Designing Maintainable XML Transformations

Siim Karus
University of Tartu, Estonia
siim.karus@ut.ee

Marlon Dumas
University of Tartu, Estonia
marlon.dumas@ut.ee

***Abstract* – Modern applications often rely on XML to represent data internally and to interact with other applications and with end users. XSL transformations are commonly employed to transform between the internal representations of XML documents manipulated by an application and representations used for interaction with end-users and with other applications. These XSL transformations need to be updated whenever the underlying XML formats evolve. To address this maintenance problem, we formulate a number of guidelines for designing XSL transformations that are resilient to changes in the schema of the input XML documents. These guidelines are evaluated experimentally on the basis of three case studies. The evaluation shows that the use of these guidelines leads to more concise XSL transformations and to significant reductions in the amount of changes required to adapt existing XSL transformations in response to changes in the input schema.**

Keywords – Software maintenance, forward compatibility, XML, XSL Transformations

I. Introduction

Extensible Stylesheet Language Transformations (XSLT) [1] is a popular language for transforming XML documents [2]. Trends such as Service-Oriented Architectures have heightened the role played by XML document transformations in Web information systems. XSLT is often the preferred choice for implementing these transformations due to its platform-independence and performance [3]. Even graphical editors used for defining XML transformations often generate XSLT code. In many cases, XSLT is used in a supportive role as a language for accomplishing specific tasks and amounts to a rather small percentage of the total code base of a project. However, considerable amounts of XSLT code can be found in projects that rely heavily on document transformation and data viewing. Reportedly, some applications in the financial domain contain libraries with tens of thousands lines of XSLT code.[1] This and similar examples show that the amount of XSL transformations is non-negligible and maintaining these transformations is an issue worth consideration.

The question addressed in this paper is how to design XML transformations in a way that minimizes the amount of effort required to maintain these transformations when changes occur in the source schema. To address this question, the paper formulates a set of guidelines and studies the hypothesis that XML transformations designed according to these guidelines offer higher maintainability than XML transformations that do not follow these guidelines. To test this hypothesis, we conducted three case studies with different characteristics, and we measured the impact of the guidelines on the number of Lines of Code (LOC) added or changed in order to adapt the XSL transformations in response to different types of changes (the *churned LOC* measure as defined in [4]). A secondary hypothesis tested in the study is that the guidelines have a positive impact on performance (execution time) of the transformations. We also assessed the impact of individual guidelines on various other metrics related to the size of the transformations, such as number of elements of different types.

To complement the empirical evaluation, the paper also presents a conceptual analysis of the guidelines with respect to an existing taxonomy of changes in XML schemas. This analysis provides insights into the types of changes for which the guidelines provide forward-compatibility.

The paper is structured as follows. The following section introduces the guidelines. Next, Section III presents the case studies, while Section IV discusses the evaluation results and Section V presents the conceptual analysis of the guidelines. Finally, Section VI discusses related work and Section VII concludes and outlines future work.

II. Guidelines

By drawing upon the principles of service-orientation [5] and forward-compatible software engineering [6], we propose the following four guidelines for designing forward-compatible XSL transformations:

1. Identical output document fragments with more than one node should be created once, and stored in a variable or parameter for reuse. Functionally or structurally similar output document fragments should be produced using parameterised templates instead of multiple templates or multiple similar inline code blocks.
2. If a transformation task needs to be performed whenever a fragment of the source document matches one among multiple conditions, this task should be encapsulated as a *generic transformation service*.
3. Complex transformation services should be composed from individually *addressable* and *subscribable* transformation services.
4. XSL transformations should exploit the metadata carried by the source document, specifically: the morphology of element/attribute names, the number of child entities of an entity and their structure.

In a nutshell, guidelines 1 and 3 seek to increase modularity, while guidelines 2 and 4 seek to increase abstraction, both of which are recognized attributes of forward-compatible software [6]. Below we present each

[1] http://www.nycircuits.com/xml/xsl-consulting.html

guideline in details and we analyse the impact of these guidelines according to various metrics.

A. *Common controls.*

According to this guideline, when designing an XSL transformation, the developer should identify recurrent output document fragments. Once these common fragments are identified, they should be checked for possible simplifications or unifications with other common fragments. If possible, these simplifications or unifications must be applied. Unifications can be based on functional similarity (e.g. tables used for HTML layout can be unified with divisions used for the same purpose) or structural similarity (e.g. unifying elements that share common child elements).

Next, the developer should identify input parameters required to generate instances of the identified recurrent output document fragment. If the contents of the recurrent document fragment is constant and comprises more than one node, a constant (global) variable containing the document fragment should be created. Otherwise, a parameterized XSL template should be designed to produce the recurrent document fragment. We use the term *common control* to refer to the common output elements, global variables, and templates resulting from the above refactoring process. When designing the XSL transformation, if one or multiple input elements need to be translated to one of the identified recurrent output element types, the corresponding common control should be reused. By using existing or common controls, duplicate code is avoided and this is likely to lead to increased maintainability.

