A Framework for Integrating XML Transformations

Ce Dong James Bailey

NICTA Victoria Laboratory

Department of Computer Science and Software Engineering
The University of Melbourne, VIC 3010, Australia
{cdong, jbailey}@csse.unimelb.edu.au

Abstract. XML is the de facto standard for representing and exchanging data on the World Wide Web and XSLT is a primary language for XML transformation. Integration of XML data is an increasingly important problem and many methods have been developed. In this paper, we study the related and more difficult problem of how to integrate XSLT programs. Program integration can be particularly important for server-side XSLT applications, where it is necessary to generate a global XSLT program, that is a combination of some initial XSLT programs and which is required to operate over a newly integrated XML database. This global program should inherit as much functionality from the initial XSLT programs as possible, since designing a brand new global XSLT program from scratch could be expensive, slow and error prone, especially when the initial XSLT programs are large or/and complicated. However, it is a challenging task to develop methods to support XSLT integration. Difficulties such as template identification, unmapped template processing and template equivalence all need to be resolved. In this paper, we propose a framework for semi-automatic integration of XSLT programs. Our method makes use of static analysis techniques for XSLT and consists of four key steps: i) Pattern Specialization, ii) Template Translation, iii) Lost Template Processing and iv) Program Integration. We are not aware of any previous work that deals with integrating XML transformations.

1 Introduction

XML [6] is rapidly emerging as a dominant standard for data representation and exchange on the Web [11]. The eXtensible Stylesheet Language Transformations (XSLT) standard [8, 26] is a primary language for transforming, reorganizing, querying and formatting XML data. In particular, server-side XSLT [23] is an extremely popular technology for processing and presenting results in response to user queries issued to a server side XML database. An XSLT program consists of a set of templates. Execution of the program is by recursive application of individual templates to the source XML document.

The availability of large amounts of homogeneous Web databases necessitates XML integration [5, 7, 12, 15, 20, 22, 27, 29], e.g. when two organizations which have similar XML information databases are amalgamated. Such XML integration is typically DTD-directed, that is, the integration task is constrained by a predefined DTD, to which the target XML document is required to conform [11]. A set of mapping rules between the initial DTDs and the global DTD must be provided.

However, when databases are amalgamated, it is not just static information which needs to be combined. XML repositories will often have associated dynamic aspects as well, such as XSLT programs or stylesheets, that have been designed to transform or present the XML information. When repositories are combined, so too must be the dynamic aspects. In other words, we require a new (global) XSLT program to access the integrated XML database. It is likely that this program will be required to inherit much of the functionality that was present in the initial XSLT programs, which operated over the original XML repositories.

Different from the language XQuery [4], an XSLT program consists of templates, which can be regarded as the basic program unit for building the global XSLT program during integration. Also, different from static XML data or schema integration [5, 7, 12, 15, 20, 22, 27, 29], XSLT integration is additionally challenging, because it must deal with the dynamic aspects. Some difficulties are faced: 1) A specific XSLT template might match, by means of selection patterns, multiple XML elements. This can cause confusion when mapping the template from the initial XSLT program to the global XSLT program, using the element mapping rules. 2) Two initial templates (from different initial programs) which match the same XML element, will need to be combined together within the global XSLT program. However, it is difficult to identify the conflicts and relationships (equivalence, containment and intersection) between their functionalities, when generating the global template body. 3) Some initial templates might not be mapped to and included in the global XSLT program, based on the element mapping rules. However, their absence might strongly affect the execution result and thus they must be properly combined within the global XSLT program. 4) Some templates contain functionality which is valid for an initial XSLT program, but which is no longer useful or even invalid for the global XSLT program. This needs to be detected and reconciled.

The integration framework proposed in this paper has four main components: 1) *Pattern Specialization* is used to specialize the template selection patterns and construction patterns and consequently lessen element reference ambiguity; 2) *Template Translation* is used to translate template selection patterns and construction patterns to conform to the global DTD; 3) *Lost Template Processing* is used to process the templates which match XML elements not existing in the mapping rule list; 4) *Program Integration* is used to generate the global XSLT program and mark any problematic templates for further consideration by the program designer.

The problem of XSLT integration is a new and challenging research issue. We are not aware of any other similar work that addresses this topic.

The remainder of this paper is organized as follows. We first review some basic concepts in section 2. Then, in section 3 we introduce XML integration approaches and related terminology. Next, in section 4, we propose the XSLT integration framework step by step. Related work is surveyed in section 5 and finally in section 6, we conclude our research and give the discussion of future work.

2 Background

We begin by briefly reviewing some concepts regarding DTDs, XSLT and XPath, assuming the reader already has basic knowledge in these areas.

2.1 DTDs and DTD-Graph

An XML DTD [6, 19] provides a structural specification for a class of XML documents and is used for validating the correctness of XML data. Based on the DTD, we can create a data structure to summarize the hierarchical information within a DTD, called the DTD-Graph. It is a rooted, node-labeled graph, where each node represents either an element or an attribute from the DTD and the edges indicate element nesting. The DTD-Graph developed in our previous work [10] is similar to the Dataguide structure described by Goldman and Widom in 1997[13]. It is an important data structure used to validate the XPath expressions (selection patterns and construction patterns) of XSLT programs during XSLT integration.

