Static Analysis of XSLT Programs

Ce Dong James Bailey

Department of Computer Science and Software Engineering
The University of Melbourne
Victoria 3010, Australia

{cdong, jbailey}@cs.mu.oz.au

Abstract

XML is becoming the dominant standard for representing and exchanging data on the World Wide Web. The ability to transform and present data in XML is crucial and XSLT (Extensible Stylesheet Language Transformations) is the principal programming language that supports this activity. Methods for analysis of XSLT programs are currently an important open issue. In this paper, we discuss new methods for analysing XSLT programs, which return information about *reachability*, *invalid calling relationships* and *termination* properties. Our methods are based on the determination of the associations which can exist between components of an XSLT program, refined by the knowledge from a DTD. Such analysis is important for debugging and verification of XSLT programs and also their optimisation.

Keywords: XSLT, XML, termination analysis

1 Introduction

The extensible markup language XML has recently emerged as a new standard for information storage, representation and exchange on the World Wide Web. By virtue of its self-describing and textual nature, XML is expected to be used in large volumes and extracted from diverse data sources and applications on the Web. Extensible Stylesheet Language Transformations (XSLT) (Clark 1999) is a popular language for processing XML, especially in data transformation, reorganization, querying and formatting. Indeed, XSLT is used as a primary technology for XML data and server side XSLT applications have become extremely important for XML data exchange and publishing.

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. This recursive application of templates is an essential aspect of XSLT. However, important problems can arise in designing XSLT templates. Firstly, some templates within an XSLT stylesheet [1] may never be applied during execution, regardless of the XML source being input. We call such templates *unreachable templates*. Secondly, there may exist pairs of templates, which appear to call each other, based on the syntactic structure of the program, but in fact cannot, due to underlying constraints which exist within an accompanying DTD. We call these *invalid template calling relationships*. Thirdly, the XSLT program itself may loop forever on some XML input(s). This is the problem of XSLT *termination*. Analysis methods that detect these problems would offer valuable support for the programmer in debugging and stylesheet design. Current XSLT processors and tools unfortunately do not offer any support (e.g. M. Kay 2003).

In this paper, we provide algorithms for dealing with all three of these problems. Our techniques rely on the definition of three important data structures. The *DTD-Graph*, which captures hierarchical information within the accompanying DTD, plus two variations of a *Template and Association Graph*(*TAG*), for modelling components within the XSLT program and relationships that exist between them. The *Raw-TAG*, for modelling the original XSLT designed by the user and the *Refined-TAG,* which further uses information from the *DTD-Graph*, to achieve a more precise model.

Our principal contributions in this paper are twofold:

- The identification of four important static analysis questions for XSLT and the definition of algorithms for determining them: 1) Analysing reachability, 2) Analysing invalid calling relationships, 3) Analysing missing templates, 4) Analysing termination. We are not aware of any other work which has addressed these problems.
- The definition of important data structures that are used in the analysis and which also have more general applicability (e.g. for potential XSLT program optimisation).

The remainder of this paper is organized as follows: Section 2 reviews some basic concepts about DTDs, XSLT and XPath and introduces some definitions. In section 3, we introduce the *Raw-TAG* and the *Refined-TAG* and methods for their construction. Section 4 overviews the steps of our analysis and section 5 formally defines the analysis properties. Section 6 discusses related

[1] In this paper we will use the terms XSLT stylesheet and XSLT program interchangeably.

work and section 7 provides a summary and ideas for future work.

2 Background

We begin by briefly reviewing some concepts regarding DTDs, XPath and XSLT, though we assume the reader already has a basic knowledge of these. We also introduce *DTD-Graph*s and define an abstract view of XSLT syntax, which forms the basis for our techniques.

2.1 DTDs

An XML DTD (Bray et al 2000) provides a structural specification for a class of XML documents. It is used for validating the correctness of XML data. An example DTD is in figure 1.

```
<!ELEMENT PLAY (TITLE, PERSONAE, SCNDESCR, PLAYSUBT,
ACT+)>
<!ELEMENT PERSONAE (TITLE, (PERSONA | PGROUP)+)>
<!ELEMENT PGROUP (PERSONA+, GRPDESCR)>
<!ELEMENT ACT (TITLE, SUBTITLE*, SCENE+)>
<!ELEMENT SCENE (TITLE, SUBTITLE*, (SPEECH |
STAGEDIR)+)>
<!ELEMENT SPEECH (SPEAKER+, (LINE)+)>
<!ELEMENT TITLE (#PCDATA)>
<!ELEMENT PERSONA (#PCDATA)>
<!ELEMENT GRPDESCR (#PCDATA)>
<!ELEMENT SCNDESCR (#PCDATA)>
<!ELEMENT PLAYSUBT (#PCDATA)>
<!ELEMENT SPEAKER (#PCDATA)>
<!ELEMENT LINE (#PCDATA)>
<!ELEMENT STAGEDIR (#PCDATA)>
<!ELEMENT SUBTITLE (#PCDATA)>
<!ATTLIST PLAY CATEGORY CDATA #REQUIRED>
```

Figure 1: play.dtd for the XML document of Shakespeare's plays

<!ELEMENT> in the DTD is used for declaring the syntax of an element. *<!ATTLIST>* is used to declare the syntax of attribute(s) of an element.

