An example based generator of XSLT programs

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.

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.

This approach visual XSLT programming has obvious limitations. Only a subset of all possible XSLT transformations is programmable by pairing texts on a source and target documents. For instance, second order transformations or recursive templates are out of its scope. Use cases for Vishnu are formatting XML documents in XHTML and conversion among similar formats. For instance, creating an XHTML view of an RSS feed and converting metadata among several XML formats are among the possible uses of Vishnu. Moreover, we do not expect the automated features of Vishnu to produce the final version of an XSLT program. We view its final result as a skeleton of a transformation that can be further refined using other tools already available in Eclipse.

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. In the refiner component we highlight the two types of refinements: simplifications and abstractions. 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.

RELATED WORK

The first step to start editing XSLT files is choosing the editor that most suits one's programming environment. There are several environments for programming in XSLT. Usually these tools are integrated in XML IDE's or in general purpose IDE's such as Eclipse. In the former we can highlight StyleVision and Stylus Studio. StyleVision [1] is a commercial visual stylesheet designer for transforming XML. It allows drag and dropping XML data elements within an WYSIWYG interface. An XSLT stylesheet is automatically generated and can be previewed using the FOP built-in browser. Stylus Studio's [2] is another commercial XML IDE that includes a WYSIWYG XSLT designer. The edition process is guided by simple drag-and-drop operations without requiring prior knowledge of XSLT.

There are also several plugins for Eclipse for editing XSLT and the Tiger XSLT Mapper [3] is the most prominent. It is a simple development environment that supports automatic mappings between XML structures and can be edited using the drag-and-drop visual interface. While the mappings and XML structures are modified, the XSLT template is automatically generated and modified. Other examples of Eclipse plugins address the XSLT edition [4, 5, 6] and the XSLT execution [7,8].

There are other tools analogue to Vishnu that are not integrated into Eclipse, as the dexter-xsl [9] which is intended to be used from the command line, the VXT [10] a visual programming language for the specification of XML transformations in an interactive environment and FOA [11] an XSL-FO graphical authoring tool to create XSL-FO stylesheets. It includes a tree visualization scheme to represent the source XML document and the target FO tree structure. FOA generates an XSLT stylesheet that transforms XML content into an XSL-FO document.

Despite the existence of several environments for programming in XSLT, usually integrated into IDE's, 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.

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 **Erro! A origem da referência não foi encontrada.**. 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.

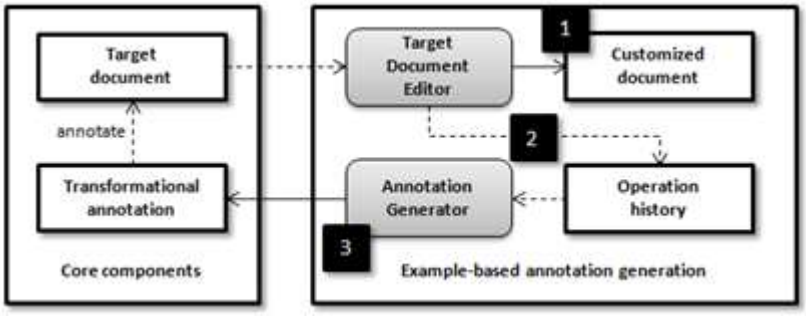

Figure 1. History based document transformation.

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.

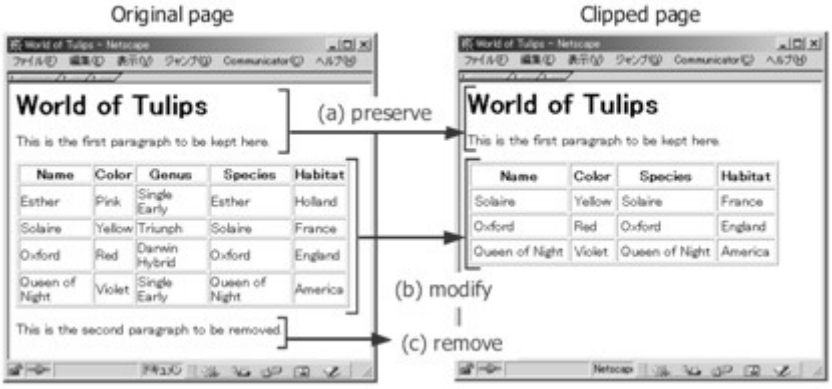

Figure 2. Simple example of an HTML page clipping.

