-descendant pair should be handled as expected, yet regarded as special cases of a general rule.

The approach requires a *definition of the structural relationship* existing between two sets of locations. The relationship is expressed as a navigation mapping loctions from the first set to locations from the second set. Note that when applied to *nodes* which are instances of the locations of the first set, the chosen navigation path determines which *subset* of the nodes which are instances of the locations of the second set are reached. For example, compare the path `child::title` with `parent::books/child::book/child::title`. Applied to a location `book`, both paths lead to child location `title`. Applied to a node which is an instance of location `book`, the first path selects only child elements of the `book` node, whereas the second path matches all `title` elements under any `book`. Note that the rule how to construct the navigation path from parent alignment to child aligment is arbitrary, as there is no conceptual base for deciding the "correctness" of the solution. The challenge is to define a rule which matches intuitive expectations best and under the broadest set of circumstances.

We define the structural relationship between the alignments of parent and child locations as the **shortest navigation path** leading from the parent alignment (one or more source locations) to the child alignment (one or more source locations). The shortest path between two locations A and B is defined as follows: (a) if A is descendant (ancestor) or B, the shortest path is along the descendant:: (ancestor::) axis; (b) otherwise the shortest path is a prefix leading from A to the nearest common ancestor C, followed by the path from C to B along the descendant:: axis. The following table compiles a few examples.

Table 7. Application of the “shortest-path-principle”.

Source alignment of parent node	Source alignment of child node	Shortest path
A/B/C	A/B/C/D	D
A/B/C	A/B/C/D/E	D/E
A/B/C	A/B	parent::B
A/B/C	A	parent::B/parent::A
A/B/C	A/B/X/Y	parent::B/X/Y

Future work

The shortest-path-principle needs elaboration. When the alignments of the target parent and child locations comprise both several source locations, inferred navigation is *not* the union of all navigation paths leading from any source location of the parent alignment to any source location of the child alignment. To illustrate the problem, let us assume that the alignment of parent location `publication` comprises `book` and `journal`, and the alignment of child location `title` comprises `title` under `book` and `journalTitle` under `journal`. Given a parent `publication` with a source context consisting of a `book` node, we want to ignore the shortest path from `book` to `journalTitle`. The example shows that the shortest-path-principle requires further elaboration which is work in progress.

Metadata model SAAT (Source Alignment Annotated Target tree)

In the preceding section we have shown that alignments can be mapped to navigation which plays the role of context propagation, as defined by the SNAT model of document transformation. What is the potential value of a location tree augmented by metadata specifying source alignments? This would be the minimal version of a **SAAT** tree, a Source Alignment Annotated Target tree.

Minimal SAAT

A SAAT tree can be mapped to a SNAT tree, and therefore it implies a precisely defined document to document transformation:

- *Propagation of the source context* is derived from source alignments according to the shortest-path-principle; this enables populating the SNAT metadata related to context propagation (@ctxt, @for-each, @src)
- *Distribution* of the propagated source context over target instances is decided by a default rule (one target instance per node belonging to the propagated source context, unless the maximum cardinality of the target domain is one); SNAT metadata @group-by and @sort-by are not used
- *Atomization* of the source context is decided by a default rule (string values of all context nodes are concatenated, using a blank as separator)

No matter how sensible the defaults are, they are only defaults, meeting the actual requirements in some cases and not meeting them in others. A minimal SAAT does define a transformation precisely, but it cannot be used for expressing arbitrary transformations.

Alignment qualifiers

This limitation may be addressed by **alignment qualifiers** – metadata modifying the interpretation of the alignments. In particular, the propagation, distribution and atomization of the source context might be adapted to special requirements not handled by the default rules. This approach requires a **metadata model** defining the names and semantics of “aqua items”, metadata items playing the role of alignment qualifiers. Embarking on this task, one should remember a basic difference between the SNAT model and the SAAT model: the SNAT model is based on expressions and closer to a software developer’s perspective; the SAAT model focuses on alignments which can be identified by a subject matter expert. Simplicity is an important benefit of the SAAT approach, without which it might not be worthwhile, considering the superior expressiveness of the SNAT model. The definition of a SAAT-based metadata model may be perceived as the challenge to find a tradeoff between simplicity and expressiveness.

