Techniques for Text Planning with XSLT

Mary Ellen Foster and **Michael White**
Institute for Communicating and Collaborative Systems
School of Informatics, University of Edinburgh
Edinburgh EH8 9LW
{mef,mwhite}@inf.ed.ac.uk

Abstract

We describe an approach to text planning that uses the XSLT template-processing engine to create logical forms for an external surface realizer. Using a realizer that can process logical forms with embedded alternatives provides a substitute for backtracking in the text-planning process. This allows the text planner to combine the strengths of the AI-planning and template-based traditions in natural language generation.

1 Introduction

In the traditional pipeline view of natural language generation (Reiter and Dale, 2000), many steps involve converting between increasingly specific tree representations. As Wilcock (2001) points out, this sort of tree-to-tree transformation is a task to which XML—and particularly XSLT template processing—is particularly suited.

In this paper, we describe how we plan text by treating the XSLT processor as a top-down rule-expanding planner that translates dialogue-manager specifications into logical forms to be sent to the OpenCCG text realizer (White and Baldridge, 2003; White, 2004a; White, 2004b). XSLT is used to perform many text-planning tasks, including structuring and aggregating the content, performing lexical choice via the selection of logical-form templates, and generating multiple alternative realizations for messages where possible.

Using an external realizer at the end of the planning process provides two advantages. First, we can use the realizer to deal with those aspects of surface realization that are difficult to implement in XSLT, but that the realizer is designed to handle (e.g., syntactic agreement via unification). Second, we take advantage of OpenCCG's use of statistical language models by sending multiple alternative logical forms to the realizer, and having it make the final choice of surface form. Allowing the text planner to produce multiple alternatives also obviates the need for backtracking, which is not something that is otherwise easily incorporated into the a system based on XSLT processing.

We have implemented this approach in two dialogue systems. In this paper, we concentrate on how text is planned in the COMIC multimodal dialogue system (den Os and Boves, 2003). Similar techniques are also used in the FLIGHTS spoken-dialogue system (Moore et al., 2004), which generates user-tailored descriptions and comparisons of flight itineraries.

The rest of this paper is organized as follows: Section 2 gives an overview of the COMIC dialogue system and the OpenCCG text realizer. Section 3 then shows how the COMIC text planner generates logical forms for the realizer from high-level dialogue-manager specifications. Section 4 describes how the interface between the text planner and the realizer allows us to send multiple alternative logical forms, and shows the advantages of this approach. Section 5 discusses related work, while Section 6 outlines the future plans for this work and gives some conclusions.

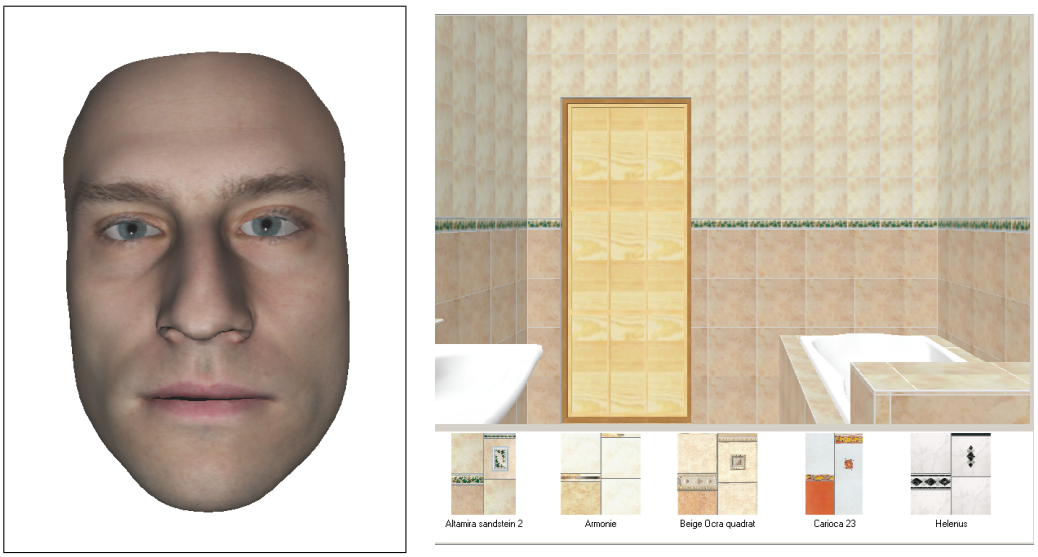
2 Systems

2.1 COMIC

COMIC[1] (den Os and Boves, 2003) is an ongoing project investigating multimodal dialogue systems. The demonstrator adds a dialogue interface to a CAD-like application used in bathroom sales situations to help clients redesign their rooms. The input to the system includes speech, handwriting, and pen gestures; the output combines synthesized speech, a "talking head" avatar, and control of the underlying application. Figure 1 shows screen shots of the avatar and the bathroom-design application.