The term *common control* (in reference to the Windows Common Control Library) [2] refers to presentation-level components used to perform common input and output tasks. For example, a pull-down menu is a common control that displays a list of items and allows the user to choose one of these items. HTML form elements for example define a number of common controls. In a more general definition applicable to XML-to-XML transformations, we define an XSLT template that produces as output a target document fragment, which is identified as "recurring" – like for example an instance of an "address" element type that may appear instantiated in different places in an XML document.

```
<!-- link that opens in new window -->
<xsl:template name="t_component_iconbutton ">
  <xsl:param name="href"/>
  <xsl:param name="text"/>
  <xsl:param name="icon"/>
  <xsl:param name="iconclass"/>
  <xsl:param name="title">View more</xsl:param>
  <a href="{$href}" title="{$title}"
onclick="javascript: window.open('{$href}',
'vvv_link'); return false;">
    <xsl:if test="$icon != ''">
      <img src="{$icon}" alt="{$title}"
class="{$iconclass}"/>
    </xsl:if>
    <xsl:if test="$text != ''">
      <xsl:value-of select="$text"/>
</xsl:if></a></xsl:template>
```

Listing 2. Complex common control – Template for displaying an "icon-button".

```
<!-- line break in HTML document -->
<xsl:variable name="newline">
  <br /><xsl:text>&#x000d;&#x000a;</xsl:text>
</xsl:variable>
```

Listing 1. Simple common control. – XSLT variable for HTML newline.

A common control can be simple or complex. A simple common control corresponds to one output XML element. Many (X)HTML elements, e.g. `img`, can be factored out as simple common controls since they occur frequently. Similarly, in the context of RosettaNet business documents[3] one frequently encounters `FreeFormText` elements used for various purposes. Thus, element `FreeFormText` is to be identified as a simple common control in transformations that output RosettaNet documents. An element that can only be used once in the target document is not a common control.

A special case of simple common controls are variables or parameters containing constant re-occurring document fragments with more than one node. The variable defined in Listing 1 is an example of a simple common control that we used in one of our case studies. Introducing such simple common controls can improve maintainability of the XSLT code, since it factors out re-occurring output fragments.

Complex common controls are templates that generate recurring combinations of common controls (simple or complex). These templates are usually called by name and take parameters to control the output. The template for creating an "icon-button" (an icon that acts like a button) shown in Listing 2, is an example of a complex common control. Complex controls can also be used to simplify the generation of XML elements that have similar inner structure. When having a choice between simple or complex common controls, one should prefer simple common controls.

Below, we analyze the impact of this guideline with respect to several factors:

- *Number of similar code sections*. Using common controls does not reduce the number of similar code sections. [4] Long similar code sections and code sections with similar output are replaced with similar common controls. So the number of similar code sections can increase when using common controls.
- *Length of similar code sections*. Long similar code sections are replaced with references to common controls, which are generally shorter than the common control itself. In other words, using common controls factors away lengthy sections of

[2] http://msdn.microsoft.com/en-us/library/bb775493(VS.85).aspx

[3] http://www.rosettanet.org

[4] Two code fragments A and B are similar if they each contain at least one XML node, and every XML node in A can be matched to a node in B that has the same structure and at most one difference in element or attribute name.

code and thus reduces the length of duplicate or similar code sections.

- *Number of elements*. Using common controls reduces the length of similar code sections, reducing the number of elements in the transformations.
- *Number of templates*. Complex common controls are templates and thus they increase the number of templates in a transformation. However, using a simple common control instead of a custom template reduces the number of templates. Therefore, this guideline may either increase or decrease the total number of templates depending on which of the factors has a stronger effect.
- *Number of parameters and variables*. Variables and parameters are used to encode simple common controls. Thus simple common controls can increase the number of variables and parameters. But complex common controls can reduce the number of local parameters/variables by eliminating duplicate declarations.
- *Number of template calls and XSLT copy-of elements*. References to common controls are normally encoded using XSLT copy-of and call-template elements and target document element declarations. Thus, the number of template calls and XSLT copy-of elements will increase.

B. *Generic transformation services.*

Generic transformation services are fragments of XSLT code (e.g. one or multiple related templates) used to perform common tasks like extracting data from elements, or performing aggregations on datasets. In the case of XML-to-HTML transformation, these tasks also include presenting formatted lists of items, notifying errors and other user-oriented tasks. Unlike common controls, generic services expose functionality used to accomplish a user task or a data manipulation task, as opposed to being dedicated to produce a recurrent type of output element (e.g. generating an icon-button or free-form element). The guideline aims at minimizing code duplication as similar code fragments are replaced with generic services.

```
<!-- Matches all elements ending with 's' -->
<xsl:template name="t_component_list"
match="*[substring(name(.), string-
length(name(.))) = 's']">
  <xsl:param name="container" select="."/>
  <!-- Container block -->
  <div class="box">
    <!-- Header -->
    <div class="top">
      <!-- Display header --></div>
    <!-- Display container items -->
    <div class="contents">
      <xsl:apply-templates
select="$container/*"/></div>
    <!-- Display navigation for next and previous
page if present -->
    <xsl:call-template
name="t_component_list_navi"/>
</div></xsl:template>
```

Listing 3. Template for displaying lists of items.

