An Engine for Generating XSLT from Examples

José Paulo Leal[1] and Ricardo Queirós[2],

[1] CRACS & DCC-FCUP, University of Porto, Portugal
zp@dcc.fc.up.pt
[2] CRACS & DI-ESEIG/IPP, Porto, Portugal
ricardo.queiros@eu.ipp.pt

Abstract. XSLT is a powerful and widely used language for transforming XML documents. However its power and complexity can be overwhelming for novice or infrequent users, many of which simply give up on using this language. On the other hand, many XSLT programs of practical use are simple enough to be automatically inferred from examples of source and target documents. An inferred XSLT program is seldom adequate for production usage but can be used as a skeleton of the final program, or at least as scaffolding in the process of coding it. It should be noted that the authors do not claim that XSLT programs, in general, can be inferred from examples. The aim of Vishnu - the XSLT generator engine described in this paper – is to produce XSLT programs for processing documents similar to the given examples and with enough readability to be easily understood by a programmer not familiar with the language. The architecture of Vishnu is composed by a graphical editor and a programming engine. In this paper we focus on the editor as a GWT web application where the programmer loads and edits document examples and pairs their content using graphical primitives. The programming engine receives the data collected by the editor and produces an XSLT program.

Keywords: XSLT, Transformations, Refactoring.

1 Introduction

Generating a XSLT program from a pair of source and target XML documents is straightforward. A transformation with a single template containing the target document solves this requirement, but is valid only for the actual example. Using the information from the source document we can abstract this transformation. The simplest way is to assume that common strings in both documents correspond to values that must be copied between them. If we explicitly identify these correspondences we can have more control over which strings are copied and to which positions. However, a transformation created in this fashion is still too specific to the examples and cannot process a similar source document with a slightly different structure. For instance, if the source document type accepts a repeated element e and the example has n repetitions of the element e then the generated program would accept exactly n repetitions of that element.

Although too specific, a simple XSLT program can be used as the starting point for generating a sequence of programs that are more general and are better structured, ending in a program with a quality similar to one coded by a human programmer. To refine an XSLT program we can use second order XSLT transformations, i.e. XSLT transformations having XSLT transformations both as source and target documents. In this approach the role of an XSLT generation engine is to receive source and target examples, and an optional mapping between the strings of the two documents, generate an initial program and control the refinement process towards the final XSLT program.

The aim of this paper is the presentation of Vishnu – an XSLT engine for generating readable XSLT programs from examples of source and target documents. Readability is an essential feature of the generated programs so that they can be easily understood by a programmer not familiar with the language. The architecture of Vishnu is composed by a graphical editor and a programming engine. The former acts as a client where the programmer loads and edits document examples and pair their content using graphical primitives. The latter receives the data collected by the editor and produces an XSLT program.

There are several use cases for an XSLT generation engine with these features. The Vishnu generator was designed to interact with a component that provides text editing functions for the end-user or programmer. A client of Vishnu can be a plug-in of an Integrated Development Environment (IDE) such as Eclipse or NetBeans. In this case the IDE provides several XML tools (highlighting, validation, XSLT execution) and the plug-in is responsible for binding the content of text buffers and editing positions with the engine and retrieving the generated XSLT program. Vishnu can also be used as the back-end of a web environment for XSLT programming. In this case the web front-end is responsible for editing operations and invokes engine functions for setting the example documents and mappings, and retrieving the generated program. The generator can also be used as a command line tool as part of a pipeline for generating and consuming XSLT programs. In this last case the generator processes example documents in the local file systems, making mostly use of default mappings.

The rest of the paper is organized as follows. Section 2 presents work related to XSLT editing and generation. In the following section we present the inner structure of the XSLT generator that is composed of three main components: the context, the generator and the refiner. Then, we evaluate the Vishnu XSLT generation engine from three complementary and interrelated approaches, focusing: the consistency of generation and refinement process; the coverage of the existing rules; and the adequacy of the Vishnu API to XSLT editing environments. Finally, we conclude with a summary of the main contributions of this work and a perspective of future research.

2 Related Work

The first step to start editing XSLT files is choosing the editor that most suits one's programming environment. There are tools integrated in XML IDEs [1, 2], tools integrated in general purpose IDEs as plug-ins [3, 4, 5, 6, 7, 8] and even standalone applications [9, 10, 11]. Despite the existence of several environments for programming in XSLT, usually integrated into IDEs, they do not use visual editing for programming. Moreover, as far as we know, none of the graphical XSLT programming environment generates programs from examples as source and target documents.