In figure 1, *<!ELEMENT PLAY (TITLE, PERSONAE, SCNDESCR, PLAYSUBT, ACT+)>* shows that the *PLAY* element contains an element sequence of *TITLE, PERSONAE, SCNDESCR, PLAYSUBT* and *ACT*. *<!ELEMENT TITLE (#PCDATA)>* shows the declaration for the leaf element *TITLE*, namely, an element with parsed character text value. *<!ATTLIST PLAY CATEGORY CDATA #REQUIRED>* shows that there exists an attribute *CATEGORY* of element *PLAY*.

DTDs use well-known regular expression syntax for describing the structure of child elements:

- +: one or more occurrences of a child element.
- *: zero or more occurrences of a child element.
- ?: zero or one occurrence of a child element .
- element_1 , element_2 : element_1 followed element_2.
- element_1 | element_2: Either element_1 or element_2 not both.

2.2 DTD-Graph

We define a structure we call the *DTD-Graph*, for summarizing the hierarchical information within a DTD and removing information that isn't necessary for our analysis. This is a rooted, node-labelled graph, where each node represents either an element or an attribute from the DTD and the edges indicate element nesting.

Rules used when generating the *DTD-Graph* are

- The *, ? and + symbols are ignored. For example, both SUBTITLE* and SUBTITLE+ are treated as SUBTITLE in the *DTD-Graph*.
- The alternation operator "|" is treated as conjunction. For example, <!ELEMENT PERSONAE (TITLE, (PERSONA | PGROUP)+)> will be regarded as <!ELEMENT PERSONAE (TITLE, PERSONA, PGROUP)>
- All attribute declarations are treated as element declarations in the variation file of original DTD and start with symbol @. For example, *CATEGORY* attribute declared in *<!ATTLIST PLAY CATEGORY CDATA #REQUIRED>* will be merged into its parent element declaration as a child element use the form as : *<!ELEMENT PLAY (@CATEGORY, TITLE, PERSONAE, SCNDESCR, PLAYSUBT, ACT)>*

The *DTD-Graph* is a graphical version of the resulting DTD after the application of these rules. It is a lossy transformation of the original DTD. This loss is essentially with respect to information about "*how many"* children can occur under a parent element. Information about the "*possibility of existence*" of a particular child element is retained. This means that all documents adhering to the original DTD, are also valid with respect to the *DTD-Graph*. The *DTD-Graph* is similar to the Dataguide structure described in (Goldman and Widom 1997).

For example, the DTD in figure 1 can be represented by the graph in figure 2:

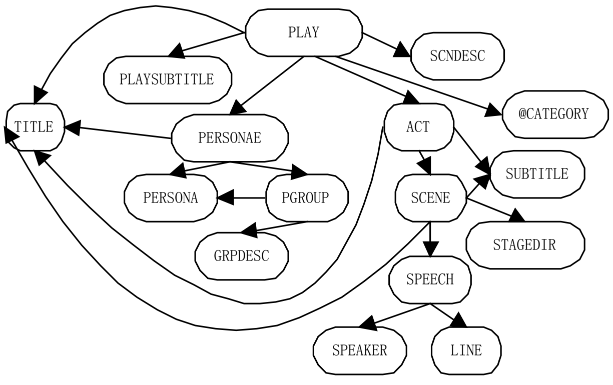


Figure 2: A DTD-Graph of DTD in figure 1

We will later use the *DTD-Graph* to help eliminate potential calling relationships in the XSLT program. A more detailed description of *DTD-Graph* generation is given in figure 3.

```
Algorithm_1: DTD-GraphGeneration
Input:    DTD
Output:   DTD-Graph
//Pre-generation
[01]retrieve <!ELEMENT> and <!ATTLIST>  tags from DTD
[02]delete all "*", "+" and "?" symbols in <!ELEMENT> tags
[03]replace symbol "|" in <!ELEMENT>  with symbol ","
[04]delete all inner parentheses in  <!ELEMENT> tags
[05]foreach <!ATTLIST> do
[06]        merge ATTIBUTE_NAME into corresponding
                  <!ELEMENT> tag within  the outer parentheses
                  as children ELEMENT_NAME
[07]        delete <!ATTLIST>
[08]endforeach
//Generation
[09]Graph DTD_Graph=EMPTY
[10]foreach <!ELEMENT> tag do
[11]        create a node in DTD_Graph
[12]        if (<!ELEMENT> is NOT a leaf node) then
[13]                create nodes for its child ELEMENTs in the
                          DTD-Graph
[14]                create edges from ELEMENT node to each child
                          ELEMENT
[15]        endif
[16]endforeach
[17]output DTD-Graph
```

Figure 3: Algorithm for DTD-Graph generation

2.3 Templates in XSLT

We next outline an abstract view of XSLT syntax. We will assume familiarity with the basics of XSLT. Our analysis techniques are applicable to XSLT programs that make use of the following constructs: *<xsl:apply-templates>, <xsl:template>, <xsl:for-each>, <xsl:if>, <xsl:choose>, <xsl:value-of>, <xsl:copy-of>*. This represents a reasonably powerful and commonly used fragment of the language. We also place some restrictions on XPath expression syntax, described later.

It is well known that an XML document can be modelled as a tree. In XSLT, one defines templates (specified using the command *<xsl:template>*) that match a node or a set of nodes in the XML document tree. XSLT templates enable the designer to specify how the transformation should be carried out. Execution of an XSLT program essentially corresponds to 'walking' through the tree, applying the appropriate templates. Each template has a *selection pattern,* specified using the *match* attribute of the *<xsl:template>* element, to indicate for which nodes the template is applicable. The content of the template specifies how that node or set of nodes should be transformed. Before getting into the details of XSLT, we first give a working example of an XSLT program in figure 4, that will be used throughout the remaining part of this paper.

