

# Pipelines, Templates and Transformations: XML for Natural Language Generation

Graham Wilcock  
University of Helsinki  
00014 Helsinki, Finland  
graham.wilcock@helsinki.fi

## Abstract

The paper discusses a number of ways in which XML can be used in natural language generation, including XML-based pipeline architectures, template-based generation with XSL templates, and tree-to-tree transformations. The ideas are based on practical experience in building an experimental XML-based generation component for a spoken dialogue system. Prototype implementations using DOM, XSL and Translets are briefly compared.

## 1 Introduction

The paper discusses several ways in which XML can be used in work on natural language generation (NLG). The discussion is based on practical experience in developing an experimental XML-based generation component for a spoken dialogue system.

In this paper we are not proposing a new approach to generation. What we are proposing is the use of new XML-based tools to implement existing well-known approaches to NLG. The software tools which we use for parsing and transforming XML are open-source, Java-based, and freely available. While we wish the software to be as up-to-date as possible, we wish the basic design of the generator to be rather orthodox and straightforward.

The most common architecture for NLG is the *pipeline architecture*. In Section 2 we briefly describe the standard approach and suggest that it is natural to implement this

architecture in XML, using DOM trees as internal representations.

One of the most widely-known approaches is *template-based generation*, which has recently become more respected in the NLG community. In Section 3 we suggest that it is natural to implement NLG templates by means of XSL templates, and describe the form of template-based generation used in the prototype generator.

In Section 4 we describe the implementation of *tree-to-tree transformations* in the different stages of the pipeline. We suggest that, at least for relatively straightforward applications, the processing in the different stages of the pipeline can be performed by a sequence of XSL transformations.

In Section 5, we discuss whether XSL can be used as an all-purpose tool for building NLG systems. We then compare three different ways in which XML transformations can be implemented: (1) by XSL stylesheet processing using the TrAX standard, (2) by Java programs using the DOM document model, and (3) by using Translets (compiled XSL). We suggest that the third method combines the advantages of the other two methods.

### 1.1 Spoken Dialogues

Before describing the XML implementation, we illustrate the kind of responses generated by the system. The design of the generation component is described by Wilcock and Jokinen (2001). A more detailed description of response generation and its role within a spoken dialogue system is given by Jokinen and Wilcock (2001), whose examples and descriptions are partly repeated here.

In this approach, response planning starts from the *new information focus*, known as *NewInfo*. One of the tasks of the generator is to decide how to present the NewInfo to the user: whether it should be presented by itself or whether it should be *wrapped* in a link to the Topic.

(1) User: *Which bus goes to Malmi?*

System: *Number 74.*

(2) User: *How do I get to Malmi?*

System: *By bus - number 74.*

In example (1) NewInfo is the information about the bus number, while in (2) NewInfo concerns the means of transportation. In both cases, NewInfo is presented to the user by itself, without linking to the Topic.

(3) *When will the next bus leave for Malmi?*

(a) *2.20pm*

(b) *It will leave at 2.20pm*

(c) *The next bus to Malmi leaves at 2.20pm*

Whether NewInfo should be wrapped or not depends on the changing dialogue context. When the context permits a fluent exchange of contributions, wrapping is avoided and the response is based on NewInfo only, as in example (3a). When the context requires more clarity and explicitness, NewInfo is wrapped by Topic information as in (3b) in order to avoid misunderstanding. When the communication channel is working well, wrapping can be reduced, but when there are uncertainties about what was actually said, wrapping must be increased as in example (3c) to provide implicit confirmation. These examples are discussed in more detail by Jokinen and Wilcock (2001).

## 2 Pipelines

We now describe the use of pipelines in natural language generation, and how pipelines can be implemented using XML. Although it can be debated whether or not a pipeline is

the best architecture for generation, it is certainly well-known and the most widely-used (Reiter, 1994).