Hori and Ono [12, 13] use an example-based annotation tool which relies on a target document editor. The main concepts of their approach are depicted in Figure 1. An annotator can edit a target document (e.g., an HTML page) by using the capabilities of a WYSIWYG authoring tool (1). The editing actions are recorded into an operation history (2). When the editing is finished, the annotation generator creates transformational annotation for the document customization (3), which can be further used by XSLT processor to replicate the transformation from the initial document to the customized document.

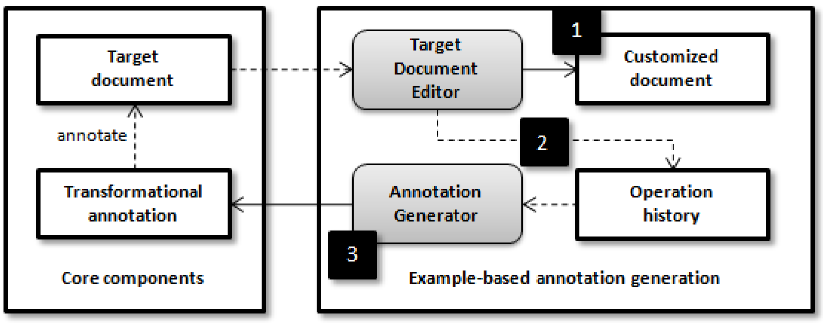


Fig. 1 History based document transformation.

Spinks [14] presents an annotation-based page-clipping engine providing a way of performing Web resources adaptation. At content delivery time, the page-clipping engine modifies the original document based on: 1) the page-clipping annotations previously generated in a WYSIWYG authoring tool and 2) the user-agent HTTP header of the client device. The page-clipping annotation language uses the `keep` and `remove` elements in the annotation descriptions to indicate whether the content being processed should be preserved or removed.

3 The Vishnu engine

The Vishnu engine [15] concentrates all the tasks related with the automatic generation of an XSLT program from examples using second order transformations. Nevertheless, it was designed to interact with a client. A client of the Vishnu engine concentrates all the tasks related with user interaction where the programmer loads and edits document examples and pairs their content using graphical primitives.

The communication between these two components is regulated by the Vishnu API. Hence, the architecture of the Vishnu application is composed by a **Graphical Editor** and a **Programming Engine** as depicted in Figure 2.

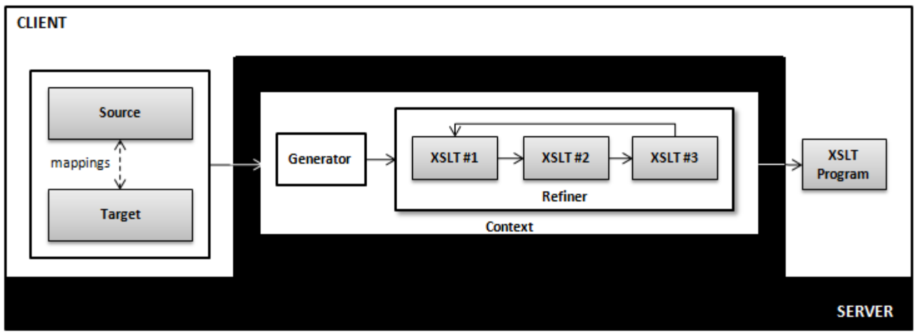


Fig. 2 The architecture of Vishnu.

The former acts as a client where the programmer loads and edits document examples and pair their content using graphical primitives. The design and implementation of a client for the Vishnu engine is presented in the next section to validate the adequacy of the Vishnu API to XSLT editing environments.

The latter receives the data collected by the editor and produces an XSLT program. The engine relies on the Vishnu API that includes methods for setting the source and target documents as streams of characters, setting a mapping between the strings of these documents using editing locations (offsets), and retrieving the resulting XSLT program. The Vishnu API includes also functions for supporting graphical interaction in the editor and for configuring the generation process. The functions for selecting strings in the XML documents (text and attribute nodes) from editing locations are example functions for supporting graphical interaction. The Vishnu façade class implements this API and hides the inner structure of the XSLT generator that is composed of three main components: the **context**, the **generator** and the **refiner**.