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

<xsl:template match="/">
      <xsl:apply-templates select="PLAY"/>
</xsl:template>

<xsl:template match="PLAY">
      <xsl:for-each select="PERSONAE/PERSONA">
            <xsl:copy/>
      </xsl:for-each>
      <xsl:for-each select="STAGEDIR">
            <xsl:copy/>
      </xsl:for-each>
      <xsl:if test="ACT/TITLE='ACT I'">
            <xsl:apply-templates select="ACT/SCENE/TITLE"/>
      </xsl:if>
      <xsl:if test="ACT/TITLE='ACT II'">
            <xsl:for-each select="ACT/SCENE/LINE">
                  <xsl:copy/>
                  </xsl:for-each>
      </xsl:if>
      <xsl:apply-templates select="PERSONAE"/>
      <xsl:apply-templates select="ACT/STAGEDIR"/>
</xsl:template>

<xsl:template match="PERSONAE">
      <xsl:apply-templates select="PGROUP/PERSONA"/>
</xsl:template>

<xsl:template match="PGROUP/PERSONA">
      <xsl:apply-templates select="//PERSONAE"/>
</xsl:template>

<xsl:template match=" ACT/STAGEDIR ">
      <xsl:copy/>
</xsl:template>

</xsl:stylesheet>
```

Figure 4: XSLT example designed using the DTD shown in figure 1

Looking at the example, we see the template element *<xsl:template match="PLAY/ACT">* represents the node or node set in the XML tree matching the XPath *selection pattern* "*PLAY/ACT*". i.e. all *ACT* nodes with *PLAY* as parent node. We will henceforth use <t> to denote use of the *<xsl:template>* element.

When the XSLT processor finds a node that matches <t>'s pattern, that node becomes the context node, and further processing is then performed with respect to that node.

2.4 XSLT Template Calling Relationships

We now briefly outline the kinds of calling elements and relationships within an XSLT program.

2.4.1 <xsl:apply-templates>

The *<xsl:apply-templates>* element is used to instruct the XSLT processor that it should find any matching templates for the child node of the current context node. A *construction pattern* may optionally be specified using the *select* attribute in *<xsl:apply-templates>,* to select the nodes for which template matches need to be found. If no *construction pattern* appears in a template, then all children of the current node are processed. In this case, the processor will provide a series of built-in templates, to process corresponding nodes. For specifying *template calling relationships* in XSLT, we use <a> to denote the *<xsl:apply-templates>* statement with (possibly) a *construction pattern*. A simple skeleton example of an XSLT *template calling relationship* is shown in figure 5.

```
<t0>
        <a1 >
        …
</t0>
<t1> … </t1>
```

Figure 5: A simple template calling relationship activated by element <a> inside <t>

In this figure, template <t0> calls template <t1> by application of the statement <a1> inside <t0>. Any node satisfying the *construction pattern* of <a1> must also satisfy the *selection pattern* of <t1>, for it to be applicable.

An XSLT fragment extracted from figure 4 that uses the calling structure <t>-<a>-<t> is listed in figure 6.

```
[01]<?xml version="1.0" encoding="UTF-8"?>
[02]<xsl:stylesheet version="1.0"
            xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
[03]
[04] <xsl:template match="/">
[05]      <xsl:apply-templates select="PLAY"/>
[06] </xsl:template>
[07]
[08] <xsl:template match="PLAY">
      …
[25] </xsl:template>
      …
[35]</xsl:stylesheet>
```

Figure 6: An XSLT fragment with the template calling relationship structure of <t>-<a>-<t>

In figure 6, the template calling will occur from a <t> template *<xsl:template match="/">* to another <t> template *<xsl:template match="PLAY">,* via the <a> calling element *<xsl:apply-templates select="PLAY"/>.* Here, the XSLT processor will check the correspondence of the sets of nodes obtained by evaluating the construction pattern of <a> and the sets of nodes obtained by evaluating the selection pattern of <t>. If the two sets intersect, there is a calling relationship.

2.4.2 <xsl:for-each>

Similar to functional programming, one can write XSLT programs using either a *folding* or an *unfolding* style. The example in figure 5 presented a *folding* style calling relationship. It is generally used for the template calling with multiple callers. To use the *unfolding* style, the element *<xsl:for-each>* is used within a template. This is generally used for template calling relationships having a single caller. We use <f> to denote the *<xsl:for-each>* element with a *construction pattern*. A simple *unfolding* XSLT template calling relationship example is shown in figure 7.

```
<t0>
                <f1>….. </f1>
</t0>
```

Figure 7: A simple example for the unfolding style XSLT template calling relationship

In this example the processor would try to find matches for all children of the context node, satisfying the *construction pattern* in template <f1>. We will denote <t>-<a>-<t> and <t>-<f> for the *folding* and *unfolding* template calling relationship styles respectively.

An XSLT fragment extracted from figure 4 that uses the template calling structure from figure 7 is listed in figure 8. The *<xsl:for-each>* element is in fact considered to be a template and so in this example, there is a calling relationship between the outer template in line 8 and the inner template in line 12.

