Determining the Output Schema of an XSLT Stylesheet

Sven Groppe and Jinghua Groppe

University of Innsbruck, Technikerstrasse 21a, A-6020 Innsbruck, Austria
{Sven.Groppe, Jinghua Groppe}@uibk.ac.at

Abstract. The XSLT language is used to describe transformations of XML documents into other formats. The transformed XML documents conform to *output schemas* of the used XSLT stylesheet. Output schemas of XSLT stylesheets can be used for a static analysis of the used XSLT stylesheet, to automatically detect the XSLT stylesheet, which has been used for the transformation, of target XML documents or to reason on the output schema without access to the target XML documents. In this paper, we describe how to automatically determine such an output schema of a given XSLT stylesheet, where we only consider XML to XML transformations. The input of our proposed output schema generator is the XSLT stylesheet and the schema of the input XML documents. The experimental evaluation shows that our prototype can determine the output schemas of nearly all typical XSLT stylesheets.

1 Introduction

Among other usages of XML, XML is the most widely used data model for exchanging data on the web and elsewhere. For the exchange of data, we have to transform the data from one format into another format whenever the two exchange partners use different formats. The exchange partners can use different formats, which might be a proprietary company standard, a proprietary application format or other standard formats, for historical, political or other reasons. We focus on XSLT [23] as transformation language for the XML data. In this paper, we describe how to determine output schemas of XSLT stylesheets, where we use XML Schema [24] to express the output schema. We define the term *output schema* as follows:

Definition 1 (*output schema*): The output schema `OS` of an XSLT stylesheet `V` must be valid for all results of the XSLT stylesheet `V`, i.e. we require the following to hold: ∀ XML documents `D`: `V(D)` is valid according to `OS`.

Output schemas of XSLT stylesheets can be used in several application scenarios:

In the first application scenario, we have an already transformed XML document and a collection of XSLT stylesheets. In this application scenario, we want to exclude those XSLT stylesheets, which were not used to retrieve the given transformed XML document. This application scenario is useful, whenever it is unknown, which XSLT stylesheet from a set of XSLT stylesheets has been used for transformation. However, the user wants to execute this unknown XSLT stylesheet (which is available in a set of

XSLT stylesheets) again, because the source data has been changed. The output schema of the XSLT stylesheets can be used for this test in the following way: If the given transformed XML document is *not* valid according to the output schema of the XSLT stylesheet, then we are sure that this XML document was not transformed by the considered XSLT stylesheet.

Another application scenario is the optimization of the execution times of XSLT stylesheets. The contributions [10] and [12] describe a search on the output nodes of XSLT stylesheets as part of a static analysis of the XSLT stylesheets. This search could be analogously done in the output schemas of the XSLT stylesheets.

Within further application scenarios (see Figure 1), we use a given XSLT stylesheet as XML view. Then we can use the output schema of the given XSLT stylesheet to reason on it by using a satisfiability tester, an intersection tester, a containment tester (which is sometimes called subsumption tester) or an equivalence tester. The input of these testers is the output schema and queries. Figure 1 summarize some application scenarios, where these testers can be used to reason on the output schema and given queries for a special purpose, which is also described in the table of Figure 1 .

Application Scenario	**Used Tester**				**Description**
	?	∩	⊆	≡	
Querying a cache of original data or of transformed data		x	x	x	Caches are designed to return answers faster compared to querying the original source (and optionally transforming its data) so that the answer time is reduced whenever results can be retrieved from the cache. Given a query `Q` and the content of the cache described by a query `C`. `Q` can be partially answered by the cache if proving `Q∩C≠{ }` is successful. `Q` can be fully answered by the cache if `Q⊆C` can be proved. The content of the cache can be returned as answer of `Q` if proving `Q≡C` is successful.
Update of the original data of a cache		x			Given an update query `U` and the content of the cache described by a query `C`. `U` describes the part of the original data that has been updated. We can check whether the content of the cache is valid by proving `U∩C={ }`.
Querying distributed sources		x			Given a query `Q` and the content of a source described by a query `C`. We can avoid querying the source if `Q∩C={ }` can be proved. Avoiding querying the source saves transportation costs and reduces the load of the processor where the source is located.
Optimization of disjunctive queries			x		We can optimize a disjunctive query `Q1|Q2` to `Q2` if `Q1⊆Q2`. We can save processing costs by applying this optimization.
Access control		x			We can check access rights of users by using the approach of [3].
Optimization of answering queries	x				Given a query `Q`. We can avoid querying a source (or a view respectively) if we can prove `Q={ }`. Particularly in the case of views with cost intensive operations, we can save processing costs.
Check of user query	x				We can check a user query `Q` by testing `Q={ }`, which is a hint that the user query `Q` is not meaningful.

Fig. 1: Application scenarios of the testers, where ? represents the satisfiability tester, ∩ represents the intersection tester, ⊆ represents the containment tester and ≡ represents the equivalence tester

We have implemented the output schema generator for XSLT stylesheets in a prototype. The experimental evaluation shows that our prototype can determine 95% of the output schemas of typical XSLT stylesheets.

The rest of the paper is organized as follows. Section 2 describes the algorithm of the output schema generator. Section 3 presents the experimental evaluation. Section 4 deals with the related work. We end up with the summary in Section 5.