As a simple example, an HTML page and its clipped results are shown in Figure 2. In this example, the header and the first paragraph are preserved. The table element is modified by deleting the third column and the second row. In addition, the whole of the second paragraph is removed. All the structural changes in HTML documents can be easily done by using a WYSIWYG HTML editor. The XML excerpt in **Erro! A origem da referência não foi encontrada.** shows an annotation document that realizes the page clipping.

The `description` element defines a unit of an annotation statement in the annotation language. The `target` attribute is an XPath expression identifying the node on which the annotation will be applied, and the `take-effect` attribute indicates whether the annotation is applied before or after the target node. The following code shows a simple example of an HTML page clipping.

```
<?xml version='1.0' ?>
<annot version="2.0">
	<!-- (a) Set the default clipping state to 'keep' -->
	<description take-effect="before"
		target="/HTML[1]/BODY[1]/*[1]">
		<keep/>
	</description>
	<!-- (b) Remove a column and a row of the first -->
	<!-- table, and change a cellpadding -->
	<!-- attribute value -->
	<description take-effect="before"
		target="/HTML[1]/BODY[1]/TABLE[1]">
		<keep/>
		<table>
			<column index="3" clipping="remove"/>
			<column index="*" clipping="keep"/>
			<row index="2" clipping="remove"/>
			<row index="*" clipping="keep"/>
		</table>
		<insertattribute name="cellpadding" value="4"/>
	</description>
	<!-- (c) Set the clipping state to 'remove' -->
	<description take-effect="before"
		target="/HTML[1]/BODY[1]/P[2]">
		<remove/>
	</description>
	<!-- (d) Set the clipping state back to 'keep' -->
	<description take-effect="after"
		target="/HTML[1]/BODY[1]/P[2]">
		<keep/>
	</description>
</annot>
```

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 **Erro! A origem da referência não foi encontrada.**.

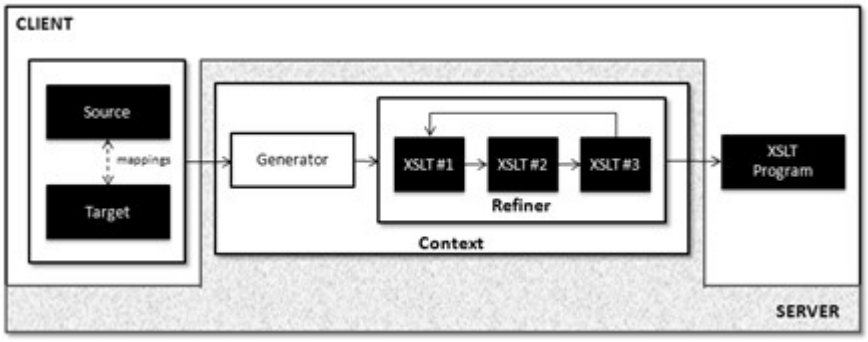

Figure 3. The architecture of Vishnu.

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 facade 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**.

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. The mapping can be set manually through a GUI client or inferred. When in automatic pairing mode Vishnu tries to identify pairs based on:

- Text matches (text or attribute nodes);
- 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 this mode a single occurrence of a string in the source document may be paired with several occurrences in the target document, as depicted in **Erro! A origem da referência não foi encontrada.**.