For example, having one transformation template (i.e. *generic service*) to display lists of items or components (see Listing 3) instead of many specialized transformations is more maintainable. Templates exposing generic services are typically invoked by matching source document sections to template's match expression.

Below we analyze the impact this guideline with respect to several factors:

- *Number of similar code sections*. Using one general template instead of many similar ones reduces the number of similar code sections by reducing the number of similar templates.
- *Length of similar code sections*. Generic services allow one template to be used for many similar (not necessarily identical) code sections. Therefore, similar code sections get replaced with calls to the template or an apply-templates directive, both of which are usually more compact than the original code sections.
- *Number of elements*. Using one general template instead of many similar ones reduces the number of elements by reducing the number of similar templates.
- *Number of match, select and test expressions that use implicit metadata*. Generic services use implicit metadata in order to factor out similar but non-identical code sections. Therefore, increased number of generic services results in increased number of match, select and test expressions that use implicit metadata.
- *Number of templates*. Using one generic template instead of many similar ones reduces the number of similar templates.
- *Number of variables and parameters*. Replacing similar templates with one generic template is likely to reduce the number of parameters and variables. Indeed, some parameters used by different similar templates will be factored out in the generic template. However, it might be necessary to introduce additional parameters to differentiate between different uses of the generic template. Thus generic services generally reduce the number of parameters and variables, but there can be exceptions
- *Number of XSLT apply-templates elements*. Abstractions are used by match, select and test expressions. Thus, increased number of abstractions results in increased number of XSLT apply-templates elements.

C. *Addressable and subscribable services.*

Complex services allow using one invocation to fulfil many tasks. Complex services built as monolithic services are difficult to upgrade or modify as requirements for the services change. It might happen that there will be need for services with minor differences in the details, which would

require almost full rewrite of a monolithic service, but can be solved by defining one service composed of smaller subservices, which can be called individually (i.e. they are *addressable*) and made available individually (i.e. they are *subscribable*).

Each template exposes a service. Therefore, what this guideline entails is that templates should not be too specific. In fact, one can use calls to templates or template matching in complex templates to achieve the required flexibility. The templates called or matched are services, which together form a complex service. Listing 4 shows how a complex template can be separated into two independently addressable templates.

An analysis of the potential impact of applying this guideline is given below:

- *Number of elements*. Decomposing templates into individually addressable templates, variables and parameters adds some overhead due to additional elements needed for addressing and declaring new services. Therefore, this guideline increases number of elements used in stylesheets.
- *Number of templates*. In XSLT stylesheets, templates can be addressed individually and independently. Other constructs that can be used to achieve similar results (like choice, if or for-each) can not be called from multiple locations and must be used only inline. Therefore, improving addressability increases the number of templates. Match and mode attributes used by templates are also convenient when controlling access and execution of stylesheet functionality (requirement of *subscribability*).
- *Number of parameters and variables*. Just like templates, global variables and parameters can be addressed from multiple locations in the stylesheet. This is useful when target documents have many identical sections (code clones) that can be created once and stored in a parameter or variable for repeated use. Thus, the number of global variables and parameters increases.
- *Number of template calls and XSLT copy-of elements*. New services need to be addressed from the decomposed services. The addressing is done by using XSLT copy-of and call-template elements. Therefore, the guideline increases the number of template calls and XSLT copy-of elements in stylesheets.
- *Length of templates*. As templates can use addressable content and services instead of re-implementing these in every occurrence they need to be used, the templates will be shorter due to reduced number of repetitions. Additionally, breaking longer templates into individually addressable services reduces length of templates as inline code gets replaced by service calls.

D. *Use of metadata.*

Implicit metadata captured in the names of elements or attributes of the input document, and in the structure and

```
<!-- Non-adressable template for two cases -->
<xsl:template match="*">
  <xsl:choose>
    <xsl:when test="@Name or
name()='Resources'">...</xsl:when>
    <xsl:otherwise><xsl:apply-templates
/></xsl:otherwise>
</xsl:choose></xsl:template>
<!-- Adressable templates -->
<!-- Template for Resources -->
<xsl:template
  match="*[@Name or name()='Resources']">
  ...</xsl:template>
<!-- Default template -->
<xsl:template match="*">
  <xsl:apply-templates/></xsl:template>
```

Listing 4. Example of non-addressable template and individually addressable templates.

constructs used in the input document, should be used to identify general cases that can be transformed using generic templates as suggested by the second guideline. This guideline is illustrated by the template in Listing 3, where element names that end with an 's' and that contain sub-elements, are identified as lists and displayed using `div` elements.

