MMAX2 Annotation Tool
Style Sheet Guide

© Christoph Müller
EML Research gGmbH
http://mmax.eml-research.de

1st February 2005

Contents

1 About this Document

This document describes how XSL style sheets are used to create the display of the MMAX2 annotation tool. Though writing a style sheet for this purpose is pretty simple, a certain familiarity with XSL is assumed. This document describes mainly those points in which MMAX2 style sheets are different from 'regular' style sheets. In addition to what is detailed in this document, the whole range of XSL functions can be used anywhere in a MMAX2 style sheet. Also, the functionality described here is not considered to be complete. Rather, additional functions are likely to be added in the future. If you have suggestions, we would like to hear from you!

Important Note: The version of MMAX2 that accompanies this document (1.0 BETA 3 or later) introduces considerable changes to the MMAX2 style sheet engine and the methods supported by it. These changes have been implemented in order to optimize and simplify the use of style sheets. They also render superfluous the base data <sentences> and <turns> files that earlier versions required (cf. Section 2). As a consequence, style sheets from earlier releases (up to and including 1.0 BETA 2), including the ones supplied as samples, will probably not work any more. The sample files supplied with the latest distribution of MMAX2 have been updated to not use deprecated functionalities any more. If you have written your own style sheets and run into problems now, please write to mmax@eml-research.de, and we will be happy to assist you!

2 Introduction

The MMAX2 annotation tool uses XSL stylesheets to create different display renderings (or *views*) for a given set of base data files and annotation levels. Depending on the degree of sophistication and the number of annotation levels, an XSL style sheet might become rather complex. On the other hand, a basic display can be created with just a few lines.

The style sheets that have have been defined for a given document are associated with this document through the common_paths.xml file that has to be present in the root directory of every annotation project (or set of projects). In the <views> Section of this file, there is one reference to every defined style sheet file. Since references to style sheets are defined in the common_paths.xml file (and not in each .mmax file individually), adding one reference to this file is sufficient to make a new style sheet accessible for all .mmax files in the annotation project. The first style sheet in the <views> Section is the *default* style sheet that is always loaded when a .mmax file is loaded. After that, the available style sheets can be found by selecting in the 'Markable level control panel' the menu 'Settings' and then the menu item 'Style Sheet'. **Note:** This will probably change in a later version, since the 'Markable level control panel' is likely to be abandoned. If more than one style sheet is available, the display can be changed by selecting a different one. The style sheet currently in use can be reapplied (e.g. after it was modified) by selecting in the main window the 'Display' menu and then the 'Reapply current style sheet' menu item. Note that this means that style sheets can be modified and tested interactively without the need to reload the entire .mmax file. The tool and the style sheet engine is also sufficiently robust to handle even serious style sheet errors: If a style sheet crashes after modification and reapplication, simpy undo the last change, save the style sheet again, and again reapply it. In most cases, the tool will recover from the style sheet crash without need to restart.

The fact that MMAX2 does support multiple levels of annotation (as opposed to the first version which supported only one such level) means that the once obligatory <sentences> or <turns> base data files can be abandoned in favour of an additional markable level. The fact that one of both files was required as part of the base data was a nuisance in earlier versions of MMAX, since sentence and particularly turn segmentation was simply not always available a priori, i.e. before starting the annotation. A common work-around was to use a dummy <sentences> or <turns> file which contained only one entry, spanning the entire discourse.

From this version on, both files are optional, meaning that both the <sentences> and the <turns> entry in a .mmax file can remain empty. Instead, an additional markable level can be defined, which has the advantage that sentence or turn segmentation can be done using the tool, i.e. as part of the annotation proper, rather than prior to the annotation. Already existing <sentences> or <turns> files can easily be **converted into markable files** by

- renaming the elements (incl. their ID namespaces) from `sentence` or `turn` to `markable`,
- replacing the root element with a <markables> element, including a proper namespace (i.e. *sentences* or *turns*),
- adding a markables doctype declaration, including a SYSTEM reference to the file markables.dtd, and
- adding a reference to the new markable layer to the .mmax file.

This document uses for illustration purposes the sample data supplied with this distribution. This data follows **the data format that is now favoured**, and the style sheets described in the following are slightly

incompatible with earlier versions.

3 Simple Style Sheets

3.1 A Basic Style Sheet Template

The most basic MMAX2 style sheet is one that simply outputs the base data, i.e. the text that is represented in the <word> elements in the base data words.xml file. More complex style sheets (described later) include ones that add line breaks, markable handles or markable attribute values at certain positions. Note, however, that the style sheet is NOT responsible for specifying e.g. the font color for markables: This is the domain of markable customizations. As a rule of thumb, style sheets are responsible for the display *layout*, i.e. what appears where, while markable customizations are responsible for *how* things appear, i.e. using which font attributes.[1]
Each MMAX2 style sheet must have *at least* the following elements:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:mmax="org.eml.MMAX2.discourse.MMAX2DiscourseLoader"
                version="1.0">
 <xsl:output method="text" indent="no" omit-xml-declaration="yes"/>
 <xsl:strip-space elements="*"/>
 <xsl:template match="words">
  <xsl:apply-templates/>
 </xsl:template>
</xsl:stylesheet>
```

The <xsl:stylesheet> tag does not only define the standard xsl name space (first argument), but also the mmax name space (second argument), which is associated with the name of a MMAX2 Java class. This name space is required to enable the style sheet to execute the Java methods that MMAX2 needs when building the display. The <xsl:output> tag mainly specifies that the style sheet should produce plain text without any indentations or markup. The <xsl:strip-space> tag makes sure that white space and line breaks from the base data and annotation level xml files do not show up in the display. Finally, there is one <xsl:template> element to match the root node of the document submitted to the style sheet (i.e. the parsed base data words.xml file). This template is matched only once, and calls <xsl:apply-templates/> for each of its childern, i.e. for every individual word in the base data.[2]
The above style sheet will not yet produce any output (and probably even throw some non-critical errors). In order to actually let text appear in the display, an <xsl:template> element for elements of type <word> has to be added. The minimal form that this template can take is the following:

```
<xsl:template match="word">
 <xsl:value-of select="mmax:registerDiscourseElement(@id)"/>
 <xsl:value-of select="mmax:setDiscourseElementStart()"/>
  <xsl:apply-templates/>
 <xsl:value-of select="mmax:setDiscourseElementEnd()"/>
 <xsl:text> </xsl:text>
</xsl:template>
```

[1] As will become clear later, this division of labour is not really strict, because the font attributes for *literal text* inserted into the display are specified by the style sheet inserting it.

[2] For those interested, this is the major point of difference between this and earlier versions of MMAX: Earlier versions, which still required a <sentences> or <turns> file would match this elements, calling the now deprecated method mmax:getDiscourseElementsAsNodes(String ID) for each sentence or turn.

The first line in the <word> template must be a call to mmax:registerDiscourseElement(String ID). The above template demonstrates the easiest way to realize this call, i.e. in the `select` part of a <xsl:value-of> instruction. Although this instruction is normally used to insert text into the style sheet output stream (i.e. in the result of the transformation), it can also be used like this if the function that is called simply does not return anything. The next three lines are responsible for adding the actual word string to the display. The middle line simply calls <xsl:apply-templates/> for the current <word> element's textual child, causing the text string of this child to be inserted into the display. This call has to be immediately surrounded by calls to mmax:setDiscourseElementStart() and mmax:setDiscourseElementEnd(), respectively. These calls create an association between the current element's ID and the portion of the display that contains the string of this element's textual child. The last line in the template inserts a space separator directly after the word just added. When applied to the sample HTC data, the above minimal MMAX2 style sheet produces the display shown in Figure 1. The source code for the above style sheet can be found in file htc_1.xsl the

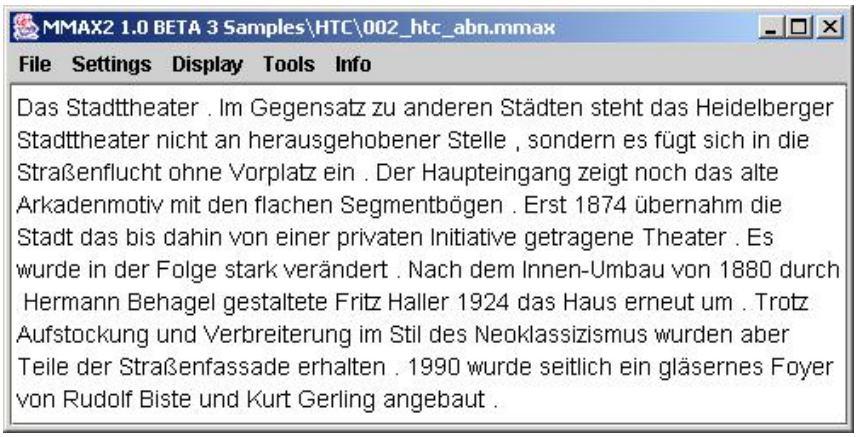

Figure 1: MMAX2 display with minimal output

`Styles` folder of the HTC sample data. Note that for the above screen shot (and for all following ones, unless otherwise noted), markable customizations have been switched off.[3]

[3]Markable customizations can be deactivated for a level by unchecking the box at the right of the Validate button in the 'Markable level control panel'.

4 More Complex Style Sheets

Most displays (except for the most simple ones) will have to incorporate somehow information about the actual annotation that is available for a given set of base data elements. In order to do that, they have to access markables, since markables are the carriers of annotation information.

4.1 Accessing Markables

MMAX2 is a multi-level annotation tool. Therefore, several levels of annotation do normally exist, each of which resides in a separate xml file. In order for the style sheet to be able to distinguish markables from different levels, each level has to be associated with an xsl name space, which is normally the name of the annotation level as defined in the .mmax file. A markable level is associated with a name space by adding a name space declaration to the root <markables> element in the markable xml file. In the HTC sample data, e.g., the markables from the *coref* level have the root element <markables xmlns="www.eml.org/NameSpaces/coref">, the markables from the *sentences* level have the root element <markables xmlns="www.eml.org/NameSpaces/sentences">.
The name space of each markable level to be accessed in a style sheet has to be declared in the style sheet header. This is done by adding lines *into* the <xsl:stylesheet> element, like this:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:mmax="org.eml.MMAX2.discourse.MMAX2DiscourseLoader"
                xmlns:coref="www.eml.org/NameSpaces/coref"
                xmlns:sentences="www.eml.org/NameSpaces/sentences"
                version="1.0">
```

In a style sheet like this, markable elements from different levels can be distinguished by prepending their associated name space to the generic 'markable' element name: coref:markable vs. sentences:markable.

4.2 Adding Simple Markable Handles

One of the most useful ways to incorporate markables into the display is by adding *markable handles*. These markable handles surround a markable and allow its immediate selection, which is particularly useful in cases of multiple embedding. In addition, they also nicely visualize the extent of a markable. By default, they are mouse-sensitive, i.e. when the mouse pointer rests over a markable handle, the matching handle will be highlighted.[4]
Adding handles for markables from a particular level requires two things: The <xsl:template> for <word> elements has to be slightly modified, and templates for actually adding the handles have to be written.
The first step consists of adding two instructions, like in the following:

```
<xsl:template match="word">
 <xsl:value-of select="mmax:registerDiscourseElement(@id)"/>
 <xsl:apply-templates select="mmax:getStartedMarkables(@id)" mode="opening"/>
  <xsl:value-of select="mmax:setDiscourseElementStart()"/>
   <xsl:apply-templates/>
  <xsl:value-of select="mmax:setDiscourseElementEnd()"/>
```

[4] In fact, pairs of handles will *always* be highlighted if the mouse pointer rests over a position that is *uniquely associated* with a markable. This is true for all markable handles, but also if only one markable level with only one markable exists at a given position.

```
  <xsl:apply-templates select="mmax:getEndedMarkables(@id)" mode="closing"/>
  <xsl:text> </xsl:text>
 </xsl:template>
```

As the names suggest, each of the inserted instructions returns a node set of markables that start or end at the respective position. Note that the calls have to be added exactly as shown above, i.e. immediately before mmax:setDiscourseElementStart() and after mmax:setDiscourseElementEnd(). Also note the `mode` argument, which is important to select the correct template to match the started or ended markable nodes (cf. below).
What remains to be done is adding <xsl:template> elements to actually match the markables returned by the calls to mmax:getStartedMarkables(String ID) and mmax:getEndedMarkables(String ID).
The domain of a template, i.e. the set of elements to which it is to apply, is determined by the template's `match` argument. For markables from the *coref* level, this is coref:markables. In order to further distinguish between templates for starting and ending markables, the template's `mode` argument is used, which is set to `opening` or `closing`, just like in the corresponding calls in the <xsl:template> for <word> elements, cf. above. Thus, the most simple markable handles are created with the following pair of templates:

```
<xsl:template match="coref:markable" mode="opening">
 <xsl:value-of select="mmax:addLeftMarkableHandle(@mmax_level, @id, '[')"/>
</xsl:template>

<xsl:template match="coref:markable" mode="closing">
 <xsl:value-of select="mmax:addRightMarkableHandle(@mmax_level, @id, ']')"/>
</xsl:template>
```

This will produce a display with simple markable handles as shown in Figure 2.