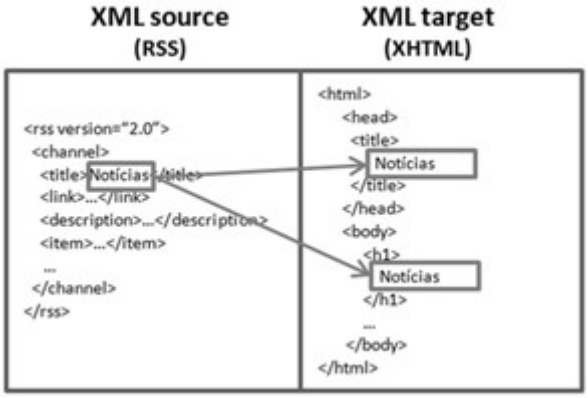

Figure 4. Automatic mapping - exact match between single texts.

In the second mode Vishnu tries to aggregate strings in the source document to create a string in the target document. In **Erro! A origem da referência não foi encontrada.** we illustrate with a simple case where 3 strings occurring in attributes and text nodes can be concatenated into a part of the text node on the target document. In this mode several strings on the source document can be paired with strings on the target document.

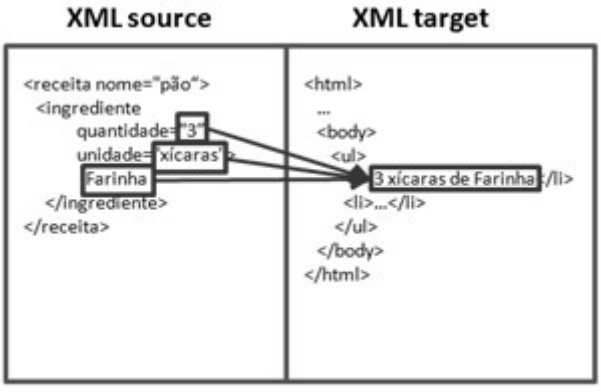

Figure 5. Automatic mapping - subset of aggregation of texts.

After automatic pairing, the inferred correspondences are presented in the GUI with lines connecting the two XML documents. The user can then manually reconstruct the pairing of string between both documents.
The result of pairing the examples is a document including the actual documents and a list of pairs of XPath expressions relating them. This document is formally defined by an XML schema depicted diagrammatically in **Erro! A origem da referência não foi encontrada.**.

Figure 6. The mapping XML language.

The pairing XML language has an element `vishnu` as the root element with three top elements:

- `source` - a copy of the source document;
- `target` - a copy of the target document;
- `pairings` - list of pairing relating the two documents.

Each correspondence is defined by a pairing element with two attributes for selecting textual occurrences in both documents: `source` and `target`. The `source` attribute includes a valid XPath expression selecting the text to map in the source document. The `target` attribute includes a valid XPath expression selecting the text of the target document.

As said before 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:

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

It should be noted that locating nodes using their editing positions and the reverse (locating an editing position of a node) are not operations supported by XML processing APIs. The implementation of these operations by the PathLocator class is not trivial. The current version is not yet supporting indexes on references to text nodes. With this limitation we were not able yet to apply Vishnu no mixed content scenarios. However, upgrading the PathLocator to supported sibling text nodes is a comparatively easy task that we expect to complete in the next version of Vishnu.

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 XML excerpt in Doc 2 (based on **Erro! A origem da referência não foi encontrada.**) shows an example of a source, target and a list of pairs of XPath expressions relating them merged in a file called `vishnu.xml`.

This file will serve as input for the Generator component to produce a XSLT program. The following code shows the XML pairing file.

```
<vishnu xmlns="http://www.dcc.fc.up.pt/vishnu">
<!—Source document -->
<source>
<rss version="2.0" xmlns="http://backend.userland.com/rss2"/>
     <channel>
       <title>Notícias</title>
       <link>... </link>
```

```
       <description>…</description>
       <item>
       ...
       </item>
      </channel>
</rss>
</source>
<!—target document -->
<target>
<html xmlns="http://www.w3.org/1999/xhtml">
      <head>
        <title>Notícias</title>
      </head>
      <body>
        <h1>Notícias</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>
```