2.2 XSLT and Functionality Blocks

XSLT is a recursive XML transformation language [8, 16, 17, 18]. An XSLT program can be thought of as an ordered collection of templates. Each template has an associated pattern (selection pattern) and contains a nested set of construction rules. A template processes XML-tree [8] nodes that match the selection pattern and constructs output according to the construction rules [23].

An XSLT program is also an XML document, with a corresponding tree structure, having a 'root element' node of <*xsl:stylesheet*> that has <*xsl:template*> child nodes. We refer to the sub-trees which are children of the <*xsl:template*> nodes as “functionality blocks”.

2.3 XPath

The primary purpose of XPath is to address parts of an XML document using path expressions. It also provides basic facilities for manipulation of strings, numbers and booleans.[28]. A *location path* is an XPath expression which selects a set of nodes relative to the context node. If we remove ‘predicate(s)’ from the location path, we can get an XPath expression consisting of ‘axes’, ‘steps’ and ‘/’, called a *distinguished* XPath [2] expression. The selection patterns and construction patterns in an XSLT program are expressed using XPath. Selection patterns can only use the axes of ‘child’ and ‘attribute’, whereas construction patterns may be full XPath expressions. XPath expressions starting with ‘/’ or ‘//’ are called *absolute* XPath expressions. Otherwise (e.g. starting with ‘.’ or ‘node name’), they are called *relative* XPath expressions. *Simple* XPath (similar to [2]) is a fragment of XPath which disallows the use of any ‘function’, ‘predicate’ and ‘axes’ other than ‘child’, ‘self’, and ‘descendant-or-self’. Oppositely, XPath expressions which contain ‘functions’ or ‘predicates’ or ‘axes’ other than those above, we will term *rich* XPath. Our XSLT integration framework can deal with simple XPath expressions automatically and handles rich XPath expressions via human interaction (to be discussed in section 4).

We further define *full-absolute* XPath expressions to be those starting with '/', followed by a sequence of node names separated by '/' (e.g. '*/a/b/c/d*'). We define *full-relative* XPath expressions to be those starting with '*./*' or 'node name', followed by a sequence of node names separated by '/' (e.g. '*./b/c/d*' and '*b/c/d*'). These concepts are important for supporting the descriptions of the XSLT integration framework in section 4.3 and 4.4.

2.4 The Template and Association Graph (TAG) of an XSLT Program

XSLT syntactic structure gives rise to calling relationships between templates [14, 17]. In our previous work [10], we designed a *Template and Association Graph* (TAG), which is a rooted node-labeled directed graph used to describe the calling relationships between XSLT templates. The TAG can be used to analyze an XSLT program and help to find bugs in XSLT program design [10]. In this paper, we use the TAG to eliminate unreachable templates, missing templates and invalid calling relationships [10], that are generated as 'side-effects' during the XSLT integration process.

2.5 Server-Side XSLT

Server-side XSLT [23] is a popular solution for data exchange and querying on the Web. It is often deployed in e-commerce, e-publishing and information services applications. Transforming the content on the server has advantages such as providing convenience for business logic design and code reuse, cheaper data access and security and smaller client downloads [18]. XSLT integration is more meaningful for server-side XSLT (as opposed to client side XSLT), since a global XSLT program must be constructed after the server XML databases are merged.

3 XML Integration

Suppose we have XML databases associated with a server-side XSLT system. There are then two major different approaches which can be used for XML integration [3, 22, 24]. One is *virtual* integration, where no physically integrated XML needs to be built. Specifically, *virtual* integration publishes a global XML schema (e.g. a DTD) which is 'integrated' from the initial distributed XML database schemas. A user query over the global schema passed to the system is then re-written into distributed queries (i.e. parameters to distributed XSLT programs) to access the distributed XML databases (initial XMLs). A combined result is returned to the user. Another kind of XML integration is called *instance* integration, since a global XML is physically built. Specifically, based on a predefined global XML schema, the data of the initial XMLs is merged into the global XML. A user query based on the global DTD is evaluated directly over the integrated XML database. Our XSLT integration framework is designed to integrate the initial XSLT programs according to instance based integrated XML. Hereafter, when we refer XML integration, this should be understood to mean instance based XML integration. In the following definitions, Doc_XML1 and Doc_XML2 denote the initial XMLs and Doc_XML3 denotes the global XML.

- **Mapping rule:** A pair containing an initial element and a global element. It indicates that the initial element describes the same object as the global element. The XML elements are expressed using full-absolute XPath expressions. For example, (‘*/a/b/c*’, ‘*/X/Y/Z*’) denotes that the ‘c’ node of parent node ‘b’ and grand parent node ‘a’ under the ‘root’ in the initial XML is mapped to the ‘Z’ node of parent node ‘Y’ and grand parent node ‘X’ under the ‘root’. XML integration refers to two sets of mapping rules: i) MAP1 contains all the mapping rules from Doc_XML1 to Doc_XML3, ii) MAP2 contains all the mapping rules from Doc_XML2 to Doc_XML3.
- **Name Change***:* This term refers the situation when the name of element of an initial XML element is mapped to a different name in the global XML, based on the mapping rules (e.g. initial element ‘c’ is mapped to global element ‘Z’).
- **Structure Change:** This term is used to refer the situation when a parent-child relationship between elements in the initial XML doesn’t exist between their mapped elements in the global XML, based on the mapping rules.
- **Lost Element:** This term is used to refer to an element in an initial XML document which doesn’t have a corresponding (mapped) element in the global XML document, according to the mapping rules.

