Composing XSL Transformations with XML Publishing Views

Chengkai Li[*]
University of Illinois at Urbana-Champaign
cli@uiuc.edu

Philip Bohannon
Lucent Technologies
Bell Labs
bohannon@lucent.com

Henry F. Korth[†]
Lehigh University
korth@cse.lehigh.edu

P.P.S. Narayan
Lucent Technologies
Bell Labs
ppsnarayan@lucent.com

ABSTRACT

While the XML Stylesheet Language for Transformations (XSLT) was not designed as a query language, it is well-suited for many query-like operations on XML documents including selecting and restructuring data. Further, it actively fulfills the role of an XML query language in modern applications and is widely supported by application platform software. However, the use of database techniques to optimize and execute XSLT has only recently received attention in the research community. In this paper, we focus on the case where XSL transformations are to be run on XML documents defined as views of relational databases. For a subset of XSLT, we present an algorithm to compose a transformation with an XML view, eliminating the need for the XSLT execution. We then describe how to extend this algorithm to handle several additional features of XSLT, including a proposed approach for handling recursion.

1. INTRODUCTION

As XML has continued to gain popularity as a standard for information representation and exchange, tools to render and present XML are increasingly supported by common application platforms. Many of these tools implement a W3C standard, the XML Stylesheet Langauge [16]. This language is divided into *transformation* and *formatting* subsystems. As implied by the name, the transformation subsystem reorganizes the tree structure of an XML document and the formatting subsystem renders the result into a display format. In fact, the transformation sublanguage, appropriately called XML Stylesheet Language Transformations, or XSLT, has proven very popular with developers and is often implemented as a stand-alone tool. Some of the well-known implementations of XSLT are XT [18], SAXON [10] and XALAN [13]. Common uses of XSLT include translating XML to HTML, and modifying or selecting part of an XML document. Unlike XQUERY [14], XSLT was not expressly designed as a query language. Nevertheless, XSLT can easily be used for "query-like" transformations. It also shares with XQUERY the use of XPATH [15] for path evaluation.

[*]Work performed while the author was visiting Bell Labs.
[†]Work performed while the author was with Bell Labs.

Despite widespread use of XML standards for business data *exchange*, the vast majority of business data is *stored* and maintained by relational database systems. In fact, XML-publishing middleware technology, proposed by the research community [2, 4, 6, 12], is rapidly being implemented by relational database vendors to ensure that XML-centric applications are well supported. Such middleware provides a declarative *view query* language with which to specify the desired mapping between the relational tables and the resulting XML document. Based on the mapping defined by the view query, a portion of the database can be exported as XML.

Given the respective importance of XML views and XSLT, it is important to propose efficient execution of XSLT stylesheets against XML-publishing views. A straightforward approach for accomplishing this would be to fully materialize the XML view as an XML document, upon which an XSLT stylesheet is evaluated.

This approach is problematic for large documents or complicated stylesheets, simply due to performance problems that commonly arise with XSLT evaluation on large documents. Recent work has explored optimizing the execution of XSLT transformations by incorporating XSLT processing into database query engines [9]. However, even if XSLT evaluation can be made scalable and efficient, it may not be a good solution for XML documents published from relational database systems. For example, it is not actually necessary to materialize many of the nodes in the view query to produce the correct XSLT result document. First, node types not referenced by any XPATH expressions in the XSLT stylesheet need not be materialized. Similarly, nodes that do not match selection conditions in the appropriate XSLT templates are not useful. Finally, intermediate nodes along the XPATH expression need not be materialized if they are not part of the result.

These observations argue for a *view composition* approach to supporting XSLT in relational XML middleware. In this paper, we take the view composition approach and develop techniques that support execution of XSLT against XML-publishing views by first composing, as much as possible, the transformations in a stylesheet with a publishing view, resulting in a composed *stylesheet view*. Evaluating the stylesheet view on a database instance results in the same XML document that would be produced by evaluating the XSLT stylesheet on the original XML view. The stylesheet view does not generate the unnecessary nodes described above, promising improved efficiency. In addition, this approach removes the cost of parsing and XSLT processing. Rather, the XSLT processing is pushed into SQL queries processed by relational query engines. In fact, such view composition algorithms have been designed for user XQUERY or XPATH queries to be efficiently executed by XML publishing middlewares and relational engines [11, 5]. We argue that similar techniques should be developed with XSLT.

To accomplish this, we define a restrictive subset of XSLT that we term XSLT$_{basic}$, and give an algorithm that can compose an XSLT$_{basic}$ stylesheet with an XML view definition to produce a new XML view definition, the *stylesheet* view. At a high level, the algorithm consists of three steps. First, a graph representing the processing done by the XML stylesheet is constructed. Second, this graph is combined with a graph representation of the XML view query by matching pairs of nodes from the two graphs in a manner similar to the creation of a cross-product automaton. Finally, the resulting graph is pruned to remove unnecessary nodes and modified to handle formatting instructions, producing the stylesheet view.

An alternative to composing XSLT with view queries is to translate XSLT to XQUERY [1], and then use techniques such as those proposed in [11] to compose the resulting XQUERY expression with the view query. While this approach is promising, we are not aware of any published techniques for performing this conversion, and it may introduce new complexities. Further, techniques developed in this paper (such as merging select-expressions with match-patterns) would apparently be required for such a conversion.

The only prior work of which we are aware on using SQL engines to execute a class of XSLT transformations is [7]. While a detailed comparison with [7] is deferred to Section 6, our approach differs in three key ways: 1) our algorithm produces an XML view query rather than an SQL query, 2) [7] divides the XSLT transform into a set of paths and processes each separately, while our algorithm composes the stylesheet as a whole with the publishing query, and finally the extensions to our algorithm in Section 5 address features of XSLT such as priority that are not addressed by [7].

The focus of this paper is on the view-composition algorithm, rather than on optimization of either the composition algorithm or the resulting queries. We defer experimental evaluation and full consideration of optimized execution strategies for XSLT view composition queries to future research.

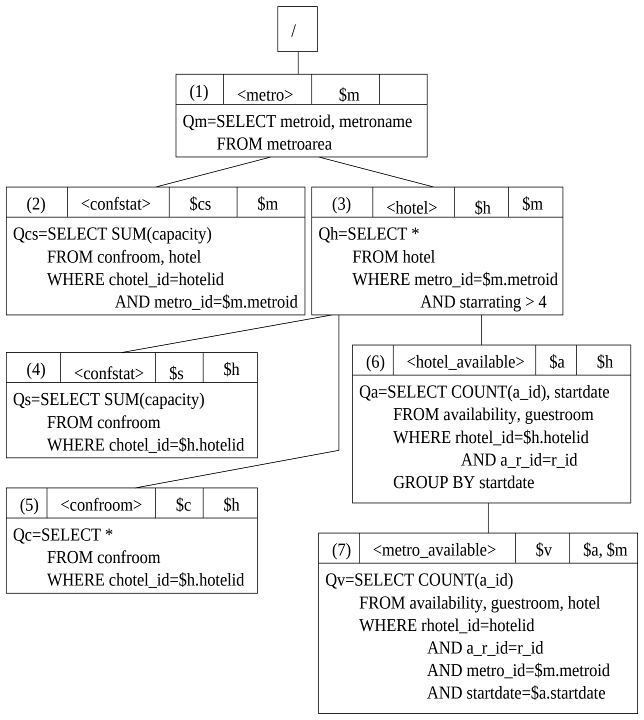
The outline of the remainder of the paper is as follows. In Section 2, we introduce the concepts and notions of XML publishing views and XSLT stylesheets. In Section 3, we overview the algorithm by describing the data structures generated in the four steps of the algorithm. In Section 4, we describe the details of the algorithm, and in Section 5, we describe how to extend the algorithm to handle more general XSLT stylesheets. Related work is discussed in Section 6, followed by our conclusion and a discussion of future work in Section 7.

2. BACKGROUND

In this section, we briefly introduce XML-publishing views and XSLT stylesheets.

2.1 Schema-Tree Queries

In this section, we introduce the notion of a view query in XML-publishing middleware. SILKROUTE [5, 6] defines XML views using RXL and queries them with XML-QL and XPERANTO [4, 11, 12] uses XQUERY. Throughout this paper, we use the view-query specification format as defined in ROLEX [2, 3]. This query format, referred to as a *schema-tree query*, is meant to capture a rich set of XML view queries, and is adapted from the intermediate query representation of SILKROUTE. While ROLEX focuses on a particular system architecture with tight integration between the application and the DBMS, our algorithm for composing XSLT with XML views does not rely on any particular features of ROLEX, and thus we expect it to be readily adaptable to other XML view languages. Details of schema-tree queries can be found in [2]. Below we briefly introduce schema-tree queries using a definition and an illustrative example.



```
Qm=SELECT metroid, metroname
     FROM metroarea

Qcs=SELECT SUM(capacity)
      FROM confroom, hotel
      WHERE chotel_id=hotelid
            AND metro_id=$m.metroid

Qh=SELECT *
     FROM hotel
     WHERE metro_id=$m.metroid
           AND starrating > 4

Qs=SELECT SUM(capacity)
     FROM confroom
     WHERE chotel_id=$h.hotelid

Qa=SELECT COUNT(a_id), startdate
     FROM availability, guestroom
     WHERE rhotel_id=$h.hotelid
           AND a_r_id=r_id
     GROUP BY startdate

Qc=SELECT *
     FROM confroom
     WHERE chotel_id=$h.hotelid

Qv=SELECT COUNT(a_id)
     FROM availability, guestroom, hotel
     WHERE rhotel_id=hotelid
           AND a_r_id=r_id
           AND metro_id=$m.metroid
           AND startdate=$a.startdate
```

Figure 1: Example schema tree view query.

hotelchain(chainid, companyname, hqstate)
metroarea(metroid, metroname)
hotel(hotelid, hotelname, starrating, chain_id
 metro_id, state_id, city, pool, gym)
guestroom(r_id, rhotel_id, roomnumber, type, rackrate)
confroom(c_id, chotel_id, croomnumber, capacity, rackrate)
availability(a_id, a_r_id, startdate, enddate, price)

Figure 2: Hotel reservation schema.