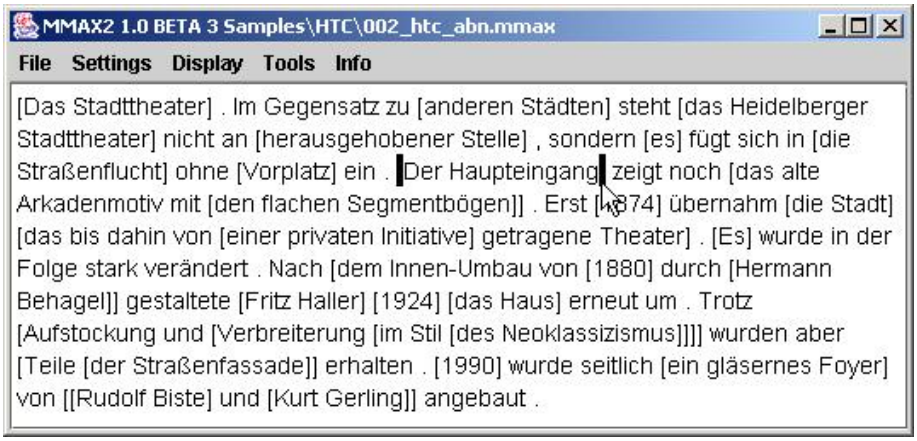

Figure 2: MMAX2 display with simple markable handles

The source code for the above style sheet can be found in file htc_2.xsl the `Styles` folder of the HTC sample data. Note that opening and closing handles are automatically inserted in the correct embedding order, i.e. the handle that is opened first is closed last.

The actual string that is used for the handle can be specified in the last parameter of the mmax:addXMarkableHandle(...) call. Note that using a string with more than one character is also possible, creating a correspondingly longer handle. However, while markable handles are mouse-sensitive over their entire length (cf. above), only the leftmost and rightmost characters of left and right handles, respectively, will be highlighted when the mouse pointer rests over a handle.[5] Note also that, in contrast to e.g. the call to mmax:registerDiscourseElement(String ID), the calls for adding markable handles actually do return a string, which is then inserted into the display. Since markable handles constitute *literal text* (as opposed to text from the base data), their font attributes have to be set by the style sheet inserting them. Font attributes for literal text (not just markable handles) can be set using the methods described in Section 5.3. Generally, text to which attributes are to be applied must be surrounded by a pair of calls to the respective start / end methods. Thus, in order to let the markable handles appear in bold font, the following modifications to the above templates are necessary:

```
<xsl:template match="coref:markable" mode="opening">
 <xsl:value-of select="mmax:startBold()"/>
  <xsl:value-of select="mmax:addLeftMarkableHandle(@mmax_level, @id, '[')"/>
 <xsl:value-of select="mmax:endBold()"/>
</xsl:template>

<xsl:template match="coref:markable" mode="closing">
 <xsl:value-of select="mmax:startBold()"/>
  <xsl:value-of select="mmax:addRightMarkableHandle(@mmax_level, @id, ']')"/>
 <xsl:value-of select="mmax:endBold()"/>
</xsl:template>
```

This will produce the display with bold markable handles shown in Figure 3).

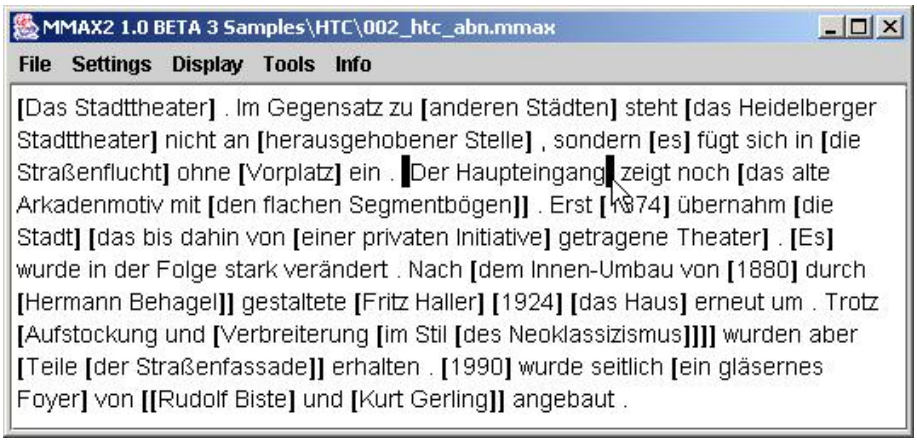

Figure 3: MMAX2 display with simple markable handles in bold font

The source code for the above style sheet can be found in file htc_3.xsl the `Styles` folder of the HTC sample data. By default, markable handles will always appear in black. Different colours can be defined by means of markable customizations, by defining an appropriate pattern (e.g. `pattern="{all}"`) and

[5] This holds for the simple versions of the addXMarkableHandle(...) methods only. Section 4.4 describes methods that allow to override this behaviour.

using the format `handles=colour` in the style string (e.g. `style="handles=yellow"`). Note that this mechanism is flexible and powerful enough to also have attribute-dependent markable handle colours. Note also that only handle *colours* can be set this way.

4.3 Structuring the Display

A second major use of incorporating markables into the display is by using them to put the display text in a particular structure. Structuring can be as simple as adding a line break after every sentence, or visualizing the turn and utterance structure of a spoken dialogue. Adding line breaks is simply done by writing a template that adds a line break after each *sentence* markable. This can be done like this:

```
<xsl:template match="sentence:markable" mode="closing">
 <xsl:text>
 </xsl:text>
</xsl:template>
```

This will produce the output in Figure 4. The source code for the above style sheet can be found in file