An important usecase of SAAT models is the support of graphical mapping tools: the graphical representation of a document to document transformation can be captured by a SAAT tree, which

can in turn be mapped to a SNAT tree providing both – a *specification* of the transformation, and its *implementation* (obtained as output of a SNAT consuming code generator).

The construction of a SAAT based metadata model is work in progress, geared towards supporting a graphical mapping tool which is part of a framework supporting data transformations between large type systems.

Code generator SAAT

A simple SAAT-based source code generator is integrated into the SNAT-based source code generator described in the section called “Code generator SNAT”. Currently, the metadata model is restricted to alignments (@al attributes), and alignment qualifiers are not yet supported. Code generation is based on a transformation of the supplied SAAT document into a SNAT document. The SAAT is prepared by hand-editing an initial version generated from an XSD description of the target documents. Let target documents have a root element `<publications>` and be described by an XSD `publications.xsd`. An initial version of a SAAT document is generated by the following call:

```
basex –b "request=saat?xsd=/a/b/c/publications.xsd,
                   ename=publications"
      –o publications.saat.xml
$HOME_XSDPLUS/xsdplus.xq
```

After editing the SAAT document, the transformation code (`publications.xq`) is generated:

```
basex –b "request=saat2xq?saat=publications.saat.xml"
      –o publications.xq
$HOME_XSDPLUS/xsdplus.xq
```

Given an input document (e.g. `books.xml`), the transformer can be tested, creating a target document `publications.xml`:

```
basex –i books.xml –o publications.xml publications.xq
```

Changes of the XSD require an upgrade of the SAAT document. After renaming the current SAAT document (e.g. to `publications.saat.v100.xml`), this is achieved by the following command:

```
basex –b "request=saat?xsd=/a/b/c/publications.xsd,
                   ename=publications,
                   upgrade=publications.saat.v100.xml"
      –o publications.saat.xml
$HOME_XSDPLUS/xsdplus.xq
```

Like an upgraded SNAT document, an upgraded SAAT document must usually be hand-edited in order to supply the settings required for new elements and attributes.

AT Map Machine

The previous sections explained the use of annotated target tree models for generating the source code of document transformations. Now we explore the possibility of generalization, enabling the transformation of non-XML data sources into XML documents. We propose the idea of an annotated target tree as a concept supporting the generation of various kinds of data mappers.

Does SNAT presuppose XML?

A SNAT-based code generator is driven by metadata enabling primitive operations, which serve as the functional building blocks of a document transformation. These operations map a node sequence to a value which is another node sequence (context-propagator), an array of node sequences (context-distributor) or a string (context-atomizer). Does the central role played by node sequences mean that the SNAT model is restricted to XML data sources?

The apparent limitation can be overcome if we generalize the notion of a “node sequence” to mean a *sequence of distinct items*. The following table gives a few examples.

Table 8. Examples for a possible generalization of the node concept, meaning distinct items of which the resource is composed.

Source data media type	What is a "node"?
HTML	DOM node
JSON	JSON item (object, array, string, number, boolean)
CSV	table, row, cell
SQL	db, table, row, column
RDF	RDF node

For each media type based on distinct items of information a SNAT-based code generator may be defined, following a scheme of actions discussed below. Note, however, that with some media types (e.g. HTML, JSON, CSV) a simpler approach to transformation is to use an XML representation of the original input, readily obtained by Open Source products (like BaseX ([2]) with its extension functions for parsing non-XML resources into XML). Such a trivial preprocessing step enables the reuse of code generators for XML-to-XML transformation (like the one described in the section called “Code generator SNAT”) for non-XML data sources.

Does SAAT presuppose XML?

A source alignment based model (as defined in section "Source alignment based model") is a set of rules how to generate a SNAT document from a SAAT document. A SAAT document describes the alignment of target locations with source locations and thus presupposes a location tree describing the source data set. However, the concept of a location tree has not yet been defined for non-XML resources. For this reason, an exploration of SAAT-based code generation for the transformation of non-XML data sources is beyond the scope of this paper.