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 in **Erro! A origem da referência não foi encontrada.** the output of this second order stylesheet based on the example included in the previous subsection. The following code shows an example of the output of the second order stylesheet.

```
<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>
      <ol>
         <li>
            <a>
              <xsl:attribute name="href">
                <xsl:value-of
select="/rss[1]/channel[1]/item[1]/link[1]/text()"/>
               </xsl:attribute>
```

```
              <xsl:value-of
select="/rss[1]/ channel[1]/item[1]/title[1]/text()"/>
            </a> -
            <i>
               <xsl:value-of
select="/rss[1]/channel[1]/item[1]/description[1]/text()"/>
            </i>
          </li>
          ...
       </ol>
    </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.
Doc 3 illustrates the verbosity of the initial template with XPath expressions are very long and difficult to read, with indexes in every path branch. Actually, the expressions are even more complex that those presented here since all elements must be prefixed when namespaces are used. For the sake of clarity namespace prefixes in XPath expressions where omitted in this example. The initial template has also plenty of redundancy. This XSLT transformation generates an XHTML ordered list element (<ol>) with the exact number of item elements (<li>) need to reflect the given example. Thus, the part of the document marked with ellipsis corresponds almost to a repetition of the of the first XHTM item element, with changes only on the index of the RSS item element referred by XPath expressions.
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.

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, $i.e. \forall S, T\ \rho_n(S) = T \Rightarrow r_i(\rho_n)(s) = T$. 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.

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.

As an illustration we present the final output of the refinement process based on the example included in the previous subsection in **Erro! A origem da referência não foi encontrada.**. The following code shows the final output of the refinement process.

```
<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 control of the refinement process is implemented in Java, rather than in XSLT. This separation encourages the modularity and reusability of the refinement transformations

which would be harder to achieve if the whole refinement process was encoded in a single XSLT. With this approach is easy to introduce new refinements or to temporarily switch them off. It is easier to change a single and simple XSLT file than to change the code and recompile the application. There are two types of refinements [15] - simplifications and abstractions – that are detailed on following sub-subsections, after which are presented implementation details these second order transformations.

Simplifications are refinements that preserve the semantics of the program while changing its syntax. Preserving the semantics means that, for all documents S and T, if a program P transforms document S in document T then the program P', resulting from a simplification refinement, will also transform S to T.
Simplifications can be used for different purposes. They can be used to improve the readability of XPath expressions or to extract global variables. The following paragraphs illustrate this concept with concrete simplifications and examples of the refinements they introduce.

- **Context**: extracts the common prefix of all the XPath expressions from value-of elements in the same template and append it as a suffix of the match attribute on the template element

Table 1 *Applying the Context refinement.*

Source XSLT	Result XSLT
<xsl:template match="a"> ... ...<xsl:value-of select="b/c"/> ...<xsl:value-of select="b/d"/> </xsl:template>	<xsl:template match="a/b"> ... ...<xsl:value-of select="c"/> ...<xsl:value-of select="d"/> </xsl:template>

- **Melt:** two or more templates with the same containers are merged into one in which the match attribute is an expression that combines the terms of th original attributes match using the operator (|) that computes two or more node-sets.

Table 2 *Applying the Melt refinement.*

Source XSLT	Result XSLT
<xsl:template match="a"> ... </xsl:template> <xsl:template match="b"> ... </xsl:template>	<xsl:template match="a | b"> ... </xsl:template>

- **Extract:** strings inside the templates are assigned to global variables;

Table 3 *Applying the Extract refinement.*

Source XSLT	Result XSLT
<xsl:template ...> xpto </xsl:template>	<xsl:variable name="x" select="'xpto'"/> ... <xsl:template ...> <xsl:value-of select="$x"> </xsl:template>

- **Join:** different variables within the same scope and the same content are replaced by a single variable;

Table 4 *Applying the Join refinement.*

Source XSLT	Result XSLT
<xsl:variable name="x1" select="'xpto'"/> <xsl:variable name="x2" select="'xpto'"/> ... <xsl:value-of select="$x1"/> ... <xsl:value-of select="$x2"/>	<xsl:variable name="x1" select="'xpto'"/> ... <xsl:value-of select="$x1"/> … <xsl:value-of select="$x1"/>