The textbook version of the NLG pipeline architecture for a text generation system is described by Reiter and Dale (2000), who list the modules and tasks required as:

- Document Planning
  - content determination
  - document structuring
- Microplanning
  - lexicalization
  - referring expression generation
  - aggregation
- Realization
  - linguistic realization
  - structure realization (e.g. HTML)

For a spoken dialogue system, however, all the pipeline stages described by Reiter and Dale for text generation systems are not required. The content determination stage has already been done by the dialogue manager. Document structuring is greatly simplified because, as neatly put by Santamarta (1999), “in a dialogue system the length of the generated text is rather short, namely a turn. A turn can consist of one word up to a couple of sentences.” The starting point for the generation component in the spoken dialogue system is therefore the specification of the utterance content which is determined by the dialogue manager. We describe how this is represented in XML in Section 2.1.

We will describe the subsequent stages of the pipeline in later sections, according to the different types of processing involved. The aggregation stage is described in Section 3.1 which is concerned with template-based generation. The lexicalization and referring expression generation stages are described in Section 4.1 which is concerned with tree-to-tree transformations. The possible requirement for embedded functions implemented in a general-purpose programming language,

for example to handle complex morphological generation, is discussed briefly in Section 4.2. The final structure realization stage, rendering the text in a speech markup language to be used by the speech synthesizer, is described in Section 4.3.

## 2.1 Input: an Agenda in XML

The starting point for the generation pipeline is the content determination which has been performed by the dialogue manager. This is specified in the form of an *agenda*, a set of concepts marked with Topic and NewInfo tags, the tags also being determined by the dialogue manager. Here we show how the agenda can be represented using XML.

In the above example (1) *Number 74*, the bus number information is supplied by the task manager, which consults the timetable database. The dialogue manager then puts the following concepts into the agenda.

```
<agenda>
  <concept info="Topic">
    <type>means-of-transportation</type>
    <value>bus</value>
  </concept>
  <concept info="Topic">
    <type>destination</type>
    <value>malmi</value>
  </concept>
  <concept info="Topic">
    <type>bus</type>
    <value>exists</value>
  </concept>
  <concept info="NewInfo">
    <type>busnumber</type>
    <value>74</value>
  </concept>
</agenda>
```

The root node of the XML tree is `<agenda>`, and its children are `<concept>` nodes, which here represent an unordered set. The dialogue manager labels each concept as NewInfo or Topic, using its knowledge of how the concepts relate to the current dialogue situation. These labels are represented in the XML agenda by attributes. Here, the concept 'busnumber' is labelled as NewInfo, and the other three concepts are labelled as Topic.

In the case of example (2) *By bus - number 74*, the dialogue manager specifies the means of transportation as NewInfo and the destination to Malmi as Topic. In addition, the

dialogue manager provides further new information about the bus number, following a cooperative dialogue strategy. The agenda is as follows:

```
<agenda>
  <concept info="NewInfo">
    <type>means-of-transportation</type>
    <value>bus</value>
  </concept>
  <concept info="Topic">
    <type>destination</type>
    <value>malmi</value>
  </concept>
  <concept info="NewInfo">
    <type>bus</type>
    <value>exists</value>
  </concept>
  <concept info="NewInfo">
    <type>busnumber</type>
    <value>74</value>
  </concept>
</agenda>
```

## 2.2 An XML Pipeline

The starting point for the response generator is the agenda of concepts specified in XML, as shown in the examples in Section 2.1. The finishing point is the linguistic utterance to be passed to the speech synthesizer, specified in a speech mark-up language which is also XML, as shown in the examples in Section 4.3. We are thus generating XML from XML, and we can use XML for all the internal representations along the stages of the pipeline.

In some stages we need to create a completely new XML tree. For example, in the aggregation stage described in Section 3.1 we create an aggregation tree as a new XML document. In other stages we simply add information to the existing XML tree, or replace some nodes with new nodes. For example, in the referring expression stage described in Section 4.1 we replace domain concepts with linguistic referring expressions, within the existing aggregation tree.

Independently of whether we need to create a new XML tree or simply add information to the existing XML tree, we can choose whether to use the DOM document model for explicit manipulation of the tree nodes, or to use XSL stylesheets to specify XML transformations. We have built prototype generators using both approaches, which we compare in Section 5.1.

In both cases the entire DOM tree can easily be held and manipulated in memory (there is no need to spend time writing it out to a file). The size of the DOM tree is small, as the utterance is only a few words or a couple of sentences.

