Copyright © 1994-1995 Centre National de la Recherche Scientifique

Task 1.2
Integration, testing, and refinement

Task leader:

Nancy Ide

Table of Contents

1. Introduction

The LRE project MULTEXT was established explicitly to develop generally usable and reusable tools for text analysis, corpus encoding standards for mrkup of linguistic data, and a multi-lingual corpus marked up and annotated using these standards and tools, for use within the Language Engineering community. This necessitated the development of encoding and processing strategies which form a consistent and coherent whole, and which are widely reusable, flexible, and extensible. This document describes the overall processing and data architectures developed within the MULTEXT project, their rationale and interdependencies, and their specific formats.

In the final phase of the MULTEXT project, one of the most important and essential activities has been the testing, refinement, and final integration of all of the MULTEXT materials, including standards, corpora, and tools, into a harmonious and generally usable whole, and the development and use of tools to make the MULTEXT materials easily and maximally usable. This document also describes the testing and refinement of the MULTEXT standards and tools and the development of additional modules for the final package.

2. Integrated processing chain

MULTEXT has developed a chain of annotation tools for linguistic corpora. Typically, in language engineering applications, an original text is input to corpus annotation tools, which may accomplish tasks such as segmentation of texts into words and sentences, grammatical annotation, alignment of parallel translations, etc. The annotation tools produce annotation information (sentence and word boundaries, grammatical annotation, etc.). Annotation information must be added to the original corpus in some way which makes it easily stored, interchanged, and accessible and manipulatable by other annotation tools and, ultimately, by retrieval and exploitation tools.

Because a corpus is manipulated and enhanced by the annotation tools, the data architecture for the corpus and the process architecture underlying the annotation tools are obviously mutually constraining. Therefore, the processing architecture for the MULTEXT tools and the data architecture for the MULTEXT corpus have necessarily co-evolved during the course of their development, in order to ensure compatibility and consistency. The full integrated corpus annotation environment encompasses both the processing and data architectures. This section of this document presents an overview of these components.

2.1. The MULTEXT Processing Model

2.1.1. Design and architecture

All of MULTEXT's tools have been developed utilizing what has been called the "Software Lego® Approach". The fundamental idea underlying this approach is the development of a set of small tools that can be used alone or combined to create larger, more complex programs, rather than the development of a large, integrated system that is difficult to adapt or extend. Such atomic modules, or "software bricks" can be bundled to perform more complex tasks, as needed.

To implement its Software Lego® philosophy for tool design, MULTEXT adopted the operator/stream approach, as implemented in the Unix system, which has already been widely used in the language engineering community (see for example, Liberman and Marcus, 1992), as the basis for its processing model. The object-oriented paradigm, which is very popular at the moment in large-scale development, could have offered the modularity needed to implement the Software Lego® approach. However, it usually imposes:

an object class system common to a project;
complex data types and use of APIs;
programming language dependence, at least for the time being (e.g. C++);

These constraints were determined to be too strong in the context of heterogeneous distributed development (that is, for the broader community, beyond MULTEXT), especially with non-expert intervention (for assembling of modules, development of new modules, customization, etc.).

In the operator/stream approach, data flows in uni-directional "streams" (or "pipelines") between tools.[1] Each of these tools is an "operator" (or "filter") that transforms the data as it passes by. Component tools are independent, and at no point are they compiled together in a single program. This approach has several advantages:

each tool can be implemented in a different language, developed by different people, tested independently, etc.
new tools can be plugged into the stream as needed, and tools are re-usable in other contexts.
the choice of Unix as a platform enables exploiting the use of built-in Unix tools and utilities.
Unix scripting facilities enable rapid and easy bundling of tools, easy modification and prototyping, etc.