2 Determining the Output Schema

The goals of output schema generators are to compute the output schema definition as restrictive as possible, but to guarantee the requirement of Definition 1. We describe the output schema generator used in our prototype in the following sections in detail.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:template match="table">
    <table><xsl:apply-templates/></table>
  </xsl:template>
  <xsl:template match="row">
    <address id="{id}" firstname="{firstname}" lastname="{lastname}"
             street="{street}" city="{city}" state="{state}" zip="{zip}"/>
  </xsl:template>
</xsl:stylesheet>
```

Fig. 2: The XSLT stylesheet `avts.xsl` of the XSLTMark benchmark [6]

```
<xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'>
 <xsd:group name='name0'>
  <xsd:sequence>
   <xsd:element name='table'>
    <complexType mixed='true'>
     <xsd:sequence>
      <xsd:choice>
       <xsd:group ref='name0' minOccurs='0' maxOccurs='1'/>
       <xsd:group ref='name1' minOccurs='0' maxOccurs='1'/>
      </xsd:choice>
     </xsd:sequence>
    </complexType>
   </xsd:element>
  </xsd:sequence>
 </xsd:group>
 <xsd:group name='name1'>
  <xsd:sequence>
   <xsd:element name='address'>
    <complexType mixed='true'><xsd:sequence></xsd:sequence></complexType>
      <xsd:attribute name='id' type='xsd:string'/>
      <xsd:attribute name='firstname' type='xsd:string'/>
      <xsd:attribute name='lastname' type='xsd:string'/>
      <xsd:attribute name='street' type='xsd:string'/>
      <xsd:attribute name='city' type='xsd:string'/>
      <xsd:attribute name='state' type='xsd:string'/>
      <xsd:attribute name='zip' type='xsd:string'/>
   </xsd:element>
  </xsd:sequence>
 </xsd:group>
 <xsd:sequence>
  <xsd:choice><xsd:group ref='name0' minOccurs='0' maxOccurs='1'/></xsd:choice>
 </xsd:sequence>
</xsd:schema>
```

Fig. 3: Output schema of the XSLT stylesheet of Figure 2 computed with the output schema generator

Example 1 (*Output schema of XSLT stylesheet*): Figure 3 presents the output schema of the XSLT stylesheet of Figure 2 computed with the output schema generator. The output schema generator expects name attributes in the `<xsl:template>` instructions, a start template matching "/" (in Figure 2 hidden as built-in template as specified in [23]) and long forms when generating element or attribute nodes, i.e.