4 XSLT Integration

XSLT program integration concerns not only schema mapping, but also comparisons between template selection patterns and the relationships between template bodies (functionality). We now define some terminology that will be useful when we discuss comparison of templates.

Definition_1: *Potentially Conflicting Template Pair* is used to refer a pair of XSLT templates, each from different *initial* XSLT programs that are awaiting integration, and which have the same *distinguished* XPath selection pattern.

Definition_2: *Rich Template* is used to refer to templates whose selection pattern or/and construction pattern(s) are *rich* XPath expressions.

We also have some restrictions and assumptions on our model.

- The initial XSLT programs are well-formed and valid (error free).
- The output of the XSLT transformations is HTML or XML (the most popular cases used in XSLT transformations).
- The template(s) for the ‘*root*’ (‘*/*’) and ‘*root element*’ must exist (XSLT program traverses the XML-tree from the top).

For simplicity, in this paper, the DTD-Graphs of Doc_XML1, Doc_XML2 and Doc_XML3 are denoted by DG1, DG2 and DG3 respectively. XSL1, XSL2 and XSL3 denote two initial XSLT programs and the global XSLT program respectively. Their corresponding Template and Association Graphs are denoted by TAG1, TAG2 and TAG3 respectively. *<T m=’selection pattern’>* denotes XSLT element *<xsl:template match=’selction pattern’>* and *<A s=’construction pattern’>* denotes *<xsl:apply-templates select=’construction pattern’>*.

4.1 Overview of XSLT Integration

Our framework addresses the XSLT integration task in four principal steps.

Step_1: Pattern Specialization: The system converts all selection patterns and absolute construction patterns into full-absolute XPath and specializes the relative construction patterns containing ‘*’ and/or ‘//’ into full-relative XPath expressions. Human interaction is required for processing ‘rich’ templates.

Step_2: Template Translation: This step translates all XPath expressions that conformed to the initial DTD-Graphs (DG1 and DG2) into corresponding XPath expressions conforming to the global DTD-Graph (DG3), based on mapping rules (MAP1 and MAP2). Human interaction is also required to handle some special situations of element mapping.

Step_3: Lost Template Processing: This follows the template translation step and invokes special processing for templates or construction statements which refer to lost elements. Human interaction is asked before applying the default processes.

Step_4: Program Integration: The pre-processed initial XSLT programs are integrated into the global XSLT program XSL3, by means of integration algorithms. Human interaction is required for *rich* templates and static analysis.

Finally, all problematic templates in XSL3 are detected and marked based on TAG3, which can then be used as support for program further revision.

We use human interaction as a supplement to our XSLT integration framework. A completely automatic method is clearly impossible, due to the undecidable nature of much of the analysis required. This is also in line with the requirement of human interaction for static and schema integration [5, 7, 12, 15, 20, 22, 27, 29]. The overall aim of our framework though, is to alleviate the burden on the designer as much as possible, presenting them with a clear set of choices which need to be made. Furthermore, different methods and static analysis techniques can be ‘plugged in’ to the framework, according to their availability.

4.2 XSLT Integration Example

An XSLT integration example is provided here to help explain our method. It includes i) two synthetic initial server-side XSLT programs (XSL1 and XSL2), ii) the corresponding DTD-Graphs (DG1 and DG2) and iii) the corresponding mapping rules (MAP1 and MAP2). The scenario is based on integration between two XML employee information databases. We omit the XMLs, since it is the structure of the data which determines the XSLT integration workflow, not the data values.

Firstly, the initial DTD-Graphs (DG1 and DG2) and the global DTD-Graph (DG3) are shown in figures 1 (a), (b) and (c) respectively.

Secondly, the sets of mapping rules of MAP1 (map from DG1 to DG3) and MAP2 (map from DG2 to DG3) are listed respectively in tables 1 and table 2. For example, the second row of table 1 shows that DTD-Graph node ‘/Factory/Name’ of DG1 is mapped as node ‘/Factory/FN’ in the global DTD-Graph). From figure 1 and tables 1 and table 2 we can see that the underlying XML integration covers scenarios of ‘name change’, ‘structure change’ and ‘lost element’.

Thirdly, we show the initial XSLT programs to be integrated (i.e. XSL1 and XSL2). Their functionality is for retrieving and displaying the information about factory employees. Due to the space restrictions, we only show fragments of the programs (figures 2 (a) and (b)).