```
      …
[08] <xsl:template match="PLAY">
      …
[12]      <xsl:for-each select="STAGEDIR">
[13]            <xsl:copy/>
[14]      </xsl:for-each>
      …
[25] </xsl:template>
      …
```

Figure 8: An XSLT fragment with the template calling relationship structure of <t>-<f>

It is worth noting that there are some important differences between function calling in imperative languages such as C or Java and XSLT template calling. In such languages, functions are called explicitly by name, whereas the XSLT template calling mechanism is based on *pattern matching*. Templates are not explicitly called by name, rather, *pattern matching* is activated by a calling statement (e.g. <a> in <t>) or for-each structure (e.g. <f> in <t>) and a template which matches the corresponding pattern expression is chosen for execution.

2.5 <xsl:if> and <xsl:choose>

XSLT uses the elements *<xsl:if>* and *<xsl: choose>* for achieving branching. For both of these cases, we need to describe a more complex *conditional* template calling relationship. In this paper, we use <c> to denote all conditional elements in XSLT, including *<xsl:if>* and *<xsl:choose>*. A simple example is shown in figure 9.

```
<t0>
        <c1><a1></c1>
        <c2><f2>…</f2></c2>
</t0>
<t1> …</t1>
```

Figure 9: A simple example for conditional template calling relationship

The example in figure 9 means that <t0> is the parent template(caller) match for the context node, <t1> will be called iff <c1> is true, the <a1> is a calling statement within <t0>; <f2> will be called iif <c2> is true. We use <t0>-<c1>-<a1>-<t1> and <t0>-<c2>-<f2> to denote the cases of two *conditional* template calling relationship in figure 9.

Two fragments of the XSLT program from figure 4 are shown below in figure 10, which present the *conditional* template calling relationships. The apply statement in line 16 will only occur if the 'if' test in line 15 is true. The 'for-each' template in line 19 will only be called if the 'if' test in line 18 is true.

```
[15]    <xsl:if test="ACT/TITLE='ACT I'">
[16]        <xsl:apply-templates select="ACT/SCENE/TITLE"/>
[17]    </xsl:if>

        …
[18]    <xsl:if test="ACT/TITLE='ACT II'">
[19]        <xsl:for-each select="ACT/SCENE/LINE">
[20]            <xsl:copy/>
[21]            </xsl:for-each>
[22]    </xsl:if>

        …
```

Figure 10: An example of conditional template calling relationship

Table 1 provides a summary for the XSLT template calling relationships we have discussed.

	Folding	Unfolding
unconditional	<t>-<a>-<t>	<t>-<f>
Conditional	<t>-<c>-<a>-<t>	<t>-<c>-<f>

Table 1: Summary for the basic structures of XSLT template calling relationship

2.6 XPath

*Selection pattern*s (S. Maneth and F. Neven 2000) in XSLT are specified using a subset of the XPath language (a separate W3C recommendation) and can be used in the *match* attribute of the *<xsl:template>* elements. For example "*PLAY/ACT/SCENE/SPEECH*" is an XPath expression that selects "SPEECH" nodes from the XML tree along the path of "*PLAY/ACT/SCENE/*".

*Construction pattern*s are specified using the full XPath language and can be used in the *select* attribute of the element *<xsl:apply-templates>*, *<xsl:for-each>* and *<xsl:value-of>*. If the *construction pattern* is missing from an *<xsl:apply-templates>* element, we assume it is equal to '//*' (which is a safe approximation of the semantics for our analysis).

In this paper, we place some further restrictions on the syntax of XPath for use in construction and selection patterns (since the full XPath language is very difficult to analyse precisely). We define a fragment we call simple-XPath(similar to J. Bailey et al. 2002). This disallows the use of any axis other than child, parent, self, descendant-or-self and the use of functions. It represents a useful and reasonably expressive fragment. The syntax is given by the following grammar.

e denotes an XPath expression, *P* denotes a path expression, *q* denotes a qualifier, *n* denotes an element or attribute:

e::= ('/' | '//' | './' | './/') *p*

P::= *P* '/'*P* | *P* '//' *P* | *P* '[' *q* ']' | *n* | * | @*n* |@* | .

q::= *e* | *p*

So an XPath expression starts by establishing a context, followed by a path expression *p* and then an optional qualifier. We allow the use of any construction patterns that conform to this grammar. We allow the use of any *selection pattern*s which conform to this grammar and additionally do not make any use of the symbol '.'.

We now further define some useful terminology and operators for XPath expressions. Expressions enclosed in '[' and ']' in an XPath expression are called *qualifier*s. If we delete all *qualifier*s (along with the enclosing brackets) from an XPath expression, we are left with a path of nodes. We call this path the *distinguished path* of the expression and the node at the end of the distinguished path is the *distinguished leaf* of the expression. The *nodeset* of an XPath expression *e*, is a set of nodes, namely those matched by the *distinguished leaf* of the expression. The *nodeset* of *e,* denoted *nodeset(e)*, is one of an element name *n*, or * (where * denotes any element name), or @*n*, or @* (denoting any attribute). The *nodeset* can be determined as follows.

Let *p* be the distinguished path of *e*. If the *leaf* of *p* is @*n* or @*, *nodeset(p)* is either {@*n*} or {@*}. If the *leaf* of *p* is *n* or *, then *nodeset(p)* is either {*n*} or {*}. Otherwise *nodeset(p)* is {*, @*}, which represents 'all possibilities'.

XPath is also used within *<xsl:if>* elements for describing test conditions. E.g. *<xsl:if test="ACT/TITLE='ACT II' ">*, where *ACT/TITLE* is the XPath test expression. We require XPath test expressions to have the same syntax as *selection pattern*s described above.

Our algorithms will require a function *Eval,* for selecting the set of nodes in the *DTD-Graph*, that match a simple XPath expression *p*. *Eval(p)* applies the XPath expression *p* to the *DTD-Graph*, treating the *DTD-Graph* as a rooted XML document tree (if the *DTD-Graph* has cycles, some extra work to ensure termination is needed, but we omit the details). It thus returns the set of matching nodes in the graph for the expression *p*.

3 XSLT Raw Template and Association Graph (Raw-TAG)

We now describe construction of a *Template and Association Graph* for modelling the structure and calling relationships between the components of the XSLT program. To construct this graph, we first create a succinct version of the original XSLT program. This uses the previously mentioned abbreviations for the xsl element names (*t* for *template*, *a* for *apply-templates*, *f* for *for-each*, etc) and deletes any occurrences of *<xsl:copy-of>* (since it can have no effect on calling relationships). E.g. our previous example now becomes:

```
[01]<?xml version="1.0" encoding="UTF-8"?>
[02]<xsl:stylesheet version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
[03]
[04] <t match="/">
[05]      <a select="PLAY"/>
[06] </t>
[07]
[08] <t match="PLAY">
[09]      <f select="PERSONAE/PERSONA">
[10]
[11]      </f>
[12]      <f select="STAGEDIR">
[13]           <xsl:copy/>
[14]      </f>
[15]      <c test="ACT/TITLE ='ACT I'">
[16]           <xsl:apply-templates select="ACT/SCENE/TITLE"/>
[17]      </c>
[18]      <c test="ACT/TITLE ='ACT II'">
[19]           <f select="ACT/SCENE/LINE">
[20]
[21]           </f>
[22]      </c>
[23]      <a select="PERSONAE"/>
[24]      <a select="ACT/STAGEDIR"/>
[25] </t>
[26]
[27] <t match="PERSONAE">
[28]      <a select="PGROUP/PERSONA"/>
[29] </a>
[30]
[31] <t match="PGROUP/PERSONA">
[32]      <a select="//PERSONAE"/>
[33] </t>
[34]
[35] <t match=" ACT/STAGEDIR ">
[36]
[37] </t>
[38]
[39]</xsl:stylesheet>
```

Figure 11: The succinct XSLT of figure 4

We term the graph that will be created the *Raw-TAG* (Raw Template and Association Graph). The term *Raw* is used to indicate it is based on the "raw'' XSLT program (without reference to the DTD). The *Raw-TAG* is a rooted node-labelled graph. There are two kinds of nodes in *TAG*. Nodes used to represent template elements in XSLT (<t> and <f>), are called *template nodes* and are represented graphically by rounded rectangles. Nodes used to represent calling statement elements (<a>) and conditional elements (<c>) in XSLT and value selections (*<xsl:value-of>*) are called *association nodes* and are represented graphically by diamonds. Information such as *element name*, *pattern value* and *line number* in the XSLT program are recorded as the node label. We use edges between template nodes and association nodes to represent the structure and calling relationships between components of the program.

We note that the *Raw-TAG* will be an approximation of the calling relationships. Edges really represent the 'possibility' that calling might occur. The lack of an edge between nodes, represents the fact that a calling relationship is impossible.

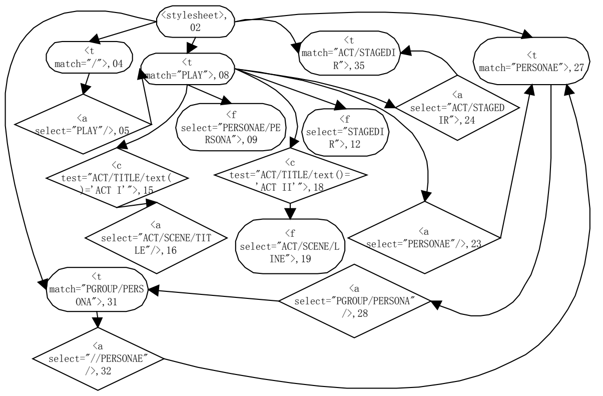
We now describe the rules for creating edges between the template nodes and association nodes in table 2. After creating all the nodes, all pairs of template and association nodes are examined and edges created if certain conditions hold.

Structure	Edge Creation
<t>-<e>	Edge(t,e), if e is one of a,f,v or c and parent(t,e)
<f>-<e>	Edge(f,e), if e is one of a,f,v or c and parent(t,e)
<c>-<e>	Edge(c,e), if e is one of a,f,v or c and parent(t,e)
<a>-<t>	Edge(a,t), if NodeSet(C_a) $\cap$ NodeSet(C_t) <> empty

Table 2: The rules of creating the edge of Raw-TAG

The first three rules in this table are based on checking the existence of parent child relationships between node pairs and follow straightforwardly from the program syntax. The intuition is that if the second element is nested inside the first, then a call can be said to happen between the first and the second. The fourth <a>-<t> rule checks whether the nodes that could be matched by the *construction pattern* of <a> intersect with the nodes that could be matched by the *selection pattern* of <t>. If the intersection is true, then the calling relationship is possible, otherwise it would be impossible.

Figure 12 is the *Raw-TAG* of the XSLT stylesheet shown in figure 4. The root (stylesheet node is a special case and we assume it has an implicit *apply-templates* call).



<stylesheet>, 02
<t match="/">, 04
<t match="PLAY">, 08
<t match="ACT/STAGEDIR">, 35
<t match="PERSONAE">, 27
<a select="PLAY"/>, 05
<f select="PERSONAE/PERSONA">, 09
<f select="STAGEDIR">, 12
<a select="ACT/STAGEDIR">, 24
<c test="ACT/TITLE/text()='ACT I'">, 15
<c test="ACT/TITLE/text()='ACT II'">, 18
<a select="ACT/SCENE/TITLE"/>, 16
<f select="ACT/SCENE/LINE">, 19
<a select="PERSONAE"/>, 23
<t match="PGROUP/PERSONA">, 31
<a select="PGROUP/PERSONA"/>, 28
<a select="//PERSONAE"/>, 32

Figure 12: The Raw-TAG of the XSLT stylesheet in figure 4

It can be seen that there is an edge from the stylesheet to each of the five templates that have been defined. As an example, the template from line 31 (in the bottom left of the graph) can call the template from line 27 (in the top right of the graph), via the apply templates statement on line 32 (represented using the diamond association).

3.1 Refined-TAG