```
<xsl:element name="address">
  <xsl:attribute name="id"><xsl:value-of select="id"/></xsl:attribute>
</xsl:element>
```

instead of `<address id="{id}"/>`. Note that XSLT stylesheets in general and in particular the XSLT stylesheet of Figure 2 can be easily transformed into such a representation.

2.1 Processes of the determination of output schemas of XSLT stylesheets

The processes of the determination of the output schemas of XSLT stylesheets is presented in Figure 4. First, we have to pre-process the schema of the input XML documents of the XSLT stylesheets (see Section 2.2). Second, we parse an expression, which is here an XSLT stylesheet. The result of the parsing is the abstract syntax tree of the expression. For example, Figure 5 presents the abstract syntax tree of the XSLT stylesheet of Figure 2. Third, we evaluate the attribute grammar described in Section 2.3 on this abstract syntax tree, which computes the attributes of the nodes in the abstract syntax tree in the way we describe in Section 2.3. A special attribute of the root node of the abstract syntax tree contains the result after the overall evaluation of the attribute grammar. Fourth and at last, we have to post-process the determined output schema, which we describe in Section 2.4.

2.2 Preprocessing the XML Schema definition of the input XML document

In order to integrate parts of the XML Schema definition of the input XML document, we perform the following tasks in the XML Schema definition of the input XML document. We first embed all element declarations `<xsd:element name=N1>` on the top level of the XML Schema definition of the input XML document in `<xsd:group name=N2>` elements so that we can refer to these element declarations. In order to correct the references to the embedded element declarations, we replace all `<xsd:element ref=N1>` elements with `<xsd:group ref=N2>` elements. Furthermore, we rename all names, which are also used in the generated output schema so that they are not in conflict any more with the names of the generated output schema. We will embed all referred groups of the XML Schema definition of the input XML document in the output schema.

Example 2: Figure 7 presents the pre-processed XML Schema definition of Figure 6.

2.3 Attribute Grammar

For the description of the algorithm of the output schema generator, we use *attribute grammars*.

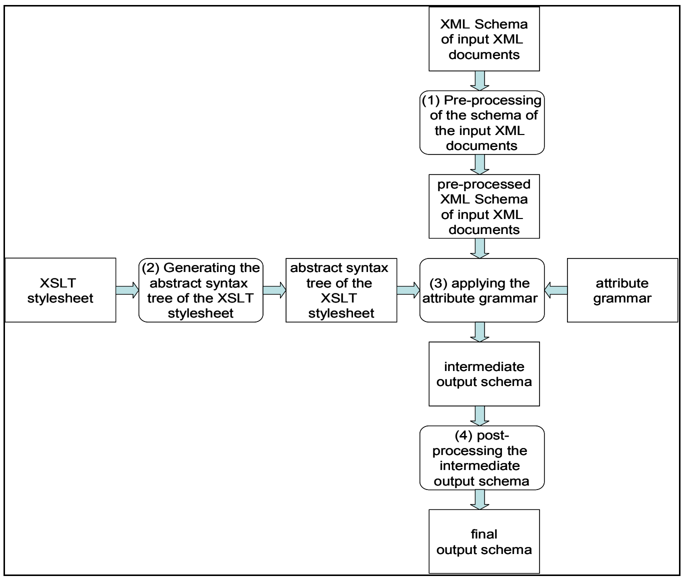
XML Schema of input XML documents
(1) Pre-processing of the schema of the input XML documents
pre-processed XML Schema of input XML documents
XSLT stylesheet
(2) Generating the abstract syntax tree of the XSLT stylesheet
abstract syntax tree of the XSLT stylesheet
(3) applying the attribute grammar
attribute grammar
intermediate output schema
(4) post-processing the intermediate output schema
final output schema

Fig. 4: Processes of the determination of the output schema of an XSLT stylesheet

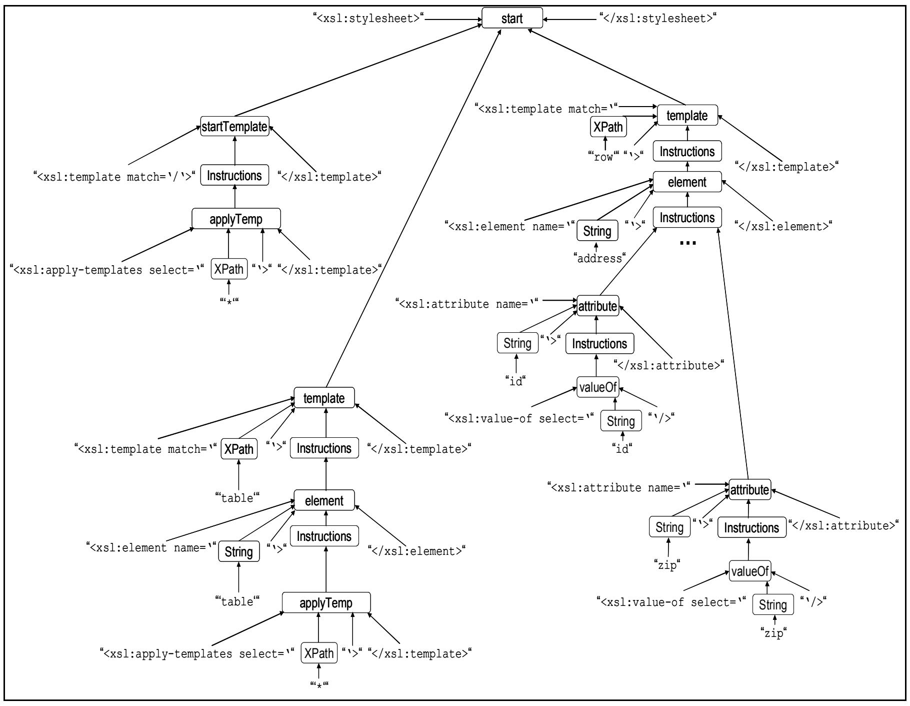

Fig. 5: Abstract syntax tree of the XSLT stylesheet of Figure 2, where one built-in template is included in the figure, but only the first and the last attribute XSLT instructions of the `address` element XSLT instruction are represented due to space limitations

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="table" minOccurs="1" maxOccurs="1">
  <xsd:complexType>
   <xsd:element name="row" minOccurs="0" maxOccurs="unbounded">
    <xsd:complexType>
     <xsd:sequence minOccurs="1" maxOccurs="1">
      <xsd:element name="id" minOccurs="1" maxOccurs="1">
       <complexType mixed='true'/></xsd:element>
      <xsd:element name="firstname" minOccurs="1" maxOccurs="1">
       <complexType mixed='true'/></xsd:element>
      <xsd:element name="lastname" minOccurs="1" maxOccurs="1">
       <complexType mixed='true'/></xsd:element>
      <xsd:element name="street" minOccurs="1" maxOccurs="1">
       <complexType mixed='true'/></xsd:element>
      <xsd:element name="city" minOccurs="1" maxOccurs="1">
       <complexType mixed='true'/></xsd:element>
      <xsd:element name="state" minOccurs="1" maxOccurs="1">
       <complexType mixed='true'/></xsd:element>
      <xsd:element name="zip" minOccurs="1" maxOccurs="1">
       <complexType mixed='true'/></xsd:element>
     </xsd:sequence>
    </xsd:complexType>
   </xsd:element>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

Fig. 6: XML Schema definition of the XML document used in the XSLTMark benchmark [6] and the `dbX.xml` XML documents of the XSLTMark benchmark.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:group name="N1">
  <xsd:element name="table" minOccurs="1" maxOccurs="1">
   <xsd:complexType>
    <xsd:element name="row" minOccurs="0" maxOccurs="unbounded">
     <xsd:complexType>
      <xsd:sequence minOccurs="1" maxOccurs="1">
       <xsd:element name="id" minOccurs="1" maxOccurs="1">
        <complexType mixed='true'/></xsd:element>
       <xsd:element name="firstname" minOccurs="1" maxOccurs="1">
        <complexType mixed='true'/></xsd:element>
       <xsd:element name="lastname" minOccurs="1" maxOccurs="1">
        <complexType mixed='true'/></xsd:element>
       <xsd:element name="street" minOccurs="1" maxOccurs="1">
        <complexType mixed='true'/></xsd:element>
       <xsd:element name="city" minOccurs="1" maxOccurs="1">
        <complexType mixed='true'/></xsd:element>
       <xsd:element name="state" minOccurs="1" maxOccurs="1">
        <complexType mixed='true'/></xsd:element>
       <xsd:element name="zip" minOccurs="1" maxOccurs="1">
        <complexType mixed='true'/></xsd:element>
      </xsd:sequence>
     </xsd:complexType>
    </xsd:element>
   </xsd:complexType>
  </xsd:element>
 </xsd:group>
</xsd:schema>
```

Fig. 7: Pre-processed XML Schema definition of Figure 6

Definition 2 (*attribute grammar*): An *attribute grammar* consists of a grammar `G` in EBNF notation, attributes of symbols of `G`, and computation rules for the attributes added to a production rule of `G`.

In the following, we use the notation `P { C }`, where `P` is a production rule of `G` in the EBNF notation and `C` contains the computation rules for attributes of symbols, which occur in `P`. We use a slightly different variant of the Java notation in `C`. We refer to an attribute `a` of the `m`-th symbol `n` in `P` as `n[m].a`. If there is only one symbol `n` in `P`, we use `n.a` instead of `n[1].a`. If there exists an arbitrary number of a symbol `n` in `P`, then `i` represents the concrete number of occurrences of the symbol `n` in `P`. `i1` and `i2` respectively represent the concrete numbers of occurrences if two symbols `n1` and `n2` respectively can both occur an arbitrary number of times in `P`.

