Beyond XSL:
Generating XML-Annotated Texts with EXEMPLARS

Michael White
Ted Caldwell
CoGenTex, Inc.
November 17, 1998

Abstract

In this paper, we present EXEMPLARS, a practical, object-oriented framework for generating XML-annotated texts, and compare its capabilities with XSL, an emerging W3C standard for transforming XML documents which can be used for simple text generation applications. We illustrate the framework's unique features—object-oriented specialization, ability to define higher-order exemplars, and ability to perform lookahead—and suggest that these features make EXEMPLARS better suited than XSL for developing many practical text generation applications.

1 Introduction

The EXEMPLARS framework is an object-oriented, rule-based tool designed to support practical, dynamic text generation. To date, the EXEMPLARS framework has been used to develop one commercial product (Project Reporter [CoGenTex 97, 99]) and two research prototypes (CogentHelp [Caldwell & White 97] and EMMA [McCullough et al. 98]) at CoGenTex, as well as one custom application, a natural language query tool for a large data warehousing company. In [White and Caldwell 98], the framework is compared to other hybrid natural language generation systems,[1] emphasizing its similarity to Reiter and Mellish's [92] classification-based approach (and, to a lesser degree, systemic approaches such as [Vander Linden & Martin 95]), as well as to the Command and Visitor object-oriented design patterns in [Gamma et al. 95]. In this paper, we compare EXEMPLARS with XSL (Extensible Stylesheet Language) [W3C 99a], an emerging World

Wide Web Consortium standard for transforming XML documents, which employs a similar processing model and can be used for simple text generation applications. In so doing, we illustrate the framework's unique features—object-oriented specialization, ability to define higher-order exemplars, and ability to perform lookahead—and suggest that these features make EXEMPLARS better suited than XSL for developing many practical text generation applications. We conclude with a discussion of how EXEMPLARS can be used to generate XML input to downstream NLG components.

```
<?xml version="1.0"?>
<xml>

<person id="heather" gender="female">
  <name>Heather</name>
  <department>Accounting</department>
  <meetings>
    <meeting ref="401k-switchover"/>
  </meetings>
</person>

<!-- ... etc. ... -->

<meeting id="new-water-cooler">
  <title>New Water Cooler</title>
  <date>November 18, 1999</date>
  <participants>
    <person ref="frank"/>
    <person ref="alice"/>
  </participants>
</meeting>

</xml>
```

Figure 1

2 Background: XSL

XSL (Extensible Stylesheet Language) [W3C 99a] is a stylesheet language for XML. It consists of two parts, namely XSLT [W3C 99b], which is a language for transforming XML documents into other XML documents, and an XML vocabulary for specifying formatting. XSL specifies the

[1] According to [Reiter 95], a *hybrid* natural language generation system is one that mixes template-based processing (a.k.a. 'mail merge') with more sophisticated techniques; cf. e.g. [Knott et al. 96, Milosavljevic et al. 96, Busemann & Horacek 98, Pianta & Tovena 99].

```
<?xml version="1.0"?>
<xsl:transform xmlns:xsl="http://www.w3.org/XSL/Transform/1.0">
  <xsl:output method="html"/>

  <xsl:template match="/">
    <html>
      <head><title>Who, What, When</title></head>
        <body bgcolor="white">
          <h2> People </h2>
          <hr size="1"/>
          <xsl:for-each select="xml/person">
            <xsl:apply-templates select="."/>
            <hr size="1"/>
          </xsl:for-each>
          <!-- ... etc. ... -->
        </body>
    </html>
  </xsl:template>

  <xsl:template match="person">
    <xsl:value-of select="name"/>
    <xsl:text> works in the </xsl:text>
    <xsl:value-of select="department"/>
    <xsl:text> department. </xsl:text>
    <xsl:apply-templates select="meetings"/>
  </xsl:template>

  <!-- ... etc. ... -->

</xsl:transform>
```

Figure 2

styling of an XML document by using XSLT to describe how the document is transformed into another XML document that uses the formatting vocabulary.