The *Raw-TAG* is based solely on the syntactic structure of the XSLT program. Tests for calling relationships rely only on either element nesting or simple nodeset equivalences. We now describe how to construct a more precise structure, we call the *Refined-TAG*, which uses information from a supplied DTD (assuming one exists) that describes the nature of the XML input. The nodes of the *Refined-TAG* are the same as the nodes of the *Raw-TAG*. The set of edges in the *Refined-TAG* is a subset of the set of edges in the *Raw-TAG* (i.e. some of the *Raw-TAG* edges are eliminated based on DTD information). We will shortly define the rules for selecting which edges are in the *Raw-TAG*, but not in the *Refined-TAG*. We begin with some more notations.

Let C_a, C_v, C_f be arbitrary construction patterns of <a>, <v> and <f> nodes respectively. Let C_t be an arbitrary selection pattern for the <t> node and let C_c be the test expression for an <if> node. Let C_x be one of C_a, C_v and C_f. Then C_x' is the expression C_x modified so that it will be evaluated with respect to all descendants of the root node. The function *Strip* removes an initial './' if one exists.

$C_x' = C_x$, if C_x begins with '/' or '//'

$= //\text{Strip}(C_x)$, otherwise

e.g. If C_x=q/r/s[a], then C_x' = //q/r/s[a]. If C_x = ./q/r/s[a], then C_x'=//q/r/s[a].

Special case: If C_x = '.', then define C_x' = '//*'.

The function *Concat* is used to concatenate two XPath expressions C_x and C_y and make the concatenated result which will be evaluated with respect to all descendants of the root node. If C_y is an expression with respect to the root node then the output is just C_y.

Concat(C_x, C_y) = C_x' /*Strip*(C_y) , if C_y does not begin with '/' or '//'

$=C_y$,otherwise

e.g. *Concat*(q/r/s, ./a/b/c)=/q/r/s/a/b/c.

Concat(q/r/s, /a/b/c)=/a/b/c.

The rules for removing edges are now described. These rules use the function *Eval* to check if XPath expressions are valid with respect to the DTD. Assume *e* is one of {*c,v,a,f*}:

Edge Type	Edge Removal Condition
edge(t, e)	Eval(Concat(C_t, C_e))= empty
edge(a, t)	Eval(C_a') $\cap$ Eval(C_t')= empty

Table 3: The rules of removing the edges from Raw-TAG and create Refined-TAG

The first rule removes an edge if the concatenated path between a <t> node and a {c,v,a,f} node is invalid with respect to the *DTD-Graph*. The second rule removes an edge if none of the nodes that could be selected by a construction pattern in the <a> element intersect with any of the nodes that could be selected by the selection pattern in the <t> element. The *DTD-Graph* is used to find the sets of possible nodes that could be selected by a given pattern.

Figure 13 is the *Refined-TAG* of the XSLT program from figure 4. The edges which have been removed from the *Raw-TAG* are the edge from *<t match="PLAY">* to *<f select="STAGEDIR">*, the edge from *<t match= "PLAY">* to *<f select="ACT/SCENE/LINE">* , the edge from root node(*<stylesheet>)* to *<t match="ACT/STAGEDIR">* and the edge from *<t select= "PLAY">* node to *<t match="ACT/STAGEDIR">*. The first is impossible because the path "*//PLAY/STAGEDIR"* doesn't exist in the DTD graph, the second is impossible because the path "*//PLAY/ACT/SCENE/LINE"* doesn't exist in the DTD graph , the third is impossible because the path "*//ACT/STAGEDIR*" doesn't exist in the *DTD-Graph* and The forth is impossible because the path "*//PLAY/ACT/STAGEDIR"* doesn't exist in the DTD graph (we assume the root node has an implicit *apply-templates* statement).

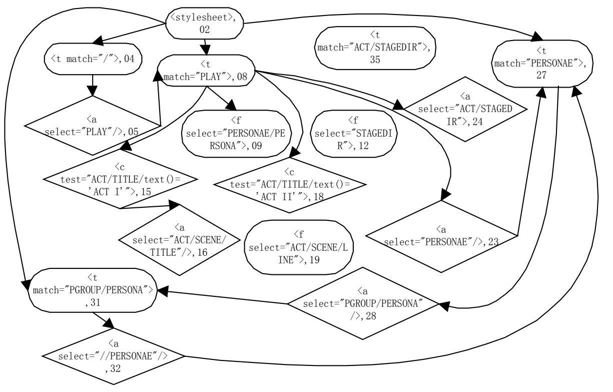

<stylesheet>, 02
<t match="/">, 04
<t match="PLAY">, 08
<t match="ACT/STAGEDIR">, 35
<t match="PERSONAE">, 27
<a select="PLAY"/>, 05
<f select="PERSONAE/PERSONA">, 09
<f select="STAGEDIR">, 12
<a select="ACT/STAGEDIR">, 24
<c test="ACT/TITLE/text()='ACT I'">, 15
<c test="ACT/TITLE/text()='ACT II'">, 18
<a select="ACT/SCENE/TITLE"/>, 16
<f select="ACT/SCENE/LINE">, 19
<a select="PERSONAE"/>, 23
<t match="PGROUP/PERSONA">, 31
<a select="PGROUP/PERSONA"/>, 28
<a select="//PERSONAE"/>, 32

Figure 13: The Refined-TAG of the XSLT stylesheet in figure 4

4 The overview of processing

Figure 14 summarises our analysis approach.

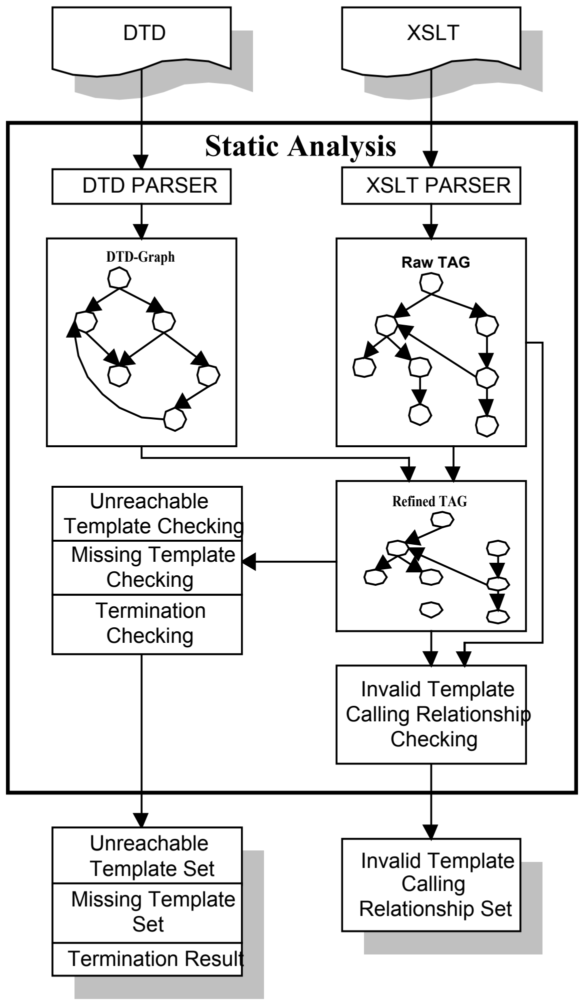


Figure 14: The procedure of XSLT analysis

As can be seen, the main steps in the process are:

Step_1: Parsing the supplied DTD and generating the *DTD-Graph*.

Step_2: Generating the *Raw-TAG* by parsing the given XSLT stylesheet and applying the rules in Table 2

Step_3: Generating the *Refined-TAG* from the *Raw-TAG* by using the *DTD-Graph* and the rules in Table 3.