This approach is readily applicable in the MULTEXT context. For example, the text segmentation process is accomplished with a set of micro-tools like the following (see deliverable ???):

[1] In Unix, streams are linear. We use the term "stream" here in a more general way, since non-linear streams can be emulated in Unix using temporary files.

mtsegspace	Produces a rough tokenization by segmenting the text at each space and tabulation.
mtsegpunct	Strips punctuation attached to tokens on the left or right.
mtsegfeatures	Detects and marks the typographic features of the token (uppercase letters, final period, etc.).
mtsegabbrev	Handles abbreviations.
mtsegclitics	Separates proclitics and enclitics from word itself (e.g., in French, "sais-je" = "sais" + "je").
mtsegcompound	Detects and recombines some multiword units (e.g. fixed locutions such as "parce que" in French).
mtsegnumdigit	Handles numbers written in digits, and recombines them if necessary.

etc.

These tools can be pipelined together to accomplish the full segmentation of a text. At the same time, each tool is small, self-contained, and therefore easy to understand, debug, and modify. A given tool can be easily replaced or deleted from the chain, and additional tools can be introduced anywhere in the chain as required to serve specific user needs. The only requirement to use the tools effectively is to understand the input and output format of each tool; therefore, it is necessary to provide a formal description of the I/O formats for the tools, discussed in the following section.

2.1.2.Tabular formats

1.1.1.1. Logical formats

Tables (or "relations") are familiar in the database world, consisting of rows and columns corresponding to specific fields of data. Relational databases map readily into Unix sequential files. Rows in the tables correspond to Unix lines, columns correspond to fields. A simple example:

ID	TOKEN	POS	LEMMA
1	La	DMF	le
2	petite	AFS	petit
3	porte	NCFS	porte
...			

In a non-disambiguated corpus, you may have redundancies in the rows of the tables, as in the following:

ID	TOKEN	POS	LEMMA
1	La	DMF	le
1	La	PPJ	le
2	petite	AFS	petit
2	petite	NCFS	petit
3	porte	NCFS	porte
3	porte	VM1S	porter
3	porte	VM3S	porter
...			

To avoid this redundancy, values can be "packed" in a non-first normal form relation (NF^2), that is, a relation in which values can be either atomic values or sub-tables (Abiteboul and Bidoit, 1986; Roth et al. 1988). A new table containing sub-tables can be constructed on this basis:

<table>
<tr><th>ID</th><th>TOKEN</th><th colspan="2">(POS LEMMA)*</th></tr>
<tr><td rowspan="2">1</td><td rowspan="2">La</td><td>DMF</td><td>le</td></tr>
<tr><td>PPJ</td><td>le</td></tr>
<tr><td rowspan="2">2</td><td rowspan="2">petite</td><td>AFS</td><td>petit</td></tr>
<tr><td>NCFS</td><td>petit</td></tr>
<tr><td rowspan="3">3</td><td rowspan="3">porte</td><td>NCFS</td><td>porte</td></tr>
<tr><td>VM1S</td><td>porter</td></tr>
<tr><td>VM3S</td><td>porter</td></tr>
<tr><td>...</td><td></td><td></td><td></td></tr>
</table>

The notation (ID TOKEN (POS LEMMA)*)* denotes the type of the table in the example above, which consists of three attributes. The notation (POS LEMMA)* denotes a single (relation-valued) attribute. This type of notation will be used to describe the input and output format of tools (see for example Deliverable 2.2.1).

Sub-tables such as those in the example above can be considered (and processed) as a whole. We provide below a Unix representation for embedded tables that packs sub-tables in a single field. This enables skipping a sub-table altogether, performing a **sort**, a **join**, a **uniq**, etc. on an entire sub-relation, building an index on a sub-table, etc. We limit the nesting of relations to one level, which is a reasonable compromise between added expressiveness and added notational complexity. In addition, we have not yet found examples in the chain of annotation tools where a deeper embedding would be useful.

1.1.1.2. Physical formats

There is a substantial inconsistency within Unix commands and tools as to what constitutes a field or (conversely) what constitutes a separator. The space vs. tab problem is well known. In addition, tools do not always recognize 8-bit characters, and lines have various length limitations.

We therefore have defined the following conventions for (possibly nested) tables :

Each row is a line.
(Top level) separator is a tab.
Subtables are coded using '\' (backslash) as field separator and '|' (pipe) as row separator.
Fields cannot contain end-of-line, tab, backslash, pipe or space.
Fields can use any other printable ISO 8859-1 character.
When needed in a field, space must me recoded as underscore (e.g. New_York).
If end-of-line, tab, backslash, pipe or a non ISO 8859-1 character is needed in a field it should be coded using an appropriate SGML entity.
Line length is not limited.