Definition 1: A **schema-tree query** v is a set of nodes $\{n_i\}$, each of which is a 6-tuple ($id(n_i)$, $tag(n_i)$, $bv(n_i)$, $parameters(n_i)$, $\mathcal{Q}_{bv(n_i)}$, $children(n_i)$), where $id(n_i)$ is a unique identifier, $tag(n_i)$ is the tag, $bv(n_i)$ is the *binding variable* of n_i, $parameters(n_i)$ is the set of parameters of $\mathcal{Q}_{bv(n_i)}$, which is the *tag query* of n_i, and $children(n_i)$ is the set of child nodes of n_i.

Figure 1 shows an example of schema-tree query that defines an XML view on the tables of Figure 2, in order to support conference planning by showing candidate hotels along with information about availability of rooms in the same metro area.

The id of a node is used to identify a node uniquely in v. For example, in Figure 1, there are two nodes that have the same tag `<confstat>`, but different ids, 2 and 4. Each tuple returned by the tag query $\mathcal{Q}_{bv(n_i)}$ becomes an element in the resulting XML document with XML tag $tag(n_i)$; this element is said to have been *generated* by n_i. For example, the node with id 1 in Figure 1 has associated with it the tag `<metro>` and the tag query "$\mathcal{Q}_m$ = SELECT metroid, metroname FROM **metroarea**." This query defines a list of metropolitan areas that become sibling nodes in the resulting XML document, each tagged with the `<metro>` tag (a unique document root is implied). Relational attributes from the SELECT clause appear as XML attributes the generated element.

Tag queries may be parameterized by zero or more parameters, associated with binding variables. We refer to the query that defines binding variable $bv(n_i)$ as $\mathcal{Q}_{bv(n_i)}$, with $parameters(n_i)$ as the binding variables used in the body of the query. The binding variable $bv(n_i)$, a tuple variable ranging over the results of $\mathcal{Q}_{bv(n_i)}$, is used as a parameter when specifying the tag queries of descen-

```
<xsl:template match="pattern" mode=integer>
  <Result>
    <xsl:apply-templates select="expression"/>
  </Result>
</xsl:template>
```

Figure 3: Skeleton of XSLT template rule.

```
<xsl:template match="/">                        (R1)
  <HTML>
    <HEAD></HEAD>
    <BODY>
      <xsl:apply-templates select="metro"/>
    </BODY>
  </HTML>
</xsl:template>

<xsl:template match="metro">                    (R2)
  <result_metro>
    <A></A>
    <xsl:apply-templates
       select="hotel/confstat"/>
  </result_metro>
</xsl:template>

<xsl:template match="confstat">                 (R3)
  <result_confstat>
    <B></B>
    <xsl:apply-templates
     select="../hotel_available/../confroom"/>
  </result_confstat>
</xsl:template>

<xsl:template match="metro/hotel/confroom">  (R4)
  <xsl:value-of select="."/>
</xsl:template>
```

Figure 4: An example of XSLT stylesheet.

dant nodes of n_i. For example, the variable m associated with `<metro>` is used as a parameter in tag queries for `<hotel>` and `<metro_available>` to refer to the attribute m.metroid.

The remainder of the view in Figure 1 defines the following. The tag query, $\mathcal{Q}_h(m)$ for `<hotel>` is parameterized by the tuple variable m ranging over metropolitan areas and gives a list of hotels in that metropolitan area. The tag query, $\mathcal{Q}_a(h)$, for `<hotel_available>` counts available rooms at the given hotel in a certain fixed time period, whereas the tag query, $\mathcal{Q}_v(m, a)$ for `<metro_available>` counts the total available rooms in the entire metropolitan area for that same time period. In separate branches of the schema-tree, summary and detail information about conference rooms is given by the nodes with tags `<confstat>` and `<confroom>` respectively.

2.2 XSLT

In this section, we briefly introduce XSLT, show how stylesheets are modeled, and introduce a working example.

Definition 2: An XSLT **stylesheet** x is a set of template rules $\{r_i\}$, each of which is a 4-tuple $(match(r_i), mode(r_i), priority(r_i), output(r_i))$, where $match(r_i)$ is the *match pattern* of r_i, $mode(r_i)$ is the *mode* of r_i, $priority(r_i)$ is the *priority* of r_i, and $output(r_i)$ is the *output-tree fragment* of r_i.

The skeleton of a template rule is shown in Figure 3, and an example stylesheet in Figure 4, which contains four template rules $R1$ through $R4$. The match pattern of a template rule, $match(r_i)$, is a *pattern* [16] and is essentially a subset of XPATH path expressions containing only *child*, *descendant* ("//"), and *attribute* axes [15]. For example, as shown in Figure 4, $match(R2)$ is *"metro"*. The mode of a rule, $mode(r_i)$, is a symbol that allows rules to be partitioned; that is, rule invocations must match in mode as well as match pattern. If there is no mode attribute, the XSLT processor will set it to be a default value. Similarly the priority of a rule, $priority(r_i)$, is an integer, used in conflict resolution and is briefly discussed in Section 2.2.1. The output tree fragment for a rule, $output(r_i)$ controls the structure of a rule's output. For (R2) of Figure 4, the output tree fragment consists of the `<result_metro>` tag and its contents. $output(r_i)$ may contain a set of `<xsl:apply-templates>` nodes, $apply(r_i) = \{a_j\}$, as defined below.

Definition 3: An **apply-templates node**, a_j, is a 2-tuple of the form $(select(a_j), mode(a_j))$, where $select(a_j)$ is the *select expression* of a_j, and $mode(a_j)$ is the *mode* of a_j.

$select(a_j)$ is a subset of XPATH *expressions* [15] intended to ensure that results of the expression are nodes rather than atomic values. In (R2) of Figure 4, there is only one apply-templates node and its select expression is *"hotel/confstat"*. $mode(a_j)$ limits the rules which may match as described above.

2.2.1 XSLT Processing Model

Basic XSLT processing consists of context transitions from a given XML document context node to a new context node recursively, starting from the root as the original context node, concatenating results in traversal order. This is shown as a function $PROCESS$ in Figure 5. Context transition is realized by two functions, $MATCH$ and $SELECT$ [17].

For an XML document context node d_{con} and a rule r_i, the function $MATCH(d_{con}, r_i)$ returns true if $match(r_i)$ matches some suffix of the *incoming* path from the document root to d_{con}. The semantics of a select expression is a function $SELECT$. For an XML document node d_{con} matched by a rule r_i and an apply-templates node $a_j \in r_i$, $SELECT(d_{con}, a_j)$ returns a set of nodes selected by $select(a_j)$, with d_{con} as the document context node.

In step 3 of Figure 5, the algorithm checks that $mode(r_i)$ is the desired mode and that $MATCH(d_{con}, r_i)$ returns true. If multiple rules match (step 2), the XSLT processor employs a conflict detection and resolution scheme that applies only the rule with the highest priority [16]. Once a rule is activated, then, in steps 4-10 d_{con} is used to instantiate $output(r_i)$ to construct the resulting document fragment. $select(a_j)$ evaluated on d_{con} results in a node set D_{new_con}. The apply-templates nodes $a_j \in apply(r_i)$ are replaced with the concatenation of resulting document fragments produced by recursively processing each node $d_{new\_con} \in D_{new\_con}$ with $mode(r_i)$ as the desired mode. Therefore the result of executing stylesheet x on XML document d is $PROCESS(x, root, 0)$, where $root$ is the root node of d.

In XSLT, there are special template rules called *built-in* template rules [16] which cause the input document to be copied to the output. In all examples (including Figure 4), we list all template rules that will be processed and the reader should assume that built-in rules have been overridden.

2.2.2 XSLT$_{basic}$

We first develop an algorithm to process a subset of XSLT, that we call XSLT$_{basic}$, which has the following restrictions on XSLT: (1) no type-coercion, (2) no document order, (3) no recursion, (4) no predicates, (5) no flow-control elements, (6) no conflict resolution for template rules, *i.e.*, no conflicting rules, (7) no functions and aggregations, and (8) no variables and parameters, (9) no use of the descendent (//) axis, (10) $select$ attribute of `<value-of>` or `<copy-of>` element can only be "`.`" or an attribute "@$attribute$".

Refer to [16] for details of (1)-(9). Restriction (10) is related to the simple model of output formatting used in this paper in which

```
Function $PROCESS(x, d_{con}, mode)$
Input: XSLT stylesheet as $x$, Context document node as $d_{con}$,
    Desired mode of the matched rule as $mode$
Output: XML document fragment as $result$
 1: $result \leftarrow empty$
 2: for $r_i \in x$ in decreasing order of $priority(r_i)$ do
 3:     if $mode(r_i) = mode$ and $MATCH(d_{con}, r_i)$ then
 4:         $result \leftarrow output(r_i)$
 5:         for $a_j \in apply(r_i)$ do
 6:             $D_{new\_con} \leftarrow SELECT(d_{con}, a_j)$
 7:             $sub\_result \leftarrow empty$
 8:             for $d_{new\_con} \in D_{new\_con}$ do
 9:                 $sub\_result \leftarrow concatenate(sub\_result,$
                                  $PROCESS(x, d_{new\_con}, mode(a_j)))$
10:             Replace $a_j$ in $result$ with $sub\_result$
11:         return $result$
```

Figure 5: Algorithm for XSLT processing.

values produced from the database always appear as attributes of a node. We do not consider document order in this paper and consider it part of future work. In Section 5, we extend our algorithms to process supersets of XSLT$_{basic}$ that include (4), (5) and (6). We believe the resulting subset of XSLT will cover a reasonable variety of XSLT stylesheets applied to XML-publishing views, but plan to extend the fragment we cover in future work. Finally in Section 5, we propose by example an approach for handling recursion.