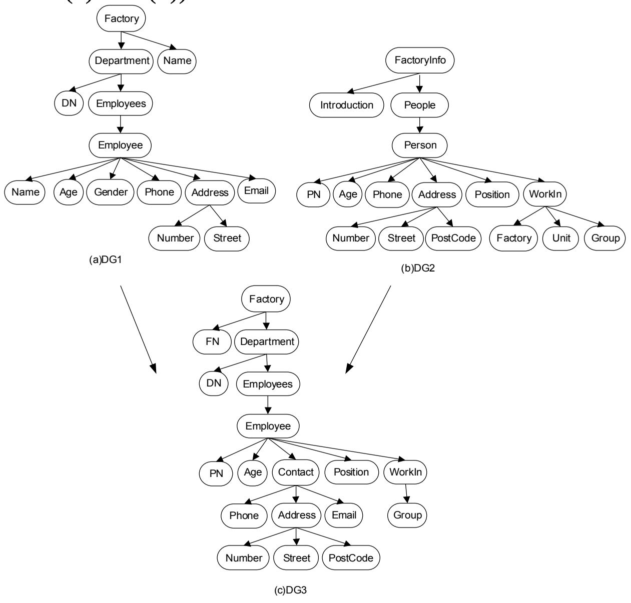


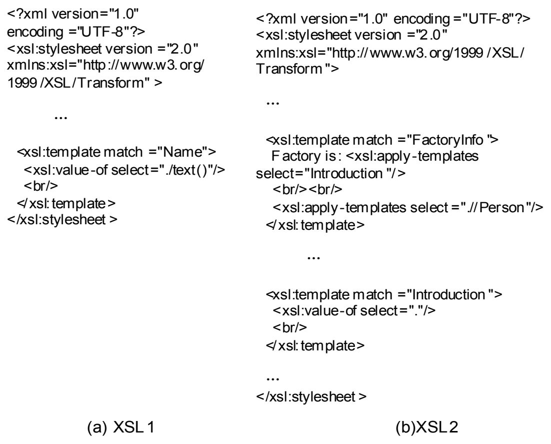
Fig. 1. DTD-Graphs of Doc_XML1, Doc_XML2 and Doc_XML3

Table 1. The fragment of mapping rules between DG1 and DG3

('/Factory', '/Factory')
('/Factory/Name', '/Factory/FN')
('/Factory/Department', '/Factory/Department')
('/Factory/Department/DN', '/Factory/Department/DN')
('/Factory/Department/Employees/Employees', '/Factory/Department/Employees/Employees')
('/Factory/Department/Employees/Employees/Employee', '/Factory/Department/Employees/Employees/Employee')
('/Factory/Department/Employees/Employee/Name', '/Factory/Department/Employees/Employee/PN')
…

Table 2. The fragment of mapping rules between DG2 and DG3

('/FactoryInfo', '/Factory')
('/FactoryInfo/Introduction', '')
('/FactoryInfo/People', '/Factory/Department/Employees')
('/FactoryInfo/People/Person', '/Factory/Department/Employees/Employee')
('/FactoryInfo/People/Person/WorkIn', '/Factory/Department/Employees/Employee/WorkIn')
('/FactoryInfo/People/Person/WorkIn/Factory', '/Factory/Name')
('/FactoryInfo/People/Person/WorkIn/Unit', '/Factory/Department/DN')
. . .



```
<?xml version="1.0"
encoding ="UTF-8"?>
<xsl:stylesheet version ="2.0"
xmlns:xsl="http://www.w3.org/
1999/XSL/Transform" >

        ...

  <xsl:template match ="Name">
    <xsl:value-of select="./text()"/>
    <br/>
  </xsl:template>
</xsl:stylesheet>
```

(a) XSL1

```
<?xml version="1.0" encoding ="UTF-8"?>
<xsl:stylesheet version ="2.0"
xmlns:xsl="http://www.w3.org/1999/XSL/
Transform">

  ...

  <xsl:template match ="FactoryInfo">
    Factory is: <xsl:apply-templates
select="Introduction"/>
    <br/><br/>
    <xsl:apply-templates select=".//Person"/>
  </xsl:template>

          ...

  <xsl:template match ="Introduction">
    <xsl:value-of select="."/>
    <br/>
  </xsl:template>

  ...
</xsl:stylesheet>
```

(b)XSL2

Fig. 2. Fragments of the initial XSLT programs to be integrated

Next, based on the example shown above, we explain the details of our XSLT integration framework step by step.