Example:

```
1   La        DMF\le|PPJ\le
2   petite    AFS\petit|NCFS\petit
3   porte     NCFS\porte|VM1S\porter|VM3S\porter
...
```

Implications for the tools:

they should process 8 bit characters;
they should accept long lines;
separators should be re-definable;
character counts, etc. must be adapted to SGML entities.

Many public tools (e.g. GNU) satisfy the first two or three constraints. The use of SGML entities in the stream is very unfortunate, but seems unavoidable--there will always be some character that does not belong to the chosen character set.

1.1.1.3. Example

At the moment, the segmentation and lexical lookup tools use the tabular formats defined in figure 3 for input and output. The atomic-valued fields and the sub-tables are as follows:

ACT	action associated with a token
CLASS	class
CNAME	class name
COM	comment
CORPTAG	tags specific for corpus processing
CPOS	vector of possible positions of a token in a compound
FEAT	typographical features of tokens
LEM	lemma
LEXDESCR	lexical description
POS	position of a token within an SGML element
TREELOC	location of element in SGML tree
TOK	token

CORPINFO	= (LEM CORPTAG)*
LEXINFO	= (LEM LEXDESCR)*
REF	= (TREELOC POS)*

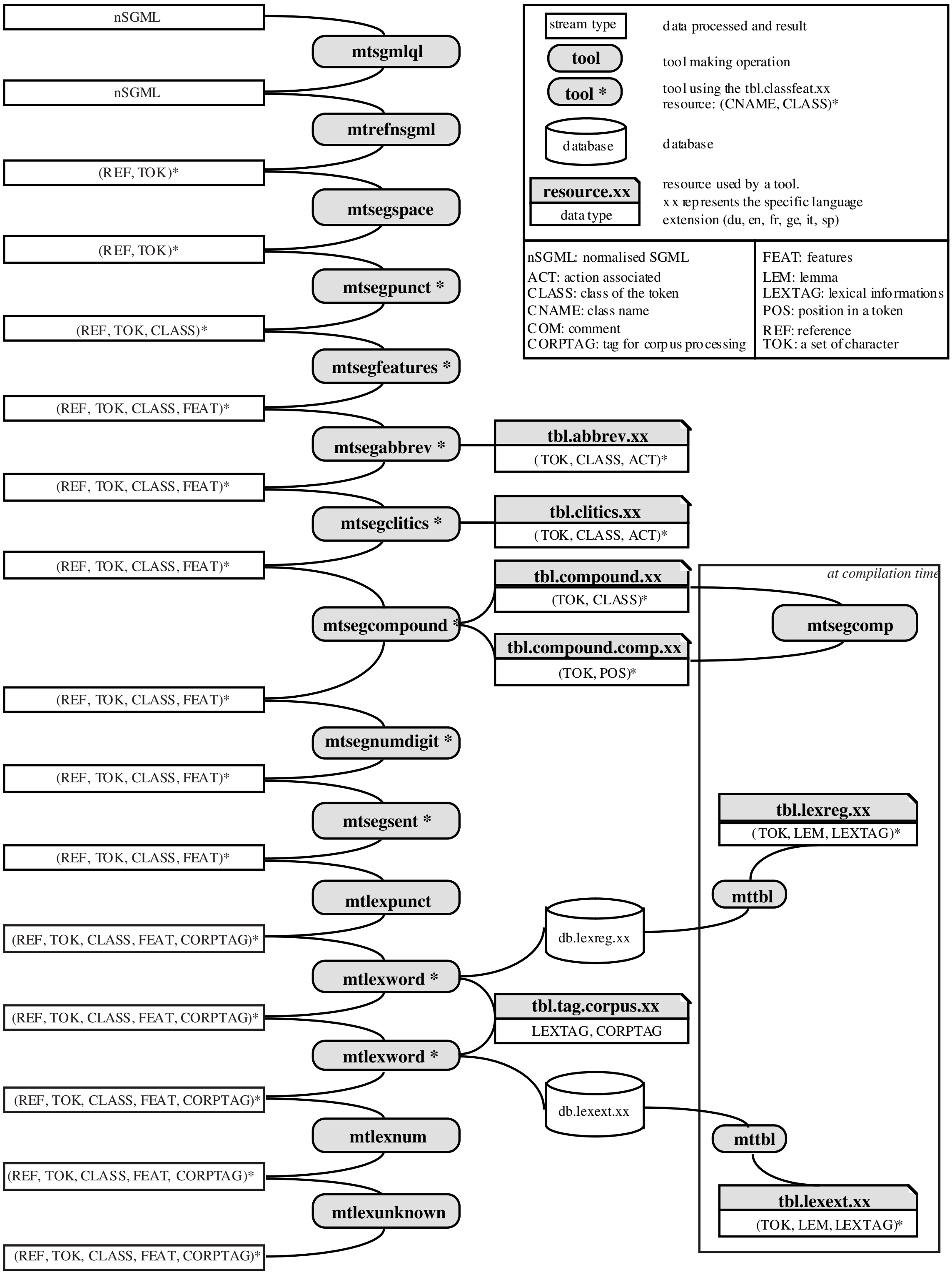

Figure 3. Tabular formats for a subset of tools

2.2. The MULTEXT document handling model

In MULTEXT, original documents are CES-compliant SGML documents marked for gross structural features ("level 1", i.e. down to the level of paragraph--see deliverable ???), since SGML has become widely accepted as a text markup scheme in the language engineering community.[2] It is essential to be able to retain all the information in the SGML original--including information about structure, rendition, etc.--throughout the annotation process, for use by the exploitation tools and possibly even the annotation tools themselves.

When the tools process documents originally encoded in SGML, it is very important for many applications that the SGML structure is fully preserved. For example :

¶ **Translation:** it might be necessary to reproduce the document in the target language with the same features (divisions, layout, etc.) as the original.

- **Navigation and display:** it would be useful to display the document in its original format and have hypertext-like capabilities based on the added annotation, e.g.:

  - show the results of searches in the actual document (e.g.: highlight all the instances of the some POS or pattern of POS in the text);
  - click on a word and get the morphosyntactic information;
  - select a fragment and have the corresponding fragment highlighted in the parallel text;

  etc.