Another example were this guideline is applicable is for elements that denote addresses. Addresses, or elements thereof, can be identified from their names and their structure. Once identified, generic services can be used to transform these elements as illustrated in Listing 5.

```
<!--Matches all elements containing address-->
<xsl:template match="*[ends-with(.,'Address') and
count(./*) &gt; 0]">
  <!—Address element -->
  <PhysicalAddress>
    <!-- City -->
    <cityName><xsl:value-of select="./City"/>
</cityName>
    <!-- Address lines and region info -->
    ...
</PhysicalAddress></xsl:template>
```

Listing 5. Template that uses metadata to match elements containing addresses.

An analysis of the impact of applying this guideline is given below:

- *Number of elements*. Using implicit metadata reduces the need for specific tests and allows reuse of code for processing different source document sections. Therefore, the number of elements in stylesheet is reduced.
- *Number of similar code sections*. Using implicit metadata reduces the need for specific tests and allows reuse of code for processing different source document sections, which reduces the number of similar code sections. However, template invocations will be using apply-templates instead of inline code or call-template XSLT element, which may increase the number of similar code sections due to increased number of similar template invocations.

- *Number of template calls and XSLT copy-of elements*. Using implicit metadata requires using match, select or test expressions in stylesheets. This means the guideline is favouring XSLT element apply-templates over XSLT element call-template, which has no means of using implicit metadata present in the source document. As copy-of elements are used to copy constant document fragments, they are not affected by this guideline. Therefore, this guideline reduces number of template calls in stylesheet.
- *Number of XSLT apply-templates elements in stylesheet*. This guideline increases the number of XSLT apply-templates elements due to preferring these to XSLT call-template elements.
- *Number of match, select and test expressions*, which make use of implicit metadata in the source document. The guideline is directed at increasing the use of implicit metadata, which can only be used by match, select and test expressions. Therefore, this guideline increases the number of match, select and test expressions, which make use of implicit metadata in the source document.
- *Number of apply-templates elements with select expression, that uses wildcards*. Since implicit metadata can only be used by match, select and test expressions, template match clauses and apply-templates select clauses will be used to identify the appropriate actions for source entities. If implicit metadata is not used, test clauses or template calls or apply-templates (and templates) with simple select (match) expressions will be used instead. Thus, this guideline increases the number of apply-templates elements with select expression that uses wildcards.

Table 1 summarizes the expected effect of each guideline. For each guideline, the table shows whether the use of the guideline increases the value of each metric (upwards arrow), decreases its value (downwards arrow), has an ambivalent effect (horizontal arrow) or no effect on the metric.. For example, guideline 1 increases the number of similar code sections, however, guideline 2 decreases the number of similar code sections. Guideline 3 does not affect the number of similar code sections and applying guideline 4 can result in either increased or decreased number of similar code sections.

TABLE 1. OVERVIEW OF GUIDELINES' IMPACT.

Metric	Guideline 1	Guideline 2	Guideline 3	Guideline 4
Number of Similar code sections	↑	↓		→
Length of similar code sections	→	→	→	→
Number of elements in stylesheet	→	↓	↑	↓
Number of local variables and parameters in stylesheet	→	→	→	→
Number of global variables and parameters in stylesheet	↑		↑	
Number of template calls and copy-of elements	↑		↑	↓
Number of match, select and test expressions, which use metadata		↑		↑
Number of templates in stylesheet	→	↓	↑	→
Number of apply-templates elements in stylesheet		↑		↑
Length of templates in stylesheets	→	→	↑	→

III. EMPIRICAL EVALUATION

We conducted an empirical evaluation aimed at testing the following three hypotheses:

- (*Size*) Transformations that follow the above guidelines are more concise as measured by LOC. Studies have shown that LOC influences maintainability, though it does not determine maintainability by itself [7].
- (*Code Churn*) The code churn required to update an XSL transformation in response to a change in the source document schema, is lower for transformations that follow the guidelines than for those that do not follow them. Code churn between two versions of a set of transformation is measured by means of *churned LOC* [4]: the sum of the added and changed lines of code between the previous version and the new version. The *cumulative churned LOC* is the sum of the churned LOC between every two consecutive versions along a given versioning branch.
- (*Execution Time*) Transformations that follow the guidelines take shorter to process than transformations that do not follow the guidelines.

To this end, we collected XSL transformations from the following three case studies representing different XSLT usage scenarios:

1. (*Web application scenario*) Transformations for generating XHTML in VabaVaraVeeb (VVV)[5] – a portal dedicated to freeware. These transformations underwent changes in response to evolutions in the underlying schemas over the lifespan of the development of the portal. For this research, we analyzed the XSL transformations corresponding to 8 change iterations that resulted in changes in the source XML schemas.
2. (*Desktop application scenario*) Transformations for creating HTML pages to display analysis results for FxCop[6] – a tool for analysing .NET assemblies.
3. (*Business application scenario*) A transformation for converting purchase order documents between two business-to-business data exchange standards, namely xCBL[7] (XML Common Business Library v1) and RosettaNet.