Scheme for building SNAT-based code generators

The concept of a SNAT-based code generator spans two layers:

- The ground layer is the decomposition of document construction into primitive operations
- The top layer is a metadata model enabling the implementation of those primitives

The ground layer is independent of a particular source data media type, as long as the source data can be viewed as a collection of distinct items. The metadata model, on the other hand, must specify the construction of code which can be applied to sets of source data items. Such code will usually rely on a particular media type. We conclude that

- A SNAT-based code generator is tied to a particular source data media type
- The dependency is restricted to the metadata model

Note. As mentioned before, the dependency on a single media type can be removed by a preliminary step translating various media types into a single media type whose transformation is natively supported. For example, an XML-dependent code generator can be easily extended to support JSON, CSV and HTML data sources.

In general, the task of designing a SNAT-based code generator can be redefined as the design of a metadata model which enables a generic implementation of the primitive operations using metadata as sole input. The model comprises three submodels:

- **metadata item model** – defines the names and semantics of metadata items
- **metadata value model** – defines the interpretation of metadata item values
- **code assembly model** – defines how to assemble the source code of primitives from metadata

Metadata item model

Metadata items fall into two categories: *location* metadata items and *prolog* metadata items. Location metadata items are properties of an individual location. Prolog metadata items set up a context facilitating the specification and interpretation of location metadata items. The metadata model described in section "Metadata model SNAT", for instance, defines various kinds of location metadata items (see table "Metadata items SNAT") and two kinds of prolog metadata.

Metadata value model

The metadata model must specify how to interpret the metadata item values. Note that the interpretation may depend on the kind of metadata item. There are many possibilities, for example:

- Expressions from a standardized query language (XQuery, SQL, SPARQL, ...)
- Code from an imperative programming language (Java, Python, ...)
- Expressions from a purpose-defined language
- Data structures (XML, JSON, CSV, ...)
- Atomic values (strings, numbers, …)
- Invocations passing arguments to functions defined by the metadata model
- Any combination of the above

The metadata value model described in section "Metadata model SNAT" uses the following combination:

- Location metadata items: the value is interpreted as XQuery expression, unless syntactically marked up as invocation of a mapping function (reflecting a user-supplied value mapping) or one of a set of functions defined by the metadata model (functions not described in this paper)
- Prolog metadata items: the value is interpreted as a data structure (item kind `valueMap`) or an XQuery function (item kind `function`)

Code assembly model

The code assembly model is a set of rules which map location metadata items to implementations of the three primitive operations defined by the SNAT model - context-propagator, context-distributor and context-atomizer. As an example, table "The assembly of primitive operations" summarizes the code assembly model of the code generator described in section "Metadata model SNAT".

Using SNAT as a meta model

We have introduced the concept of a SNAT (Source Navigation Annotated Target tree) as a document which specifies the transformation of source data sets of a particular kind into instances of a particular document model. A SNAT document is a location tree obtained from the target document model and augmented by metadata. The names, semantics and scope of the metadata items are defined by a metadata model (see for example table "Metadata items SNAT"). Relying on such a metadata model, a SNAT-based code generator transforms a SNAT document into executable code implementing primitive operations and composing them into the transformation as a whole.

Defined in such general terms, the concept of a SNAT-based code generator becomes an abstract model which may guide the design of concrete code generators, distinguished by a particular metadata model and expecting source data with a particular media type. To illustrate this possibility, we introduce a SNAT-based code generator for constructing XML documents from RDF data.

Proof of concept: RDF-to-XML, SNAT-based

This sections sketches a simple code generator for RDF-to-XML transformations, which is driven by a Source Navigation Annotated Target tree. As a SNAT-based code generator is distinguished by a particular metadata model, the description can be structured according to the main parts of the metadata model.

Metadata item model

Although dealing with RDF data sources, we may reuse the metadata item model introduced in section "Metadata model SNAT" virtually unchanged. Again, table "Metadata items SNAT" gives us the names of the most important metadata items and describes the role of the item values during document construction. For example, @ctxt and @for-each items provide the mapping of a current context (set of RDF resources) to a new context (another set of RDF resources); and @src items map a context to the RDF nodes supplying data values. Note, however, that with RDF data sources the metadata item *values* cannot be XQuery expressions, as RDF nodes are not part of the XQuery data model, so that XQuery expressions cannot consume RDF nodes.