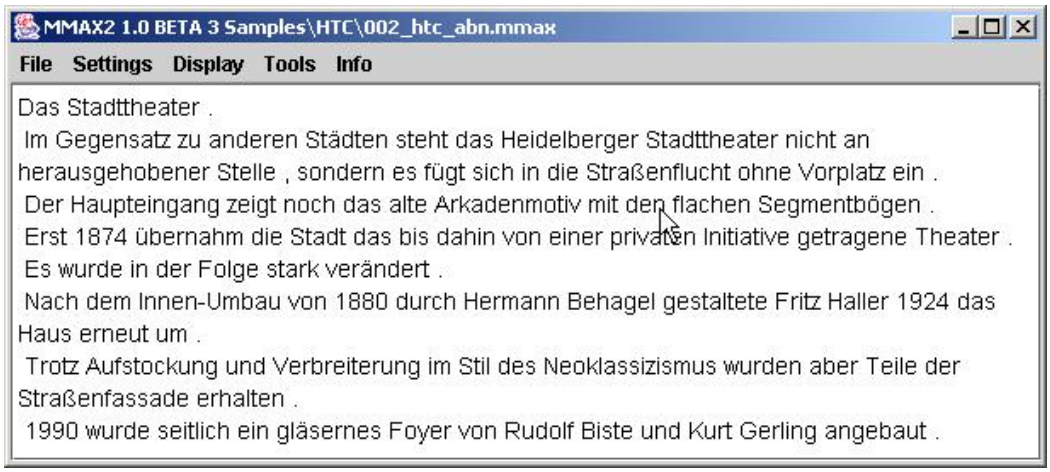

Figure 4: MMAX2 display with sentences separated by line breaks

htc_4.xsl the `Styles` folder of the HTC sample data.
A more refined layout with the first line (i.e. the head line) set off with an extra empty line can also be created by using the fact that the first markable in the sample file (and in all other files from the HTC, for that matter) has a particular id.

```
<xsl:template match="sentence:markable" mode="closing">
<xsl:choose>
 <xsl:when test="@id='markable_1'">
  <xsl:text> <!-- Add extra line break after sentence markable with id markable_1 --

  </xsl:text>
 </xsl:when>
 <xsl:otherwise>
```

```
  <xsl:text> <!-- Add normal line break only after all other sentence markables -->
  </xsl:text>
 </xsl:otherwise>
</xsl:choose>
</xsl:template>
```

This is also a good example for how standard XSL control structures can be used for creating MMAX2 displays. The output of the above template looks as in Figure 5: The source code for the above style sheet

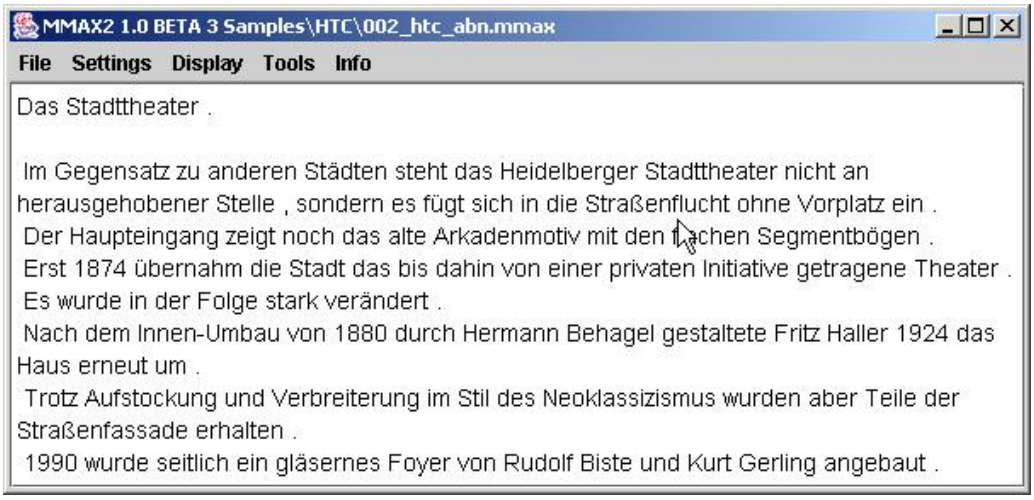

Figure 5: MMAX2 display with sentences separated by line breaks and head line set off

can be found in file htc_5.xsl the `Styles` folder of the HTC sample data.
As a slightly more complex structuring example, consider the Trainline data. Each Trainline dialogue contains markables representing turn and utterance segments. The information on these levels should be used to render the display of the dialogue as readable as possible. In addition, the *turn* level contains information about speaker and number of turn, which one would also like to see on the display, although it is not part of the actual data.
To start with, the following templates (in combination with the header and <word> template described above) demonstrate how to add this type of information to the display:

```
<xsl:template match="turns:markable" mode="opening">
 <xsl:text>
 </xsl:text>
 <xsl:value-of select="mmax:startBold()"/>
  <xsl:value-of select="@speaker"/>
  <xsl:text>.</xsl:text>
  <xsl:value-of select="@number"/>
  <xsl:text>: </xsl:text>
 <xsl:value-of select="mmax:endBold()"/>
 <xsl:text>  </xsl:text> <!-- This is a tab character! -->
 <xsl:value-of select="mmax:startBold()"/>
 <xsl:value-of select="mmax:addLeftMarkableHandle(@mmax_level, @id, '[')"/>
```

```
 <xsl:value-of select="mmax:endBold()"/>
</xsl:template>

<xsl:template match="turns:markable" mode="closing">
 <xsl:value-of select="mmax:startBold()"/>
  <xsl:value-of select="mmax:addRightMarkableHandle(@mmax_level, @id, ']')"/>
 <xsl:value-of select="mmax:endBold()"/>
 <xsl:text>
 </xsl:text>
</xsl:template>
```

This will produce the type of display as in Figure 6. The source code for this style sheet can be found in file