5 http://vabavara.net/

6 http://blogs.msdn.com/fxcop/

7 http://www.xcbl.org

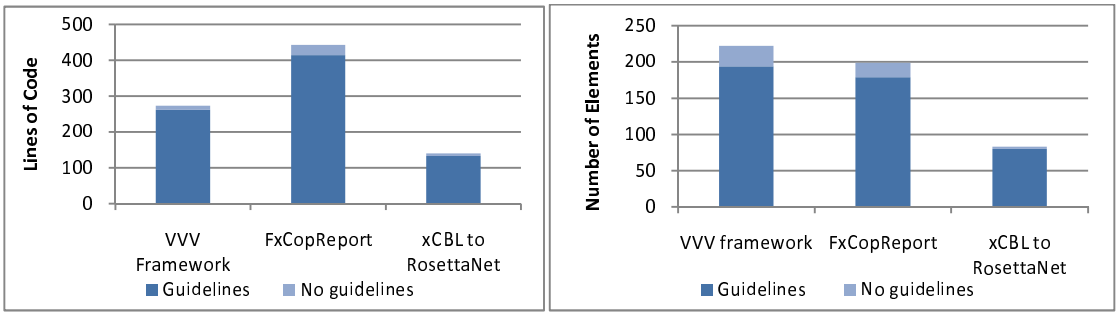


Figure 2. Lines of code and number of XML elements in the XSL transformations

Two versions of each transformation were developed: one created without following the guidelines and one created by changing the first transformation in order to comply with the guidelines. In the case of VVV, both the original and the guideline-compliant versions of the XSL transformations were developed by the first author of this paper. In the case of the FxCop and the xCBL-to-RosettaNet case studies, the original transformations were developed by third parties, and the first author made them compliant with the guidelines. The FxCop transformations were taken from FxCop's web site, while the original xCBL-to-RosettaNet transformation is from [8].

In the case of VVV, we tracked two evolution branches of the transformations: a non-guidelines-compliant branch and a guidelines-compliant branch. Each branch started with one set of transformations. These initial transformations were extended and revised 8 times in response to new features that needed to be added to the portal and that led to changes in the source document schemas. Along each branch, we calculated the LOC of the initial version of the branch (*initial LOC*), and the cumulative churned LOC along that branch. The sum of these two measures (initial plus cumulative churned LOC) was used as a measure of maintainability. The transformations were formatted using Microsoft Visual Studio 2008 "Format Document" command before calculating the initial LOC and the churned LOC. The churned LOC between consecutive versions was calculated using the classical "diff" method.

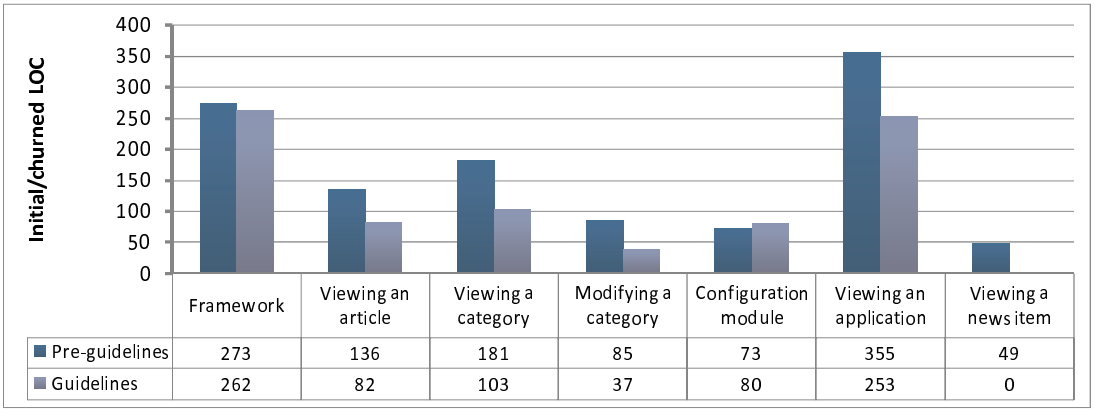
Execution times were measured using PHP 5 XSLT profiling tools. Each transformation was run 1000 times and the total execution time was recorded. PHP 5 XSLT profiling tools only measure execution time and number of calls to templates. They do not provide data on memory usage or usage of other resources. All tests were made on a PC with dual core Intel processor, 2GB RAM running Windows Vista.

IV. Results and Discussion

We first report on the results regarding the conciseness of guidelines-compliant XSL transformations (cf. the "*Size*" hypothesis). The results show minor but visible decrease (4-6%) in the LOC of XSL transformations that followed the guidelines versus those that did not follow them (see the stacked bar chart in Figure 2). In the case of VVV, Figure 2 displays the LOC only for the initial versions of each of the branches (called the "VVV framework").