To process the XML data using the DOM model explicitly, we use the Xerces XML parser (Apache XML Project, 2001b) with our own Java programs. Simple examples of how to “pipe” DOM trees between programs are included in the Xerces distribution.

Alternatively, to process the XML data using XSL stylesheets, we use the Xalan XSL processor (Apache XML Project, 2001a). With Xalan it is also easy to set up a sequence of transformations, in which the output of one transformation becomes the input to the next transformation.

In both cases, we believe that this kind of “piping” is a natural way to implement the pipeline architecture commonly used in natural language generation systems.

### 3 Templates

There has been considerable debate in the NLG community on the role of template-based generation (Becker and Busemann, 1999). Although it is one of the most widely-known approaches, it has been considered to be restricted to simple string manipulations with no linguistic knowledge. However, we do not take it to be restricted in this way.

Following van Deemter et al. (1999), we take *template-based* to mean “making extensive use of a mapping between semantic structures and representations of linguistic surface structures that contain gaps”. On this interpretation, templates can clearly include linguistic knowledge and the ways in which the gaps can be filled can be highly flexible.

In this section we suggest that it is natural to implement NLG templates (in the above sense) by means of XSL templates. This is especially appropriate at those points in the pipeline where new XML trees are created. We therefore describe the form of template-based generation used in the generator by looking at the aggregation stage.

### 3.1 Aggregation

Following the new information focus model of generation described in Section 1.1, the aggregation stage selects those concepts marked as *NewInfo* as the basis for generation, and also decides whether *NewInfo* will be the only output, or whether it will be preceded by the Topic linking concepts.

To initiate the change from the unordered set of concepts in the agenda to an ordered syntactic structure in the generated response, we use *aggregation templates* as a form of sentence plan specification. The aggregation templates are implemented by means of XSL named templates, as in the following simplified example:

```
<xsl:template name="NUM-DEST-TIME">
<aggregation>
  <tree><node>S</node>
    <tree><node>NP</node>
      <xsl:copy-of select="."
        /concept[type='busnumber']"/>
    </tree>
    <tree><node>V</node>
      <xsl:copy-of select="."
        /concept[type='bus']"/>
    </tree>
    <tree><node>PP</node>
      <xsl:copy-of select="."
        /concept[type='destination']"/>
    </tree>
    <tree><node>PP</node>
      <xsl:copy-of select="."
        /concept[type='bustime']"/>
    </tree>
  </tree>
</aggregation>
</xsl:template>
```

The selected aggregation template creates a new XML document instance, with root node `<aggregation>`. Its child nodes are one or more `<tree>` nodes, containing syntactic categories and other features. The trees contain variable slots, which will be filled in later by the lexicalization and referring expression stages. In the aggregation stage, the concepts from the agenda are copied directly into the appropriate slots by means of `<xsl:copy-of>` statements.

The selection of an appropriate aggregation template is based on which concept types are in the agenda and on their information status as *Topic* or *NewInfo*. The logic is implemented by means of nested `<xsl:choose>`

statements, as in the following example:

```
<!-- CHOOSE TEMPLATE BASED ON AGENDA -->
<xsl:template match="agenda">
<xsl:choose>
<xsl:when test="concept[@info='NewInfo']
    /type='means-of-transportation'">
    <xsl:call-template name="BY-TRANSPORT"/>
</xsl:when>
<xsl:when test="concept[@info='NewInfo']
    /type='bus'">
    <xsl:choose>
    <xsl:when test="concept[@info='NewInfo']
        /type='busnumber'">
        <xsl:call-template name="NUM-DEST-TIME"/>
    </xsl:when>
    ...
    </xsl:choose>
</xsl:when>
<xsl:when test="concept[@info='NewInfo']
    /type='busnumber'">
    <xsl:call-template name="NUMBER-ONLY"/>
</xsl:when>
...
</xsl:choose>
</xsl:template>
```

Here, if means-of-transportation is NewInfo as in example (2), the template named BY-TRANSPORT is selected. If means-of-transportation is not NewInfo, but bus and busnumber are both NewInfo, the template NUM-DEST-TIME is selected. If only busnumber is NewInfo as in example (1), the template NUMBER-ONLY is selected.