COMIC produces a variety of output, using its full range of modalities. In this paper, we will concentrate on the textual content of those turns in which the system describes one or more options for

[1] COnversational Multimodal Interaction with Computers; http://www.hcrc.ed.ac.uk/comic/.



(a) Avatar (b) Bathroom-design application

Figure 1: Components of the COMIC demonstrator

decorating the user's bathroom, as in the following description of a set of tiles:

(1) *Here is a country design. It uses tiles from Coem's Armonie series. The tiles are terracotta and beige, giving the room the feeling of a Tuscan country home. There are floral motifs on the decorative tiles.*

2.2 OpenCCG

The OpenCCG realizer (White and Baldridge, 2003) is a practical, open-source realizer based on Combinatory Categorial Grammar (CCG; Steedman, 2000). It employs a novel ensemble of methods for improving the efficiency of CCG realization, and in particular, makes integrated use of *n*-gram scoring of possible realizations in its chart realization algorithm (White, 2004a; White, 2004b). The *n*-gram scoring allows the realizer to work in "anytime" mode—able at any time to return the highest-scoring complete realization—and ensures that a good realization can be found reasonably quickly even when the number of possibilities is exponential.

Like other realizers, the OpenCCG realizer is partially responsible for determining word order and inflection. For example, the realizer determines that *also* should preferably follow the verb in *There are also floral motifs on the decorative tiles*, whereas in other cases it typically precedes the verb, as in *It also has abstract shapes*. It also enforces subject-verb agreement, e.g., between *are* and *motifs*, and *it* and *has*, respectively. Less typically, in COMIC and FLIGHTS, the OpenCCG realizer additionally determines the type of pitch accents, and the type and placement of boundary tones, based on the information structure of its input logical forms.

3 Text Planning in COMIC

Broadly speaking, text planning in COMIC follows the standard pipeline model of natural language generation (Reiter and Dale, 2000). The input to the COMIC text planner, from the dialogue manager, specifies the content of the description at a high level; the output consists of logical forms for the OpenCCG realizer.

The module is implemented in Java and uses Apache Xalan[2] to process the XSLT templates. The initial implementation of the presentation-planning module—of which the XSLT-based sentence planner described here is just a part—took approximately one month. After that, the module was debugged and updated incrementally over a period of several months, during which time additional templates were created to support updates in the OpenCCG grammar. The development process was made easier by the ability to use OpenCCG to parse a target sentence, and then base a template on the resulting logical form.

The current presentation planner uses 14 templates for content structuring and aggregation (Section 3.2), and just over 100 to build the logical forms (Section 3.3). The tasks described here take little time to perform (i.e., hundreds of milliseconds);

[2] `http://xml.apache.org/xalan-j/`

```
<rdf:Description rdf:about="#Tileset9">
  <rdf:type>
    <daml:Class rdf:about="#Tileset"/>
  </rdf:type>
  <comic:has_id>
    <xsd:string xsd:value="9"/>
  </comic:has_id>
  <comic:has_commentary
    rdf:resource="#Commentary9"/>
  <comic:has_decoration>
    <xsd:string xsd:value="floral-motifs"/>
  </comic:has_decoration>
  <comic:has_series>
    <xsd:string xsd:value="Armonie"/>
  </comic:has_series>
  <comic:has_manufacturer>
    <xsd:string xsd:value="Coem"/>
  </comic:has_manufacturer>
  <comic:has_colour rdf:resource="#Terracotta"/>
  <comic:has_colour rdf:resource="#Beige"/>
  <comic:has_style rdf:resource="#Country"/>
</rdf:Description>
```

Figure 2: Ontology properties of tileset 9

```
<object type="describe">
  <slot name="has_object">
    <object type="Tileset">
      <slot name="has_id">
        <value type="String">9</value>
      </slot>
    </object>
  </slot>
  <slot name="has_feature">
    <value type="String">has_colour</value>
  </slot>
</object>
```

Figure 3: Dialogue-manager specification

most of the module's time is spent communicating with other modules in the system.