Here we will be concerned with XSLT, and also XPath [W3C 99c], a language for addressing parts of an XML document that is used by XSLT. For convenience, we will continue to use the term XSL, rather than the more specific term XSLT, when there seems to be little chance of confusion.

2.1 Combining template-based and data-driven processing

An XSL stylesheet primarily consists of a list of *template rules* that specify how to create an XML (or HTML) result document from an XML input document. The output specified by each template rule can combine recursive template-based processing—i.e., simple fill-in-the-blank (or 'mail merge') processing, augmented with recursive rule expansion—with more data-driven processing. With template-based processing, the structure of the output is determined primarily by the output

```
<xsl:template match="person/meetings[count(meeting) > 1]">
    <xsl:call-template name="mention-person-using-pronoun">
      <xsl:with-param name="person" select=".."/>
    </xsl:call-template>
    <xsl:text> is involved in </xsl:text>
    <xsl:value-of select="count(meeting)"/>
    <xsl:text> meetings, </xsl:text>
    <xsl:for-each select="meeting">
      <xsl:call-template name="mention-meeting-and-date">
        <xsl:with-param name="meeting" select="key('id-key',@ref)"/>
      </xsl:call-template>
      <xsl:if test="not(position()=last() or position()=last()-1)">
        <xsl:text>, </xsl:text>
      </xsl:if>
      <xsl:if test="position()=last()-1">
        <xsl:text> and </xsl:text>
      </xsl:if>
    </xsl:for-each>
    <xsl:text>. </xsl:text>
  </xsl:template>

  <xsl:template match="person/meetings[count(meeting) = 1]">
    <xsl:call-template name="mention-person-using-pronoun">
      <xsl:with-param name="person" select=".."/>
    </xsl:call-template>
    <xsl:text> is involved in only one meeting, </xsl:text>
    <xsl:call-template name="mention-meeting-and-date">
      <xsl:with-param name="meeting" select="key('id-key',meeting/@ref)"/>
    </xsl:call-template>
    <xsl:text>. </xsl:text>
  </xsl:template>

  <xsl:template name="mention-person-using-pronoun">
    <xsl:param name="person"/>
    <xsl:choose>
      <xsl:when test="$person/@gender='male'">
        <xsl:text>He</xsl:text>
      </xsl:when>
      <xsl:otherwise>
        <xsl:text>She</xsl:text>
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>
```

Figure 3

template; with data-driven processing, properties of the input data primarily determine the output structure.

For example, consider the XML input data shown in Figure 1. This data can be converted into an HTML page via the transform[2] partially shown in Figure 2. The resulting page will contain a section entitled "People", which contains a short description of each person in the input data together with a list of his or her meetings. One such entry is "Heather works in the Accounting department. She is involved in only one meeting, 401k Switchover (November 15, 1999)." This

[2] In XSL, the keywords *stylesheet* and *transform* are interchangeable.

text is generated via the second template rule[3] in Figure 2, which matches on `person` elements. Using a simple fill-in-the-blank approach, this template rule first outputs the sentence "<name> works in the <department> department." It then continues recursive processing of the subordinate meetings element (via rules not yet shown). Now let us consider the first, top-level template rule in Figure 2, whose match condition selects the root of the input document. In addition to setting up the output HTML page, this rule recursively processes each `person` element under the `xml` root element. This rule demonstrates how XSL's template-based processing can be used to generate output with new element structures, as well as how its `xsl:for-each` construct can be used for data-driven processing.

2.2 Rule selection and conditional processing in XSL

XSL includes further support for data-driven processing through its rule selection mechanism and its conditional processing constructs. To illustrate, consider the template rules shown in Figure 3. The first two template rules provide variants for describing a person's meetings (in the context of the rules shown in Figure 2) depending on whether the person is listed as attending exactly one or more than one meeting. Earlier we saw an example involving the second template rule; an example involving the first rule is as follows: "Frank works in the Shipping department. He is involved in 2 meetings, New Water Cooler (November 18, 1999) and Product Design (November 22, 1999)."