The aggregation templates are quite similar to the syntactic templates described by van Deemter et al. (1999), who argue that this approach resembles generation with Tree-Adjoining Grammars, and that the approach is fundamentally well-founded.

## 4 Transformations

At other stages of the pipeline it is not necessary to create a new XML tree, but further information needs to be added to the existing tree, or nodes in the tree need to be replaced by new nodes. Of course, this kind of tree-to-tree transformation is a basic technique in all forms of natural language processing, including generation.

Again, there are two ways in which this can be done using XML - either by explicit tree node manipulation using the DOM document model, or by specifying transformations in XSL stylesheets. We have implemented prototype generators using both approaches,

but suggest that these transformations can be most naturally expressed using XSL. This is done by combining XPath expressions to find the relevant part of the tree, and XSLT expressions to specify the transformations.

In the generation pipeline, the stage which is responsible for generating referring expressions (names, pronouns or descriptions) is one of the stages in which nodes of the existing tree must be replaced by new nodes. We therefore illustrate the XSL transformations using this stage.

### 4.1 Referring Expressions

In the stages of the generation pipeline which deal with referring expression generation and with lexicalization, the domain concepts in the aggregation templates are replaced by noun phrases for referring expressions and by other language-specific lexical items.

We illustrate how these stages can be implemented in XML by means of the following simplified examples of XSL templates. The basic idea is that concepts which are marked as Topic are realized as pronouns, whereas concepts which are marked as NewInfo are realized as full descriptions.

```
<!-- REFERRING EXPRESSIONS: PRONOUNS -->
<xsl:template
    match="concept[@info='Topic']"
    mode="referring-expression">
<xsl:choose>
<xsl:when test="type='busnumber'">
    <xsl:text> it </xsl:text>
</xsl:when>
<xsl:when test="type='destination'">
    <xsl:text> there </xsl:text>
</xsl:when>
<xsl:when test="type='bustime'">
    <xsl:text> then </xsl:text>
</xsl:when>
</xsl:choose>
</xsl:template>
```

Here, a destination concept which is marked as Topic is pronominalized as *there*. By contrast, if the same destination concept were marked as NewInfo, it could be realized as a full description by the following template, which generates a prepositional phrase with the preposition *to* followed by the actual text value of the destination placename, obtained by the `<xsl:value-of>` statement.

```

<!-- REFERRING EXPRESSIONS: DESCRIPTIONS -->
<xsl:template
  match="concept[@info='NewInfo']"
  mode="referring-expression">
  <xsl:choose>
  <xsl:when test="type='busnumber'">
    <xsl:text> number </xsl:text>
    <xsl:value-of select="value/text()"/>
  </xsl:when>
  <xsl:when test="type='destination'">
    <xsl:text> to </xsl:text>
    <xsl:value-of select="value/text()"/>
  </xsl:when>
  <xsl:when test="type='bustime'">
    <xsl:text> at </xsl:text>
    <xsl:value-of select="value/text()"/>
  </xsl:when>
  </xsl:choose>
</xsl:template>

```

Note that the above examples are simplified to show simple text output. In fact the generator performs further syntactic and morphological realization and produces output in a speech mark-up language, as described in Section 4.3.

## 4.2 Embedded functions

The Finnish language is famous for its highly complex morphology. A generator for Finnish must therefore be able to handle complex morphological processing as part of the linguistic realization stage. XSL would be unsuitable for this kind of processing, and in any case existing morphological generation software is available, which we wish to re-use.

In such situations, XSL can be combined with general purpose programming languages by embedding *extension functions* in XSL templates. These functions can be written in Java (Apache XML Project, 2001a).

## 4.3 Output: Speech Markup

The final stage of the textbook pipeline (Reiter and Dale, 2000) renders the generated text in a specific output presentation format such as HTML. This kind of transformation is of course the most well-known application of XSL, the task it was originally designed for.

In spoken dialogue response generation, the output must be in the specific format required by the speech synthesizer. In the prototype we use the Festival speech synthesis system (Black et al., 1999), which takes input marked up in SABLE (Sproat et al., 1998).