3.1 Context

The central piece of the engine is the generation **context**. The context holds the source and target documents and the mapping between the two and is responsible for converting between the external textual representation provided by the client and the internal XML representation required by the Vishnu. In particular this component is responsible for converting document position into XPath expressions and vice-versa.

The conversion is managed by the PathLocator class. This class converts text locations (offsets) into *IdPaths* expressions and vice-versa. An *IdPath* is an absolute XPath expression which selects either single texts or attribute nodes in an XML document. The general form of an *IDPath* is:

$/n^1[p^1]/.../n^n[p^n]/\texttt{text()}$
$/n^1[p^1]/.../n^n[p^n]/\texttt{@attr}$

It should be noted that locating nodes from using their editing positions and the reverse are not operations supported by the APIs for processing XML documents.

The Context component is also responsible for the generation of the mapping between the source and the target documents. It maintains an XML map file identifying the correspondences between both. These identifications can be inferred automatically or manually set through the Editor. The following XML excerpt shows an example of a source, target and a list of pairs of XPath expressions relating them merged in a file called `vishnu.xml`.

```
<vishnu xmlns="http://www.dcc.fc.up.pt/vishnu">
<!—Source document -->
<source>
<rss version="2.0" xmlns="http://backend.userland.com/rss2"/>
     <channel>
       <title>News</title>
       <link>…</link>
       <description>…</description>
       <item>
       …
       </item>
      </channel>
</rss>
</source>
<!—target document -->
<target>
<html xmlns="http://www.w3.org/1999/xhtml">
      <head>
        <title>News</title>
      </head>
      <body>
        <h1>News</h1>
         …
      </body>
</html>
</target>
<!—pairing document-->
<pairings>
      <pairing
         source = "/rss[1]/channel[1]/title[1]/text()"
         target = "/html[1]/head[1]/title[1]/text()"/>
      <pairing
         source = "/rss[1]/channel[1]/title[1]/text()"
         target = "/html[1]/body[1]/h1[1]/text()"/>
</pairings>
</vishnu>
```

This file will serve as input for the Generator component to produce a XSLT program.

3.2 Generation

The purpose of the **generator** is to produce an initial XSLT program from the source and target, using a string mapping. If no mapping is provided by the client then it uses a default mapping inferred by the context component, linking text or attribute nodes in both documents with equal character strings. The generator component receives as input the paring file and, using a second order transformation, produces a specific XSLT program. As an illustration we present the output of this second order stylesheet based on the example included in the previous subsection.

```
<xsl:template match="/">
 <html>
   <head>
     <title>
      <xsl:value-of
select="/vishnu/source/rss[1]/channel[1]/title[1]/text()"/>
      </title>
    </head>
    <body>
      <h1>
        <xsl:value-of
select="/vishnu/source/rss[1]/channel[1]/title[1]/text()"/>
      </h1>
      …
    </body>
  </html>
</xsl:template>
```

The initial XSLT program has a single template containing an abstraction of the target document. To abstract the target document the target positions in the mapping are replaced with `xsl:value-of` instructions referring corresponding source positions in the mapping. As explained previously, with this level of abstraction the initial transformation is only able to process a document with the exact same structure of the source document provided as input. To be of any practical use this program is submitted to a refinement process.

3.3 Refinement

The **refinement** process produces a sequence of XSLT programs ρ_n starting with the initial program ρ_0 by applying $R = \{r_i\}$ set of second order XSLT transformations called refinements. Refinements can be divided in two categories: **simplifications** and **generalizations**.

Let S_0 and T_0 be respectively the example source and target documents. All refinements r_i have the following invariant: $\rho_n(S_0) = T_0 \Rightarrow r_i(\rho_n)\,(S_0) = T_0$ that is, if a program maps the example source document to the example target document then the refined program has the same property. A simplification refinement is even more restrictive and any document S that is converted by program S_0 is equally converted by its refinement, $i.e. \forall\, S, T\; \rho_n(S) = T \Rightarrow \mathrm{r_i}(\rho_\mathrm{n})(\mathrm{S}) = T$. Simplifications are "safe" refinements but fail to introduce the level of abstraction needed for a transformation to be effective, hence this stronger requirement is relaxed for abstractions.