3. ALGORITHM OVERVIEW

Given a schema-tree query v and an XSLT$_{basic}$ stylesheet x, our algorithm generates a new schema-tree query v' (called a *stylesheet view*), ensuring that for any relational database instance I, the result of query v' on I is the same as the result of running x on the result of query v on I, *i.e.* $v'(I) = x(v(I))$. The complete algorithm is given in Figure 9. In this section, we introduce the key data structures and functions used in view composition. A detailed description of the algorithm appears in Section 4.

For schema-tree query v and XSLT stylesheet x, our algorithm begins by generating a *context transition graph* (CTG). This graph (a) combines the selecting and matching steps in XSLT processing, and (b) captures the context transitions that occur when evaluating x on a document produced by v. From the CTG, a *traverse view query* (TVQ) is generated, which by definition is a schema-tree query. The TVQ captures the traversal actions of x on v. The final step of our algorithm generates *output tag trees* that are combined with the TVQ to generate the output *stylesheet view*, v'.

In this section, we introduce the above structures and describe how these structures relate to XSLT processing. A detailed description of the steps to generate the CTG, TVQ, and stylesheet view are given in Section 4.

3.1 Context Transition Graph (CTG)

The context transition graph for an XSLT$_{basic}$ stylesheet x executed on a schema-tree query v, CTG(v, x), is a multigraph with a set of nodes M and a set of edges E. Each node $m \in M$, is annotated by a pair (n, r), where n is a node in the schema-tree query v and r is an XSLT$_{basic}$ template rule in x. The CTG for Figure 4 is shown in Figure 6, where we use ($id(n)$, $tag(n)$) to represent the node n. (Although $id(n)$ can identify a node n uniquely, the redundant $tag(n)$ eases presentation.) Intuitively, the existence of a node (n, r) in CTG means that one or more of the XML document nodes generated by node n *may* be matched by rule r. An edge, say $e = ((n', r'), (n, r), a_j)$, incoming to this node indicates that

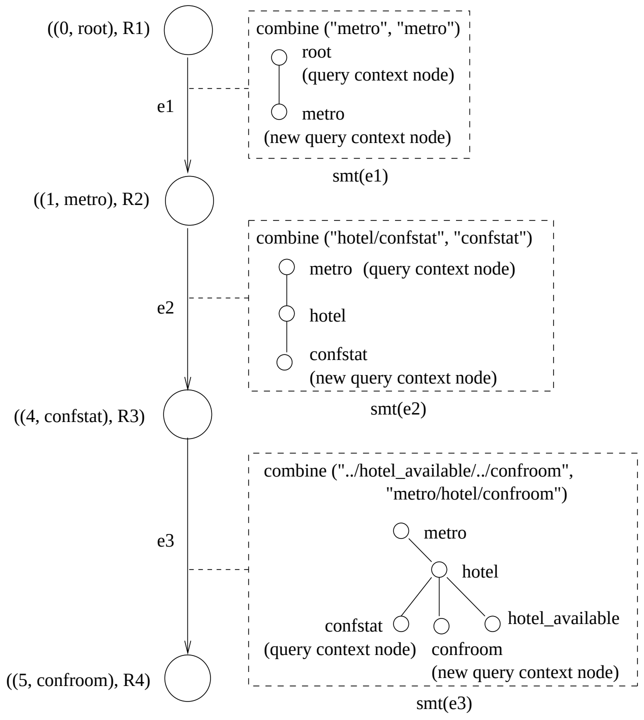


Figure 6: Context Transition Graph for Figure 4.

the firing of rule r' on a document node d generated by n', might lead to one or more document nodes generated by n appearing in $select(a_j)$ for some $a_j \in apply(r')$. (Since this may be true for multiple apply-template nodes, CTG(v, x) is a multigraph.) Associated with each edge is a tree-pattern query, $smt(e)$, referred to as the *select-match subtree* for e. Intuitively, $smt(e)$ combines the select-expression of a_j with the match-pattern of r, and is produced by the function $COMBINE$, described in Section 3.5.

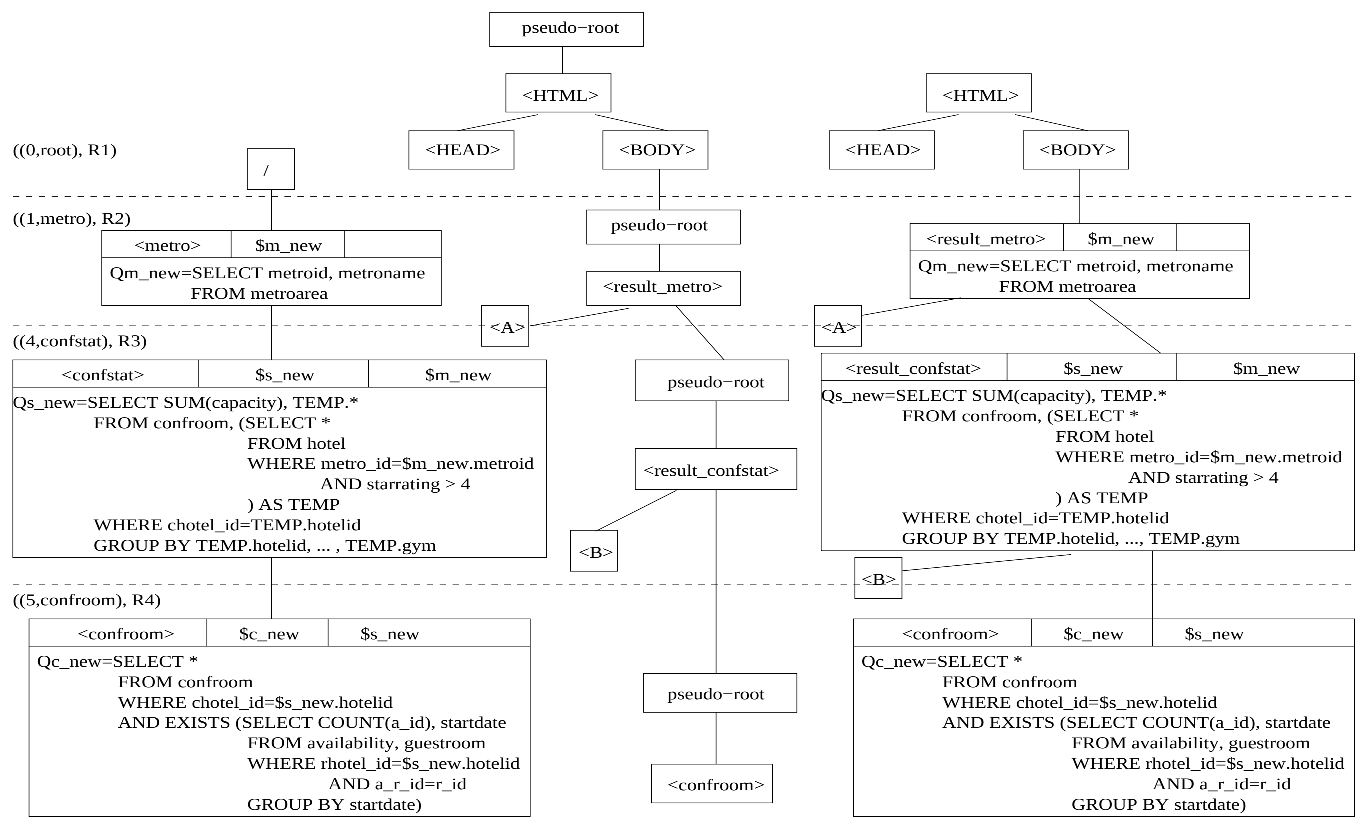
The left-hand side of Figure 6 shows the CTG produced while composing the stylesheet shown in Figure 4 with the schema-tree query of Figure 1 (the right-hand side of this figure is discussed in Section 3.5). Consider edge $e2$; this edge is present because applying the select expression *"hotel/confstat"* which appears in rule $R2$ to a document node produced by the *metro* node (id=1) in Figure 1 can potentially lead to nodes produced by the *confstat* node (id=4) being matched against rule $R3$. Note that, while Figure 6 is a simple path, CTGs for XSLT$_{basic}$ stylesheets are actually directed acyclic multigraphs, and can be general multigraphs if recursion is allowed.

3.2 Traverse View Query (TVQ)

The *traverse-view query* is a schema-tree query produced from the CTG. Intuitively, the nodes in the TVQ will generate those document nodes that may become context nodes during stylesheet evaluation. In this sense, it supports the *traversal* of the original XML document by the XSLT$_{basic}$ stylesheet. One or more nodes appear in the traverse-view query for each node in the context transition graph from which it is generated, with nodes being duplicated so that each node has a single incoming edge. The tree-pattern query of $smt(e)$, where $e = ((n_1, r_1), (n_2, r_2), a)$, in the context transition graph, is translated into a tag query in the query node associated with (n_2, r_2). The TVQ of Figure 4 is shown in Figure 7(a).

3.3 Output Tag Tree (OTT)

The generated TVQ traverses the document nodes, but does not



```
Qm_new=SELECT metroid, metroname
        FROM metroarea
```

```
Qs_new=SELECT SUM(capacity), TEMP.*
          FROM confroom, (SELECT *
                          FROM hotel
                          WHERE metro_id=$m_new.metroid
                                AND starrating > 4
                          ) AS TEMP
          WHERE chotel_id=TEMP.hotelid
          GROUP BY TEMP.hotelid, ... , TEMP.gym
```

```
Qc_new=SELECT *
          FROM confroom
          WHERE chotel_id=$s_new.hotelid
          AND EXISTS (SELECT COUNT(a_id), startdate
                      FROM availability, guestroom
                      WHERE rhotel_id=$s_new.hotelid
                            AND a_r_id=r_id
                      GROUP BY startdate)
```

(a) Traverse View Query (b) Output Tag Tree (c) Stylesheet View

Figure 7: (a) Traverse View Query, (b) Output Tag Trees, and (c) Stylesheet View for Figure 4.

generate the expected output. To produce output, for each node (n, r) in the traverse view query, we generate an *output tag tree* t corresponding to r. The output tag trees for Figure 4 are shown in Figure 7(b). We connect all the output tag trees to form a single output tag tree. The details about how the trees are generated and connected are discussed in Section 4.3.