3.1 Content Selection

The features of the available designs are stored in the system ontology. This is represented in DAML+OIL (soon to be OWL) and includes tile properties such as style, colour, and decoration. There is also canned-text commentary associated with some features (e.g., the *Tuscan country home* text in (1)). The ontology instance corresponding to design ("tileset") 9 is shown in Figure 2.

For a description like (1), the dialogue-manager specifies only the tileset to be described, and optionally a set of features to include in the description. Figure 3 shows a dialogue-manager message[3] indicating that tileset 9 should be described, and that the description must include the colour.

To select the content of the description, we first retrieve all of the features of the indicated design from the ontology, using the Jena semantic web framework.[4] We then use the system dialogue history to filter the retrieved features by removing any that have already been described to the user. Finally, we add back to the set any features specifically requested by the dialogue manager, even if they have been included in a previous description.

3.2 Content Structuring

The result of content selection is an unordered set of tileset features; this set is converted into a text plan as follows. First, for each selected feature, a message is created in XML that combines the information gathered from the ontology with information from the system dialogue history. Figure 4 shows the messages corresponding to the colour feature and to the associated canned-text commentary. The dialogue-history information is included in the *same-as-last* (i.e., whether this value is the same as the corresponding value of the previous tileset) and *already-said* attributes.

The unordered set of messages is converted to an ordered list using a small number of heuristics: for example, features requested by the dialogue manager are always put at the start of the list, while canned-text commentary always goes immediately after the feature to which it refers. These heuristics provide a partial ordering, which is then converted to a total ordering by breaking ties at random.

The next step is to aggregate the flat list of messages. In many NLG systems, aggregation is a task that is done at the syntactic level; in COMIC, we instead work at the conceptual level. Thanks to the fact that we produce multiple alternative syntactic structures (see Section 4), we can be confident that, whatever the final set of messages, there will be some syntactic structure available to realize them.

The aggregation is done using a set of XSLT templates that combine adjacent messages based on various criteria. For example, the template shown in Figure 5 combines a feature-value message with the associated canned-text commentary.[5] Figure 6 shows the combined message that results when the messages in Figure 4 are processed by this template.

The sentence boundaries in the final text are determined by the content structure: each aggregated message after aggregation corresponds to exactly one sentence in the output.

[3]The object-slot-value syntax used here allows messages containing ontology instances to be validated easily against an XML schema.

[4]`http://jena.sourceforge.net/`

[5]This template is simplified; there are actually many more tests, and aggregation is performed in several passes to allow multi-level aggregation. The `set` namespace refers to a Java *Set* instance that stores message IDs to avoid processing a message twice.

```
<messages>
  <msg id="t2-1-5" prop="has_colour" type="prop-has-val" same-as-last="false" already-said="false">
    <slot name="object" value="tileset9"/>
    <slot name="value" value="terracotta beige"/>
  </msg>
  <msg full-sentence="false" id="t2-1-6" prop="has_colour" type="canned-text">
    <slot name="object" value="tileset9"/>
    <slot name="value" value="give-tuscan-feeling"/>
  </msg>
</messages>
```

Figure 4: Initial messages

```
<xsl:template match="messages">
  <xsl:variable name="void" select="set:clear()"/>
  <messages>
    <xsl:for-each select="msg">
      <xsl:variable name="next" select="following-sibling::msg[1]"/>
      <xsl:choose>
        <!-- Return nothing if we've already processed this message. -->
        <xsl:when test="set:contains(@id)"/>

        <!-- Add canned text to a sentence. -->
        <xsl:when test="@prop=$next/@prop and @type='prop-has-val'
          and $next/@type='canned-text' and not($next/@full-sentence='true')">
          <msg type="same-prop-canned-text" id="{concat(@id, '+', $next/@id)}">
            <slot name="prop"> <xsl:copy-of select="."/> </slot>
            <slot name="text"> <xsl:copy-of select="$next"/> </slot>
          </msg>
          <xsl:variable name="void" select="set:add(string(@id))"/>
          <xsl:variable name="void" select="set:add(string($next/@id))"/>
        </xsl:when>
        <!-- ... other tests ... -->

        <!-- Nothing matched: just copy the message across. -->
        <xsl:otherwise> <xsl:copy-of select="."/> </xsl:otherwise>
      </xsl:choose>
    </xsl:for-each>
  </messages>
</xsl:template>
```

Figure 5: Aggregation template (simplified)

3.3 Sentence Planning

After the content of a description has been selected and structured, the logical forms to send to the realizer are created by applying further XSLT templates. Every such template matches a message with particular properties, and produces a logical form for the realizer, possibly combining the results of other templates to produce its own final result. XSLT modes are used to select different templates in different target syntactic contexts.