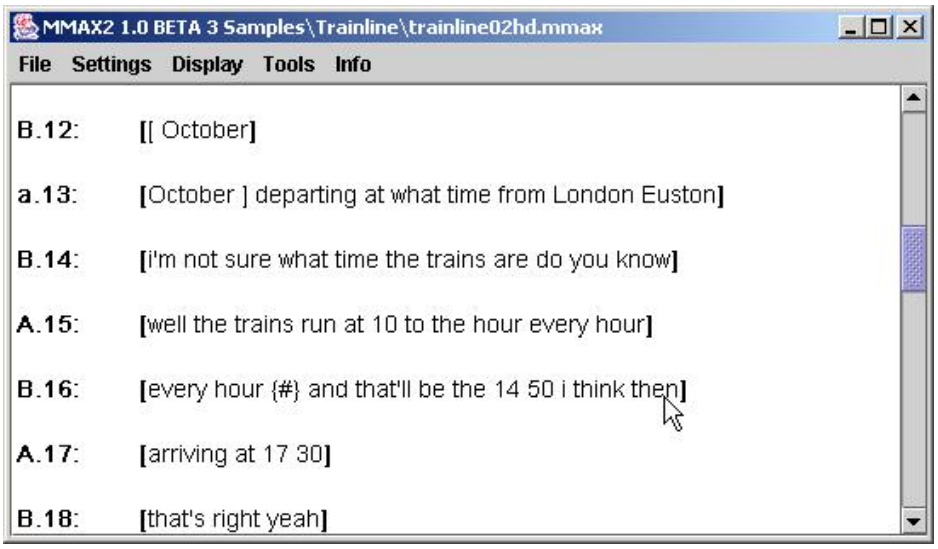

Figure 6: MMAX2 display with one turn per line

trainline_1.xsl the `Styles` folder of the Trainline sample data. There are two interesting things to note in this style sheet. First, note how the *speaker* and *number* attributes in the turn level markables are accessed by using the standard XSL notation `@speaker` and `@number`. Second, a tab character is used to separate the turn number from the actual turn. This way, the proper turn alignment is produced.
It is important to understand the status of the literal text (i.e. the speaker and number strings) added to the display: Although this text consists of information *taken* from markables, it is part of the display only, and cannot be annotated. Thus, the text cannot be highlighted by left-clicking and dragging the mouse (just like markable handles), so that consequently it cannot be part of markables.
The obvious problem with the above display layout is that utterance segmentation is not properly visible. Adding a markable handle for every utterance within a turn would be simple, but would still render every turn in one long line. It would be preferable to have a display that gives each utterance in a separate line, grouping together utterances from the same turn. In order to that, one simply has to add a line break before every utterance but the turn-initial one. Adding the following templates for handling utterance markables does just that.

```
 <xsl:template match="utterances:markable" mode="opening">
```

```
 <xsl:if test="mmax:startsMarkableFromLevel(@id, @mmax_level, 'turns')=false">
  <xsl:text>
        </xsl:text> <!-- This is a tab character! -->
 </xsl:if>
 <xsl:value-of select="mmax:addLeftMarkableHandle(@mmax_level, @id, '[')"/>
</xsl:template>

<xsl:template match="utterances:markable" mode="closing">
 <xsl:value-of select="mmax:addRightMarkableHandle(@mmax_level, @id, ']')"/>
</xsl:template>
```

This will produce the type of display as in Figure 7. The source code for this style sheet can be found in file

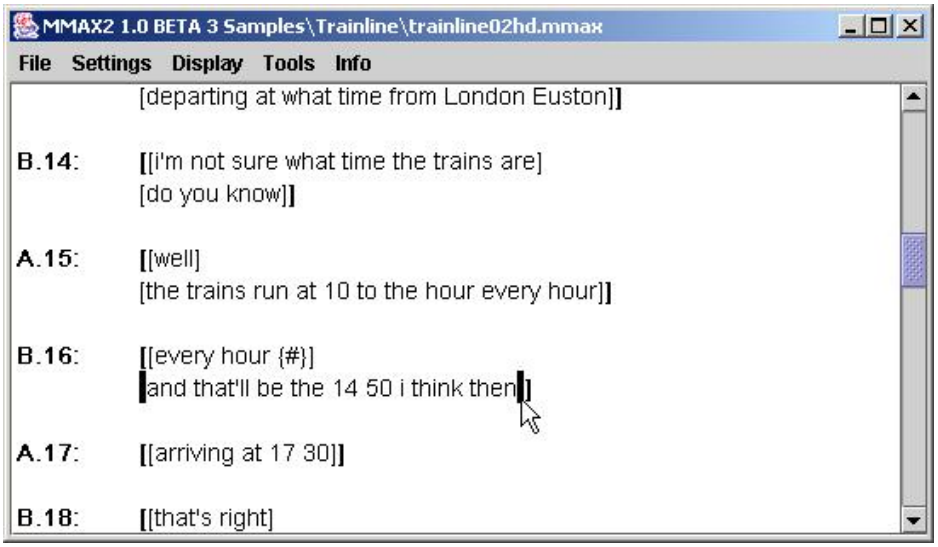

Figure 7: MMAX2 display with one utterance per line, grouped by turns

trainline_2.xsl the `Styles` folder of the Trainline sample data. The most interesting part in the above style sheet is the call to mmax:startsMarkableFromLevel(String ID, String ownLevel, String targetLevel). This is a boolean function that returns `true` if there is a markable on level *targetLevel* that the current markable *starts*, i.e. whose first word is identical to the current markable's first word, and `false` otherwise. In the latter case, a line break plus a tab is added to the display. This method call (and the call to the corresponding mmax:finishesMarkableFromLevel(String ID, String ownLevel, String targetLevel)) can be used wherever a standard XSL boolean-returning function can be used. Note that neither function does care *how many* markables from level *targetLevel* the current markable starts or finishes; rather, it returns `true` if there is at least one, `false` otherwise.

Using markable handles for markables from different markable levels simultaneously (like in the above example) makes it necessary to pay some attention to the markable level *ordering*. For each annotated document, this ordering is initially defined by the ordering of the <level> elements in the associated .mmax file. By default, the ordering is such that levels of markables that are *larger* units (e.g. turns) are **below** levels containing markables that are *smaller* units (e.g. utterances). It is only with this setting that the above style sheet will produce the desired output. The ordering of the markable levels can be changed

temporarily by moving them up or down in the 'Markable level control panel'. In order for changes to the markable level ordering to take effect, the current style sheet has to be reapplied. The style sheet currently in use can be reapplied by selecting in the main window the 'Display' menu and then the 'Reapply current style sheet' menu item.

4.4 Adding Textual Markable Handles

As already mentioned in Section 4.2, markable handles need not be restricted to single characters. They do not even have to consist of *literal* text only. Instead, the **values of markable attributes** can be used as handles. This has the great advantage that attributes can very easily be inspected without the need to select each markable. Only if some error is detected, the markable has to be selected in order to correct the error. Aftzer the markable modification in the MMAX2 attribute window has been applied, a simple reapplication of the current style sheet suffices to also display the corrected value in the display.
Depending on how fancy the character attributes for the handles are, textual handles can be created very easily.