- **Braces**: attribute values defined by XSL elements are replaced by braces with XPath expressions

Table 5 *Applying the Braces refinement.*

Source XSLT	Result XSLT
<a> <xsl:attribute name="href" select="item/url"/>… 	…

- **Mode**: Removes modes that do not contribute to differentiate templates from template definitions and related apply-templates;

Table 5 *Applying the Mode refinement.*

Source XSLT	Result XSLT
<xsl:template mode="m"> … </xsl:template> <xsl:apply-templates mode="m"/>	<xsl:template> … </xsl:template> … <xsl:apply-templates/>

- **Orphan**: remove template with just a single xsl:apply-templates (orphan) with the same mode (must be applied after removing unneeded modes)

Table 6 *Applying the Orphan refinement.*

Source XSLT	Result XSLT
<xsl:template mode="m"> <xsl:apply-templates mode="m"> </xsl:template>	

Abstractions are refinements that change both the syntax and the semantics of the program, although retaining the intended semantics of the example documents. This means that, for the documents S and T given as example, if a program P transforms document S in document T then the program P', resulting from a abstraction refinement, will also transform S to T.
Abstractions can be used for different purposes. For instance, they can be used to generalize templates and to restructure large templates in several smaller ones. The

following paragraphs illustrate this concept with concrete abstractions and examples of the refinements they introduce:

- **Generalize:** two or more templates with the same container and a match attribute differing only in the "index" are merged into one and is removed the last predicate of the attribute match. The original transformation accepts only document with a precise number of elements of a certain kind and the abstracted transformation accepts an undetermined number of elements of that kind.

Table 7 *Applying the Generalize refinement.*

Source XSLT	Result XSLT
<xsl:template match="a[1]"> ... </xsl:template> <xsl:template match="a[2]"> ... </xsl:template> <xsl:template match="a[3]"> ... </xsl:template	<xsl:template match="a"> ... </xsl:template>

- **Structure:** fragments templates that contain Xpath expressions with a common prefix. The extracted template will match with elements outside the scope of the original template.

Table 8 *Applying the Structure refinement.*

Source XSLT	Result XSLT
<xsl:template match="a"> <X> <xsl:value-of select="b/x"> <xsl:value-of select="b/y"> </X> <xsl:value-of select="c"> </xsl:template>	<xsl:template match="a"> <xsl:apply-templates select="b"/> ... <xsl:value-of select="c"> </xsl:template> <xsl:template match="b"> <X> <xsl:value-of select="x"> <xsl:value-of select="y"> </X> </xsl:template>

Implementation details

The refinements implemented in Vishnu are XSL 1.0 transformations. Selecting the version of XSL, both as a target language and for implementation of refinements was a major design decision in Vishnu. After careful consideration it was decided to use version 1.0 as a target language as this is more disseminated and easier for novice XSLT programmers. Using version 1.0 as target would difficult the use of version 2.0 for refinements. We would have to use two XSLT processors in the generation process, one for refining transformations and another for testing them. Since this would be an extra burden we preferred to use consistently a single language version in Vishnu, and selected version 1.0.
The main reason considering XSLT 2.0 was the use of features that otherwise would have to rely on extensions. Fortunately most of the features needed are available in standard extensions. Thus we used the Xalam XSLT processor with extensions for

string handling, function definition and basic elements and functions. The following code shows an example of a refinement: the Orphan simplification.

```
<!DOCTYPE xsl:stylesheet [
<!ENTITY xsl "http://www.w3.org/1999/XSL/Transform">
]>
<xsl:stylesheet version="1.0"
      xmlns:func=”http://exslt.org/functions”
      xmlns:vishnu="http://www.dcc.fc.up.pt/vishnu"
      xmlns:exslt="http://exslt.org/common"
    xmlns:str=”http://exslt.org/strings”
    xmlns:xsl="&xsl;">

  <xsl:import href="../common.xsl" />
  <xsl:output indent="yes" />

  <xsl:template match="xsl:template">
    <xsl:if test="not(
       count(xsl:apply-templates) = 1                and
       (
         (not(@mode) and not(xsl:apply-templates/@mode))
              or
         @mode = xsl:apply-templates/@mode
       )                                             and    count(*) = 1
                          and
       normalize-space(text())=''
    )">
       <xsl:copy-of select="."/>
 </xsl:if>
 </xsl:template>

</xsl:stylesheet>
```