3.4 Stylesheet View

Combining the traverse view query and output tag trees, we generate the stylesheet view v' for the XSLT stylesheet, as is shown in Figure 7(c). The combination entails copying the tag query for each node (n, r) in the traverse view query into the root of the output tag tree for (n, r). This is discussed in Section 4.4.

3.5 Functions Used in Composition

$SELECTQ$ and $MATCHQ$ are analogous to $SELECT$ and $MATCH$ presented in Section 2.2.1, but they operate on schema-tree nodes rather than data nodes. Recall that both $SELECT$ and $MATCH$ involve applying patterns to context nodes, either to derive a set of *new* context nodes or to determine whether the context nodes are matched. Correspondingly, when we apply a pattern abstractly on a schema-tree query, there is a *query context node*, and a *new query context node*. For example, we might apply the pattern "hotel/confstat" abstractly to the query context node $(1, \mathrm{metro})$ in Figure 1 and determine that node $(4, \mathrm{confstat})$ is a new query context node. We also introduce a third function, $COMBINE$, which produces the select-match subtrees associated with each edge in the CTG.

Given query node n and rule r, $MATCHQ(n, r)$ checks if the template path $match(r)$ matches some suffix of the path from the root to n in the schema-tree query. In XSLT, the $match(r)$ contains only child or descendant ("//") axis location steps. Since XSLT$_{basic}$ does not have the descendant axis, any match will correspond to a unique, simple path in the schema-tree query. If such a path exists, it is returned as a tree-pattern query, otherwise NULL is returned. As a result, if d_{con} is an instance of n, $MATCH(d_{con}, r)$ returns true if $MATCHQ(n, r)$ returns a non-NULL value. For example, in Figure 6, there is a node $((5, \mathrm{confroom}), R4)$, and $match(R4)$=*"metro/hotel/confroom"*. Figure 8 shows the corresponding tree-pattern query, which has three nodes.

Given query nodes n_1 and n_2, rule r, and `<apply-templates>` element $a \in apply(r)$, $SELECTQ(n_1, a, n_2)$ returns a tree-pattern query in which n_1 is the query context node and n_2 is the new query context node, or NULL. The tree-pattern query (if one is returned) is derived from $select(a)$ by using n_1 as the context node and associating n_2 with the final selection-step in $select(a)$. With selection step axes limited to `<child>` and `<parent>`, the resulting tree-pattern will be unique. For example, if a is the *apply-templates* element in $R3$, then $select(a)$= *"../hotel_available/../confroom"* and the corresponding tree-pattern query is the one shown in the top left Figure 8.

Given the tree-pattern query t returned from $SELECTQ$ and the tree-pattern query p returned from $MATCHQ$, $COMBINE$ creates a combined tree-pattern query as shown in Figure 8. To accomplish this, the two patterns are combined into a new pattern which is turned into a tree by a simple unification process, as follows. First, the node marked 'new query context node' in t and the node marked 'query context node' in p are unified. (Unification succeeds if the two nodes have the same ids.) If parents of the nodes just-unified exist in both queries, they are unified, and this

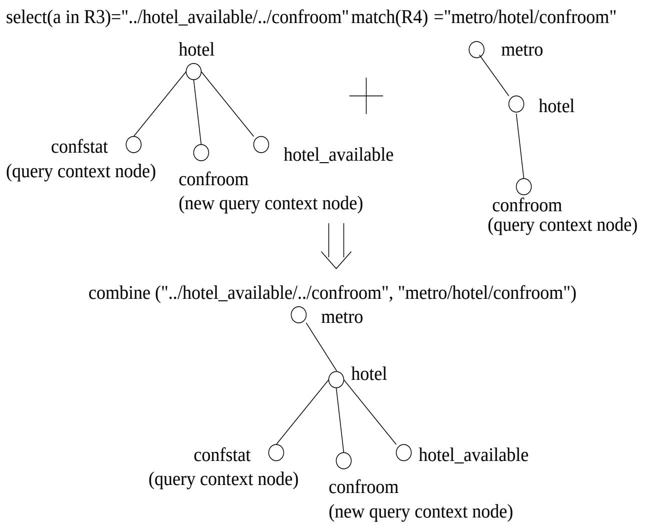


Figure 8: Combining tree-pattern queries for $select(a)$ and $match(r)$

process is repeated as many times as possible. Note that since both t and p are obtained from the schema-tree query, the result of this process will be a tree. Obviously, the function will fail if the initial unification fails, but as $COMBINE$ is used in this paper, they are guaranteed to be the same schema-tree node.

4. DETAILED ALGORITHM

The detailed stylesheet-composition algorithm is shown in Figure 9 and described below in four steps corresponding to the four data structures introduced in Section 3.

4.1 Step 1: Generating the CTG

Given a view query v and a stylesheet x, lines 3 to 14 in Figure 9 create $ctg = CTG(v, x)$. An edge $e = ((n_1, r_1), (n_2, r_2), a)$ will appear in ctg if and only if the following conditions are satisfied: (1) $MATCHQ(n_1, r_1) \neq$ NULL, (2) $MATCHQ(n_2, r_2) \neq$ NULL, (3) $SELECTQ(n_1, a, n_2) \neq$ NULL, and (4) $mode(a) = mode(r_2)$.

Edge e is labeled with the select-match subtree generated by the $COMBINE$ function, which combines the tree-patterns for $select(a)$ and $match(r_2)$. The new query context node in the tree-pattern for $select(a)$ is the same as the query context node in the tree-pattern for $match(r_2)$.

4.2 Step 2: Generating the TVQ

The generation of the traverse view query in Figure 7 takes place in lines 16–22 of Figure 9 and proceeds by copying the CTG (line 16), turning the resulting structure into a tree (line 17), and substituting new binding variables (lines 18). Next the tree-pattern query for each edge is translated to a parameterized SQL query as described below. The translated SQL query becomes the tag query associated with the target node of the edge.

4.2.1 Generating the Select-Match Subtree Query

This section explains how to translate a select-match subtree smt to an SQL query. This procedure is shown in the *UNBIND* function of Figure 10. Suppose the query context node and new query context node of smt are m and n respectively, and, n has a tag query $\mathcal{Q}_{bv(n)}(s_1, s_2, \ldots, s_k)$, parameterized by k binding variables, $s_1, s_2, ..., s_k$. We recursively replace the binding variables appearing in $\mathcal{Q}_{bv(n)}(s_1, s_2, \ldots, s_k)$ with tag queries of ancestor nodes of n until we reach n_j, the lowest common ancestor of m and n. The new query is $\mathcal{Q}^{s_1, s_2, \ldots, s_{j-1}}_{bv(n)}(s_j, \ldots, s'_k)$, where s_j is the binding variable for n_j and s_{j-1} is the binding variable for $child_n(n_j)$, the child node of n_j along the path from n_j to n. Further query transformations like those described in [8] can be applied to this query. Since the procedure is essentially to remove the binding variables in tag queries, it is named the *UNBIND* function.

As an example, the select-match subtree $smt(e2)$ in Figure 6 is translated into an SQL query as follows. It is generated by unbinding the tag query $\mathcal{Q}_s(h)$ of node `<confstat>` (with id 4), which is the new query context node in smt. All occurrences of binding variable $\$h$ in $\mathcal{Q}_s(h)$ are replaced with references to a sub-query, the tag query $\mathcal{Q}_h(m)$. Since `<metro>` is the lowest common ancestor of query context node `<metro>` and new query context node `<confstat>`, the unbinding stops. The resulting unbound query is shown below:

```
Procedure Compose(v,x)
Input: v: original schema-tree view query; x: XSLT stylesheet
Output: stylesheet view
1:  $ctg$: a Context Transition Graph
2:  $tvq$: a Traverse View Query
3:  $ottree$: an Output Template Tree
4:  for $n \in v$ do
5:      for $r \in x$ do
6:          if $MATCHQ(n, r) \neq$ NULL then
7:              add $(n, r)$ to $ctg$
8:  for $(n_1, r_1) \in ctg$ do
9:      for $(n_2, r_2) \in ctg$ do
10:         for $a \in apply(r_1)$ do
11:             $t \leftarrow SELECTQ(n_1, a, n_2)$, $p \leftarrow MATCHQ(n_2, r_2)$
12:             if $t \neq$ NULL and $mode(a) = mode(r_2)$ then
13:                 add an edge $e = ((n_1, r_1), (n_2, r_2), a)$ to $ctg$
14:                 $smt(e) \leftarrow COMBINE(t, p)$
15: (repeatedly) Delete all nodes without incoming edge, except $(root, r)$
16: $tvq \leftarrow ctg$ {(copy)}, $bvmap$(root of $tvq$) $\leftarrow$ empty
17: (repeatedly) Duplicate nodes with multiple incoming edges, splitting
    incoming edges and copying outgoing edges
18: replace binding variables in $tvq$ with new, unique binding variables
19: for $e = (w_1 = (n_1, r_1), w_2 = (n_2, r_2), a)$ in edges of $ctg$ do
20:     $(\mathcal{Q}_{bv(w_2)}, bvmap(w_2)) \leftarrow$
        $UNBIND(smt(e), n_1, n_2, bv(w_2), bvmap(w_1))$
21:     for all binding variables $bv$ referenced in $\mathcal{Q}_{bv(w_2)}$ do
22:         rename $bv$ as $bvmap(w_2).get(bv)$
23: for $w = (n, r) \in tvq$ do
24:     $ott(w)$ = $GENERATE\_OTT(n, r)$
25: $ottree \leftarrow ott(w) \forall w \in tvq$ {initially a forest}
26: for $e = (w_1, w_2, a)$ in edges of $tvq$ do
27:     $a' \leftarrow$ the "apply-template" node in $ott(w_1)$ which is a copy of $a$
28:     replace $a'$ with an edge from $parent(a')$ to $root(ott(w_2)))$
29: for $w = (n, r) \in tvq$ do
30:     $bv(root(ott(w))) \leftarrow bv(w)$
31:     $\mathcal{Q}_{bv(root(ott(w)))} \leftarrow \mathcal{Q}_{bv(w)}$
32: Remove the topmost "pseudo-root" node in $ottree$
33: while there is any "pseudo-root" node $pr$ left do
34:     for each child node $c$ of $pr$ do
35:         add edge $e = (parent(pr), c)$
36:         if $\mathcal{Q}_{bv(c)}$ is empty then
37:             $bv(c) \leftarrow bv(pr)$, $\mathcal{Q}_{bv(c)} \leftarrow \mathcal{Q}_{bv(pr)}$
38:         else
39:             $\mathcal{Q}_{bv(c)} \leftarrow UNBIND(parent(pr), c)$
40:             add the $SELECT$ columns of $\mathcal{Q}_{bv(pr)}$ to $\mathcal{Q}_{bv(c)}$
41:             change "$bv(pr)$" as "$bv(c)$" in the tag queries of $c$'s descen-
                dents
42:     remove edge $e = (parent(pr), pr)$ and node $pr$
43: return $ottree$ {new stylesheet view}
```