Two sample templates are shown in Figure 7. The first template produces the logical form for a sentence (`mode="s"`) describing the colours of a tileset (e.g., *The tiles are terracotta and beige*). The second template creates a logical form representing a commentary message as a verb phrase[6] (`mode="vp"`), and then appends it as an elaboration to a sentence about the same property. When the messages in Figure 6 are transformed by these templates, the result is the logical form shown in Figure 8, which corresponds to the sentence *The tiles are terracotta and beige, giving the room the feeling of a Tuscan country home*.

Referring expressions are generated based on the number of mentions of the referent: the first reference gets a full NP (e.g., *this design*), while subsequent mentions are pronominalized.

The logical form created for each top-level message is sent to the OpenCCG realizer, which then generates and returns the corresponding surface form. As described below, the logical forms may incorporate alternatives, in which case the realizer chooses the logical form to use.

4 Sending Alternatives to the Realizer

Many messages can be realized by several different logical forms. For example, to inform the user

[6] Canned-text commentary is represented in the realizer lexicon as a multi-word verb.

```
<messages>
  <msg id="t2-1-5+t2-1-6" type="same-prop-canned-text">
    <slot name="prop">
      <msg id="t2-1-5" prop="has_colour" type="prop-has-val" same-as-last="false" already-said="false">
        <slot name="object" value="tileset9"/>
        <slot name="value" value="terracotta beige"/>
      </msg>
    </slot>
    <slot name="text">
      <msg full-sentence="false" id="t2-1-6" prop="has_colour" type="canned-text">
        <slot name="object" value="tileset9"/>
        <slot name="text" value="give-tuscan-feeling"/>
      </msg>
    </slot>
  </msg>
</messages>
```

Figure 6: Combined messages

```
<xsl:template match="msg[@type='prop-has-val' and @prop='has_colour']" mode="s">
  <node pred="be" tense="pres">
    <rel name="Arg"> <node pred="tile" det="the" num="pl"/> </rel>
    <rel name="Prop"> <xsl:apply-templates select="slot[@name='value']" mode="np"/> </rel>
  </node>
</xsl:template>

<xsl:template match="msg[@type='same-prop-canned-text']" mode="s">
  <node pred="elab-rel">
    <rel name="Core"> <xsl:apply-templates select="slot[@name='prop']/msg" mode="s"/> </rel>
    <rel name="Trib"> <xsl:apply-templates select="slot[@name='text']/msg" mode="vp"/> </rel>
  </node>
</xsl:template>
```

Figure 7: Sentence-planning templates (simplified)

that a particular design is in the country style, the options include *This design is in the country style* and *This design is country*. Often, the text planner has no reason to prefer one alternative over another. Rather than picking an arbitrary option within the text planner (as did, e.g., van Deemter et al. (1999)), we instead defer the choice and send all of the valid alternatives to the realizer, in a packed representation. This makes the implementation of the text planner more straightforward. Figure 9 shows an example of such a logical form, incorporating both of the above options under a `<one-of>` element.

To process a logical form with embedded alternatives, the COMIC realizer makes use of the same *n*-gram language models that it uses to guide its search for the realization of a single logical form. Since OpenCCG cannot yet handle the realization of logical forms with embedded alternatives directly (though this capability is planned), in the current system the packed alternatives are first multiplied out into a list of top-level alternatives, whose order is randomly shuffled. The realizer then computes the best realization for each top-level alternative in turn, keeping track of the overall best scoring complete realization, until either the anytime time limit is reached or the list is exhausted. To allow for some free variation, a new realization's score must exceed the current best one by a certain threshold before it is considered significantly better.

As a concrete example, consider the case where the system must confirm that the user intends to refer to a tileset with a specific feature. The feature could be included in the logical form in two ways: it could be attached directly to the *design* node (2–3), or it could instead be included as a non-restrictive modifier (4).