Festival also accepts Java Speech Markup Language (Sun Microsystems, Inc., 1997). JSML provides speech markup in VoiceXML prompts (VoiceXML Forum, 2000) and forms the basis of the W3C SSML working draft (World Wide Web Consortium, 2001). Note that JSML, SABLE and SSML are all XML. The brief response *Number 74* in example (1) is marked up in JSML as follows.

```

<?xml version="1.0"?>
<jhtml>
  <sent> Number
    <sayas class="number"> 74 </sayas>
  </sent>
</jhtml>

```

Here `<sent>` marks sentence boundaries. `<sayas>` tells the speech synthesizer how to say something - here it shows that 74 should be pronounced “seventy-four”. The response *By bus - number 74* in example (2) is marked up in JSML as follows.

```

<?xml version="1.0"?>
<jhtml>
  <sent> By
    <emp> bus </emp>
  </sent>
  <break size="large"/>
  <sent> Number
    <sayas class="number"> 74 </sayas>
  </sent>
</jhtml>

```

Here `<emp>` shows that the word *bus* should be spoken with emphasis, and `<break>` shows that a pause is required before the second part of the response.

## 5 Discussion: XSL or Java?

We have suggested that it is easy to set up a pipeline architecture with XSL, that it is easy to perform template-based generation with XSL, and that it is easy to perform tree-to-tree transformations with XSL. This clearly raises the wider question of whether XSL is really suitable as a general tool for building complete NLG systems.

This is discussed by Cawsey (2000), who concludes that relatively simple XSL transformations can be used for generation when the input is fairly constrained, but that XSL is not suitable for less constrained input, when we need to turn to general purpose programming languages or NLG tools.

XSL transformations are intended to be used to convert information content represented in XML into a desired presentation format, such as HTML. In such typical applications of XSL, there is usually no need for complex re-ordering of the content. Here however, the generator must convert the unordered set of concepts in the agenda into a syntactically correct ordered sequence of words.

Simple reordering can certainly be performed in XSL, for example by using XSL *modes*. We have experimented with applying XSL templates first with a Topic mode (if required), followed by a NewInfo mode. The usefulness of XSL modes for such purposes is noted by Cawsey (2000).

However, we suggest that even quite complex reordering can be performed in XSL, provided it is broken down into separate stages which can be organised on some principled basis. As van Deemter et al. (1999) argue, the syntactic template-based approach rather resembles generation with Tree-Adjoining Grammars, and should be considered fundamentally well-founded.

White and Caldwell (1999) compare their Java-based generation system EXEMPLARS with XSL and suggest that their system has advantages because it is more object-oriented. However, the practical problem with object-oriented systems and general purpose programming languages is of course that they require the participation of skilled programmers. One of the major advantages of XSL is that it is *not* a general-purpose programming language, but falls into the category of scripting languages which can be used by non-programmers.

However, XSL can be combined with general purpose programming languages when required, by embedding extension functions in the XSL templates. This means that even where general purpose programming languages are required for specific purposes, a pipeline of XSL transformations can still be used as a framework.

## 5.1 TrAX, DOM or Translets?

We have experimented with three different implementations of the design described here. In one implementation, the XML-to-XML transformations required at each stage of the pipeline are specified by means of XSL stylesheets, as illustrated in Sections 3.1 and 4.1. The XSL stylesheets are processed by the Xalan processor using the TrAX standard API. This means that the same unchanged stylesheets are parsed again, every time a dialogue response is generated. This implementation is therefore too slow for use in a real-time dialogue system.

In a second implementation, the XML-to-XML transformations at each stage of the pipeline are performed by Java programs which manipulate the XML structures using the DOM document model. This implementation using compiled Java code is therefore fast, and can be used in a real-time dialogue system. However, the programs require detailed navigation around the nodes in the document tree structures using the DOM API, and of course the use of Java requires the appropriate object-oriented programming skills, so one of the main advantages of using XML and XSL is lost.

The third implementation uses Translets.<sup>1</sup> Translets are compiled from XSL stylesheets into Java classes (bytecode). The translets, after they have been developed and tested, only need to be compiled once and can then be executed repeatedly, giving the same speed as other compiled Java classes.

This appears to combine the advantages of the other two approaches: on one hand, the XML transformations can be written in XSL stylesheets, thus avoiding the need for skilled Java programming and low-level node navigation, but on the other hand the resulting translets offer high speed because they are pre-compiled, avoiding the need for repeated stylesheet parsing.