Figure 9: Query composition algorithm.

```
Function $UNBIND(m, n)$
Input: Query context node as $m$, Target node to unbind as $n$
Output: Unbound query for $n$ as $q$
1: $n_j \leftarrow$ the lowest common ancestor of $m$ and $n$
2: $s_j \leftarrow bv(n_j)$
3: $child_n(n_j) \leftarrow$ child node of $n_j$ along the path from $n_j$ to $n$
4: $s_{j-1} \leftarrow bv(child_n(n_j))$
5: $q \leftarrow \mathcal{Q}_{bv(n)}^{s_1,s_2,..,s_{j-1}}(s_j, ..., s'_k)$, which is the unbound tag query
   of $n$, $\mathcal{Q}_{bv(n)}(s_1, s_2, ..., s_k)$
6: return $q$
```

Figure 10: Naive UNBIND function for node in select-match subtree.

```
Function $NEST(p, p')$
Input: Node as $p$, Child node as $p'$
Output: Tag query for $p$ after nesting as $q$
1: $q \leftarrow \mathcal{Q}_{bv(p)}$
2: for each child node $c$ of $p$, except $p'$ do
3:     $q_c \leftarrow NEST(c, NULL)$
4:     add $EXISTS\ q_c$ in $WHERE$ clause of $q$
5: return $q$
```

Figure 11: Generating nested sub-query for subtree under node in select-match subtree.

```
SELECT SUM(capacity), TEMP.*
FROM confroom, (SELECT * FROM hotel
                WHERE metro_id=$m.metroid
                AND starrating > 4) AS TEMP
WHERE chotel_id=TEMP.hotelid
GROUP BY TEMP.hotelid,...,TEMP.pool,TEMP.gym
```

This query becomes the tag query of $((4, \text{confstat}), R3)$ as shown in Figure 7(a), after the following transformation: the binding variable $\$m$ in the above query is renamed as $\$m\_new$, which is the binding variable of $((1, metro), R2)$. Note that a GROUP BY clause on all columns of **TEMP** are added into the query by unbinding to preserve the semantics of the aggregation SUM(capacity) in $\mathcal{Q}_s(h)$.

Another example of this process is the query for $smt(e3)$ in Figure 6. The result of unbinding as described so far is shown here:

```
SELECT * FROM confroom
WHERE chotel_id=$h.hotelid
```

However, this query is incorrect due to three subtle issues. First, the select-expression for $smt(e3)$ is "*../hotel_available/../confroom*", so there must exist at least one <hotel_available> node, but the unbound query above does not check for one. Second, such a <hotel_available> node is not arbitrary, but must be a sibling node of the <confstat> node with the same parent. In case of a missing <hotel_available> sibling, the XSLT stylesheet would not process the <confroom>. Third, in a more complex pattern <hotel_available> may itself be the root of a subtree, requiring that the process be carried out recursively. To handle these three issues (*existence*, *sibling* and *recursion* conditions), the procedure of generating the query for select-match subtree smt is modified as shown in Figure 13. We introduce a new function $NEST$ (Figure 11) that is invoked from the procedure. Note that care must be taken in $NEST$ to rename tables during processing to avoid namespace collision, but these details are not shown. Line 5 of Figure 10 is also modified, as shown in Figure 12, to satisfy the three conditions.

Suppose for smt, its query context node is m and its new query context node is n with the tag query $\mathcal{Q}_n(s_1, \ldots, s_k)$. First the

```
1: {Replace line 5 of Figure 10}
2: $P \leftarrow$ {nodes along the path from $child_n(n_j)$ to $n$}
3: for node $p \in P$ do
4:     $p' \leftarrow$ child of $p \in P$, otherwise $NULL$
5:     $\Theta_{bv(p)} \leftarrow NEST(p, p')$
6: $q \leftarrow \Theta_{bv(n)}^{s_1,s_2,..,s_{j-1}}(s_j, ..., s'_k)$, which is the unbound and
   nested tag query of $n$, $\Theta_{bv(n)}(s_1, s_2, ..., s_k)$, and decorrelation
   of $bv(s_i)$ is done by $\Theta_{s_i}$.
```

Figure 12: Changes to UNBIND functions in Figure 10

.

```
Function $UNBIND(smt, m, n, bv', bvmap)$
Input: Select-match subtree as $smt$, Query context node of $smt$
   as $m$, New query context node of $smt$ as $n$, New binding
   bariable as $bv'$, Binding variable map as $bvmap$.
Output: Unbound query for $smt$ as $q$, New binding variable map
   as $bvmap'$.
1: $q \leftarrow UNBIND(m, n)$
2: $n_j \leftarrow$ the lowest common ancestor of $m$ and $n$
3: $child_n(n_j) \leftarrow$ child node of $n_j$ along the path from $n_j$ to $n$
4: $R \leftarrow$ {nodes along the path from $child_n(n_j)$ to $n$}
5: for node $p \in R$ do
6:     add the $SELECT$ columns of $\mathcal{Q}_{bv(p)}$ to $q$
7: $P \leftarrow$ {nodes along the path from root of $smt$ to $m$}
8: for node $p \in P$ do
9:     for each child node $c$ of $p$, such that $c \notin P$ and $c \notin R$ do
10:        $q_c \leftarrow NEST(c, NULL)$
11:        add $EXISTS\ q_c$ in $WHERE$ clause of $q$
12: $bvmap' \leftarrow bvmap$
13: for node $p \in R$ do
14:     $bvmap'.insert(bv(p), bv')$
15: $child_m(n_j) \leftarrow$ child node of $n_j$ along the path from $n_j$ to $m$
16: $S \leftarrow$ {nodes along the path from $child_m(n_j)$ to $m$}
17: for node $s \in S$ do
18:     $bvmap'.remove(bv(s))$
19: return $(q, bvmap')$
```

Figure 13: UNBIND function for select-match subtree.

unbound query $\mathcal{Q}_n^{s_1,\ldots,s_{j-1}}(s_j, \ldots, s'_k)$ is generated (line 1 in Figure 13), using the function shown in Figure 12. The unbound query involves every node along the path from $child_n(n_j)$ to n, which we denote as $nodeset(child_n(n_j) \rightarrow n)$. The existence of every node along the path from n_j to m, denoted as $nodeset(n_j \rightarrow m)$, is ensured because of the existence of m itself. Therefore, we need to ensure that for every node in the set $(nodeset(smt) \setminus (nodeset(n_j \rightarrow m) \cup nodeset(child_n(n_j) \rightarrow n)))$, there should exist at least one matching document instance node. For a node $p \in nodeset(child_n(n_j) \rightarrow n)$, the existence of child nodes of p is checked by the $EXISTS$ clause inserted at line 4 of $NEST$. Nesting is performed recursively for the nodes in the subtree under p. Similarly, for the child nodes (and the subtrees rooted at them) of nodes in $nodeset(n_j \rightarrow m)$, $EXISTS$ clauses are inserted at lines 10–11 of Figure 13. For example, the resulting query for tag query $\mathcal{Q}_c(h)$ should be as shown below:

```
SELECT * FROM confroom
 WHERE chotel_id=$h.hotelid
   AND EXISTS (
       SELECT COUNT(a_id), startdate
         FROM availability, guestroom
        WHERE rhotel_id=$h.hotelid
          AND a_r_id=r_id
        GROUP BY startdate)
```

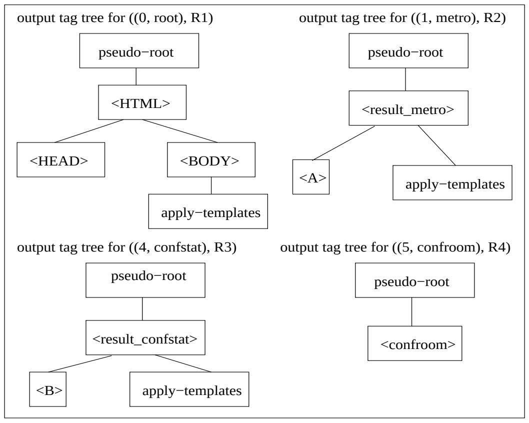


Figure 14: Output Tag Trees for nodes in TVQ of Figure 7(a).