A more striking difference can be observed if we measure not the LOC, but the number of elements in the XSL transformation – which makes sense since XSL transformations are XML documents themselves. The number of elements used in the guidelines-compliant transformations was consistently lower than the non-guidelines-compliant ones: 10% lower in the FxCop case, 13% lower in the VVV case and 4% lower in the xCBL-to-RosettaNet case.

The xCBL-to-RosettaNet case was the only one for which the number of attributes in the guidelines-compliant version was higher than in the non-compliant version (19% more). In other words, there were less elements in the guidelines-compliant version of the xCBL-to-RosettaNet



	Framework	Viewing an article	Viewing a category	Modifying a category	Configuration module	Viewing an application	Viewing a news item
Pre-guidelines	273	136	181	85	73	355	49
Guidelines	262	82	103	37	80	253	0

Figure 1. Initial and churned LOC for VVV transformation.

TABLE 2. PERFORMANCE TEST RESULTS.

stylesheet test		templates	calls	Time (ms)	Difference (ms)	Difference (%)
xCBL 2 RosettaNet	*Original*	2	60000	204590		
	guidelined	2	60000	210602	6012	2,9%
FxCopReport	*original*	12	2450000	8051419		
	guidelined	13	2450000	7830934	-220485	-2,7%

TABLE 3. EFFECT OF THE GUIDELINES ON THE NUMBER OF TEMPLATES AND TYPES OF EXPRESSIONS

		Templates	Apply Templates	Match		Select		Test	
				Simple	*Complex*	*Simple*	*Complex*	*Simple*	*Complex*
VVV	*pre-guidelines*	16	3	4	0	20	0	9	0
	guidelines	13	7	1	2	18	5	12	5
FxCopReport	*pre-guidelines*	12	13	11	1	44	9	4	18
	guidelines	13	13	11	2	39	9	4	13
xCBL to Rosettanet	*pre-guidelines*	1	0	1	0	26	0	1	0
	guidelines	2	5	1	1	26	0	1	0

transformation but more attributes. FxCop and VVV guidelines-compliant transformations had 6% and 11% less attributes than the non-compliant ones. This difference can be explained by the fact that the xCBL-to-RosettaNet transformation was the simplest original transformation with less duplicate or similar code segments.

An analysis of the evolution of VVV's transformations shows that transformations that follow the guidelines can in most cases be extended with significantly less effort than the original transformations (usually more than 40% decrease in churned LOC). A breakdown of the initial LOC and churned LOC of the VVV transformations is shown as a dual bar chart in Figure 1. Each column denotes one version of the transformation. The leftmost column corresponds to the initial version and plots the LOC of this initial version (the "Framework"). The remaining columns represent subsequent versions and plot the churned LOC with respect to the previous version. Each version corresponds to introducing a new feature into the portal. The chart shows that the use of the guidelines consistently leads to lower code churn.

Interestingly, transformations into XHTML used more templates than the transformation to RosettaNet, suggesting the latter transformation was generated using a schema mapping tool like BizTalk Mapper[8]. As mapping tools move focus from the source document to the output document, the guidelines are more difficult to apply to such transformations.

The performance tests show almost no difference between guidelines-compliant and non-compliant transformations (Table 2). The guidelines lead to more templates with complex match and select expressions, thus increasing the load on the XPath matcher but decreasing memory requirements and the tree-depth of the transformations. These changes may have different effects on XSLT processors, depending on how these processors are optimised. The performance results should thus be interpreted with care. In any case, they indicate that the guidelines do not have a performance penalty, but at the same time they do not provide evidence to back hypothesis 2.

Table 3 shows the observed effects of applying the guidelines on a representative subset of the number of templates, number of apply-templates, and number of match, select, and test expressions. We classify expressions into simple and complex: simple expressions are those that match entities by their exact name, while complex expressions are those that use additional metadata. We count the number of complex expressions in order to determine if the transformations use implicit metadata.

It is difficult to automatically check whether a match, select or test expression is complex (in the sense that it uses implicit metadata). Accordingly, we make the following assumption: simple expressions are those that consist of exact entity names only, while complex expressions are those that contain functions or wildcards. This is only an approximation. The fact that an expression uses wildcards or functions, does not always mean that the expression uses implicit metadata. For example, consider the XPath expression: "`*[name()='Resources']`". Even though this expression contains a wildcard and a function, it is does not use implicit metadata since it can be re-written simply as "`Resources`".

Having adopted this definition, a lower ratio of simple expressions to complex expressions indicates (but does not necessarily guarantee) higher use of implicit metadata. This is consistent with the results in Table 3 showing that guidelines-compliant transformations have a lower ratio of simple- to-complex expressions.

The other results in Table 3 are also in line with the observations in Table 1. In particular, the number of apply-template elements increases when the guidelines are applied. Also, the total number of templates decreased in the first case study, but increased in the two other case studies. This is in line with the analysis in Table 1, since guideline 2 decreases the number of templates, but guideline 3 increases

[8] http://msdn.microsoft.com/en-us/library/ee253382(BTS.10).aspx

it, and the two other guidelines have ambivalent effects on the number of templates.