An XSL transformation engine is responsible for selecting the template rule to use in a given context; if there is more than one rule whose matching conditions are met, a rather complex priority scheme is invoked to choose a single rule to apply (see [W3C 99b] for details). It is also possible to call another template rule by name; this is useful when variants have part of their output in common, as shown in the calls to the template named `mention-person-using-pronoun` at the beginning of the first and second template rules in Figure 3. With named templates, the name must be unique

[3] Note that the use of an `xsl:text` element is optional, but is useful for controlling white space in the result tree, especially around punctuation.

within the stylesheet, so the rule selection mechanism is much simpler (it is however possible to override a template rule with the same name in an imported stylesheet). Finally, Figure 3 also shows XSL's conditional processing constructs, `xsl:if` and `xsl:choose`.

3 Object-Oriented Specialization in Exemplars

According to [W3C 99b], "XSLT is also designed to be used independently of XSL. However, XSLT is not intended as a completely general-purpose XML transformation language. Rather it is designed primarily for the kinds of transformations that are needed when XSLT is used as part of XSL." Given that XSLT is not intended as a completely general-purpose XML transformation language, an interesting question arises as to what extent it can be suitably employed in natural

```
exemplar DescribeEverything( Data data )
{
  void apply()
  {
    <<+
      <html>
        <head><title> Who, What, When </title></head>
        <body bgcolor="white">
          <h2> People </h2>
          <hr size=1>
          <> :: PeopleSite
          // … etc. …
        </body>
      </html>
    +>>
    for (int i = 0; i < data.people.length; i++) {
      PeopleSite <<+
        {{ DescribePerson(data.people[i]) }}
        <hr size=1>
      +>>
    }
    // … etc. …
  }
}

exemplar DescribePerson( Person person )
{
  void apply()
  {
    <<+
      { person.name } works in the { person.department } department.
      {{ ListMeetings(person) }}
    +>>
  }
}
```

Figure 4

```
exemplar ListMeetings( Person person )
{
  void apply() {
    <<+
      { person.isMale ? "He" : "She" } is involved in
      { person.meetings.length } meetings,
      <> :: ListSite
      <> :: PostListSite
      .
    +>>
    // do each item, adding commas and "and", as appropriate ...
    for (int i = 0; i < person.meetings.length; i++) { /* ... */ }
    // call method for post-list phrase
    doPostListPhrase();
  }
  // default has no post-list phrase
  void doPostListPhrase() {}
}

exemplar ListMeetingsSingleton( Person person ) extends ListMeetings
{
  boolean evalConstraints() { return (person.meetings.length == 1); }
  void apply() { /* ... */ } // override entire output
}

exemplar ListMeetingsAllInPast( Person person ) extends ListMeetings
{
  // check for at least 2 meetings, all in past
  boolean evalConstraints() { /* ... */ }
  // add generalization in post-list phrase
  void doPostListPhrase() {
    PostListSite <<+
      , { (person.meetings.length != 2) ? "all" : "both" }
      of which have already taken place
    +>>
  }
}
```

Figure 5

language generation. This question has been investigated by Cawsey [99] in the context of the Mirador project [Heriot Watt 99], who found that "though we can come up with various useful tricks to make this tractable when the input is fairly constrained (e.g., using modes; creating intermediate tree structures), XSLT is not suitable for less constrained input, when we need to turn to general purpose programming languages or natural language generation tools."

While there is not space to detail Cawsey's reasoning here, her conclusions should perhaps come as no surprise, given XSLT's emphasis; in any case, her conclusions naturally lead us to the topic of the rest of this paper, namely the ways in which the EXEMPLARS framework goes beyond XSL in supporting text generation.