This query becomes the tag query of $((5, \mathrm{confroom}), R4)$ as shown in Figure 7(a), after renaming the binding variables.

A *binding variable map*, $bvmap((n, r))$, is associated with each node in the TVQ. In Figure 13, the entries in $bvmap((m, r'))$ are copied to $bvmap((n, r))$, except those entries for schema-tree query nodes along the path $child_m(n_j) \rightarrow m$. Morever, entries are inserted for schema-tree query nodes along the path $child_n(n_j) \rightarrow n$, such that the binding variables of these nodes in the original schema-tree query are mapped to $bv((n, r))$. In the tag query for each node (n, r) in the TVQ, the referenced binding variables are renamed according to $bvmap((n, r))$.

4.2.2 Multiple Incoming Edges

It is possible that there are multiple incoming edges to a node labeled (n, r) in the CTG. Since query generation is defined for the select-match subtree associated with a single incoming edge, we need to duplicate (n, r) when we generate the TVQ (line 17). Since the children of this node now need to be duplicated (and their children, to the leaf nodes of the CTG), the size of the TVQ may be up to exponentially larger than the CTG.

4.3 Step 3: Generating Output Tag Tree

For each node (n, r) in TVQ, an output tag tree $ott((n, r))$ is generated by *GENERATE_OTT(n,r)* (lines 23–24), and then these trees are connected to form a single output tag tree $OTT(v, x)$ (lines 26–28). The output tag trees for nodes in the TVQ of Figure 7(a) are illustrated in Figure 14, and the final output tag tree is shown in Figure 7(b). Brief descriptions of the two steps are given below.

4.3.1 GENERATE_OTT(n,r) Function

We show the intuition of *GENERATE_OTT(n,r)* using the example in Figure 14. For a node (n, r), $ott((n, r))$ is the tree representation of the hypertext fragment inside the template rule r. We add a root *pseudo-root* to the fragment. The element `<xsl:value-of select=".">` in r, is represented as a node with the tag n, for example the `<confroom>` node in $ott((5, \mathrm{confroom}), R4))$. Each `<xsl:apply-templates>` element in r is represented as an *apply-templates* node in $ott((n, r))$. Not shown are `<xsl:value-of select="@`*attribute*`">` elements which would cause data from the database to be attached to a node like `<result-confstat>`.

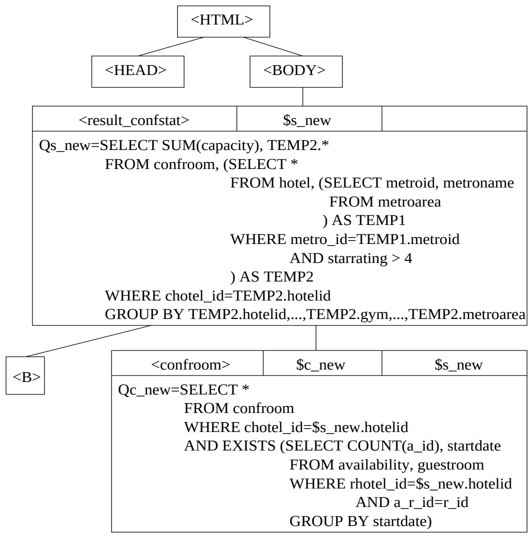
4.3.2 Connecting the Output Tag Trees

The output tag trees of the nodes in the TVQ are connected to form $OTT(v, x)$. For two nodes (n_1, r_1) and (n_2, r_2), if (n_1, r_1)

```
(R1), (R3), and (R4) are the same as Figure 4.

<xsl:template match="metro">                    (R2)
  <xsl:apply-templates select="hotel/confstat"/>
</xsl:template>
```

Figure 15: An example of forced unbinding.



```
Qs_new=SELECT SUM(capacity), TEMP2.*
         FROM confroom, (SELECT *
                           FROM hotel, (SELECT metroid, metroname
                                          FROM metroarea
                                        ) AS TEMP1
                           WHERE metro_id=TEMP1.metroid
                                 AND starrating > 4
                         ) AS TEMP2
         WHERE chotel_id=TEMP2.hotelid
         GROUP BY TEMP2.hotelid,...,TEMP2.gym,...,TEMP2.metroarea
```

```
Qc_new=SELECT *
         FROM confroom
         WHERE chotel_id=$s_new.hotelid
         AND EXISTS (SELECT COUNT(a_id), startdate
                       FROM availability, guestroom
                       WHERE rhotel_id=$s_new.hotelid
                             AND a_r_id=r_id
                       GROUP BY startdate)
```

Figure 16: Stylesheet view for Figure 15.

is the parent of (n_2, r_2), we add an edge from the parent of the *apply-templates* node in $ott((n_1, r_1))$ to the *pseudo-root* node in $ott((n_2, r_2))$, and remove the *apply-templates* node in $ott((n_1, r_1))$.

4.4 Step 4: Generating Stylesheet View

The stylesheet view for Figure 4 is shown in Figure 7(c). The stylesheet view is generated by first copying queries from the TVQ to the OTT (lines 29–31), and then removing *pseudo-root* nodes, pushing down queries (lines 32–42).

We now describe the pushing down of queries in more detail. If the child of the *pseudo-root* is an element specified in the OTT, the query will be empty and unbinding is not required. If the child already has a query (due to an `<apply-templates>` tag appearing at the top level of a rule body) then the child's tag query needs to be unbound with the tag query of the *pseudo-root*, a process we call *forced unbinding*. Note that (1) this significantly limits output formatting, but we believe better formatting can be added without affecting the rest of the algorithm and (2) pushing the query down separately into the *apply-templates* nodes will cause the results for these apply-templates to be grouped rather than interleaved; leaning more towards the assumption that document order is not supported by view composition.

For example, in Figure 4, each template rule has some output that will become part of the result tree fragment. However, in Figure 15, $((1, \mathrm{metro}), R2)$ has no output, but one child $((4, \mathrm{confstat}), R3)$. Therefore $ott((1, \mathrm{metro}), R2)$ has only two nodes *pseudo-root* and *apply-templates*, and only *pseudo-root* is kept after all output tag trees are connected into $OTT(v, x)$. The stylesheet view for Figure 15 is shown in Figure 16.

4.5 Complexity of the Algorithm

Below we first show that the time complexity of the algorithm is polynomial when there is at most one incoming edge for each node in context transition graph, and the worst-case time complexity is $O(v^v)$ when this assumption does not hold.

As shown in Figure 9, the process of generating nodes of CTG (lines 4–7) is bounded by $|v| \times |x|$, where $|v|$ is the number of nodes in schema-tree query v and $|x|$ is the number of rules in stylesheet x. Since there are no conflicting rules in XSLT$_{basic}$, the number of nodes in CTG is bounded by $|v|$ instead of $|v| \times |x|$. Therefore the process of generating edges of CTG (lines 8–14) is bounded by $|v|^2 \times max_a$, where max_a is the maximum number of apply-templates nodes in a rule in x. The number of edges is also bounded by $|v|^2 \times max_a$. Each edge is annotated with a select-match subtree, with different tag queries. The process of generating TVQ (lines 16–22) involves translating each of such select-match subtrees to a SQL query by unbinding. As shown in Figure 13, such unbinding is bound by the number of nodes in a select-match subtree, say max_b. The rest of the steps for generating output tag tree and stylesheet view are bounded by the number of nodes and edges of CTG (and also TVQ). Therefore the worst-case running time of the algorithm is bounded by $O(min(|v||x|, |v|^3 max_a max_b))$. In most cases, $|v|$ would be larger than $|x|$, therefore the running time is $O(|v|^3 max_a max_b)$.

However, the nodes with multiple incoming edges should be duplicated in the TVQ (line 17). For a node (n, r) in CTG, its number of incoming edges is at most $|v| \times max_a$. Such a node needs to be duplicated once for every incoming edge and becomes a child of every parent in the TVQ. If some of its parents (n', r') also have multiple incoming edges, (n, r) needs to be duplicated again together with (n', r'). This process of duplication will go up until reaching the root of TVQ. Therefore (n, r) could at most be duplicated $(|v| \times max_a)^{|v|-1}$ times. After the duplications are completed, the TVQ is a tree. The total number of nodes is at most $|v| \times (|v| \times max_a)^{|v|-1}$. Of course, this complexity would only be realized in extreme cases and we expect in practice that rules will be interrelated on a small scale if at all.

5. SUPERSETS OF XSLT$_{basic}$

In this section, we show how to handle several of the features omitted from XSLT$_{basic}$ in Section 2.2.2.

5.1 XSLT$_{expression}$

XPATH expressions can appear in elements of an XSLT stylesheet including `<xsl:template>` and `<xsl:apply-templates>`. Until now, we have assumed that predicates do not appear in path expressions. But in reality, each step in a path expression can have a predicate, which may be a relational expression (for example testing the value of an attribute) or another path expression (indicating that the relative path must exist). Figure 17 shows a stylesheet with predicates. The select-match subtree for the edge from $((4, \text{confstat}), R3)$ to $((5, \text{confroom}), R4)$ in this figure is shown in Figure 18. Note that there are two `<confstat>` nodes in the select-match subtree.

The mechanics of pushing expressions from tree-pattern queries into WHERE clauses differs little from [6, 11], and we limit our discussion to the effect that predicates have on our algorithm, in particular the $COMBINE$ function in Section 3.5, the $UNBIND$ function in Figure 13, and the $NEST$ function. The new $COMBINE$ function is almost identical to the previous function, except that when two nodes n_1 and n_2 with predicates p_1 and p_2 are unified to form node u, u is given predicate $[p_1 \text{and} p_2]$. Because of the existence of predicates, in addition to the procedure described in Figure 13, the unbinding of queries must also check if predicates are satisfied for all nodes in the select-match subtree. The changes to the unbinding function and $NEST$ are shown in Figure 19.

```
(R1) and (R2) are the same as Figure 4.

<xsl:template match="confstat">                (R3)
  <result_confstat>
    <B/>
    <xsl:apply-templates select=".[@sum<200]/
      ../hotel_available/../confroom
      [../confstat[@sum>100]][@capacity>250]"/>
  </result_confstat>
</xsl:template>

<xsl:template match="metro[@metroname=
              "chicago"]/hotel/confroom">   (R4)
    <xsl:value-of select="."/>
</xsl:template>
```

Figure 17: An example of stylesheet with predicates.