4.4.1 Plain

In the following example, the template for creating the *closing* handles for utterance markables has been modified to include the value of the markable's *type* attribute.

```
<xsl:template match="utterances:markable" mode="closing">
 <xsl:value-of select="mmax:addRightMarkableHandle(@mmax_level,
                                                   @id,
                                                   concat(']  ',@type),
                                                   1)"/>
</xsl:template>
```

The display produced by this template (in combination with the above ones) looks as in Figure 8. The source code for this style sheet can be found in file trainline_3.xsl the `Styles` folder of the Trainline sample data. The right handle at each utterance now contains a matching closing bracket, followed by two space characters, and then the value of the utterance's *type* attribute. The entire handle string is mouse-sensitive. What is important here is that (unlike in the standard behaviour for right markable handles, cf. Section 4.2) what is highlighted is not the right-most character, but (in this case) the first one. This is controlled by the last parameter in the call to mmax:addRightMarkableHandle(String ownLevel, String ownID, String handle, int highlight). The `highlight` parameter can be used to specify the numerical index within the handle string that is to be used for highlighting. Using a value of 1 tells MMAX2 to use the leftmost (i.e. first) character in the handle string. If the right handle string started with e.g. one leading space, followed by the closing bracket, `highlight` would have to be set to 2, and so on.
Alternatively, the *left* markable handle can also be enhanced with attribute values. Substituting the above utterance templates with the following two will display the `type` value at the *beginning* of each utterance.

```
<xsl:template match="utterances:markable" mode="opening">
 <xsl:if test="mmax:startsMarkableFromLevel(@id, @mmax_level, 'turns')=false">
  <xsl:text>
        </xsl:text> <!-- This is a tab character! -->
 </xsl:if>
```

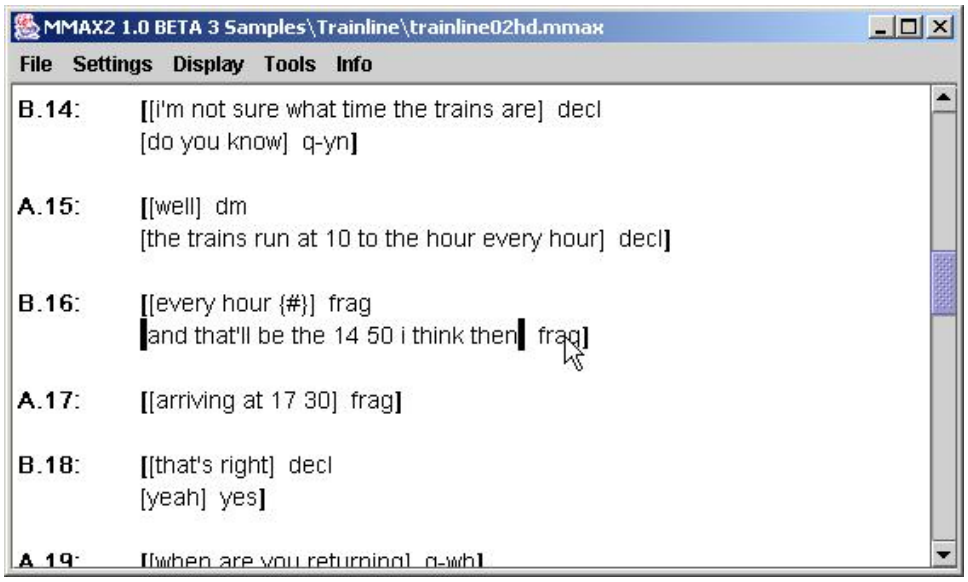

Figure 8: MMAX2 display with one utterance per line, grouped by turns, and turn-type as right handle

```
 <xsl:value-of select="mmax:addLeftMarkableHandle(@mmax_level,
                                                  @id,
                                                  concat(@type,' [’),
                                                  string-length(@type)+2)"/>
</xsl:template>

<xsl:template match="utterances:markable" mode="closing">
 <xsl:value-of select="mmax:addRightMarkableHandle(@mmax_level, @id, ’]’)"/>
</xsl:template>
```

The display will then look as in Figure 9. The source code for this style sheet can be found in file trainline_4.xsl the `Styles` folder of the Trainline sample data. Note how the `highlight` parameter is set dynamically for each utterance depending on the length of the actual type string.

4.4.2 More Fancy

The markable handles described so far have been not so fancy in that they used only very simple font attributes. More fancy handles can be created by using several font attributes within one handle, resp. by applying certain attributes to *part* of a handle only. The following pair of utterance templates will create handles where the brackets appear in normal-sized bold font, but the utterance's *type* attribute value are rendered as subscript.

```
<xsl:template match="utterances:markable" mode="opening">
 <xsl:if test="mmax:startsMarkableFromLevel(@id, @mmax_level, ’turns’)=false">
  <xsl:text>
        </xsl:text> <!-- This is a tab character! -->
 </xsl:if>
 <xsl:value-of select="mmax:addLeftMarkableHandle(@mmax_level,@id,’[’)"/>
```

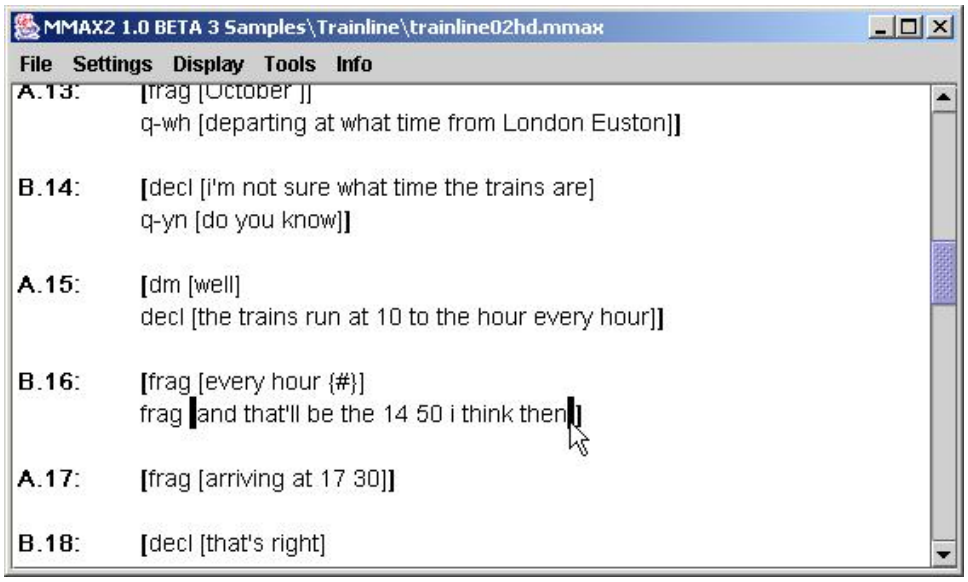

Figure 9: MMAX2 display with one utterance per line, grouped by turns, and turn-type as left handle

```
</xsl:template>

<xsl:template match="utterances:markable" mode="closing">
 <xsl:value-of select="mmax:addRightMarkableHandle(@mmax_level,
                                                   @id,
                                                   string-length(@type)+1,
                                                   1)"/>
<xsl:text>]</xsl:text>
<xsl:value-of select="mmax:startSubscript()"/>
 <xsl:value-of select="@type"/>
</xsl:value-of select="mmax:EndSubscript()">
</xsl:template>
```

The display will then look as in Figure 10. The source code for this style sheet can be found in file trainline_5.xsl the `Styles` folder of the Trainline sample data. The main difference is in mmax:addRightMarkableHandle(String ownLevel, String ownID, int handleLength, int highlight). Instead of supplying the handle string itself, this method receives only the *size* of the handle to be added. The handle itself is added as literal text, which allows to use any type of font attributes supported by MMAX2. Note that it is important that the text actually added is exactly as long as was specified in the *extent* parameter.