An example of a generalization is the refinement that unfolds a single template into a collection of smaller templates. Candidates to top elements in the new template are elements whose XPath expressions in `xsl:value-of` share a common and non-trivial prefix that can be used match of the new template. As it introduces new templates with relative expressions in the match attribute this refinement is not a simplification. The new template may match with nodes with the same tag occurring in different points in a different source document structure. To minimize the chance of unwanted matches this refinement associates a mode to the new template that is used also by the `xsl:apply-template` instruction that invokes it. An example of a simplification is the refinement that removes redundant modes from `xsl:template` and `xsl:apply-template` instructions. This refinement selects templates with non empty modes that cannot be matched by other templates. That mode is removed both from the selected template and all `xsl:apply-template` referring it. The current Vishnu implementation includes over 10 refinements.

As an illustration we present the final output of the refinement process based on the example included in the previous subsection.

```
<xsl:stylesheet version="1.0" …>

<xsl:template match="rss2:channel">
 <xhtml:html>
   <xsl:apply-templates mode="xhtml:head" select="rss2:title"/>
    <xhtml:body>
      <xsl:apply-templates mode="xhtml:h1" select="rss2:title"/>
      <xhtml:ol>
        <xsl:apply-templates select="rss2:item"/>
      </xhtml:ol>
   </xhtml:body>
 </xhtml:html>
</xsl:template>

<xsl:template match="rss2:item">
  <xhtml:li>
    <xhtml:a href="{rss2:link}">
      <xsl:value-of select="rss2:title"/>
    </xhtml:a> -
    <xsl:apply-templates select="rss2:description"/>
  </xhtml:li>
</xsl:template>

<xsl:template match="rss2:description">
   <xhtml:i><xsl:value-of select="."/></xhtml:i>
</xsl:template>

<xsl:template match="rss2:title" mode="xhtml:h1">
    <xhtml:h1><xsl:value-of select="."/></xhtml:h1>
</xsl:template>

<xsl:template match="rss2:title" mode="xhtml:head">
 <xhtml:head>
   <xhtml:title><xsl:value-of select="."/></xhtml:title>
 </xhtml:head>
</xsl:template>

</xsl:stylesheet>
```

The Vishnu engine supports different refinement **strategies** to control the application of the refinement `setR`. A refinement strategy indicates the next refinement to use is informed if the suggested refinement has changed the XSLT program and decides when the refinement process is complete. There are several refinement strategies that can be set using the Vishnu API. The most effective strategies implemented so far apply the refinements in a predefined order, repeating the application of refinement while it is effective.

4 Validation

The Vishnu engine was validated in three complementary and interrelated approaches, focusing the

consistency of the generation and refinement process;
coverage of the existing rules;
adequacy of the Vishnu API to XSLT editing environments.

By default Vishnu validates the **consistency** of the generation and refinement process by checking that each intermediate transformation converts the example source document into the examples target document. If this invariant is not satisfied then the refinement process is aborted and an error is reported to the client.

To validate the **coverage** of the existing rules different scenarios were created. Each scenario includes source and target document and a mapping, as well as the expected program.

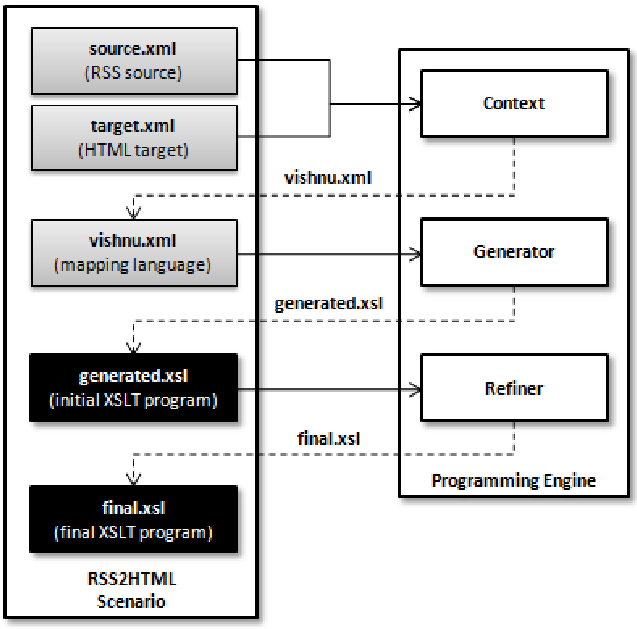


Fig. 3 The RSS to HTML scenario.