<sup>1</sup>Translets were developed by Sun Microsystems Inc. in the XSLTC compiler. The XSLTC compiler is currently being integrated into Xalan (Apache XML Project, 2001a).

## 6 Conclusion

We have discussed several ways in which XML can be used for natural language generation. We described uses of XML in a pipeline architecture, in template-based generation, and in tree-to-tree transformations. We did not aim to propose a new approach to generation, merely some ways to use new XML-based tools to implement existing approaches to NLG.

In ongoing research, we aim to increase the level of adaptivity in the responses produced by the generator. Changing levels of confidence in speech recognition accuracy can then lead to responses with more or less explicitness, as illustrated by the range of responses (a) - (c) in example (3). In order to achieve a higher level of adaptivity, the framework in which the generator is embedded should be designed specifically to support adaptivity. The Jaspis adaptive speech applications framework (Turunen and Hakulinen, 2000) is such a system, which is itself XML-based.

We are currently developing an agent-based approach, with a set of different XML-based generation agents which can be used in the Jaspis framework. This work is described by Jokinen and Wilcock (2001).

## References

- Apache XML Project. 2001a. Xalan-Java (latest version). <http://xml.apache.org/xalan-j/>.
- Apache XML Project. 2001b. Xerces-Java (latest version). <http://xml.apache.org/xerces-j/>.
- Tilman Becker and Stephan Busemann, editors. 1999. *May I Speak Freely? Between Templates and Free Choice in Natural Language Generation*. Proceedings of the KI-99 Workshop. DFKI, Saarbrücken.
- Alan Black, Paul Taylor, and Richard Cayley. 1999. Festival Speech Synthesis System. <http://www.cstr.ed.ac.uk/projects/festival>.
- Alison Cawsey. 2000. Presenting tailored resource descriptions: Will XSLT do the job? In *9th International World Wide Web Conference*. <http://www9.org/w9cdrom/>.
- Kristiina Jokinen and Graham Wilcock. 2001. Confidence-based adaptivity in response generation for a spoken dialogue system. In *Proceedings of the 2nd SIGdial Workshop on Discourse and Dialogue*, pages 80–89, Aalborg, Denmark.
- Ehud Reiter and Robert Dale. 2000. *Building Natural Language Generation Systems*. Cambridge University Press.
- Ehud Reiter. 1994. Has a consensus NL generation architecture appeared, and is it psycholinguistically plausible? In *7th International Generation Workshop*, pages 163–170, Kennebunkport, Maine.
- Lena Santamarta. 1999. Output generation in a spoken dialogue system. In Becker and Busemann (1999), pages 58–63.
- R. Sproat, A. Hunt, M. Ostendorf, P. Taylor, A. Black, K. Lenzo, and M. Edgington. 1998. SABLE: A Standard for TTS Markup. <http://www.research.att.com/rws/SABPAP/sabpap.htm>.
- Sun Microsystems, Inc. 1997. Java Speech Markup Language Specification, version 0.5. <http://java.sun.com/products/java-media/speech/forDevelopers/JSML>.
- Markku Turunen and Jaakko Hakulinen. 2000. Jaspis - a framework for multilingual adaptive speech applications. In *Proceedings of 6th International Conference on Spoken Language Processing*, Beijing.
- Kees van Deemter, Emiel Krahmer, and Mariët Theune. 1999. Plan-based vs. template-based NLG: A false opposition? In Becker and Busemann (1999), pages 1–5.
- VoiceXML Forum. 2000. Voice eXtensible Markup Language VoiceXML, Version 1.00. <http://www.voicexml.org/spec.html>.
- Michael White and Ted Caldwell. 1999. Beyond XSL: Generating XML-annotated texts with EXEMPLARS. CoGenTex, Inc.
- Graham Wilcock and Kristiina Jokinen. 2001. Design of a generation component for a spoken dialogue system. In *Proceedings of the 6th Natural Language Processing Pacific Rim Symposium (NLPRS-2001)*, Tokyo.
- World Wide Web Consortium. 2001. Speech Synthesis Markup Language Specification for the Speech Interface Framework. <http://www.w3.org/TR/speech-synthesis>.