Metadata value model

The new code generator must therefore define a *new metadata value model*. A possible approach would be to define metadata item values to be SPARQL expressions. This would work, as SPARQL expressions are capable of mapping a current context of RDF nodes to the output required by context-propagator, -distributor and –atomizer. But the difference between XQuery and SPARQL expressions with regard to simplicity and intuitiveness should be considered. With XQuery, typical annotations are small path expressions, which are easy to write and to understand. Dealing with SPARQL equivalents, this ease of writing and reading will be lost, and it must be questioned if a tree studded with SPARQL expressions will be appreciated as a straightforward way of describing an RDF to XML transformation.

Guided by these considerations, we designed a simple path notation for specifying navigation within an RDF graph in a way which is similar to XPath navigation. This tiny expression language, **RPath**, reuses the basic concepts of XPath: navigation axes, name tests and predicates. The following listing gives a few examples of RPath expressions and their translation into SPARQL queries.

Example 4. Examples of SPARQL queries generated for RPath expressions.

```
###
### SPARQL generated for RPATH:
###    lib:author
###
PREFIX a: <file:///C:/projects/seat/resources-markupuk/markupuk-input.xml#>
PREFIX lib: <http://www.example.org/ns/domain/library#>

SELECT DISTINCT ?newContext WHERE {
?context lib:author ?newContext .
FILTER (?context IN ( a:n1.1.1, a:n1.1.2 )) . # the current context

###
### SPARQL generated for RPATH:
###    //lib:book/lib:author
###
PREFIX a: <file:///C:/projects/seat/resources-markupuk/markupuk-input.xml#>
PREFIX zzz: <http://www.xsdplus.org/ns/internal#>
PREFIX lib: <http://www.example.org/ns/domain/library#>

SELECT DISTINCT ?newContext WHERE {
?context (!(zzz:NEVER)+)/lib:book ?value1 .
?value1 lib:author ?newContext .
FILTER (?context IN ( a:root-elem )) . # the current context

###
### SPARQL generated for RPATH:
###    ancestor::lib:books
###
```

```
PREFIX a: <file:///C:/projects/seat/resources-markupuk/markupuk-input.xml#>
PREFIX zzz: <http://www.xsdplus.org/ns/internal#>
PREFIX lib: <http://www.example.org/ns/domain/library#>

SELECT DISTINCT ?newContext WHERE {
?context ^(!(zzz:NEVER)+) ?newContext .
?newContext ^!(zzz:NEVER)/lib:books ?newContext .
FILTER (?context IN ( a:n1.1.1, a:n1.1.2 )) . # the current context
```

Using RPath one may concisely express graph navigation as it is typical when mapping RDF data to XML. However, RPath does not support advanced requirements. When they surface, SPARQL queries can be used as fallback: the metadata value model allows the use of both, RPath expressions and SPARQL queries.

Code assembly model

The code assembly model uses XQuery as the target language, and the generated code is very similar to the code generated for the transformation of XML input. The fact that RDF nodes, rather than XML nodes, play the role of key intermediates (the source context) does not pose a problem. RDF resources are represented by their URI (a string), and RDF values are represented by their string representation. The code generator translates any RPath expressions into SPARQL queries, collects all SPARQL queries in a library and associates each one with a query key. The generated XQuery code submits the appropriate query key along with the current context to a generic function (`updContext`) which resolves the query to a new context or a single data value, dependent on the primitive operation being executed.

Example SNAT and source code

A small example will illustrate the use of the code generator. Let the goal again be the construction of a `publications` document, but this time using an RDF data source. The following two listings show a SNAT document and the source code generated from it.