4.5 More Flexible Style Sheets (updated for version 1.0 beta 4)

From version 1.0 beta 4 on, there is a convenient way to define different views of a given set of base data files and annotations without having to create an extra style sheet for each. This version introduces so-called *user switches*. User switches are user-defined symbols (i.e. strings) that can be associated with an *on* or *off* state. User switches are defined in the `common_paths.xml` file. The following code extract (taken from the common_paths.xml file of the HTC sample data) defines a user switch with the name *coref_handle*,

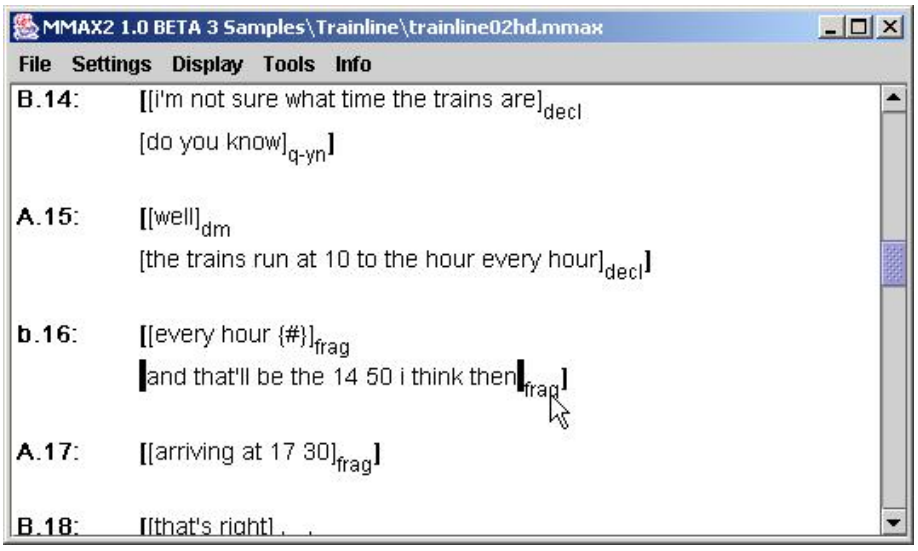

Figure 10: MMAX2 display with one utterance per line, grouped by turns, and turn-type as right handle, in subscript font

and sets it initially to *on*.

```
...
<user_switches>
 <user_switch name="coref_handle" default="on"/>
</user_switches>
...
```

For each user switch defined, a button in the 'User switches' sub menu in the 'Display' menu is created, which reflects the current state of the switch. The point is that the execution of the style sheet can be made to consider the current state of user switches. Thus, when using the style sheet htc_2.xsl on the HTC data, markable handles for the *coref* level will only be added if the *coref_handle* switch is currently set to *on*. See Section 5.2 for information about how to use user switches in your own style sheets.

5 Function Reference (updated for version 1.0 beta 4)

This Section describes the MMAX2 extension functions that can be called from within a MMAX2 style sheet. The extension functions are available *in addition* to the standard XSL functions. For reasons of clarity, the *mmax:* prefix is omitted in the following. When using the functions, the mmax name space must be supplied. Also, the mmax name space has to be declared as described in Section 3.

5.1 System Functions

System functions are those that are needed for MMAX2 to be able to create a correct display. Their use is obligatory, and they must appear at certain pre-defined positions in a style sheet (cf. Section 3.1). System functions are best called in the `select` parameter of <xsl:value-of/> functions: This is the easiest way, and it is feasible because if the function called is void (i.e. does not return anything), adding the function's return value to the output (which is what <xsl:value-of/> does) simply has no effect.

- **`registerDiscourseElement(String ownID)`**:
  This function must be the first to be called in the <xsl:template> for `word` elements. It is used for tool-internal purposes.
- **`setDiscourseElementStart()`**:
  This function must be called immediately before <xsl:apply-templates/> in the <xsl:template> for `word` elements. It is needed for creating the association between discourse elements (i.e. words) and the display positions at which they appear.
- **`setDiscourseElementEnd()`**:
  This function must be called immediately after <xsl:apply-templates/> in the <xsl:template> for `word` elements. It is needed for creating the association between discourse elements (i.e. words) and the display positions at which they appear.

5.2 Control Functions (updated for version 1.0 beta 4)

Control functions are those functions that can be used to influence the style sheet control flow. They all return boolean values (i.e. either `true` or `false`). They can be used wherever boolean-returning standard XSL functions can be used, in particular in the `test` attribute of <xsl:if> and <xsl:when> functions. Right now, there are only two such functions in MMAX2.

- **`startsMarkableFromLevel(String ownID, String ownLevel, String targetLevel)`**:
  This method returns `true` if there is at least one markable on level *targetLevel* that the markable with ID *ownID* on level *ownLevel* starts, `false` otherwise. One markable is said to start another markable if both markables have the same first word.
- **`finishesMarkableFromLevel(String ownID, String ownLevel, String targetLevel)`**:
  This method returns `true` if there is at least one markable on level *targetLevel* that the markable with ID *ownID* on level *ownLevel* finishes, `false` otherwise. One markable is said to finish another markable if both markables have the same last word.
- **`isOn(String switchName)`**:
  This function has been **added in version 1.0 beta 4**. It can be used to make the behaviour of a style sheet dependent on the settings of so-called *user switches* which can be defined in the file

`common_paths.xml`, and modified by the user via the MMAX2 GUI. This method returns `true` if the the user switch of the name *switchName* is currently switched to *on*, `false` otherwise. Using this function to control style sheet control flow can reduce the number of different style sheets that one has to write: Instead of writing e.g. three style sheets, each of which displays handles for one level, one can now simply write one style sheet in which the display of handles is controlled by a separate user switch for every level.

5.3 Functions for Setting Font Attributes

The following functions are available for setting attributes for *literal text*. Literal text is text that is added to the display by either <xsl:text></xsl:text> or <xsl:value-of/>, incl. markable handles. These functions should also be called in the `select` parameter of <xsl:value-of/> functions, cf. Section 5.1 above.