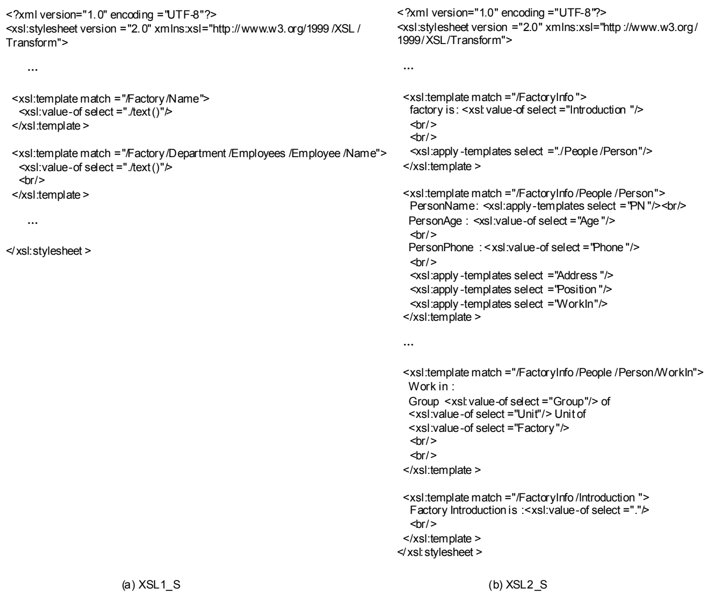
4.3 Pattern Specialization

Selection patterns in XSLT can be either full-absolute or non-full-absolute XPath expressions. A full-absolute XPath expression uniquely identifies a DTD-Graph node (i.e. the mapping relationship between a full-absolute XPath expression and a DTD-Graph node is 1 to 1), while a non-full-absolute XPath expression may identify multiple DTD-Graph nodes (i.e. the mapping relationship between a non-full-absolute XPath expression and a DTD-Graph node is 1 to N (N>=1)). Thus, when a template selection pattern is a non-full-absolute XPath expression, we might not sure which mapping rules should be chosen for translating the corresponding template from the initial DTD based XSLT program into the global DTD based XSLT program (step_2) and, consequently, can not continue the integration step to build global XSLT XSL3 (step_4). For example, consider the XSL1 fragment shown in figure 2 (a). The selection pattern of template <*T m='Name'*> can refer to the node of '/Factory/Name' and also the node of '*/Factory/Department/Employees/Employee/Name*' according to DG1 (show in figure 1 (a)). It is not clear whether 'Name' should be mapped to '*/Factory/FN*' or to '*/Factory/Department/Employees/Employee/PN*' according to MAP1 (shown in table 1), during the translation from the initial structure (DG1) to the global structure (DG3). Wrong translation can result in an integrated XSLT program which deviates from the original intentions of the initial XSLT program designers.

We choose to handle this ambiguity using a direct approach, which specialises the non-full-absolute selection patterns in XSL1 and XSL2 into full-absolute XPath expressions. This is called *pattern specialization*. In the case of a single template selection pattern matching multiple DTD-Graph nodes, we create new templates, one for each possible corresponding full-absolute selection pattern, and we then delete the original template. Let's examine the example of <*T m='Name'*> again - the template will be replaced by two new templates: <*T m='/Factory/Name'*> and <*T m='/Factory/Department/Employees/Employee/Name'*>, each with the same body as the original <*T m='Name'*>.

For the same reason and in the same way as for selection pattern specialization, we specialize construction patterns if i) they are absolute XPath expressions but not full-absolute XPath expression or ii) they are relative XPath expressions, but not full-relative XPath expressions. In the former case, the construction patterns are specialised into full-absolute XPath expressions and, in the latter case, the construction patterns are specialised into the full-relative XPath expressions. When a construction pattern indicates multiple nodes of DG1 (or DG2), we create a new construction statement for each specialized construction pattern and delete the original construction statement. For example, the construction statement <*A s='.//Person'*> of template <*T m='FactoryInfo'*> in XSL2 ('.//Person' is a non-full-relative XPath expression) is specialized to <*A s='./People/Person'*> ('*./People/Person*' is a full-relative XPath expression).

Figure 3 shows the fragments of the output of pattern specialization process, named XSL1_S and XSL2_S. We omit showing the detailed programs here due to the space restrictions.



```
<?xml version="1.0" encoding ="UTF-8"?>
<xsl:stylesheet version ="2.0" xmlns:xsl="http:// www.w3.org/1999 /XSL /
Transform">

    ...

 <xsl:template match ="/Factory /Name">
   <xsl:value-of select ="./text ()"/>
 </xsl:template >

 <xsl:template match ="/Factory /Department /Employees /Employee /Name">
   <xsl:value-of select ="./text ()"/>
   <br/>
 </xsl:template >

    ...

</ xsl:stylesheet >
```

(a) XSL1_S

```
<?xml version="1.0" encoding ="UTF-8"?>
<xsl:stylesheet version ="2.0" xmlns:xsl="http ://www.w3.org /
1999/ XSL/Transform">

 ...

 <xsl:template match ="/FactoryInfo ">
   factory is : <xsl: value-of select ="Introduction "/>
   <br/ >
   <br/ >
   <xsl:apply -templates select ="./People /Person"/>
 </xsl:template >

 <xsl:template match ="/FactoryInfo /People /Person">
   PersonName: <xsl:apply-templates select ="PN "/><br/>
  PersonAge : <xsl:value-of select ="Age "/>
   <br/ >
  PersonPhone : < xsl:value-of select ="Phone "/>
   <br/ >
   <xsl:apply -templates select ="Address "/>
   <xsl:apply -templates select ="Position "/>
   <xsl:apply -templates select ="WorkIn"/>
 </xsl:template >

 ...

 <xsl:template match ="/FactoryInfo /People /Person/WorkIn">
  Work in :
  Group <xsl: value-of select ="Group"/> of
  <xsl:value -of select ="Unit"/> Unit of
  <xsl:value -of select ="Factory "/>
   <br/ >
   <br/ >
 </xsl:template >

 <xsl:template match ="/FactoryInfo /Introduction ">
   Factory Introduction is :<xsl:value-of select ="."/>
   <br/ >
 </xsl:template >
</ xsl: stylesheet >
```

(b) XSL2_S

Fig. 3. Fragments of the XSLT programs output after pattern specialization

This kind of automatic resolution is not feasible for *rich* templates and human interaction is needed to guide the process. Specifically, the designer is asked by the system to give a new XPath expression based on the global DTD, to replace the XPath expression based on the initial DTD. Then, these templates with new selection pattern(s) and/or construction pattern(s) will be marked and the subsequent processing steps of template translation and lost template processing need not be applied.

Pattern specialization is a direct way to determine accurately the DTD-Graph node to which the selection pattern refers. However, it might generate some redundant templates which could cause unreachable template(s), missing template(s) and invalid template calling relationship(s) because i) the 'new' template selection pattern may not be harmonious with its inner construction pattern (invalid template calling relationship); ii) the created template which uses the 'new' full-absolute selection pattern might never be called by another construction statement during XSLT execution (unreachable template); iii) The newly created construction pattern might call a non existent template (missing template). These possible 'side-effects' can be detected and eliminated by using Template Association Graph (TAG) [10].