The manipulation of a scenario in Vishnu is made by the Scenario class. This class provides a set of methods for testing the Vishnu engine. Typical uses involve a set of scenarios where for each scenario the generated output of the engine is matched with the resources enclosed on the scenario itself. The current scenarios include the conversion of: 1) RSS documents to HTML; 2) Mathematical expressions in MathML to presentation MathML and 3) Meta-data in LOM (Learning Object Metadata) to RDF. The Figure 3 shows the inner workflow used for testing the RSS to HTML scenario. A mixed-content scenario has not been added yet since the context component is not supporting indexes in text nodes.

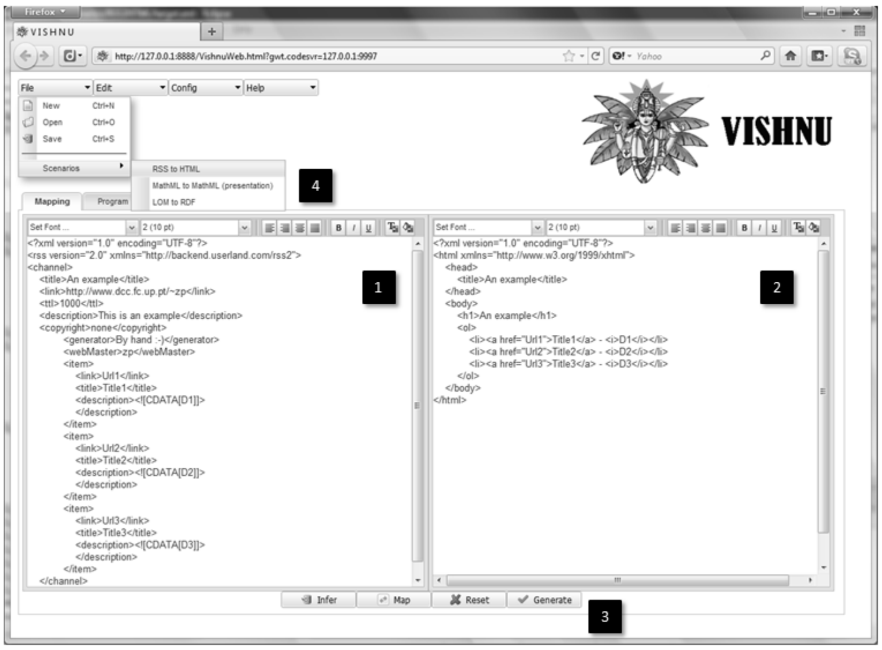
To validate the **adequacy** of the Vishnu API we developed a simple web environment for XSLT programming based on the Google Web Toolkit (GWT), an open source framework for the rapid development of AJAX applications in Java. When the application is deployed, the GWT cross-compiler translates Java classes of the GUI to JavaScript files and guarantees cross-browser portability. The specialized controls are provided by SmartGWT, a GWT API's for SmartClient, a Rich Internet Application (RIA) system.

The graphical interface of the front-end is composed by two panels: Mapping and Program. In the **Mapping panel** the "programmer" uses graphical tools to map strings in two XML documents corresponding to a source and a target documents for the intended XSLT transformation. In the **Program panel** the user obtains the resulting XSLT and can continue editing it.

Figure 4 shows the RSS-to-HTML scenario being used on the Vishnu client GUI with its main components labelled with numerals. The **Mapping panel** includes two side-by-side windows for editing respectively (1) the source and (2) the target documents. These documents may be created either from scratch or based in scenarios predefined in the Engine. Regardless of the choice the correspondences between both can be set (3) **manually** through the Editor or **inferred** by the Engine.

When setting correspondences manually the programmer is able to pair contents on these windows by selecting and highlighting with color texts where the origin is on the source document and the destination is on the target window. Origin and destination must be character data, either text nodes or attribute values.