3.1 Exemplars overview

The EXEMPLARS framework is quite similar to XSL in the basic support it provides for combining recursive template-based processing with more data-driven processing, except that it requires Java objects as input, rather than XML data.[4] In this framework, the term *exemplar* is used as a fancy name for rules similar to XSL's template rules; exemplars are so-called because they are meant to capture an exemplary way of achieving a communicative goal in a given communicative context, as determined by the system designer.

Figure 4 shows the exemplars that correspond to the XSL template rules of Figure 2. Exemplars are defined using a superset of Java including (1) exemplar signatures and (2) output statements for adding to the output tree in template-like fashion. Exemplar signatures are like Java class signatures, except that exemplars take arguments, like Java methods. Within output statements, i.e. statements delimited by `<<+ … +>>`, new elements can appear, along with string substitution expressions, delimited by single curly braces, and calls to other exemplars, delimited by double curly braces. Output statements may also contain labels for element nodes, or labels for *container nodes* that mark arbitrary locations in the output; in the `DescribeEverything` exemplar, `PeopleSite` is a label for the container node appearing under the People section heading. Output statements normally add their output to the context of the current exemplar in the output tree, though it is also possible to provide an explicit output site; within the for-loop, the output is explicitly directed to the location given by `PeopleSite`.

The EXEMPLARS framework consists primarily of a tool to compile an exemplars source file from its Java superset down to pure Java, plus a run-time *text planner* component which manages exemplar invocations. An exemplars source file normally contains a set of related exemplars, each of which is compiled into a separate Java class. The compilation process consists of (1) translating the signature into a set of constructors and methods that implement it; (2) translating any output

statements to API calls, including any embedded simple substitutions or exemplar calls; and (3) passing through any Java code in between. This approach makes it possible to reuse basic Java constructs and Java's inheritance mechanism, exceptions, threads, etc., as well as to directly and efficiently integrate with other application objects.

3.2 Augmenting or overriding default exemplars

A central idea behind the EXEMPLARS framework is to enable the designer to arrange exemplars into a specialization hierarchy, where more specialized exemplars can augment or override the more general ones they specialize. That is, in contrast to XSL, with EXEMPLARS the designer explicitly defines rule specialization; as we shall see below, explicit specialization is a crucial ingredient in providing the designer with fine-grained control over the interaction among specialized and default exemplars.

To illustrate, Figure 5 shows an exemplar named `ListMeetings`, along with two specializations (indicated by the keyword `extends`), `ListMeetingsSingleton` and `ListMeetingsAllInPast`. `ListMeetings` is the default exemplar for listing a person's meetings, and produces output like the first XSL template rule in Figure 3. `ListMeetingsSingleton` is a specialization analogous to the second XSL template rule in Figure 3; by redefining the `apply` method, it overrides the entire action of `ListMeetings`. Finally, `ListMeetingsAllInPast` is a specialization with no XSL analogue; it selectively overrides the action of `ListMeetings` by just redefining the `doPostListPhrase` method, in order to augment the output with the indicated phrase. An example output of this rule is "She is involved in three meetings, 401K Switchover (today), Technical Review (last Saturday, August 21), and Picnic Planning (the same day), all of which have already taken place."[5]

---

[4] Of course, XML data can be easily accessed in Java via XML APIs, or by first reading XML data into Java application objects.

[5] See [White & Caldwell 98] for a description of the exemplars used to produce context-sensitive date descriptions. Regarding the choice of present tense, we assume this sentence will be read as "She is involved in three meetings [of those listed in the database], …"

```
exemplar ListMeetings( Person person ) implements MakesListPhrases
{
  // make a list using the phrases defined below
  void apply()
  {
    <<+ {{ MakeList(this) }} +>>
  }

  public int getListSize() { return person.meetings.length; }

  public void doPreListPhrase( SGMLNode site )
  {
    site <<+
      { person.isMale ? "He" : "She" } is involved in
      {{ IdentifyNumber(person.meetings.length) }} meetings
    +>>
  }

  public void doListElementDescription( int i, SGMLNode site ) { /* … */ }

  // default has no post-list phrase
  public void doPostListPhrase( SGMLNode site ) {}

  public void doEmptyListDescription( SGMLNode site ) { /* … */ }

  public void doSingletonListDescription( SGMLNode site )
  {
    site <<+
      { person.isMale ? "He" : "She" }
      is involved in only one meeting,
      {{ MentionMeetingAndDate(person.meetings[0]) }}.
    +>>
  }
}
```