Example 5. SNAT document, defining the transformation of RDF data into an XML document.

```
<z:snats xmlns:z="http://www.xsdplus.org/ns/structure">
  <z:prolog>
    <z:nsMap>
      <z:ns prefix="lib" uri="http://www.example.org/ns/resources/library#"/>
    </z:nsMap>
  </z:prolog>
  <z:snat>
    <publications ctxt="$libs"
      <z:_attributes_>
        <updatedAt src="lib:lastUpdate" alt=""/>
      </z:_attributes_>
      <publication for-each="descendant::lib:book">
        <z:_attributes_>
          <publicationYear src="lib:py" alt=""/>
        </z:_attributes_>
        <isbn src="lib:isbn" dflt="'#MISSING'"/>
        <title src="lib:title" dflt=""/>
        <creator for-each="lib:author">
          <creatorRole src="'Author'" dflt=""/>
          <creatorName src="." dflt=""/>
        </creator>
```

```
      </publication>
    </publications>
  </z:snat>
</z:snats>
```

Example 6. XQuery code, generated from the SNAT document shown in the preceding listing.

```
declare variable $dataSources as element(rx:dataSources) external;
declare variable $sparqlLib :=
<sparqlLib>
  <sparql key="$libs" initialContext="$libs"/>
  <sparql key="descendant::lib:book">...</sparql>
  <sparql key="lib:author">...</sparql>
  <sparql key="lib:books/lib:lastUpdate">...</sparql>
  <sparql key="lib:isbn">...</sparql>
  <sparql key="lib:py">...</sparql>
  <sparql key="lib:title">...</sparql>
<sparqlLib>;

declare function f:updContext($dataSources as element(rx:dataSources),
                              $context as xs:string*,
                              $queryKey as xs:string) as xs:string* {...};

let $c := f:updContext($dataSources, (), "$libs")
return
<publications>{
    let $v := f:updContext($dataSources, $c, "lib:books/lib:lastUpdate")
    return
        if (empty($v)) then () else
        attribute <updatedAt {$v},

    for $c in f:updContext($dataSources, $c, "descendant::lib:book")
    return
        <publication>{
            let $v := f:updContext($dataSources, $c, "lib:py")
            return
               if (empty($v)) then () else
               attribute publicationYear {$v},
            <isbn>{
               let $v := f:updContext($dataSources, $c, "lib:isbn")
               return
                  if (exists($v)) then $v else '#MISSING'
            }</isbn>,
            <title>{f:updContext($dataSources, $c, "lib:title")}</title>,

            for $c in f:updContext($dataSources, $c, "lib:author")
            return
               <creator>{
                  <creatorRole>Author</creatorRole>,
                  <creatorName>{$c}</creatorName>
               }</creator>
        }</publication>
}</publications>
```

The XQuery source code has the same structure as the code generated for XML data sources (compare example "XQuery code generated for XML data source". Note the difference - using an XML data

source, navigation is accomplished by path expressions, whereas using an RDF data source, navigation is accomplished by submitting SPARQL queries referenced by query keys to a generic evaluator. The original RPath expressions are used as query keys, which enhances the readability of the code.

The input of the transformation is an XML document identifying the RDF graph and a set of resources bound to user-defined names. Example:

```
<rx:dataSources xmlns:rx="https://www.xsdplus.org/ns/...">
  <rx:namespace prefix="lib" uri="http://www.example.org/ns/..."/>
  <rx:graph name="main" uri="localhost:9093//...">
    <rx:resource uri="lib:rwth" name="libs"/>
    <rx:resource uri="lib:wage" name="libs"/>
    <rx:resource uri="lib:jota" name="libs"/>
  </rx:graph>
</rx:dataSources>
```

The metadata expressions can reference the resources via variable references (e.g. `ctxt="$libs"` and thus use them as starting points of navigation.

Bibliography

[1] *Altova MapForce - tool for data mapping, conversion and ETL.*. Altova. 2018. https://www.altova.com/de/mapforce.

[2] *BaseX – The XML Framework. Lightweight and High-Performance Data Processing.*. BaseX. 2018. http://basex.org.

[3] *Location trees enable XSD based tool development.*. Hans-Juergen Rennau. 2017. http://xmllondon.com/2017/xmllondon-2017-proceedings.pdf.

[4] *SPARQL 1.1 Query Language*. World Wide Web Consortium (W3C). 2013. https://www.w3.org/TR/sparql11-query/.

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