, **Structure-sensitive operations:** it is often desirable to work with selected portions of a text, e.g.:

  - count words in some element types only (e.g. excluding titles);
  - use a different tagger (or different parameters) on titles;

---

[2] Deliverable 1.5.1 defines a preferred corpus markup style which is compliant with the guidelines developed by the Text Encoding Initiative (TEI), but any SGML document can be handled. In addition, since some users potentially have "plain text" (without explicit tags) or texts in other formats (LaTeX, etc.) that can be converted to plain text, Multext tools also allow plain text input. In the following discussion, we consider SGML input only.

- find mutual information between abstracts and body of text;

etc.

Also, MULTEXT utilizes CES-compliant SGML for the exportation of data, thus dictating that all annotation be ultimately expressible in SGML markup.

The need to retain and access information about the structure of a document throughout the annotation and exploitation processes places certain constraints on the processing model, especially for corpus annotation, which is modifying the information in the corpus. If there were no need to retain and handle the structural information about the text, it would be possible to use a simple record-field approach to represent the corpus information, as in the SUSANNE corpus (Sampson, 1992) and ET10-63/CRATER (McEnery and Daille, 1993) projects. However, document structure in SGML is hierarchical, and is therefore at odds with the record/field approach inherent in Unix stream I/O. Therefore, MULTEXT developed a hybrid document processing model to enable the two to work together.

2.2.1. The hybrid model

There are at least two ways to retain information about document structure throughout the corpus annotation process. One means is to pass the entire SGML original document, together with any markup added by previous phases of annotation, to each tool in the chain. However, this approach has several disadvantages. For processing, there are the following problems:

each tool must process the entire SGML tree, even if only certain portions of it, and certain information from it, are relevant for the analysis. This can involve potentially complex analysis of the DTD and complete parsing of the SGML tree.

portions of the text with which the tools are working (e.g. titles only) must be re-selected from the full document by each micro-tool. Such "queries" can be arbitrarily complex (e.g., select **<p>** only in **<div2>** elements, and not in **<footnote>**, etc.), demanding each tool has access to an interpreter of a language in which to express the queries.