An attribute grammar `G` is evaluated on an expression `E` as follows: First the syntax tree of `E` is generated. Then we compute the attributes until all attributes are computed or no new attributes can be computed. We compute an attribute `a` of a node `N` of the syntax tree according to a rule `r` of `G`, if `r` represents the recognized situation, in which `N` is embedded, and if all necessary attributes for the computation of `a` in `r` have already been computed.

In this section, we present the algorithm of the output schema generator for XSLT stylesheets. We first present the function `getSchemaOfXPath`, which is used in the output schema generator. Afterwards, we present the algorithm of the output schema generator in form of an attribute grammar.

We define the following term for use in the function `getSchemaOfXPath`.

Definition 3 (*relative part and absolute part of an XPath expression*): An XPath expression `I` can be divided into a *relative part* `rp(I)` and an *absolute part* `ap(I)` (both of which may be empty) in such a way that `rp(I)` contains a relative path expression, `ap(I)` contains an absolute path expression, and the union of `ap(I)` and `rp(I)`, denoted as `ap(I)|rp(I)`, is equivalent to `I`. This means that the evaluation of `I` returns the same node set as the evaluation of `ap(I)|rp(I)` for all XML documents and for all context nodes in the current XML document.

```
(1)  Algorithm getSchemaOfXPath
(2)  Input:  XPath expression Q_XPath
(3)  Output: XML fragment containing the XML Schema definition of the result of
Q_XPath
(4)  XSD = XML Schema definition of the data in which context Q_XPath is applied
(5)  Q = ap(Q_XPath)|/descendant-or-self::node(/self::node()|
         /attribute::node()|/namespace::node())/rp(Q_XPath)
(6)  sp ← evaluateXPath(Q, <(Q, start node of XSD, null, null)>)
(7)  if(sp.size()=1) {s1=;s2=;}
(8)  else s1="<xsd:choice>";s2="</xsd:choice>";
(9)  for each spe ∈ sp do
(10) s3 = s3 + copy (last entry in spe).getNode() and its subtree;
(11) return createXMLFragment(s1 + s3 + s2);
```

Fig. 8: Function `getSchemaOfXPath`

Whenever an XPath expression occurs in the XSLT stylesheet, the output schema generator uses the function `getSchemaOfXPath` of Figure 8 for generating the schema of the result of the XPath expression. `getSchemaOfXPath` executes a search in the XML Schema definition of the data in which context the given XPath expression Q_{XPath} is applied (see line (4), the search is described in [10] in more detail). The context of Q_{XPath} can be XML nodes of the input XML document or of a variable. The schema of the input XML document or of the variable in which context Q_{XPath} is applied has been already determined in our static analysis of the output schema generators. In line (5), we replace the relative part of Q_{XPath} with an XPath expression that describes a superset of the XML nodes described by the relative part of Q_{XPath}, as we are not aware of the concrete context of Q_{XPath}. For determining the XML nodes of the relative part of Q_{XPath} more exactly, we refer to [10] and [12], which contains static analysises that can be used for this purpose. We start the search in the XML Schema definition in line (6). The result of the search in the XML Schema definition and its subtree is the schema of the given XPath expression Q_{XPath}. Thus, we copy these nodes of the XML Schema definition and return them as the resultant schema of the given XPath expression Q_{XPath} in line (7) to line (11).

In this section, we describe the algorithm, which generates the output schema in form of an XML Schema definition from an XSLT stylesheet V_{XSLT}. If V_{XSLT} contains an XSLT instruction `<xsl:copy-of>`, which copies XML nodes of the input XML document, then our output schema generator requires the XML Schema definition of the input XML document. We present the algorithm of the output schema generator in the following attribute grammar. Note that the chosen attribute grammar covers many XSLT stylesheets. If we consider the XSLT stylesheets of the XSLTMark benchmark [6], then we can determine the output schemas of 37 from 39 XSLT stylesheets of the XSLTMark benchmark. One of the remaining XSLT stylesheets, the output schema of which we cannot determine with the proposed attribute grammar for the determination of the output schema, contains element declarations the names of which depend on the input XML document. Another remaining XSLT stylesheet does not start with `<xsl:stylesheet>`.

The `Start` symbol is applied to the root node of the abstract syntax tree of the considered XSLT stylesheet. After evaluating the attribute grammar to the abstract syntax tree, the `schema` attribute of the `Start` symbol contains the determined output schema of the considered XSLT stylesheet.

```
Start ::= "<?xml version='1.0'?> <xsl:stylesheet>" (attributeSet|decimalFormat)*
startTemplate (template)* "</xsl:xsl:stylesheet>".
{ Start.schema = createXMLDocument(
   "<xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'>" +
   startTemplate.schema +
   template[1].schema + … + template[i].schema+"</xsd:schema>");}
```