Figure 6

At run-time, the text planner selects the most specific applicable exemplar by traversing the specialization hierarchy top-down and left-to-right,[6] starting with the given named exemplar, evaluating applicability as it goes; the traversal is similar to that of a decision tree, insofar as applicability conditions are implicitly conjoined, except that the search may end at a non-leaf node in the hierarchy. To determine whether a particular exemplar is applicable, the text planner first checks the type constraints on the exemplar's arguments, then evaluates any explicitly defined conditions. Once the most specific applicable exemplar is found, its action is invoked, which may lead to recursive calls to the text planner if the exemplar's output contains calls to other exemplars.

[6] Left-to-right order is determined by the order in which exemplar definitions appear in the source file.

Since type constraints are checked automatically, it is easy to set up exemplar specialization hierarchies that mirror the type hierarchy of the input objects (cf. [White & Caldwell 98]); this is not possible in XSL. This capability has been of significant benefit in two of the four systems built using EXEMPLARS to-date.

4 Higher-Order Text Planning

Because text generators written with exemplars are often data-driven and often geared towards a particular presentation style or modality (for example, hypertext vs. plain text), exemplars tend not to be very reusable. However, in practice we have found several types of text that crop up repeatedly in different applications, and which we would like to be able to handle using some sort of generic exemplar libraries. A good example of this is lists—for example, "Alice is involved in three meetings, *401K Switchover, Technical Review, and Picnic Planning.*" Although the kinds of items described in such lists (meetings, in this case) will vary across applications, there is a standard way of writing them: in a comma-delimited form if the list is fairly short, and using bullets if it is beyond a certain length.

Generic rules of this type can be implemented as general-purpose exemplars, which make no assumptions about particular types of input data. For example, we can define a general-purpose `MakeList` exemplar, which takes an application-specific "list-describing" exemplar as its argument. `MakeList` handles general steps like deciding whether to use bullets, inserting punctuation, etc., and it calls back to the specific exemplar to generate the pre-list and/or post-list phrases, as well as the text for each item in the list. This type of processing is not possible in XSL, since template rules

```
ListMeetings                    MakeList
  ListMeetingsAllInPast           MakeEmptyList
  ListMeetingsNoneInPast          MakeSingletonList
                                  MakeVerticalList
                                    MakeNumberedList
```

Figure 7

cannot be passed as arguments to other template rules.

To implement general-purpose exemplars, we use interfaces, another object-oriented feature of Java. The `MakeList` exemplar is accompanied by a `MakesListPhrases` interface, which specifies the methods that an application-specific exemplar must implement in order to pass itself as an argument to `MakeList`. Figure 6 shows how the `ListMeetings` exemplar from Figure 5 would be simplified (now incorporating the functionality of `ListMeetingsSingleton`) by having it implement this interface.[7]

One advantage of this approach to higher-order text planning is that the designer can make use of independent specialization hierarchies in specifying different interacting collections of exemplars. This alleviates the combinatorial explosion that can occur when one tries to represent several dimensions of textual variation using a single specialization hierarchy—for example, if the designer wants to specify different texts depending on whether all meetings are in the past or future, as well as according to the number of meetings being described. Figure 7 shows the hierarchies below the `MakeList` and `ListMeetings` exemplars.

In this example, a further dimension for variation is provided by the `MakeListOptions` class, an instance of which can optionally be passed to `MakeList` in order to specify, say, the threshold list length at which to start using bullets, or whether to use numbers instead of bullets in vertical lists.