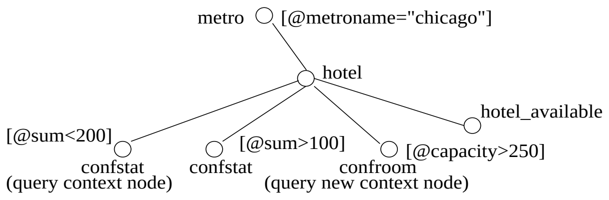


Figure 18: Tree-pattern query for smt on edge between $((4, confstat), R3)$ and $((5, confroom), R4)$ for Figure 17.

The unbound query for select-match subtree on edge between $((4, \text{confstat}), R3)$ and $((5, \text{confroom}), R4)$ of Figure 17, after renaming of binding variables, is shown in Figure 20. In this example, two predicates for `<confstat>` and `<metro>` appear, and two additional EXISTS checks appear, with the predicates inside the sub-queries.

5.2 XSLT$_{transformable}$

In this section, we show that some supersets of XSLT$_{basic}$ can be transformed to features in XSLT$_{basic}$, therefore can be processed by our algorithm.

5.2.1 Flow-Control Elements

An `<xsl:if>` element appearing in a rule r can be handled by introducing a new template rule with a previously unused mode $mnew$. The contents of the `<xsl:if>` are used as the body of the new template rule. An `<apply-templates>` statement replaces the `<xsl:if>` in r, uses the test of the `<xsl:if>` as its select and specifies $mnew$ as the $mode$. This is illustrated in Figure 21. In this figure, *nodename* is the name in the last location step in the pattern. `<xsl:choose>` can be similarly handled by viewing it as a group of template rules as shown in Figure 22. While ommitted due to space limitation, the transformation for `<xsl:for-each>` is very similar to that for `<xsl:if>`.

5.2.2 XSL:Value-of Elements

In XSLT$_{basic}$, the $select$ attribute of `<value-of>` must be ".". But, in general, it can be an XPATH expression. Similarly to how we handle flow-control elements, `<xsl:value-of>` can be handled by viewing it as a new template rule matched by implicit `<apply-templates>` in the rule in which it resides. The transformation is shown in Figure 23.

5.2.3 Conflict Resolution for Template Rules

Each template rule in a XSLT stylesheet has its priority, which varies by position in the stylesheet, and also by the order of in-

$\{$**Insert the following**
(a) after line 8 of Figure 13
(b) after line 1 of Figure 11$\}$
$e_p \leftarrow$ predicate using $tag(p)$
add e_p in $WHERE$ clause of q

Figure 19: Changes to UNBIND functions for predicates.

```
SELECT *  FROM confroom
WHERE chotel_id=$s_new.hotelid
      AND capacity > 250
      AND $s_new.SUM_capacity<200
      AND $s_new.metroname=''chicago''
      AND EXISTS (SELECT SUM(capacity)
                  FROM confroom,
                  WHERE chotel_id=$s_new.hotelid
                  HAVING SUM(capacity)>100)
      AND EXISTS (SELECT COUNT(a_id), startdate
                  FROM availability, guestroom
                  WHERE rhotel_id=$s_new.hotelid
                        AND a_r_id=r_id
                  GROUP BY startdate)
```

Figure 20: The unbound query for Figure 18.

cluded and imported stylesheets. When multiple rules match a node, XSLT needs a conflict-resolution scheme to select the one rule with the highest priority to apply. The conflict-resolution scheme itself brings no difficulty because the priorities of template rules are statically determined at view-composition time [16]. The conflict-resolution facility in [9] could be used for this purpose. The problem arises from the fact that it is impossible to determine at view-composition time whether a template rule will match a node. This is because the match pattern in a template has not only name tests but also predicates, the values of which are determinable only during XSLT evaluation at runtime.

The following approach can be used to compose conflict resolution into SQL queries in XML view. Suppose there is a set of template rules that have the same node name in their last location step (which means they potentially could be conflicting rules). First arrange them as a list of rules in the order of priority. Then in each rule add `<xsl:when>` elements to test whether the node can be matched by some higher priority rule. The transformation is shown in Figure 24. Note that the original *n* rules have the same mode `"m"`, otherwise they cannot conflict with each other. As seen in the figure, each new `<xsl:when>` element checks for an "`expression i`" which is the reverse of the "`pattern i`" in the i-th template rule. For example, if the pattern i after extension is *name1[p1]/name2[p2]/.../namen[pn]*, then expression i is *.[pn]/.../parent::name2[p2]/parent::name1[p1]*.

5.3 XSLT$_{recursion}$

Recursion between rules (or with a single rule) can easily arise if the *parent* or *ancestor* axes are allowed. In this section, we illustrate an approach to dealing with this recursion. In particular, our goal is to speed up the overall processing time by pushing computation from the stylesheet to the query processor, while leaving the recursion to be handled by the XSLT processor with a modified stylesheet.

For example, Figure 25 is a stylesheet x with recursive rules (R1 and R2). Composing it with the schema-tree query v in Figure 1 results in stylesheet view v' in Figure 26 and new stylesheet x' in Figure 27. By inspection, we see that x' on v' obtains the same result as running x on v. However, the `<result_metroavail>` nodes are generated without materializing many of the intermediate nodes such as `<hotel>`, `<confstat>` and `<confroom>`. Our algorithm for this type of transformation is currently limited to only a few cases and is not included here; however, we have included the example as we feel the approach is worthy of note and is potentially applicable to XQUERY with functions as well as XSLT.

```
<xsl:template match="pattern" mode="m">
  <xsl:if test="expression">
    template body
  </xsl:if>
</xsl:template>
```

(a)

```
<xsl:template match="pattern" mode="m">
  <xsl:apply-templates se-
lect=".[expression]" mode="mnew"/>
</xsl:template>

<xsl:template match="nodename" mode="mnew">
  template body
</xsl:template>
```

(b)

Figure 21: Transformation for `<xsl:if>`.

```
<xsl:template match="pattern" mode="m">
  <xsl:choose>
    <xsl:when test="e1">b1</xsl:when>
    <xsl:when test="e2">b2</xsl:when>
    <xsl:otherwise>b3</xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

(a)

```
<xsl:template match="pattern" mode="m">
  <xsl:apply-templates
    select=".[e1]" mode="mnew1"/>
  <xsl:apply-templates
    select=".[not(e1) and e2]"
    mode="mnew2"/>
  <xsl:apply-templates
    select=".[not(e1) and not(e2)]"
    mode="mnew3"/>
</xsl:template>

<xsl:template match="nodename"
    mode="mnew1">b1</xsl:template>
<xsl:template match="nodename"
    mode="mnew2">b2</xsl:template>
<xsl:template match="nodename"
    mode="mnew3">b3</xsl:template>
```

(b)

Figure 22: Transformation for `<xsl:choose>`.

6. RELATED WORK

XML publishing middleware SILKROUTE [6, 5] and XPERANTO [4, 12, 11] allow users to query XML views using XML-QL and XQUERY respectively. View-composition algorithms compose user queries with XML views, the result of which can be evaluated using relational database engines. However, the queries contemplated in those works differ substantially from XSLT stylesheets, as observed by [7].

In [9] Moerkotte studied the problem of incorporating XSLT processing into database query engines. XSLT stylesheets are translated into algebraic expressions, for which physical algebraic operators in database engines are implemented using an iterator. Distinctions from our work were pointed out in the introduction. Most importantly, our approach allows for materialization of fewer nodes, and thus, greater efficiency.

```
<xsl:template match="pattern" mode="m">
  <xsl:value-of select="expression"/>
</xsl:template>
```

(a)

```
<xsl:template match="pattern" mode="m">
  <xsl:apply-templates
     select="path expression" mode="mnew"/>
</xsl:template>

<xsl:template match="nodename[predicate]"
     mode="mnew">
  <xsl:value-of select="."/>
</xsl:template>
```

(b)

Figure 23: Transformation for `<xsl:value-of>`.

```
<xsl:template match="pattern 1" mode="m">
  template body 1
</xsl:template>

<xsl:template match="pattern 2" mode="m">
  template body 2
</xsl:template>
```

(a)

```
<xsl:template match="pattern 1" mode="m1">
  template body 1
</xsl:template>

<xsl:template match="pattern 2" mode="m">
  <xsl:choose>
    <xsl:when test="expression 1">
      <xsl:apply-templates select="."
             mode="m1"/>
    </xsl:when>
    <xsl:otherwise>
      template body 2
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

(b)

Figure 24: Transformation for conflict resolution.