The name of the template is used as name of a `group` construct that contains the schema information of the currently considered template. We will refer to this `group` construct whenever this template could be called. As the `startTemplate` symbol represents the initial template with which the processing of the XSLT stylesheet starts, we use the schema of this template as root nodes of the whole output schema and register the group containing the schema of the initial template by a function `rootNodesGroup`. Note that if no template matches the document node “/”, we have to consider the built-in template of XSLT that matches the document node “/”. The function `generateNewUniqueName()` returns a new unique name in order to be able to refer to the schema of the template.

```
startTemplate ::= "<xsl:template match='/' (name='" String "'")? ">" Instructions
"</xsl:template>".
{if(String occurs) nameString = String.getString();
 else nameString = generateNewUniqueName();
 startTemplate.schema = createXMLFragment(
  "<xsd:group name='" + nameString + "'>"
  + Instructions.schema + "</xsd:group>");
 template.attributes = Instructions.attributes;
 template.name = String.getString();
 rootNodesGroup(String.getString());}

template ::= "<xsl:template match='" XPath "'" (name='" String "'")? ">" Instruc-
tions "</xsl:template>".
{if(String occurs) nameString = String.getString();
 else nameString = generateNewUniqueName();
 template.schema = createXMLFragment("<xsd:group name='" +
                   nameString +
                   "'>"+ Instructions.schema + "</xsd:group>");
 template.name = String.getString();
 template.attributes = Instructions.attributes;}
```

We concatenate the schemas of several instructions in a sequence. Furthermore, we check whether the content of the instructions is fixed and store the fixed value in `Instructions.fixed` in this case.

```
Instructions ::= (applyTemplates | forEach | copyOf |  copy | variable | choose |
if | text | element | attribute | valueOf | callTemplate | attributeSet | decimal-
Format)*.
{if(only one text symbol is recognized)
   {Instructions.fixedFlag=true; Instructions.fixed=text.fixed;}
 else Instructions.fixedFlag=false;
 String attributes="";
 attributes=attributes + (Recognized Symbol 1).attributes;
 …
 attributes=attributes + (Recognized Symbol i).attributes;
 Instructions.attributes=attributes;
 Instructions.schema = createXMLFragment("<xsd:sequence>" +
                         (Recognized Symbol 1).schema + … +
                         (Recognized Symbol i).schema + "</xsd:sequence>");}
```

We refer to the schema of a called function in case of a `call-template` XSLT instruction.

```
callTemplate ::= "<xsl:call-template name='" String "'/>".
{callTemplate.schema = createXMLFragment("<xsd:group ref='" +
  String.getString() + "' minOccurs='1' maxOccurs='1'/>");
 callTemplate.attributes = (Symbol with name String.getString()).attributes;}
```

A template `<xsl:template match=M>` can be called from an `<xsl:apply-templates select=S>` XSLT instruction, if `M` and `S` intersect. For simplicity of presentation, we do not describe the determination of the concrete context of `M` that could be determined by the methods described in [10] and [12], but we replace the relative part of `M`, i.e. `rp(M)`, with a superset of the XML nodes described by `rp(M)`. This superset of the XML nodes described by `rp(M)` is /descendant-or-self::node()(/self::node()|/attribute::node()| /namespace::node())/rp(M). We can use an intersection tester, which consists of a reduction of the the intersection test to the satisfiability test of XPath expressions [13] and the satisfiability tester proposed in [8], [9], [10] or [13].

```
applyTemplates ::= "<xsl:apply-templates select='" XPath "'/>".
{T = { sy | Symbol sy representing <xsl:template match=M name=n> of the XSLT
stylesheet, where maybe (ap(XPath.getString())|/descendant-or-
self::node()(/self::node() | /attribute::node()|/namespace::node())/
rp(XPath.getString())) ∩ (ap(M)|/descendant-or-self::node()
(/self::node()|/attribute:: node()|/namespace::node())/rp(M))};
 applyTemplates.schema = createXMLFragment("<xsd:choice>"
                        + "<xsd:group ref=" + T[1].name +
                          " minOccurs='0' maxOccurs='1'/>" + …
                        + "<xsd:group ref=" + T[i].name +
                          " minOccurs='0' maxOccurs='1'/>"
                        + "</xsd:choice>");
 applyTemplates.attributes= createXMLFragment(T[1].attributes
                            + … + T[i].attributes);}
```

The returned schema stored in `Instructions.schema` already contains an `<xsd:sequence>` specification.

```
forEach ::= "<xsl:for-each select='" XPath "'>" Instructions "</xsl:for-each>".
{forEach.schema =
  "<xsd:sequence minOccurs='0' maxOccurs='unbounded'>" +
  Instructions.schema + "</xsd:sequence>";
 forEach.attributes=Instructions.attributes;}
```

The function `storeVariableSchema(name,schema,attributes)` is used in order to store the schema `schema` and the attributes `attributes` of a variable `name`. We access the schema of a variable if e.g. we analyze the schema of an XSLT instruction that copies the content of the variable with `<xsl:copy-of>`.

```
variable ::= "<xsl:variable name='" String "'>" Instructions "</xsl:variable>".
{storeVariableSchema(String.getString(), Instructions.schema,
                     Instructions.attributes);}
```

The function `getSchemaOfXPath(XP)` returns the schema to which the result of the XPath expression `XP` is valid. The function uses the schema information of the input XML document if `XP` specifies XML nodes in the input XML document, or uses the schema information of that variable the XML nodes of which are addressed by `XP`.

```
copyOf ::= "<xsl:copy-of select='" XPath "'/>".
{copyOf.schema = getSchemaOfXPath(XPath.getString());
 copyOf.attributes=;}
```

The function `getElementName(j)` returns the element name of the `j`-th element declaration at top level of the considered schema, i.e. `schema.getElementName(2)` returns "`name2`" if `schema` is

```
   <xsd:element name="name1"/>
   <xsd:element name="name2">
    <xsd:complexType><xsd:element name="name3"/></xsd:complexType>
   </xsd:element>.
```