A core feature in Vishnu is comparing two XML fragments. As result of the transformation process two XML fragments may result in different serializations but still be equivalent. To compare XML fragments Vishnu used the package XML Unit version 1.3. This package was developed for implementing by Tim Bacon and Stefan Bodewig to support unit testing in XML development. It provides a diff method to compare XML fragments and can be configured to ignore white space and differences between text and CDATA sections. This package was exposed as an extension function to refinements.

The common features we assembled in a XSLT library that is imported by most refinements. This library provides functions for XPath handling, such as finding common prefixes, and for recursively copying XML fragments while performing certain transformation, such as removing XPath prefixes. It also provides access to the extension functions mentioned previously.

In Doc 5 is presented an example of a particular refinement, the Orphan simplification introduced in sub-subsection 3.5.1 and with an example of its application in Table 5. . The root element of this second order transformation reveals all the extensions to XSLT 1.0 used. Among the top elements there is the import declaration for the common library described above. This single template in this second order transformation removes templates that contain a single apply-template in the same mode of the template, have no other elements besides the apply-templates, and have no content in text nodes but white space.

As can be seen in the Doc 5 example, refinements return the original transformation when they cannot be effectively used. Thus, the control of the refinement process is straightforward. The refinement process stops when no change is introduced to the transformation.

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.

Consistency and Coverage

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. After each refinement step the rewrite engine applies the current version of the transformation to the source example and compares the result with the target document. If this invariant is not satisfied then the refinement process is aborted and an error is reported to the client. This behaviour is the default in Vishnu. However, it can be switched off by the client to improve the efficiency of generator.
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.

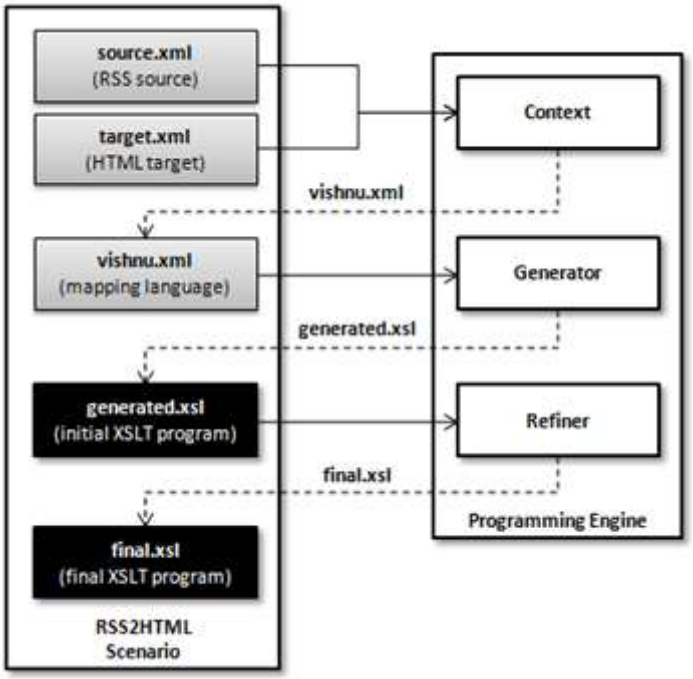

Figure 7. The RSS to HTML scenario.

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 **Erro! A origem da referência não foi encontrada.** shows the inner workflow used for testing the RSS to HTML scenario. A mixed-content scenario has not been added since the context component is not yet supporting indexes in text nodes.