The overhead of this kind of processing is unacceptable if it must be repeated for each of the MULTEXT micro-tools. The only way to avoid this overhead would be to pre-compile sets of micro-tools together (for example, the suite of segmentation tools given above). However, this grossly violates the MULTEXT atomicity principle: it precludes the possibility of skipping, replacing, or adding a given micro-tool without re-compiling. Adding micro-tools within the chain in the form of simple Unix scripts would be impossible. Adding even the most trivial tool between pre-compiled suites of micro-tools demands re-parsing the SGML document; even if an SGML API is provided to perform the parse, the range of programming language options is limited to full-blown languages such as C, and some non-trivial level of programming expertise is required. All flexibility is lost.

The all-SGML approach also locks the user into an all-SGML environment:

the tools cannot be readily used outside the SGML context, i.e., they cannot be readily used on non-SGML texts.

the ability to use and interject simple Unix tools in the chain is lost, since these tools are record/field-oriented. In addition, like most tools they necessarily operate on the entire document and are unable to select certain pieces based on a knowledge of the SGML tree. The only way to get around this problem is to re-write all Unix utilities for SGML-awareness. This is an interesting project, but it is beyond the scope of MULTEXT.

The strategy adopted in MULTEXT to retain document structure avoids the problems with the all-SGML approach outlined above. It is based on the following observations:

most corpus annotation and exploitation tools are inherently table-to-table oriented (e.g., input a token, output a set of part of speech/lemma pairs; input part of speech/lemma pairs, output the correct one; input a list of words, output a frequency list; etc.).

most tools do not need deep knowledge of the SGML tree or, indeed, any knowledge of the SGML tree. They might need to know where a paragraph or sentence boundary (as a "milestone") occurred, but not its relation to other elements in the SGML tree (e.g., that it occurred in the fourth paragraph in the third section, etc.).

in many cases it is desirable to process only some pieces of a document at a time. For example, it is typical to process only the running prose (e.g., content of **<p>** elements) in a document when doing part of speech tagging, or, alternatively, only titles (since grammatical patterns in the two may be very different).

Therefore, MULTEXT has adopted the following model (see Figure 2):

Select the right "chunks", using a query tool (see Deliverable ???) to enable users to precisely specify the pieces of the document in which they are interested. A cross-reference to the location of the selected chunk(s) in the original SGML tree is retained.

Convert the chunks and cross-references to tabular format.

Pass the chunk(s) and cross-references to the desired chain of table-oriented tools and/or Unix tools.

At any point, input from or export to SGML, for interchange, to use with SGML-aware tools, etc.

Note that the tabular format is not intended to be a translation of all information in the SGML document (which, as noted above, is not needed by most tools). At the same time, it is important to note that the full SGML structure is retained in the stored data and can be accessed at any point in the chain; and further that this structure is enriched by the addition of the annotation information when it is converted back to SGML format.