Step_4: Outputting the desired analysis information (*unreachable template* set, *missing template* set, *invalid template calling relationship* set and *termination* determination result), either by direct examination of the *Raw-TAG*, or by comparing the *Raw-TAG* with the *Refined-TAG*.

In the next section, we discuss the information which can be found by our analysis.

5 Analysis Information

We now describe the properties we can analyse using our method. We envisage all of these properties as being helpful to the XSLT designer in identifying possible program errors.

5.1 Unreachable Templates

Unreachable templates correspond to parts of the program code that will never be executed at run-time. Their presence may represent either an error in the program or a possible opportunity for code optimisation.

Definition_1: An *unreachable template* is a template that will not be matched during XSLT program execution on any input XML document conforming to the given DTD.

We can find sets of *unreachable templates* by direct examination of the *Refined-TAG*, as the following proposition shows.

Proposition_1: An *unreachable template* is a template node in the *Refined-TAG* which cannot be reached by any directed path beginning from the root (stylesheet) node.

Recall that although there exist edges in the *Refined-TAG* that could not be followed at run-time, the absence of edges between nodes has been constructed precisely and thus it is safe to deduce unreachability, if a path can't be found. In the *Refined-TAG* of XSLT example in figure 13, the template nodes labelled by *<f select="ACT/SCENE/LINE">*, *<f match="STAGEDIR">* and *<t match="ACT/STAGEDIR">* correspond to *unreachable* template nodes.

5.2 Missing templates

It is possible that within an XSLT program, there may be a construction pattern which will never be matched by any *selection pattern* of an *<xsl:template>* element (rather like a function call to a non-existent function). We term this situation as a *missing template*. i.e. There appears to be a call to a template which does not exist. This is a likely indication that there is an error in the program. Such *missing templates* can be found again by direct examination of the *Refined-TAG*.

Proposition_2: An <a> association node without any outgoing edge in the *Refined-TAG* indicates the existence of a *missing template*.

Information about such *missing templates* can then be checked by the program author to check for possible errors.

In the *Refined-TAG* of XSLT example in figure 13, the association nodes labelled by <a select="ACT/SCENE/TITLE"/> indicates a missing template.

5.3 Invalid Template Calling Relationships

When writing complex XSLT programs, it is easy for invalid template calling relationships to occur. These are calling relationships which seem to exist due to the syntactic structure of the XSLT program, but in fact cannot occur due to the constraints present within the DTD. The occurrence of such a situation may likely correspond to errors in the XSLT code design, arising from the designer having an inadequate knowledge of the DTD. Such errors would normally be more difficult to detect, since they are not the result of incorrect syntax, but instead manifested by incorrect transformation output.

Definition_2: An *invalid template calling relationship* is a pair of template nodes, with a path existing between them in the *Raw-TAG,* but no path existing between them in the *Refined-TAG*.

For example, the *<t match="PLAY">-<f select="STAGEDIR">*, the *<t match="PLAY">-<f select="ACT/STAGEDIR">* and the *<t match="PLAY">-<t select="ACT/SCENE/LINE">* are three *invalid template calling relationships* in the XSLT stylesheet shown in figure 4.

5.4 Program Termination

An infinite template calling loop can have catastrophic consequences and result in the failure of execution of the transformation. Current XSLT processors offer no support for detecting or handling such infinite behaviour. Instead, outputs are often cryptic stack overflow errors, or a blank output window in the browser. It is thus important to be able to verify whether a given XSLT program can be guaranteed as being terminating for all possible inputs.

Definition_3: An XSLT program is *terminating* if execution halts for all possible input XML documents that conform to the DTD. Otherwise it is *non-terminating*.

A conservative termination analysis can be performed with respect to the *Refined-TAG,* as the following proposition shows.