4.4 Template Translation

After pattern specialization, XPath expressions next need to be translated so that they use the vocabulary of the global DTD (DG3).

Let’s see an example. The mapping rule at row 4 of table 2 shows that XPath expression ’*/FactoryInfo/People/Person*’ over the initial schema is mapped to the XPath expression *’/Factory/Department/Employees/Employee’* over the global schema. Thus, the corresponding template *<T m=’/FactoryInfo/People/Person’>* in XSL2_S (figure 3) will be translated into *<T m=’/Factory/Department/Employees/Employee’>*.

Similar to the selection patterns, the construction patterns also need to be translated. The construction pattern using a full-absolute XPath expression can be translated based on the mapping rules directly. A construction pattern that uses a full-relative XPath expression implies a relationship between the *nodes* located by the selection and construction patterns in that template. E.g. suppose *nodes* ‘a’ and ‘b’ are in an *ancestor-descendant* relationship in one of the initial DTD-Graphs. Suppose the nodes that each maps to in the global DTD-Graph are ‘A’ and ‘B’. We then have two situations: 1) ‘B’ is a ‘descendant’ or ‘sibling’ or ‘preceding’ node of ‘A’; 2) ‘B’ is an ‘ancestor’ of ‘A’. In the former case, our method translates the initial construction pattern automatically into a full-relative XPath expression of the context node. In the latter case, human interaction is required to build the new template manually. Specifically, if ‘B’ is the ‘descendant’ node of ‘A’, the construction pattern is translated to the full-relative XPath expression based on the context node ‘A’. For example, the ancestor-descendant relationship between the selection pattern of *<T m=‘FactoryInfo’>* and the construction pattern of *<A s=‘./People/Person’>* in XSL2_S (figure 3) based on DG2 (figure 1 (b)) is preserved in their mapped nodes ‘*/Factory*’ and ‘*/Factory/Department/Employees/Employee*’ based on DG3 (figure 1 (c) and table 2). So, *<A s=‘./People/Person’>* is translated as *<A s=‘Department/Employees/Employee’>*. If ‘B’ is ‘sibling’ or ‘preceding’ node of ‘A’, and if there exists node ‘*C*’, the closest common ancestor node of both ‘A’ and ‘B’ in the global DTD-Graph, the translated construction pattern is an XPath expression which starts with ‘*ancestor::C*’, followed by the full path from ‘C’ to ‘B’. For example, in XSL2_S (figure 3 (b)), template *<T m=‘WorkIn’>* contains a construction statement *<A s=‘Unit’>* and node ‘WorkIn’ is the parent node of ‘Unit’ in DG2. Based on MAP2, they are mapped to ‘WorkIn’ and ‘DN’ in DG3 and ‘DN’ is the ‘preceding’ node of ‘WorkIn’ node. Thus, we find ‘Department’, the common and closest ancestor node of ‘DN’ and ‘WorkIn’, and then create the construction statement *<A s=‘ancestor::Department/DN’>* during the template translation step.

However, if ‘B’ is the ancestor of ‘A’, human interaction is required to do the translation, due to the high degree of change in structure. The designer is asked to i) provide the new XPath expression(s) for the selection pattern or construction pattern(s) or both or; ii) provide a new template to replace the original one.

4.5 Lost Template Processing

During XSLT integration, there may be initial XSLT templates whose selection pattern refers to XML elements which do not get mapped to any element in the global DTD. This causes a problem when translating this initial template into a global template. The same problem happens for construction patterns too. Looking back at table 2 and figure 1 (b) of the XSLT integration example in section 4.2, the node indicated by '*/FactoryInfo/Introduction*' based on DG2 doesn't have any mapped to node in DG3. The corresponding template <*T m=' Factory-Info/Introduction'*> has become a lost template in XSL2_S as a result of doing the translation. We need to correct such lost templates during the integration process.

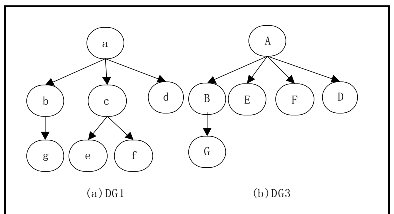


Fig. 4. The example of mapping lost

We cannot simply delete the lost template or construction statement, since i) the body of the lost template might contain valuable data processing, or ii) the inner construction statement of the *lost* template might be the only caller of another existing template, and in this case, deleting the lost template will cause a new missing template.

The integration system detects any lost templates and informs the designer, who then has the task of deciding whether to delete the lost template, or whether to provide a new XPath expression for its selection pattern and, consequently, confirm each element inside this template (i.e provide a new construction pattern for that element or create a new element to replace that element or just delete that element).