Let `n` contain the number of top level element declarations of the considered schema.

```
copy ::= "<xsl:copy" useAttributeSet? ">" Instructions "</xsl:copy>".
{SchemaInst = Instructions.schema;
 currentElementNodes = getSchemaOfXPath("/descendant::node()");
 if(useAttributeSet occurs) attributes = useAttributeSet.attributes;
 else attributes="";
 copy.schema = "<xsd:choice>" + "<xsd:element name='"+
  currentElementNodes.schema.getElementName(1)+"'><xsd:complexType mixed='true'>"+
  schemaInst+"</xsd:complexType>"+attributes+"</xsd:element>"+ … +
  "<xsd:element name='"+currentElementNodes.schema.getElementName(n)+
  "'><xsd:complexType mixed='true'>"+schemaInst+"</xsd:complexType>"+
  attributes+"</xsd:element>"+"</xsd:choice>";}
```

Let `name` contain the names of the attribute sets and `getSchemaOfAttributeSet(name)` be a function, which searches for the attribute set with name `name` in the correct scope and returns the declared attributes of its inner instructions.

```
useAttributeSet  ::= "use-attribute-sets='" name* "'".
{useAttributeSet.attributes=getSchemaOfAttributeSet(name[1])+…+
                            getSchemaOfAttributeSet(name[i]);}
```

The schemas definitions of the `attribute-set` XSLT instructions are only considered if they are referred.

```
attributeSet ::= "<xsl:attribute-set>" Instructions "</xsl:attribute-set>".
{attributeSet.schema="";}
```

Let `name` contain an arbitrary name of an attribute.

```
decimalFormat ::= "<xsl:decimal-format" + (name "='" + String + "'")* + "/>".
{decimalFormat.schema="";}

choose ::= "<xsl:choose>" when+ otherwise? "</xsl:choose>".
{if(otherwise occurs) s = otherwise.schema;
 else s="";
 choose.schema = createXMLFragment("<xsd:choice>" +
  when[1].schema + … + when[i].schema + s + "</xsd:choice>");
 choose.attributes = createXMLFragment(when[1].attributes + … +
                                     when[i].attributes);}

when ::= "<xsl:when test='" XPath "'>" Instructions "</xsl:when>".
{when.schema = Instructions.schema;
 when.attributes = Instructions.attributes;}

otherwise ::= "<xsl:otherwise>" Instructions "</xsl:otherwise>".
{otherwise.schema = Instructions.schema;
 otherwise.attributes = Instructions.attributes;}

if ::= "<xsl:if test='" XPath "'>" Instructions "</xsl:if>".
{if.schema = createXMLFragment(
   "<xsd:sequence minOccurs='0' maxOccurs='1'>" + Instructions.schema +
   "</xsd:sequence");
 if.attributes = Instructions.attributes;}
```

The content of an `<xsl:text>` XSLT instruction is used to specify fixed elements or fixed attributes.

```
text ::= "<xsl:text>" String "</xsl:text>".
{text.schema =; text.fixed  = String.getString(); text.attributes=;}

element ::= "<xsl:element name='" String "'" useAttributeSet? ">" Instructions
"</xsl:element>".
{if(Instructions.fixedFlag=true) s = "fixed='" + Instructions.fixed + "'";
 else s=;
 if(useAttributeSet occurs) attributes=useAttributeSet.attributes;
 else attributes="";
 element.schema = createXMLFragment("<xsd:element name='" +
  String.getString() + "'" + s + "><complexType mixed='true'>" +
  Instructions.schema + "</xsd:complexType>" + Instructions.attributes +
  attributes + "</xsd:element>");
 element.attributes=;}

attribute ::= "<xsl:attribute name='" String "'>" Instructions "</xsl:attribute>".
{if(Instructions.fixedFlag=true) s = "fixed='" + Instructions.fixed + "'";
 else s=;
 attribute.attributes = createXMLFragment(
   "<xsd:attribute name='" + String.getString() + "' type='xsd:string' " + s +
   "/>");
 attribute.schema=;}

valueOf ::= "<xsl:value-of select='" String "'/>".
{valueOf.schema=;valueOf.attributes=;}
```

2.4 Post-processing the XML Schema definition generated by the output schema generator

The XML Schema definition generated by the output schema generator must be post-processed for the following reasons:

1. The output schema generator for XSLT stylesheets embeds the schema definition of the start template in a group, which does not declare the root element of the schema. The root element of the schema can be declared in an `<xsd:element>` declaration as child of the `<xsd:schema>` element of the XML Schema definition.
2. The output schema generators may generate not allowed circular definitions of groups for recursive functions.
3. Different types of elements with the same name are not allowed in the same scope. However, the output schema generators generate different types of elements with the same name in the same scope whenever elements with the same name occur in a sequence of element constructors (e.g. **`<a>`**`<b/>`**`</a><a>`**`<c/>`**`</a>`**) in the XSLT stylesheet respectively. We can solve this problem by merging the schema of the types of these elements into one unique type of elements with the same name in the same scope. The disadvantage of this technique is a less precise output schema.

```
<xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'>
 <xsd:group name='name0'>
  <xsd:sequence>
   <xsd:element name='table'>
    <complexType mixed='true'>
     <xsd:sequence>
      <xsd:choice><xsd:group ref='name0' minOccurs='0' maxOccurs='1'/>
                  <xsd:group ref='name1' minOccurs='0' maxOccurs='1'/>
      </xsd:choice>
     </xsd:sequence>
    </complexType>
   </xsd:element>
  </xsd:sequence>
 </xsd:group>
 <xsd:group name='name1'>
  <xsd:sequence>
   <xsd:element name='address'>
    <complexType mixed='true'><xsd:sequence></xsd:sequence></complexType>
      <xsd:attribute name='id' type='xsd:string'/>
      <xsd:attribute name='firstname' type='xsd:string'/>
      <xsd:attribute name='lastname' type='xsd:string'/>
      <xsd:attribute name='street' type='xsd:string'/>
      <xsd:attribute name='city' type='xsd:string'/>
      <xsd:attribute name='state' type='xsd:string'/>
      <xsd:attribute name='zip' type='xsd:string'/>
   </xsd:element>
  </xsd:sequence>
 </xsd:group>
 <xsd:group  name='Start'>
  <xsd:sequence>
   <xsd:choice><xsd:group ref='name0' minOccurs='0' maxOccurs='1'/>
   </xsd:choice>
  </xsd:sequence>
 </xsd:group>
</xsd:schema>
```

Fig. 9: Generated XML Schema definition of the XSLT stylesheet of Figure 2, which has not been post-processed so far.