(2) *Do you mean this country design?*

(3) *Do you mean this design by Coem?*

(4) *Do you mean this design, with tiles by Coem?*

When the modifier can be placed before *design*, as in (2), the directly-attached structure is acceptable. However, for some features, the modifier can only be placed after the modified noun, as in (3). In these cases, the preferred structure is instead the non-restrictive one in (4); this breaks the sentence into two intonational phrases, which makes it easier to understand when it is output by the speech synthesizer. This preference is implemented by including only sentences of the preferred type when

```
<!-- The tiles are terracotta and beige, giving the room the feeling of a Tuscan country home. -->
<lf id="t2-1-5+t2-1-6">
  <node mood="dcl" info="rh" pred="elab-rel" id="n7">
    <rel name="Core">
      <node tense="pres" id="n2" pred="be">
        <rel name="Arg"> <node det="the" pred="tile" id="n1" num="pl"/> </rel>
        <rel name="Prop">
          <node id="n3" pred="and">
            <rel name="List">
              <node id="n4" kon="+" pred="terracotta"> <rel name="Of"> <node idref="n1"/> </rel> </node>
              <node kon="+" id="n6" pred="beige"> <rel name="Of"> <node idref="n1"/> </rel> </node>
            </rel>
          </node>
        </rel>
      </node>
    </rel>
    <rel name="Trib">
      <node id="n8" pred="give-tuscan-feeling"> <rel name="Arg"> <node idref="n1"/> </rel> </node>
    </rel>
  </node>
</lf>
```

Figure 8: Generated logical form

```
<lf id="t2-1-2">
  <!-- This design is ... -->
  <node tense="pres" mood="dcl" info="rh" pred="be" id="n13">
    <rel name="Arg">
      <node id="n1" num="sg" pred="design" kon="+">
        <rel name="Det"> <node kon="+" pred="this" id="n18"/> </rel>
      </node>
    </rel>
    <rel name="Prop">
      <one-of>
        <!-- ... in the country style. -->
        <node pred="in" id="n14">
          <rel name="Fig"> <node idref="n1"/> </rel>
          <rel name="Ground">
            <node num="sg" det="the" pred="style" id="n15">
              <rel name="HasProp"> <node id="n16" kon="+" pred="country"/> </rel>
            </node>
          </rel>
        </node>

        <!-- ... country. -->
        <node id="n20" kon="+" pred="country"> <rel name="Of"> <node idref="n1"/> </rel> </node>
      </one-of>
    </rel>
  </node>
</lf>
```

Figure 9: Logical form containing alternatives

building the language model for OpenCCG. The realizer will then give (4) a higher *n*-gram score than (3), and will therefore choose the desired structure.

In addition to simplifying the implementation, retaining multiple alternatives through the planning process also increases the robustness of the system, and provides a substitute for backtracking. Particularly during development, there may be times when a required template simply does not exist; for example, the second template in Figure 7 will fail if the canned-text commentary cannot be realized as a verb phrase. In such cases, the text planner prunes out the failing possibilities before sending the set of options to the realizer, using the template shown in Figure 10.