Some kinds of lost template cases (shown in figure 4) can be processed automatically based on the integration framework. Looking at figure 4, (a) is an initial DTD-Graph DG1, and (b) is a corresponding global DTD-Graph DG3. Nodes 'a', 'b', 'd', 'e', 'f', 'g' in DG1 are mapped to 'A', 'B', 'D', 'E', 'F', 'G' in DG3. Obviously, the non-terminal element node 'c' is lost during the integration. Moreover, the children nodes of 'c' (i.e. 'e' and 'f') are mapped to children nodes ('E' and 'F') of node 'A' in DG3. This is a common situation for data structure mapping in XML integration and indeed it is reasonable to expect that a parent element covers all concepts of its descendant element. Suppose the template that locates the lost element node 'c' is <*T m='/a/c'*>, lost template processing replaces the selection pattern '/a/c' with its prefix selection pattern '/a', and then, <*T m='/a'*> is translated into <*T m='A'*> in XSL1_T (or XSL2_T) if there is no <*T m='A'*> already existing in XSL1_T (or XSL2_T). If <*T m='A'*> exists in XSL1_T (or XSL2_T), the system only translates the body of <*T m='/a/c'*> and appends it at the end of the existing template <*T m='A'*>. Based on the example shown in figure 4, if template <*T m='/a/c'*> contains a construction statement <*A s='./e'*>, it will be translated to <*A s='./E'*> and appended at the end of template <*T m='A'*>.

The output XSLT programs, after the template translation and lost template processing steps have been performed, are termed as XSL1_T and XSL2_T (we omit these two programs due to the restrictions of space).

4.6 Program Integration

Following the steps of pattern specialization, template translation and lost template processing, our XSLT integration framework applies the program integration step to generate the global XSLT XSL3, based on XSL1_T and XSL2_T.

The templates among XSL1_T plus XSL2_T can be classified into two classes: i) *Unique* templates, whose *distinguished* XPath selection pattern is unique among all templates; ii) *Potentially conflicting template*s (recall Definition_1).

A unique template will be moved to XSL3 directly, without any modification, since it is the only choice of template for a specific XML node or set of nodes.

For a potentially conflicting template pair, the framework (or designer) must make its choice when generating the global template in XSL3. If one or both template(s) has a '*rich*' selection pattern or construction pattern, human interaction is required. The designer needs to decide i) what new template needs to be generated for XSL3, ii) what the template functionality should be.

The templates of potentially conflicting template pair that both use '*simple*' XPath expression(s) as selection pattern(s), can be integrated semi-automatically. We now discuss how to deal with this case.

Suppose <T1> is a template in XSL1_T with body B1 and <T2> is a template in XSL2_T with body B2 and <T1> and <T2> are a pair of *potentially conflicting templates*. Each body B is assumed to be a set of functionality blocks. There a number of possible relationships between B1 and B2. Loosely speaking, these are: 1) B1=B2 (the two templates are guaranteed to give exactly the same result), 2) $B1 \subset B2$ or $B1 \supset B2$ (the output of one template is subsumed by the output of the other), 3)$B1 \bigcap B2 = \Phi$ (the templates are independent), 4) $B1 \bigcap B2$!= Φ (the output of the templates may overlap).

Precisely determining the relationships between template bodies is undecidable. We can develop tests based on *syntactic criteria* (e.g. do a pairwise comparison between the statements in each template body). This may be effective when the components of template body are simple. More complex tests may be based on *semantic criteria*, which concerns the data retrieved from XML source tree and ignores the constant data (strings) and data format (the data order and display styles) of template output. The work in [25] describes a technique where tests for template equivalence are performed by translating the template logic into an XML query algebra [25] and then judging if two templates yield the same result by applying the evaluation rules. Different analysis techniques could also be used.

Based on the different relationships between the functionality (bodies) of the potentially conflicting template pair, our integration approach builds the functionality of the global template according to the rules described in table 3. Human interaction is required in when the static analysis is too difficult or yields imprecise results.

Table 3. Building the new functionality of the global template

Relationship	Process
B1= B2	B1 is chosen as the global functionality
$B1 \supset B2$	B1 is recommended as the global functionality.
$B1 \subset B2$	B2 is recommended as the global functionality.
$B1 \bigcap B2 = \Phi$ or $B1 \bigcap B2 != \Phi$	The designer is asked to decide.

Finally, the unreachable templates, missing templates and invalid template calling relationships are marked (based on checking TAG3) as referential information for possible further action and modification by the designer.

5 Related Work

To the authors' knowledge, no other work has been done on XSLT integration.

A number of integration systems have been developed for semi-structured data and XML. One major kind of XML integration method is view/schema based XML integration (virtual integration) [21, 25]. Another major method is called instance based XML integration [5, 7, 12, 27].

An XSLT template call-graph was described in [14] as part of a translation scheme from XSLT to SQL.

Testing equivalence of XSLT templates is examined in [25]. This work presents a powerful XML query algebra TAX and provides a collection of template equivalence rules. Based on the approach, XSLT templates are automatically translated into TAX and they are judged to be equivalent if they satisfy certain evaluation rules.

XPath analysis and XPath based XML query optimization have been considered in a large number of papers [1, 9, 21]. Any such analysis techniques can in principle be used within our framework.

6 Conclusions and Future Work

In this paper, we have proposed a novel framework for XSLT integration. Our approach is applicable for instance based XML integration methods, where server-side XSLT applications are being used. It consists of four major parts: 1) Pattern Specialization, 2) Template Translation, 3) Lost Template Processing and 4) Program Integration. We believe this new framework can be a significant aid to the designer in integration scenarios. Importantly, our framework is extensible. A variety of analysis techniques can be plugged in to provide enhanced precision.