We post-process the XML Schema definition generated by the output schema generator for XSLT stylesheets (Case 1.) by copying the declaration of the top level elements generated in the start template (or its called templates) to the child nodes of the `<xsd:schema>` instruction of the XML Schema definition.

In the case of circular definitions of groups (Case 2.), the group reference containing the circular definition of groups is not embedded in a `complexType` construct. Then we transform the circular definition by "rolling out" into an iterated sequence. Thus, we proceed as follows:

At each group reference `R` that contains a circular definition of groups, we insert an `<xsd:sequence minOccurs='1' maxOccurs='unbounded'>` instruction (embedded in another `<xsd:sequence>` instruction as `minOccurs` and `maxOccurs` attributes are not allowed in an `<xsd:sequence>` instruction directly after a named group) at the beginning of the group in which `R` is contained. If the referred group is the group in which the group reference is contained, we delete the group reference. Otherwise we replace the group reference with copies of the content of the referred group in an `<xsd:sequence>` instruction.

Example 3 (*Post-processed XML Schema definition*): Figure 3 contains the post-processed XML Schema definition of Figure 9.

In the case of different types `Ti` of elements `Ei` with the same name (Case 3.), we merge the schemas of the different types by declaring a new named `complexType`. In the new named `complexType`, we declare the different schemas in an `<xsd:choice>` instruction and we set the types in `Ei` to the named `complexType`.

Example 4 (*Post-processed XML Schema definition*): For example, the following schema definition

```
<xsd:sequence>
 <xsd:element name='a'>
  <complexType mixed='true'>
   <xsd:element name='b'><complexType mixed='true'/></xsd:element>
  </complexType>
 </xsd:element>
 <xsd:element name='a'>
  <complexType mixed='true'>
   <xsd:element name='c'><complexType mixed='true'/></xsd:element>
  </complexType>
 </xsd:element>
</xsd:sequence>
```

will be transformed into

```
<xsd:sequence>
 <xsd:element name='a' type='newName'/>
 <xsd:element name='a' type='newName'/>
</xsd:sequence>

<xsd:complexType name='newName'>
 <xsd:choice>
   <xsd:element name='b'><complexType mixed='true'/></xsd:element>
   <xsd:element name='c'><complexType mixed='true'/></xsd:element>
 </xsd:choice>
</xsd:complexType>
```

3 Experimental Evaluation

We have implemented the output schema described above by coding the attribute grammar in Java. We have used Java 1.5 and Windows XP as operation system.

We have tested our prototype with the XSLT stylesheets of the XSLTMark benchmark [6], which consists of 39 stylesheets. The designers of XSLTMark chose the XSLT stylesheets in order to cover as many aspects of the XSLT language as possible.

Our prototype first loads the XSLT stylesheet, the output schema of which we want to determine, in the main memory by the DOM parser of Java 1.5. Small problems occur in this phase, which have resulted in little changes of the original XSLT stylesheets, which are listed in the following enumeration:

1. Whenever attributes of the XSLT stylesheets, which are enclosed by " characters in the XSLTMark benchmark and contain the character ', the DOM parser of Java 1.5 raises errors. Thus, we have replaced the outer " character by the ' characters and the inner 'characters by the " characters. For example, we have replaced the line `<xsl:value-of select="concat ($bottles, ' bottles')"/>` of `bottles.xsl` with the line `<xsl:value-of select='concat ($bottles, " bottles")'/>`.
2. Whenever `&lt;` occurs in the attributes of the original XSLT stylesheets, the DOM parser reports an error that the DOM parser does not expect the < character (to which `&lt;` is expanded) here. The DOM parser interprets `&lt;` as a start sequence of a new tag and not as comparison operator within a string representing the value of an attribute. Thus, we have replaced comparison expressions `P1 &lt; P2` with `P2 > P1` in the original XSLT stylesheets of the XSLTMark benchmark. For example, we have replaced the line `<xsl:template match="foo[following-sibling::*[position()&lt;=2][self::barg] and following-sibling::*[position()&lt;=2][self::nar]]">` of the `xpath.xsl` XSLT stylesheet of the XSLTMark benchmark with the line `<xsl:template match="foo[following-sibling::*[2>=position()][self::barg] and following-sibling::*[2>=position()][self::nar]]">`.

After modifying the original XSLT stylesheets of the XSLTMark benchmark in the proposed way, our prototype can successfully determine the output schemas of 37 out of 39 XSLT stylesheets of the XSLTMark benchmark. One of the remaining XSLT stylesheets, the output schema of which we cannot determine with our prototype of the output schema generator, contains element declarations the names of which depend on the input XML document. Another remaining XSLT stylesheet does not start with `<xsl:stylesheet>`.

Overall, if we consider the XSLT stylesheets of the XSLTMark benchmark as typical XSLT stylesheets, then our prototype can determine 95% of the output schemas of the typical XSLTstylesheets.

4 Further Related work

[4], [7], [14], [18], [19] and [21] describe how to determine an XML Schema according to a set of XML documents, but they do not describe how to determine the output schema of XSLT stylesheets.

To the best of our knowledge, there are no contributions to the determination of output schemas of XSLT stylesheets formulated in XML Schema.