5 Related Work

The work presented here continues in the tradition of several recent NLG systems that use what could be called *generalized* template-based processing. By generalized, we mean that, rather than manipulating flat strings with no underlying linguistic representation, these systems instead work with structured fragments, which are often processed recursively. Other systems that fall into this category include EXEMPLARS (White and Caldwell, 1998), D2S (van Deemter et al., 1999), Interact

```
<xsl:template match="one-of">
  <!-- Recursive pruning step -->
  <xsl:variable name="pruned-alts">
    <xsl:for-each select="*">
      <xsl:variable name="pruned-alt"> <xsl:apply-templates select="."/> </xsl:variable>
      <xsl:if test="not(xalan:nodeset($pruned-alt)//fail)">
        <xsl:copy-of select="$pruned-alt"/>
      </xsl:if>
    </xsl:for-each>
  </xsl:variable>
  <xsl:variable name="num-remaining" select="count(xalan:nodeset($pruned-alts)/*)"/>

  <!-- Propagation step -->
  <xsl:choose>
    <!-- keep one-of when multiple alts succeed -->
    <xsl:when test="$num-remaining &gt; 1"> <one-of> <xsl:copy-of select="$pruned-alts"/> </one-of> </xsl:when>

    <!-- filter out one-of when just one choice remains -->
    <xsl:when test="$num-remaining = 1"> <xsl:copy-of select="$pruned-alts"/> </xsl:when>

    <!-- fail if none remain -->
    <xsl:otherwise> <fail/> </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Figure 10: Failure-pruning template

(Wilcock, 2001; Wilcock, 2003), and SmartKom (Becker, 2002).

The main novel contribution of the text-planning approach described here is in its use of an external realizer that processes logical forms with embedded alternatives. This eliminates the need to use a backtracking AI planner (Becker, 2002) or to make arbitrary choices when multiple alternatives are available (van Deemter et al., 1999). The realizer also uses a completely different algorithm than the XSLT template processing—bottom-up, chart-based search rather than top-down rule expansion—which allows it to deal with those aspects of NLG that are more easily addressed using this kind of processing strategy.

Our approach to text planning draws from both the AI-planning and the template-based traditions in natural language generation. Most previous NLG systems that use AI planners use them primarily to do hierarchical decomposition of communicative goals; the work described here uses XSLT to achieve the same end, with a substitute for backtracking provided by the realizer's support for multiple alternatives. The system is nonetheless equally based on (generalized) template processing. This demonstrates that, rather than being in conflict, the two traditions actually have complementary strengths, which can usefully be combined in a single system (contra Reiter, 1995; cf. van Deemter et al., 1999).

6 Conclusions and Future Work

We have described a successful implementation of the classic NLG pipeline that uses XSLT template processing as a top-down rule-expanding planner. Implementing the necessary steps using XSLT was generally straightforward, and the ability to use off-the shelf, well-tested and well-documented tools such as Java and Xalan adds to the ease of implementation and robustness.

Our implementation creates logical forms for the OpenCCG realizer; this allows the realizer to be used for those parts of the generation process to which XSLT is less well-suited. We also take advantage of the fact that the realizer uses statistical language models in its search for a surface form by generating logical forms with embedded alternatives, allowing the realizer to choose the one to use. This both adds robustness to the system and eliminates the need for backtracking within the text planner.

The current implementation is fast and reliable: it correctly processes all input from the dialogue manager, and the time it takes to do so is relatively short compared to that required by the other processing and communication tasks in COMIC.

The entire COMIC demonstrator will shortly be evaluated. As part of this evaluation, we plan to measure users' recall of the information that the system presents to them, where that information is generated at different levels of detail.

At the moment, the logical form for each message is created in isolation. In future versions of COMIC, we plan to use ideas from centering theory to help ensure coherence by planning a coherent *sequence* of logical forms for a description. We will implement this in a way similar to that described by Kibble and Power (2000).

We will also incorporate a model of the user's preferences into a later version of COMIC. The model will be used both to rank the options to be presented to the user, and to generate user-tailored descriptions of those options, as in FLIGHTS (Moore et al., 2004).

Finally, we plan to extend the use of data-driven techniques in the realizer, and to make use of these techniques to help in choosing among alternatives in the other COMIC output modalities.

Acknowledgements

Thanks to Jon Oberlander, Johanna Moore, and the anonymous reviewers for helpful comments and discussion. This work was supported in part by the COMIC (IST-2001-32311) and FLIGHTS (EPSRC-GR/R02450/01) projects.

References

Tilman Becker. 2002. Practical, template-based natural language generation with TAG. In *Proceedings of TAG+6*.

Kees van Deemter, Emiel Krahmer, and Mariët Theune. 1999. Plan-based vs. template-based NLG: a false opposition? In *Proceedings of "May I speak freely?" workshop at KI-99*.

Rodger Kibble and Richard Power. 2000. An integrated framework for text planning and pronominalisation. In *Proceedings of INLG 2000*.

Johanna Moore, Mary Ellen Foster, Oliver Lemon, and Michael White. 2004. Generating tailored, comparative descriptions in spoken dialogue. In *Proceedings of FLAIRS 2004*.

Els den Os and Lou Boves. 2003. Towards ambient intelligence: Multimodal computers that understand our intentions. In *Proceedings of eChallenges e-2003*.

Ehud Reiter. 1995. NLG vs. templates. In *Proceedings of EWNLG-95*.

Ehud Reiter and Robert Dale. 2000. *Building Natural Language Generation Systems*. Cambridge University Press.

Mark Steedman. 2000. *The Syntactic Process*. MIT Press.

Michael White. 2004a. Efficient realization of coordinate structures in Combinatory Categorial Grammar. *Research on Language and Computation*. To appear.

Michael White. 2004b. Reining in CCG chart realization. In *Proceedings of INLG 2004*. To appear.

Michael White and Jason Baldridge. 2003. Adapting chart realization to CCG. In *Proceedings of EWNLG-03*.

Michael White and Ted Caldwell. 1998. EXEMPLARS: A practical, extensible framework for dynamic text generation. In *Proceedings of INLG 1998*.

Graham Wilcock. 2001. Pipelines, templates and transformations: XML for natural language generation. In *Proceedings of NLPXML-2001*.

Graham Wilcock. 2003. Integrating natural language generation with XML web technology. In *Proceedings of EACL-2003 Demo Sessions*.