Because of their contradictory effects it makes sense not to apply all the guidelines at once when redesigning existing transformations. Instead, one can modify the transformation according to the first guideline and then continue with the second, third and fourth (iterative redesign) and repeat the cycle if necessary. The effects in Table 3 can then be used to check the correct application of a specific guideline at each step.

V. CONCEPTUAL ANALYSIS

Moro et al. [9] proposed a framework to analyse the forward-compatibility of XML documents and queries with respect to changes in the underlying schema. Specifically, they introduced a classification of XML Schema changes and evaluated these types of changes with respect to forward and backward compatibility of XML documents. These types of changes are classified into basic changes and complex changes. Basic changes include: Refinement (adding elements); Removing elements; Extension (adding complex types); Reinterpretation (changing the semantics of an element); and Redefinition (factoring out common constructions). Complex changes include: Element Composition (grouping related elements under a new element); Element decomposition (removing an element and splitting its sub-elements); Renaming of an *element/attribute*; Altering optionality (making an optional element required, or vice-versa); Renumbering (changing an element's cardinality); Retyping (changing the type of an element); Namespace change; Default value change; and Reordering (changing the order of the elements in a complex type).

TABLE 4. ANALYSIS OF CHANGE TYPES.

Change type	Effect of change on a guideline-compliant transformation
Refinement	Forward-compatible: the transformation does not assume the element exists
Removal	Forward-compatible if the removed entity was optional or not used by the transformation.
Extension	Same as refinement.
Reinterpretation	Syntactically compatible.
Redefinition	Forward compatible as the structure of the XML document remains unchanged.
Element composition	Forward-compatible if the old entity and new entities are generalised in the same way.
Element decomposition	Same as element composition.
Renaming	Same as element composition.
Optionality	Not forward-compatible if an entity required by the transformation becomes optional, otherwise forward-compatible.
Renumbering	Forward-compatible if the output schema allows multiple transformation target entities or the generalisation performs an aggregation (e.g. transforms a list of element values into space-separated list).
Retyping	Backward and forward-compatible unless value-specific processing is used in the transformation.
Default values	Syntactically compatible.
Namespaces	Not compatible for namespace-aware XSLT processors.
Reordering	Compatible in most cases (order of xml elements is rarely used in transformations as most systems do not guarantee their ordering).

Table 4 outlines the impact of the guidelines on the forward-compatibility of XSL transformations with respect to the types of changes identified by Moro et al.

While the guidelines do not guarantee forward-compatibility with respect to all types of changes, they do improve the level of forward-compatibility in many cases. Reinterpretation, redefinition, changes in default values and changes in namespaces in the source schema are the only types of changes whose impact on transformations is not mitigated by the use of the guidelines. The remaining types of changes are mitigated by the use of generalisations, which pre-empt changes in the source document fragments. Guidelines-compliant transformations impose fewer constraints on the source XML document due to the use of generalisations, which factor out similar entities. Specifically, guidelines-compliant transformations can continue to operate even if some entities are encoded in a different form in the new schema.

We acknowledge that XML Schema is not the only language that can be used to define document structure. One could use DTD to define document structure, for which other (simpler) taxonomies of changes have been proposed [10].

VI. RELATED WORK

Guerrini et al. studied the impact of XML Schema evolution on the validity of existing instances [11]. Their approach minimises the document fragments that need to be re-validated when an XML Schema changes. This makes it easier to identify forward and backward compatibility problems between documents that use different versions of a schema. This work and similar ones are orthogonal to our work since they do not deal with document transformations.

The first of our guidelines has similarities with output-driven approaches for XSL transformation design, such as the approach proposed by Lemmens & van Houben [12]. However, they do not have an explicit concept of common control.

There is a large body of research related to mapping-driven transformations, meaning transformations derived from a mapping between the elements in the source and the target schemas [13]. These mappings can be derived using automatic schema matching techniques [14] and may be visualized and edited by developers using graphical schema mapping tools. Graphical schema mapping tools are incorporated in enterprise application development platforms such as Microsoft BizTalk [3]. While these approaches enhance developer productivity, they are not designed to achieve change-resilience of the resulting transformations. In this respect, the guidelines studied in this paper are complementary to this body of work.

McDowell et al. proposed metrics that can be used to evaluate quality and complexity of MXL Schema and conforming documents [15]. The principles they followed when choosing metrics can be applied to choosing metrics for evaluation of other XML documents like XSL transformations. Additionally, the complexity of source and target documents of XSL transformations can be used to estimate the required complexity of the corresponding XSL transformation.

The guidelines presented in this paper were derived during the development of the VVV portal as reported in [16]. In this portal, XSL transformations were designed and altered by third-parties (outside the portal development team), and these and other guidelines had to be introduced to control the evolution of XSL transformations. In their initial formulation, the guidelines were restricted to XML-to-HTML transformations in the context of the VVV portal. In this paper, we have extended the guidelines to a wider context (including XML-to-XML transformations) and we have provided an experimental and conceptual evaluation of the guidelines.