Adequacy

Vishnu was conceived as an interactive tool integrated in Eclipse. Nevertheless, it was designed as two autonomous components: the editor and the engine. The editor is an Eclipse plug-in and concentrates all the tasks related with user interaction and integration with other Eclipse tools. The engine concentrates all the tasks related with the automatic creation of an XSLT program from examples using second order transformations. The communication between these two components is regulated by the Vishnu API.

By separating concerns in these two components we enable the non-interactive use of Vishnu. The engine has a command line interface to create XSLT programs from example files. Using Vishnu in this mode is as simple as executing the following command line.

$ java vishnu.jar source.xml target.xml > program.xsl

The Vishnu engine can also be invoked from other Java programs through the Vishnu API. This API may be used to create new user interfaces for Vishnu. For instance, a web interface based on the Google Web Toolkit (GWT) or a Swing based desktop interface. In general Vishnu may used by any application needing to create XSL transformations from examples. Java programs using the API must instantiate the engine using the static method Engine.getEngine() and use the following methods exposed by the Vishnu API:

`void setSource(Document source)`
Set source document example for the intended transformation

`Document getSource()`
Get given source document example for the intended transformation

`void setTarget(Document target)`
Set target document example for the intended transformation

`Document getTarget()`
Get given example of target document for the intended transformation

`void resetPairings()`
Reset all previously defined pairings

`void addPairing(String exprSource, String exprTarget)`
Add a pair of XPath location respectively on the source and target documents

`List<Pair> getPairings()`
Returns the list of pairings

`void inferPairings()`
Infer pairings from the given source and target documents

`Document program()`
Produce a XSLT program from the examples and their pairings

`Set<String> getFeatureNames()`
Return a list of names of features controlling the refinement process

`public boolean getFeature(String name)`
Get the given feature status

`public void setFeature(String name, boolean value)`

Set the given feature status

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.

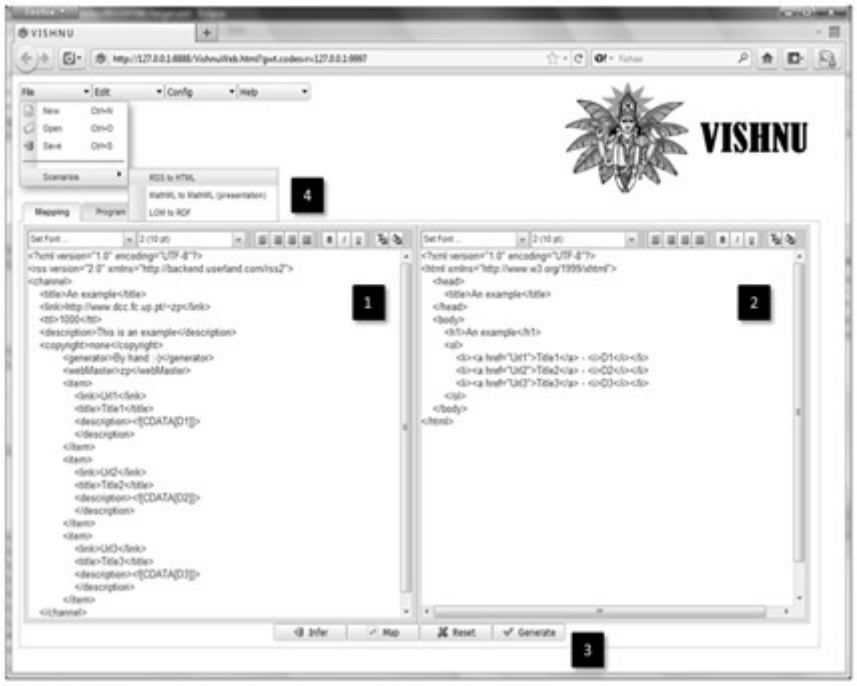

Figure 8. Vishnu client front-end.

Erro! A origem da referência não foi encontrada. 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.
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.

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. At this stage the generator has already been tested with different scenarios. It still lacks support for transformations with mixed content elements due to current limitations in its XPath locator module. Fixing this limitation is our immediate plans.

The project that lead to the development of Vishnu may 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.