There are contributions [10] and [12], which deal with a search on the output nodes of XSLT stylesheets as part of a static analysis for the optimization of the execution time of XSLT stylesheets. [11] transforms XQuery expressions into an graph, which represents the output of XQuery expressions, which is also further used for the optimization of the execution time of XQuery expressions.

[2], [15], [16], [8], [9], [13] and [1] focus on the satisifability problem of XPath queries. There is a great amount of work dealing with query containment for XPath ([5], [17], [20] and [22]). [13] describes how to reduce the containment and intersection test of XPath expressions to the satisfiability test.

5 Summary and Conclusions

We have proposed an approach for the determination of the output schema of an XSLT stylesheet. An output schema of an XSLT stylesheet is the schema to which all possible results of the XSLT stylesheet conform to. Possible application scenarios for using the output schemas of XSLT stylesheets are the test, whether a given XSLT stylesheet was not used to transform to a given XSLT stylesheet, for optimizations for XSLT stylesheets and other application scenarios (see Figure 1), where logical testers are used to reason on the output schemas.

We have described the algorithm for determining the output schema of XSLT stylesheets by an attribute grammar. The experimental evaluation shows that our prototype can determine 95% of the output schemas of typical XSLT stylesheets.

The future work will cover to extend the prototype to handle also the rare cases, which we did not consider until now. Furthermore, we will consider determining also other output schemas such as the output schemas of XQuery expressions and the output schemas of other query and transformation languages outside the XML world.

Acknowledgements

The work is funded by the European Commission under the projects ASG and TripCom; by Science Foundation Ireland under the DERI-Lion Grant No.SFI/02/CE1/I13.

Reference

1. S. Amer-Uahis, S. Cho, L. K. S. Laksmanan, D. Srivastava. Mininization of tree pattern queries. *SIGMOD 2001*, Santa Barbara, California, USA, 2001.
2. Michael Benedikt, Wenfei Fan and Floris Geerts. XPath Satisfiability in the presence of DTDs, *PODS 2005*, Baltimore, Maryland, 2005.
3. Böttcher, S., and Steinmetz, R., Adaptive XML Access Control Based on Query Nesting, Modification and Simplification. *BTW 2005*, Karlsruhe, Germany, 2005.

4. B. Chidlovskii. Schema extraction from xml: A grammatical inference approach. In *Proceedings of the International Workshop KRDB*, 2001.
5. A. Deutsch and V. Tannen. Containment and Integrity Constraints for XPath, In *KRDB 2001*, CEUR Workshop proceedings 45, 2001.
6. Developer, XSLT Mark version 2.1.0, http://www.datapower.com/xmldev/xsltmark.html, 2005.
7. M. N. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: Learning document type descriptors from xml document collections. *Data Mining and Knowledge Discovery*, 7(1):23-56, 2003.
8. Jinghua Groppe, Sven Groppe. A Prototype of a Schema-based XPath Satisfiability tester. *17th International Conference on Database and Expert Systems Applications (DEXA 2006)*, Krakow, Poland, 2006.
9. Jinghua Groppe, Sven Groppe. Filtering Unsatisfiable XPath Queries, *8th International Conference on Enterprise Information Systems (ICEIS 2006)*, Paphos, Cyprus, 2006.
10. Sven Groppe, XML Query Reformulation for XPath, XSLT and XQuery. *Sierke-Verlag*, Göttingen, Germany, 2005. ISBN 3-933893-24-0.
11. Sven Groppe, Stefan Böttcher. Schema-based query optimization for XQuery queries. *ADBIS 2005*, Tallinn, Estonia, 2005.
12. S. Groppe, S. Böttcher, G. Birkenheuer, A. Höing. Reformulating XPath Queries and XSLT queries on XSLT Views. *Data & Knowledge Engineering Journal*, 2006.
13. Sven Groppe, Stefan Böttcher, Jinghua Groppe, XPath Query Simplification with regard to the Elimination of Intersect and Except Operators, *XSDM 2006* in conjunction with IEEE ICDE 2006, Atlanta, USA, 2006.
14. J. Hegewald, F. Naumann, and M. Weis, XStruct: Efficient Schema Extraction from Multiple and Large XML Documents, *XSDM 2006* in conjunction with IEEE ICDE 2006, Atlanta, USA, 2006.
15. Hidders, Jan, Satisfiability of XPath Expressions, *DBPL 2003*, 2003.
16. Lakshmanan, L., Ramesh, G., Wang, H., and Zhao, Z. On Testing Satisfiability of Tree Pattern Queries, In *VLDB 2004*, Toronto, Canada, 2004.
17. G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment, *In Proc. PODS'02*, pages 65-76, Madison, Wisconsin, 2002.
18. J.-K. Min, J.-Y. Ahn, and C.-W. Chung. Efficient extraction of schemas for XML documents. *Information Processing Letters*, 85:7-12, 2003.
19. S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semi-structured data. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 295-306, Seattle, WA, 1998.
20. F. Neven, T. Schewentick. XPath containment in the presence of disjunction, DTDs, and variables. *Proc. Of the 9th Int. Conf on Database Theory (IDCT)*, Siena, Italy. January 8-10, 2003.
21. Q. Y. Wang, J. X. Yu, and K.-F. Wong. Approximate graph schema extraction for semi-structured data. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 302{316, 2000.
22. P.T. Wood. Containment for XPath Fragments under DTD constraints. *Proc. of the 9th Int. Conf. on Database Theory (ICDT)*. Siena, Italy, January 8-10, 2003.
23. World Wide Web Consortium (W3C), XSL Transformations (XSLT) Version 1.0, W3C *Recommendation*, http://www.w3.org/TR/1999/REC-xslt-19991116, 1999.
24. W3C: XML Schema Part 1: Structures Second Edition. *W3C Recommendation*, www.w3.org/TR/xmlschema-1, 2004.