Proposition_3: An XSLT program is terminating if no cycle exists in the *Refined-TAG*. Otherwise, it is *possibly non-terminating.*

Looking at the example XSLT shown in figure 4, we would conclude that it is possibly non-terminating, since there is a cycle including the nodes of *<t match="PGROUP/PERSONA">, <a select="//PERSONAE"/>, <t match="PERSONAE">* and *<a select="PGROUP/PERSONA"/>* in the *Refined-TAG* in figure 13. Observe that we cannot deduce definite non-termination from the presence of cycle, since edges in the *Refined-TAG* only represent possible calling relationships.

6 Related work

To the authors' knowledge, no other work has been done on graph based static analysis of XSLT.

XPath analysis and XPath based XML query optimization have been considered in a large number of papers, e.g. S. Abiteboul and V. Vianu in 1997, A. Deutsch and M. Fernandez et al in 1999, Li and Moon in 2000. A schema tree based on XML for XML view query was proposed by C. Li, P. Bohannon, H. F. Korth, P.P.S. Narayan. 2003. The schema is established based on the XML and XML query view, but not from the DTD. L. Villard, N. Layaida developed an incremental XSLT transformation processor and provided some XSLT analysis based on the incremental XML. The emphasis is on optimisation rather than analysis though. The work in (Maneth and Neven 2000) gives an automata theoretical analysis of XSLT programs, but does not include the use of a DTD. An XSLT template call-graph was described in (Jain, Mahajan, and Suciu 2002) as part of a translation scheme from XSLT to SQL. It differs from our work, since they focus on the model of <t>-<a>-<t> calling structure and the behaviour of *<xsl:variable>* and *<xsl:param>*. The *<xsl:for-each>* element was not considered and no DTD used. Neither reachability nor termination properties were considered.

7 Conclusion

In this paper, we have proposed a method for static XSLT program analysis based on using a DTD. We demonstrated how to build graphs with calling relationships that are established based on the checking of XPath expressions. We described four important analysis properties which can be deduced from our association graphs: *unreachable templates, missing templates, invalid template calling relationships* and XSLT program *termination*. We believe that discovery of these properties represents valuable information for the program designer.

As part of future work, we would like to investigate extending our analysis to handle further XSLT syntax, such as the use of functions and other XPath axes. The precision of the analysis could potentially be improved by considering extra aspects such as template priority. We also intend to examine the use of the *Refined-TAG* for program optimisation at run-time.

Reference

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

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

P. Buneman, S. Davidson, G. Hillebrand and D. Suciu. (1996): A query language and optimization techniques for unstructured data. *Proc. ACM SIGMOD*, pages 505-516.

T. Bray, J. Paoli, and C. M. Sperberg-McQueen, and E. Maler (2000): W3C Recommendation. Extensible Markup Language (XML) 1.0 http://www.w3.org/TR/REC-xml#dt-doctype.

J. Clark. (1999): W3C recommendation. XSL Transformations (XSLT) version 1.0 http://www.w3.org/TR/xslt.

A. Deutsch and V. Tannen. (2003): Containment and integrity constraints for XPath. *Proc. KRDB 2001,* CEUR Workshop Proceedings 45.

A. Deutsch, M. Fernandez, D. Florescu, A. Levy and D. Suciu. (1999): A query language for XML. *Proc.of 8th Int'l. World Wide Web Conf.*

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

S. Jain and R. Mahajan and D. Suciu (2002): Translating XSLT Programs to Efficient SQL Queries. Proceedings of WWW 2002.

M. Kay. (2000): Saxon XSLT Processor. http://saxon.sourceforge.net/.

M. Kay. (2001): Anatomy of an XSLT Processor. http://www-106.ibm.com/developerworks/library/x-xslt2/

C. Li, P. Bohannon, H. F. Korth, P.P.S. Narayan. (2003): Composing XSL transformations with XML publishing view. *SIGMOD 2003*, San Diego, CA.

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

S. Maneth and F. Neven (2000): Structured document transformations based on XSL.

L. Villard, N. Layaida. (2002): An incremental XSLT transformation processor for XML document manipulation. *Proc. World Wide Web 2002,* Hawaii, USA.

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

World Wide Web Consortium. XML Path Language(XPath) Recommendation. http://www.w3.org/TR/xpath.