When automatic correspondence is used Vishnu identifies pairs based on: text matches (text or attribute nodes) or text aggregation. In the first mode strings occurring on text and attribute type nodes on the source document are searched on the text and attribute nodes of the target document, and only exact matches are considered. In the second mode Vishnu aggregates strings in the source document to create a string in the target document. After automatic pairing, the inferred correspondences are presented in the GUI with colors mapping the two XML documents. The user can then manually reconstruct the pairing of string between both documents.



```
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0" xmlns="http://backend.userland.com/rss2">
<channel>
   <title>An example</title>
   <link>http://www.dcc.fc.up.pt/~zp</link>
   <ttl>1000</ttl>
   <description>This is an example</description>
   <copyright>none</copyright>
         <generator>By hand :-)</generator>
         <webMaster>zp</webMaster>
         <item>
            <link>Url1</link>
            <title>Title1</title>
            <description><![CDATA[D1]]>
            </description>
         </item>
         <item>
            <link>Url2</link>
            <title>Title2</title>
            <description><![CDATA[D2]]>
            </description>
         </item>
         <item>
            <link>Url3</link>
            <title>Title3</title>
            <description><![CDATA[D3]]>
            </description>
         </item>
   </channel>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml">
   <head>
      <title>An example</title>
   </head>
   <body>
      <h1>An example</h1>
      <ol>
         <li><a href="Url1">Title1</a> - <i>D1</i></li>
         <li><a href="Url2">Title2</a> - <i>D2</i></li>
         <li><a href="Url3">Title3</a> - <i>D3</i></li>
      </ol>
   </body>
</html>
```

Fig. 4 Vishnu client front-end.

In complement to creating the source and target documents from scratch, the user can fill in automatically the two rich text editors by using scenarios (4). Each scenario includes source and target document and a mapping, as well as the expected program.

5 Conclusions

In this paper we present Vishnu - an XSLT generator engine that aims to produce XSLT programs for processing documents similar to the given examples and with enough readability to be easily understood by a programmer not familiar with the language.

The project that lead to the development of the Vishnu can follow different paths: the engine can be used in other XSLT programming environments; the API of the engine can extended with new functions; and the refinement process can be extended with new refinements. First of all, the Vishnu API was validated with a web environment but the appropriate place to apply it would be an IDE with support for XML. Eclipse is particularly suited for this purpose because it is not a XML IDE but rather an IDE for programming in general with tools for handling XML, including XSLT programming. Secondly, the Vishnu engine was designed as a tool for generating simple XSLT programs from examples and can be extended for other uses. The refinement process was designed to improve the quality of a naïve XSLT program automatically generated from examples but can be used to improve any XSLT program. In fact, an interesting side effect of this research is the definition of sort of “canonical XSLT” in terms of second order XSLT transformations. In practical terms we plan to expand the Vishnu API to enable the use of the refinement process on a given XSLT program, rather than only on those generated from examples. This feature may be used in the XSLT programming environment to refractor any XSLT programs, including the generated program after it was edited by the programmer. Finally, Vishnu is an expandable system in the sense that refinements and refinement strategies can be easily integrated. We expect to create new refinements both to improve the quality of automatically generated XSLT programs and to introduce new forms of automatically refactoring existing XSLT programs.

References

1. Stylus Studio - http://www.stylusstudio.com/
2. Altova StyleVision - http://www.altova.com/stylevision.html
3. Tiger XSLT Mapper - http://www.axizon.com/
4. XSL Tools - http://marketplace.eclipse.org/content/xsl-tools
5. oXygen - http://www.oxygenxml.com/eclipse_plugin.html
6. XMLSpy Eclipse editor - http://www.altova.com/xmlspy/eclipse-xml-editor.html
7. OrangevoltXSLT - http://eclipsexslt.sourceforge.net/
8. X-Assist - http://sourceforge.net/projects/x-assist/
9. Dexter-xsl - http://code.google.com/p/dexter-xsl/
10. VXT: A Visual Approach to XML Transformations. Emmanuel Pietriga, Jean-Yves Vion-Dury and Vincent Quint. Proceedings of the 2001 ACM Symposium on Document engineering, USA
11. FOA. Formatting Objects Authoring tool - http://foa.sourceforge.net
12. Hori, M., Ono, K., Abe, M. and Koyanagi, T.: Generating transformational annotation for Web document adaptation: Tool support and empirical evaluation. Journal of Web Semantics, 2(1), pp. 1-18 (2004-12).
13. Ono, K. et al., “XSLT Stylesheet Generation by Example with WYSIWYG Editing,” Proceedings of the Symposium on Applications on the Internet (SAINT 2002), 2002, pp. 150-159.
14. Spinks, R., Topol, B., Seekamp, C., and Ims, S.: Document clipping with annotation. IBM developerWorks, http://www.ibm.com/developerworks/ibm/library/ibmclip/ (2001).
15. Leal, J.P. and Queirós, R.: Visual Programming of XSLT from Examples - 8ª Conferência - XML: Aplicações e Tecnologias Associadas, Vila do Conde, Portugal, June, 2010.