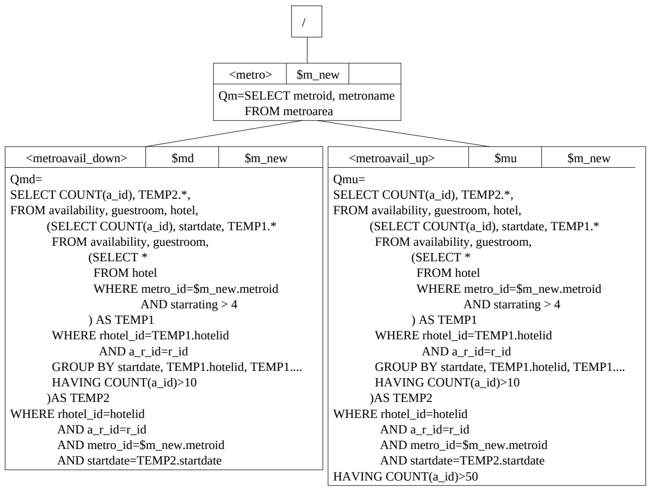
The work of Jain, Mahajan and Suciu [7] is the closest to ours. Its approach generates a single SQL query for an entire XSLT program and tags the SQL result tuples to generate XML output. The technique of [7] begins by separating the XSLT stylesheet into distinct paths. Each path is represented by a structure called *QTree*. One SQL query is generated for each path by composing its *QTree* with the XML view tree. The final SQL query for the XSLT stylesheet is a union of all such queries. Our approach improves upon theirs on a number of points. The first point is that we generate a tree-structured view query directly rather than an SQL query. Since recent work either (1) optimizes the translation from view-query to SQL-query [5] or (2) exploits knowledge of the tree structure during optimization [2], we expect this distinction will be more important over time.

Second, our construction of select-and-match subtrees avoids several issues that arise with the *QTree*: (1) it is not clear how data values for internal nodes are generated since outer joins are not done along QTree paths, and if outer joins were performed, duplicates would then need to be eliminated. [7] assumes that only the leaf node of a path contribute to the result. However, in reality all other nodes along the path could generate part of the result. Therefore for each path, one SQL query cannot be sufficient. Instead maximally n (the number of nodes in the path) SQL queries may be necessary. Therefore the final SQL query should be the union of

```
<xsl:template match="/metro">                (R1)
  <xsl:param name="idx" select="10"/>
  <result_metro>
    <xsl:apply-templates
     select="hotel/hotel_available[@count>10]
             /metro_available[@count<$idx]">
      <xsl:with-param name="idx"
               select="$idx"/>
    </xsl:apply-templates>
  </result_metro>
</xsl:template>

<xsl:template match="metro_available">      (R2)
  <xsl:param name="idx"/>
  <xsl:choose>
    <xsl:when test="$idx<=1">
      <xsl:value-of select="."/>
    </xsl:when>
    <xsl:otherwise>
      <result_metroavail>
        <xsl:apply-templates
             select="self::[@count>50]/../../..">
          <xsl:with-param name="idx"
                          select="$idx-1"/>
        </xsl:apply-templates>
      <result_metroavail>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Figure 25: An example of stylesheet with recursive rules.



```
Qm=SELECT metroid, metroname
      FROM metroarea
```

```
Qmd=
SELECT COUNT(a_id), TEMP2.*,
FROM availability, guestroom, hotel,
         (SELECT COUNT(a_id), startdate, TEMP1.*
          FROM availability, guestroom,
                  (SELECT *
                   FROM hotel
                   WHERE metro_id=$m_new.metroid
                            AND starrating > 4
                  ) AS TEMP1
          WHERE rhotel_id=TEMP1.hotelid
                   AND a_r_id=r_id
          GROUP BY startdate, TEMP1.hotelid, TEMP1....
          HAVING COUNT(a_id)>10
         )AS TEMP2
WHERE rhotel_id=hotelid
          AND a_r_id=r_id
          AND metro_id=$m_new.metroid
          AND startdate=TEMP2.startdate
```

```
Qmu=
SELECT COUNT(a_id), TEMP2.*,
FROM availability, guestroom, hotel,
         (SELECT COUNT(a_id), startdate, TEMP1.*
          FROM availability, guestroom,
                  (SELECT *
                   FROM hotel
                   WHERE metro_id=$m_new.metroid
                            AND starrating > 4
                  ) AS TEMP1
          WHERE rhotel_id=TEMP1.hotelid
                   AND a_r_id=r_id
          GROUP BY startdate, TEMP1.hotelid, TEMP1....
          HAVING COUNT(a_id)>10
         )AS TEMP2
WHERE rhotel_id=hotelid
          AND a_r_id=r_id
          AND metro_id=$m_new.metroid
          AND startdate=TEMP2.startdate
HAVING COUNT(a_id)>50
```

Figure 26: Stylesheet view v' for composing Figure 25 with Figure 1.

all SQL queries for each *QTree*, each of which is an outer union of SQL queries for each node on the path represented by the *QTree*. (2) *QTree* combines the select expressions of different rules as long as they are on the same path. Again because that not only the leaf node but also all nodes on a path could generate result fragment, the predicates in expressions of different rules should not be simply combined together. (3) *QTree* does not appear to handle the parent axis of XPath, for example, "../hotel_available/../confroom" cannot be handled by QTree because the existence of *hotel_available* node is not tested.

Finally, [7] does not cover many features of XSLT. The predicates handled in [7] are accesses of attributes or text subelements. It does not consider how to handle the case when they are also ex-

```
<xsl:template match="/metro">           (R1')
  <xsl:param name="idx" select="10"/>
  <result_metro>
    <xsl:apply-templates
         select="metroavail_down[@count<$idx]">
      <xsl:with-param name="idx"
         select="$idx-1"/>
    </xsl:apply-templates>
  </result_metro>
</xsl:template>

<xsl:template match="metroavail_down">   (R2')
  <xsl:param name="idx"/>
  <xsl:choose>
    <xsl:when test="$idx<=1">
      <xsl:value-of select="."/>
    </xsl:when>
    <xsl:otherwise>
      <result_metroavail>
        <xsl:apply-templates
             select="../metroavail_up">
          <xsl:with-param
             name="idx" select="$idx-1"/>
        </xsl:apply-templates>
      <result_metroavail>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<xsl:template match="metroavail_up">     (R3')
  <xsl:param name="idx"/>
  <xsl:choose>
    <xsl:when test="$idx<=1">
      <xsl:value-of select="."/>
    </xsl:when>
    <xsl:otherwise>
      <result_metroavail>
        <xsl:apply-templates
          select="../metroavail_down[@count<$idx]">
          <xsl:with-param
             name="idx" select="$idx-1"/>
        </xsl:apply-templates>
      <result_metroavail>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Figure 27: New stylesheet x' for composing Figure 25 with Figure 1.

pressions. It does not discuss how to handle features such as flow-control elements and conflicting rule resolution. It does not deal with recursion in template rules.

7. CONCLUSION AND FUTURE WORK

In this paper, we focus on the problem of evaluating XSL transformations on XML-publishing views. For a subset of XSLT, we present a view-composition algorithm to compose a transformation with an XML view. We then describe how to extend this algorithm to handle most features of XSLT, including certain cases of recursion. Evaluating the composed stylesheet view on a database instance results in the same XML document as evaluating the XSLT stylesheet on the original XML view. In this way, inefficient XSLT processing is replaced by queries pushed into relational database engines. In addition, the stylesheet view does not generate the unnecessary nodes. These features of our approach offer the promise of greatly improved efficiency.

Based on the view-composition algorithm proposed in this paper, we plan to focus our future work on (1) handling more features of XSLT that were excluded in this paper and (2) developing further the approach of handling recursion and other complicated features by *partially* pushing functionality from the stylesheet into the query. Finally, we expect that this last approach may be used to ensure that the composition will always run in polynomial time, at the cost of leaving more functionality to be processed by XSLT, and we plan to investigate this tradeoff.

8. REFERENCES

[1] Suggestion by anonymous reviewer.

[2] P. Bohannon, S. Ganguly, H. F. Korth, P. P. S. Narayan, and P. Shenoy. Optimizing view queries in rolex to support navigable result trees. In *Proc. 28th Int. Conf. Very Large Data Bases, VLDB*, pages 119–130, 2002.

[3] P. Bohannon, H. F. Korth, and P. Narayan. The table and the tree: On-line access to relational data through virtual XML documents. In *Proc. of WebDB*, 2001.

[4] M. J. Carey, D. Florescu, Z. G. Ives, Y. Lu, J. Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. XPERANTO: Publishing object-relational data as XML. In *WebDB (Informal Proceedings)*, pages 105–110, 2000.

[5] M. Fernandez, A. Morishima, and D. Suciu. Efficient evaluation of XML middle-ware queries. In *Proc. of the ACM SIGMOD Int'l conference on Management of Data*, pages 103–114, 2001.

[6] M. F. Fernandez, W.-C. Tan, and D. Suciu. SilkRoute: Trading between Relations and XML. In *Int'l World Wide Web Conf. (WWW)*, Amsterdam, Netherlands, May 2000.

[7] S. Jain, R. Mahajan, and D. Suciu. Translating XSLT programs to efficient SQL queries. In *Proc. of the Eleventh Int'l Conference on the World Wide Web*, pages 616–626, 2002.

[8] W. Kim. On optimizing an SQL-like nested query. *ACM Trans. on Database Systems*, 7(3):443–469, Sept. 1982.

[9] G. Moerkotte. Incorporating XSL processing into database engines. In *Proc. 28th Int. Conf. Very Large Data Bases, VLDB*, pages 107–118, 2002.

[10] The Saxon XSLT processor. http://www.blnz.com/xt/index.html.

[11] J. Shanmugasundaram, J. Kiernan, E. Shekita, C. Fan, and J. Funderburk. Querying XML views of relational data. In *Proceedings of the 27th International Conference on Very Large Data Bases(VLDB '01)*, pages 261–270, 2001.

[12] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as XML documents. *VLDB Journal: Very Large Data Bases*, 10(2–3):133–154, 2001.

[13] The-Apache-Software-Foundation. Xalan C++ XSLT stylesheet processor. http://xml.apache.org/xalan-c/index.html.

[14] W3C. W3C XML query. http://www.w3.org/XML/Query.

[15] W3C. XML path language (XPATH). http://www.w3.org/TR/xpath20/.

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

[17] P. Wadler. A formal semantics of patterns in XSLT. In *Proc. Markup Technologies*, Philadelphia, PA, USA, 1999.

[18] XT web site. http://www.blnz.com/xt/index.html.