5 Lookahead

When designing a set of recursive templates to generate text, and especially when trying to make the templates flexible and general-purpose, it is often useful to be able to find out what interacting templates are up to at runtime, using techniques such as lookahead. The Exemplars framework makes lookahead possible, because of the semantics of calls between exemplars—the output

[7] A future version of the exemplars framework will use standard XML APIs; the current version uses our own SGML trees, which are converted to XML trees in a post-processing step.

```
exemplar MakeList( MakesListPhrases mle,
                   MakeListOptions opts = new MakeListOptions() )
{
  int size = mle.getListSize();

  void apply() {
    // setup sites for parts
    <<+
      <> :: PreListSite
      <> :: ListSite
      <> :: PostListSite
    +>>

    // do each part
    mle.doPreListPhrase(PreListSite);
    doPreListSeparator();
    doListBody();
    doPostListSeparator();
    mle.doPostListPhrase(PostListSite);
    doFullStop();
  }

  /* … */

  // add period after post-list phrase
  void doFullStop() { PostListSite <<+ . +>> }
}

exemplar MakeVerticalList( MakesListPhrases mle, MakeListOptions opts )
  extends MakeList
{
  boolean evalConstraints() { return opts.useVerticalList(mle); }

  // separate pre-list phrase from list body with a colon
  void doPreListSeparator() { PreListSite <<+ : +>> }

  // do each item in a vertical list, terminated with a semicolon
  void doListBody() { /* … */ }

  // add period after post-list phrase, unless it's empty;
  // otherwise add it at the end of the last list item
  void doFullStop() {
    if (!PostListSite.isEmpty()) PostListSite <<+ . +>>
    else ListItemSite <<+ . +>>
  }
}
```

Figure 8

structure defined by each such call is created at the time of the call, so that it can be examined by the calling exemplar before execution leaves the scope of the calling method.

Returning to our example for illustration, the `MakeVerticalList` exemplar (Figure 8) overrides the `doFullStop` method in `MakeList`, in order to place the final period in the list sentence in different

places, according to whether a post-list phrase is generated—if there is a post-list phrase, the period goes after it; otherwise, the period goes after the last item in the vertical list.[8]

This type of lookahead operation is not possible in XSL 1.0, as it does not allow tests to be performed on the output tree. Note, however, that this difference is perhaps less fundamental than the ones discussed in the previous sections, since such capabilities are under consideration for future versions of XSL.

6 Conclusions and Future Work

In this paper, we have illustrated the EXEMPLARS framework's unique features—object-oriented specialization, the ability to define higher-order exemplars, and the ability to perform lookahead—and have suggested that these features make it better suited than XSL for developing many practical text generation applications. To date, the framework has been used in two commercial applications and two research prototypes that directly generate HTML. However, EXEMPLARS is not limited to generating HTML; it can be used just as easily to generate any kind of structure that can be straightforwardly encoded in XML. In future work, we plan to develop applications that use XML to integrate with downstream NLG components. Under this scenario, the framework is used in the initial, text planning phase to generate XML representations of discourse and sentential structure from the input application objects, at a possibly varying level of detail appropriate for the application. Downstream components then further process these representations in the sentence planning and surface realization phases. As a first step in this direction, we are currently working on a new, more flexible XML-based integration with RealPro [Lavoie & Rambow 97], CoGenTex's syntactic realizer, as well as a revisions module that uses XSL for matching.

[8] The framework includes a punctuation module that fixes up certain combinations of adjacent punctuation, such as a semi-colon appearing immediately adjacent to a period (cf. [Nunberg 90, White 95]). This module does not handle this case, however, as it will not move a period appearing outside of a list element into the last list item.

References

[Busemann & Horacek 98] Busemann, S., and H. Horacek. 1998. A Flexible Shallow Approach to Text Generation. In *Proceedings of the Ninth International Workshop on Natural Language Generation*, Niagara-on-the-Lake, Canada, pp. 238–247.

[Caldwell & White 97] Caldwell, D. E., and M. White. 1997. CogentHelp: A tool for authoring dynamically generated help for Java GUIs. In *Proceedings of the 15th Annual International Conference on Computer Documentation (SIGDOC '97)*, pp. 17–22, Salt Lake City, UT.