VII. Conclusion and Future Work

The study reported in this paper is a step towards a more thorough and empirically-tested understanding of how to design maintainable XSL transformations. The experimental evaluation shows that the proposed guidelines have a positive effect on the conciseness and maintainability of the transformations, while having no visible effect on the performance of the transformations (i.e. execution time).

The benefits are most notable in the case of the FxCop and the VVV transformations. Interestingly, both of these collections of transformations have human-oriented output. One can therefore hypothesize that the benefits derived from the guidelines is greater for human-oriented output than for application-oriented output. A study of possible correlations between the purpose of the transformation and the usefulness of the guidelines is thus a desirable direction for future work.

Another desirable extension of this work is to consider other maintainability metrics than those based on LOC or number of elements/templates. While these metrics provide an indication of maintainability, they do not provide a full picture. Other factors worth consideration include complexity, e.g. cyclomatic [7].

Yet another aspect not considered in this paper is that of enforcing the conformance of XSL transformations with respect to the proposed guidelines. Such an undertaking would require the definition of metrics for evaluating the degree of conformance of XSL transformations with respect to the guidelines, and techniques to pinpoint deviations with respect to the guidelines. This direction for future work could lead to quality metrics for XSL transformation with respect to maintainability.

Acknowledgments. This work is funded by Swedbank and ERDF through the Estonian Centre of Excellence in Computer Science. Thanks to D. Chappell for offering the initial set of xCBL-to-RosettaNet transformations.

VIII. References

[1] World Wide Web Consortium XSL Transformations (XSLT), November 1999. http://www.w3.org/TR/xslt

[2] M. Jazayeri, "Some Trends in Web Application Development". *Workshop on Future of Software Engineering (FOSE'07)*, May 2007, pp. 199-213.

[3] D. Probert. "Understanding the BizTalk Mapper: Part 12 - Performance and Maintainability". February, 2008 [Online]. http://tinyurl.com/cefd9l

[4] N. Nagappan and T. Ball. "Use of Relative Code Churn Measures to Predict System Defect Density". *International Conference on Software Engineering (ICSE)*, St. Louis MO, USA, 2005, pp. 284-292.

[5] K. Bennett et al., "Service-Based Software: The Future for Flexible Software". *Asia-Pacific Software Engineering Conference (APSEC),* Singapore, December 2000, pp. 214-221.

[6] C. Armbruster. Design for Evolution, White Paper, 1999. http://chrisarmbruster.com/documents/D4E/witepapr.htm

[7] H. Hayes, J. Liming, and Z. Liming, "Maintainability Prediction: A Regression Analysis of Measures of Evolving Systems," in *Proceedings of the 21st International Conference on Software Maintanance (ICSM)*, 2005, pp. 601-604.

[8] D. Chappell, "JMS and XSLT for E-Business Messaging," SOA World Magazine, vol. 2, no. 2, February 2001.

[9] M. M. Moro, S. Malaika, and L. Lim. "Preserving XML queries during schema evolution". June 2007 http://www.ibm.com/developerworks/xml/library/x-evolvingxquery.html

[10] H. Su, D. Kramer, L. Chen, K. Claypool, and E. A. Rundensteiner, "XEM: Managing the Evolution of XML Documents," in International Workshop on Research Issues in Data Engineering: (RIDE) – Document Management for Data Intensive Business and Scientific Applications, Heidelberg, Germany, April 2001, pp. 103-110.

[11] G. Guerrini, M. Mesiti, and D. Rossi, "Impact of XML schema evolution on valid documents," in *ACM International Workshop on Web Information and Data Management (WIDM 2005)*, Bremen, Germany, November 2005, pp. 39-44.

[12] P. Lemmens and G.-J. Houben, "XML to XML through XML," *Proceedings of WebNet 2001 - World Conference on the WWW and Internet*, Orlando, Florida, 2001, pp. 772-777.

[13] H. Jiang, H. Ho, L. Popa, and W.-S. Han, "Mapping-Driven XML Transformation". *WWW '07: Proceedings of the 16th international Conference on World Wide Web*, Banff, Alberta, Canada, May 2007, pp. 1063-1072.

[14] E. Rahm and P. A.Bernstein, "A survey of approaches to automatic schema matching," *The VLDB Journal*, vol. 10, no. 4, pp. 334--350, 2001.

[15] A. McDowell, C. Schmidt, and K.-b. Yue, "Analysis and Metrics of XML Schema," *International Conference on Software Engineering Research and Practice (SERP)*, Las Vegas, USA, 2004, pp. 538-544.

[16] S. Karus and M. Dumas, "Enforcing Policies And Guidelines in Web Portals: A Case Study". *International Workshops on Web Information Systems Engineering*, Nancy, France, 2007, pp. 154-165.