- **`startBold(), endBold()`**
- **`startItalic(), endItalic()`**
- **`startUnderline(), endUnderline()`**
- **`startStrikeThrough(), endStrikeThrough()`**
- **`startSubscript(), endSubscript()`**
- **`startSuperscript(), endSuperscript()`**

The names of the above functions are self-explanatory. Attributes can also be combined: e.g. text can be bold and italic at the same time. However, the font attributes underline and strikethrough resp. subscript and superscript are mutually exclusive!

5.4 Markable-Related Functions

5.4.1 Markable-Returning Functions

As with system functions (cf. Section 5.1 above), the positions at which markable-returning functions may appear are strictly defined. Unlike system functions, however, they are not obligatory, meaning that they can be left out if no access to markables from any levels is required (as in the simple display described in Section 3.1). If markables are to be accessed from within the style sheet (e.g. for adding markable handles), the following two functions have to be called exactly at the positions specified below. Since these functions return markables (in the form of NodeSets), they must be called in the `select` parameter of an <xsl:apply-templates/> call, with the `mode` parameter set to the correct value. This value is necessary so that the markables retrieved can be matched by the respective <xsl:template> elements. Note: Calling markable-returning functions alone will not have any effect on the display! They make only sense in combination with <xsl:template> elements that actually process the returned markables (cf. Section 5.4.2 below).

- **`getStartedMarkables(String ID)`**:
  If used, this function must be called directly after registerDiscourseElement(String ID) and directly before setDiscourseElementStart(). The `mode` value must be set to *opening*. This method returns a NodeSet of all markables that are started at the markable with the supplied ID. The markables

returned are in *inverse* markable level order: markables from lower levels are returned before markables from higher levels. If two markables from the same level start at the same position, the longer one is returned before the shorter one. This ordering is used to ensure that the markables are processed in an order allowing for correct embedding of markable handles.

- **`getEndedMarkables(String ID)`**:
  If used, this function must be called directly after setDiscourseElementEnd(). The `mode` value must be set to *closing*. This method returns a NodeSet of all markables that are ended at the markable with the supplied ID. The markables returned are in markable level order: markables from lower levels are returned after markables from higher levels. If two markables from the same level start at the same position, the longer one is returned after the shorter one. This ordering is used to ensure that the markables are processed in an order allowing for correct embedding of markable handles.

5.4.2 Functions for Adding Markable Handles

There are several functions for adding left and right markable handles. The all have in common that they have to be called from within <xsl:template> elements that match markable elements. Some of these functions actually return a value (i.e. a string representing the handle to be added), some don't. The preferred way to call them is in the `select` parameter of <xsl:value-of/> functions.

- – **`addLeftMarkableHandle(String level, String id, String handle)`**:
  This function has to be called in a template with `mode="opening"`. It inserts the (potentially multi-character) string *handle* as a markable handle for the markable with the ID *id* on level *level*. The left-most character in *handle* is used for highlighting matching handles. The handle is inserted by the function itself (i.e. as the return value).
  – **`addRightMarkableHandle(String level, String id, String handle)`**:
  This function has to be called in a template with `mode="closing"`. It inserts the (potentially multi-character) string *handle* as a markable handle for the markable with the ID *id* on level *level*. The right-most character in *handle* is used for highlighting matching handles. The handle is inserted by the function itself (i.e. as the return value).
- – **`addLeftMarkableHandle(String level, String id, String handle, int highlight)`**:
  This function has to be called in a template with `mode="opening"`. It inserts the (potentially multi-character) string *handle* as a markable handle for the markable with the ID *id* on level *level*. The character at position *highlight* (1-based) in *handle* is used for highlighting matching handles. The handle is inserted by the function itself (i.e. as the return value).
  – **`addRightMarkableHandle(String level, String id, String handle, int highlight)`**:
  This function has to be called in a template with `mode="closing"`. It inserts the (potentially multi-character) string *handle* as a markable handle for the markable with the ID *id* on level *level*. The character at position *highlight* (1-based) in *handle* is used for highlighting matching handles. The handle is inserted by the function itself (i.e. as the return value).
- – **`addLeftMarkableHandle(String level, String id, int extent)`**:
  This function has to be called in a template with `mode="opening"`. It inserts a (potentially multi-character) handle of length *extent* as a markable handle for the markable with the ID *id*

on level *level*. The left-most character is used for highlighting matching handles. The handle is **not** inserted by the function itself: It has to be added explicitly by means of <xsl:text> or <xsl:value-of/> after the function call.

- **`addRightMarkableHandle(String level, String id, int extent)`**:
  This function has to be called in a template with `mode="closing"`. It inserts a (potentially multi-character) handle of length *extent* as a markable handle for the markable with the ID *id* on level *level*. The right-most character is used for highlighting matching handles. The handle is **not** inserted by the function itself: It has to be added explicitly by means of <xsl:text> or <xsl:value-of/> after the function call.

- **`addLeftMarkableHandle(String level, String id, int extent, int highlight)`**:
  This function has to be called in a template with `mode="opening"`. It inserts a (potentially multi-character) handle of length *extent* as a markable handle for the markable with the ID *id* on level *level*. The character at position *highlight* (1-based) is used for highlighting matching handles. The handle is **not** inserted by the function itself: It has to be added explicitly by means of <xsl:text> or <xsl:value-of/> after the function call.

- **`addRightMarkableHandle(String level, String id, int extent, int highlight)`**:
  This function has to be called in a template with `mode="closing"`. It inserts a (potentially multi-character) handle of length *extent* as a markable handle for the markable with the ID *id* on level *level*. The character at position *highlight* (1-based) is used for highlighting matching handles. The handle is **not** inserted by the function itself: It has to be added explicitly by means of <xsl:text> or <xsl:value-of/> after the function call.

5.4.3 Other Functions

This section lists miscellaneous functions that do not fit into any other section.

- **`addHotSpot(String text, String methodCall)`**:
  This function inserts into the display a clickable area (a so-called *Hot Spot*) containing the text *text*. Clicking the hot spot in the display will cause the method specified in the *methodCall* to be executed. Right now, only one method is defined to be used in hot spots.[6]

  - *playwavsound*:
    This method call can be used to create hot spots for playing .wav files associated with the annotation. The method needs the following parameters:

    * file name: The name of the file to be played.
    * start: The position (in seconds) in the file at which to start playback.
    * end: The position (in seconds) in the file at which to end playback.

    end must be larger than *start*. Right now, the method can handle 16 kHz .wav files only.

  methodCall must be a string composed of the above parameters (separated by space) in the order shown. E.g. to add a hot spot to play the portion of the file recording.wav from 3.763 to 6.73, the string must look as follows: 'playwavsound recording.wav 3.763 6.73'.

[6] Other functions, including ones for more flexible .wav and .mp3 playback, will be added in later versions.