[Cawsey 99] Cawsey, A. 1999. Presenting tailored resource descriptions: Will XSLT do the job? Manuscript.

[CoGenTex 97] CoGenTex, Inc. 1997. Text Generation Technology for Advanced Software Engineering Environments. Final Report, Rome Laboratory Contract No. F30602-92-C-0163.

[CoGenTex 99] CoGenTex, Inc. 1997. Project Reporter. http://www.cogentex.com/products/reporter/.

[Gamma et al. 95] Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA.

[Heriot Watt 99] Heriot Watt University. 1999. The Mirador Project. http://www.cee.hw.ac.uk/~mirador/.

[Knott et al. 96] Knott, A., C. Mellish, J. Oberlander, and M. O'Donnell. 1996. Sources of Flexibility in Dynamic Hypertext Generation. In *Proceedings of the Eighth International Natural Language Generation Workshop (INLG '96)*, pp. 151–160, Herstmonceux Castle, Sussex, UK.

[Lavoie & Rambow 97] Lavoie, B., and O. Rambow. 1997. A Fast and Portable Realizer for Text Generation Systems. In *Proceedings of the Fifth Conference on Applied Natural Language Processing (ANLP '97)*, pp. 265–268, Washington, D.C.

[McCullough et al. 98] McCullough, D., T. Korelsky, and M. White. 1998. Information Management for Release-based Software Evolution Using EMMA. In *Proceedings of the Tenth International Conference on Software Engineering and Knowledge Engineering (SEKE '98)*, San Francisco Bay, CA.

[Milosavljevic et al. 96] Milosavljevic, M., A. Tulloch, and R. Dale. 1996. Text generation in a dynamic hypertext environment. In *Proceedings of the 19th Australasian Computer Science Conference*, pp. 229–238, Melbourne, Australia.

[Nunberg 90] Nunberg, G. 1990. *The Linguistics of Punctuation.* CSLI Lecture Notes No. 18, University of Chicago Press.

[Pianta & Tovena 99] Pianta, E., and L. M. Tovena. 1999. Mixing representation levels: The hybrid approach to automatic text generation. In *Proceedings of the AISB'99 Workshop on "Reference Architectures and Data Standards for NLP,"* pp. 8–13, Edinburgh, Scotland.

[Reiter & Mellish 92] Reiter, E., and C. Mellish. 1992. Using classification to generate text. In *Proceedings of the 30th Annual Meeting of the Association for Computational Linguistics (ACL '93)*, pp. 265–272, Newark, Delaware.

[Reiter 95] Reiter, E. 1995. NLG vs. Templates. In *Proceedings of the Fifth European Workshop on Natural Language Generation (EWNLG '95)*, Leiden, The Netherlands.

[Vander Linden & Martin 95] Vander Linden, K., and J. H. Martin. 1995. Expressing Rhetorical Relations in Instructional Text: A Case Study of the Purpose Relation. *Computational Linguistics*, vol. 21, no. 1, pp. 29–58.

[W3C 99a] World Wide Web Consortium. 1999. Extensible Stylesheet Language (XSL). http://www.w3.org/Style/XSL/.

[W3C 99b] World Wide Web Consortium. 1999. XSL Transformations (XSLT) Version 1.0. http://www.w3.org/TR/xslt.

[W3C 99c] World Wide Web Consortium. 1999. XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath.

[White 95] White, M. 1995. Presenting Punctuation. In *Proceedings of the Fifth European Workshop on Natural Language Generation (EWNLG '95)*, Leiden, The Netherlands.

[White and Caldwell 98] White, M. and T. Caldwell. 1998. EXEMPLARS: A Practical, Extensible Framework for Dynamic Text Generation. In *Proceedings of the Ninth International Workshop on Natural Language Generation*, Niagara-on-the-Lake, Canada, pp. 266-275. Also available from http://www.cogentex.com/papers/.