As part of future work, we would like to investigate methods for handling further XSLT syntax, such as the use of functions and other XPath axes automatically. We also plan to investigate and extend our algorithm to provide more flexible mechanisms for the designer, as part of the global template generation process.

Acknowledgement

This work is partially supported by National ICT Australia. National ICT Australia is funded by the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council.

References

[1] S. Abiteboul and V. Vianu.: Regular path queries with constraints. In *Proc.of 16th ACM SIGACT-SIGMOD-SIGSTART Symposium on Principles of Database Systems,*Tucson, AZ, US (1997) 122-133

[2] J. Bailey, A. Poulovassilis, P. T. Wood.: An Event-Condition-Action Language for XML *Proc.Conf.WWW2002,* Honolulu, Hawaii, USA (2002) 486-495

[3] E. Bertino and E, Ferrari.: XML and Data Integration. Internet Computing, IEEE (2001)

[4] S. Boag et al.: XQuery 1.0: An XML Query Language W3C Candidate Recommendation 3 November 2005. http://www.w3.org/TR/xquery/

[5] P. Bohannon, S. Ganguly, H. Korth, P. Narayan, and P. Shenoy.: Optimizing view queries in ROLEX to support navigable result tree. *In VLDB,* HongKong, China (2002) 119-130

[6] T. Bray, et al.: W3C Recommendation. Extensible Markup Language (XML) 1.0 (2000)

[7] M. J. Carey, D. Florescu, Z. G. Ives, Y. Liu, J. Shanshanmugsundaram, E. J. Shekita, and S. N. Subramanian.: XPERANTO: Publishing object-relational data as XML. In *Proc.of WebDB* (2000) 105-110

[8] J. Clark.: W3C Recommendation. XSL Transformations (XSLT) version 1.0 (1999)

[9] A. Deutsch and V. Tannen. Containment and integrity constraints for XPath. Proc. *KRDB 2001, CEUR Workshop Proceedings 45* (2001)

[10] C. Dong and J. Bailey.: The static analysis of XSLT programs. In *Proc.of The 15th Australasian Database Conference*, Vol.27, Pages 151-160, Dunedin, New Zealand (2004)

[11] W. Fan, Minos Garofalakis, Ming Xiong, Xibei Jia.: Composable XML integration grammars. In *Proc.of ACM CIKM*, Washington, D.C., USA (2004) 2-11

[12] F. M. Fernandez, A. Morishima, and D. Suciu.: Efficient evaluation of XML middle ware queries. *In SIGMOD 2001.*

[13] R. Goldman and J. Widom.: DataGuides: Enabling query formulation and optimization in semi-structured database. Proc. *Int'l Conf on VLDB*, Athens, Greece (1997) 436-446

[14] S. Jain, R. Mahajan and D. Suciu.: Translating XSLT Programs to Efficient SQL Queries. In *Proc.of WWW 2002*, 616-626

[15] Euna Jeong and Chun-Nan Hsu.: Induction of integrated view for XML data with heterogeneous DTDs. In *Proc.of CIKM*, Atlanta, Georgia, USA (2001) Pages: 151 – 158

[16] M. Kay.: Anatomy of an XSLT Processor. (2001)

[17] M. Kay.: Saxon XSLT Processor. http://saxon.sourceforge.net/

[18] C. Laird.: XSLT powers a new wave of web, 2002. http://www.linuxjournal.com/article/5622

[19]D. Lee, W. Chu.: Comparative analysis of six XML schema languages. *ACM SIGMOD Record archive Volume 29, Issue 3*. ACM Press, New York, NY, USA (2000) 76–87

[20] M. L. Lee, L. H. Yang, W. Hsu, X. Yang.: XClust: clustering XML schemas for effective integration. In *Proc.of CIKM* (2002) 292-299

[21] Q. Li, B. Moon.: Indexing and querying XML data for regular path expressions. *Proc. Int'l Conf on VLDB*, Roma, Italy (2001) 361-370

[22]H. Ma, K. Schewe, B. Thalheim, J. Zhao.: View Integration and Cooperation in Databases, Data Warehouses and Web Information Systems. *Data Semantics IV* (2005)

[23] S. Maneth and F. Neven.: Structured document transformations based on XSL. In Proc.of DBPL'99, Kinloch Rannoch, Scottland (1999)

[24] K. Passi, L. Lane, S. Madria, B. Sakamuri, M. Mohania, S. Bhowmick.: A model for XML schema integration. In *Proc.of The third International Conference on E-Commerce and Web Technologies*, Aix-en-Provence, France (2002) 193 - 202

[25] A. Trombetta and D. Montesi.: Equivalences and optimizations in an expressive XSLT fragment. In *Proc.of IDEAS 2004*, Coimbra, Portugal (2004) 171-180

[26] W3C. XSL transformations (XSLT) version 2.0. http://www.w3.org/TR/xslt20/.

[27] Wanxia Wei, Mengchi Liu, and Shijun Li.: Merging of XML documents. *In 23rd Intenational Conference on Conceptual Modelling*, ShangHai, China, November 2004

[28] W3C.: XML Path Language(XPath) Recommendation. http://www.w3.org/TR/xpath.

[29] C. Yu, L. Popa.: Constraint-based XML query rewriting for data integration, In *Proc.of The 2004 ACM SIGMOD international conferenc onference on management of data* (2004).