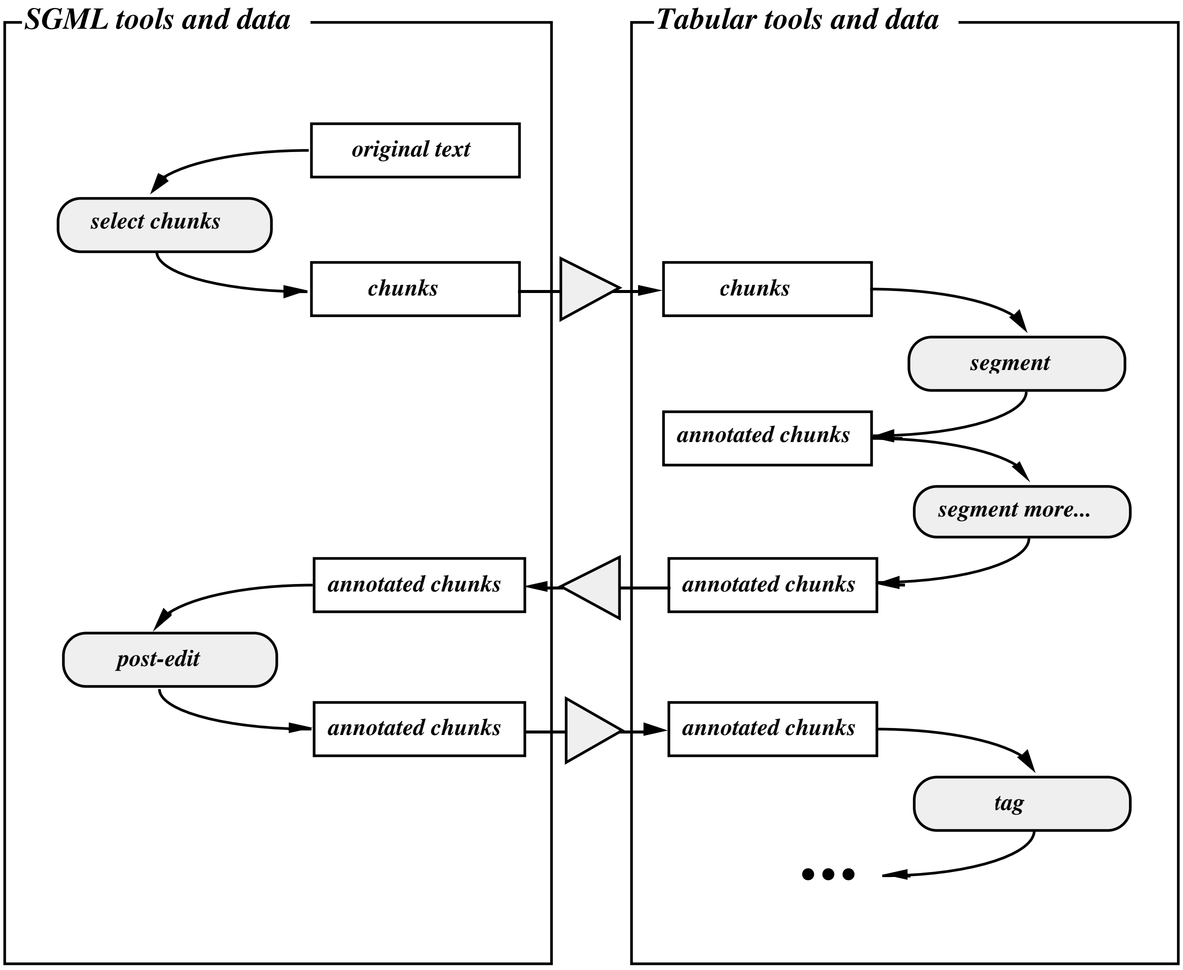

Figure 2. Hybrid model

This model has the following advantages:

simplicity of tool input and output, making it easy to read, debug, etc., as specified in the MULTEXT software principles;

ability to use Unix functions, utilities etc. anywhere desired in the chain;

input to and output from the tools can be stored in SGML format, which is rapidly becoming an interchange standard for corpora;

option to use SGML tools on the data after storage in SGML format;

non-SGML input and output is possible for users without data in SGML form, or who are not interested in retaining annotation results in SGML format.

Let us assume the following normalized SGML document (nSGML)[3] :

```
...
<TEXT LANG=FR>
<BODY>
<DIV0 TYPE=DOCUMENT>
<DIV1 TYPE=article N=4>
<TITLE>
Implantation à Eloyes
</TITLE>
<P>
L'usine, qui devrait être implantée à Eloyes (Vosges) représente un investissement
d'environ 3,7 milliards de yens. Elle fabriquera des pièces détachées pour la filiale
de Minolta en RFA.
</P>
</DIV1>
<DIV1 TYPE=article n=5>
<TITLE>
Projet Port-Mariane
</TITLE>
<P>
A Montpellier, après que le maire J. Besse ait annoncé la création du port fluvial,
se pourrait-il qu'aujourd'hui le projet Port-Mariane soit définitif ?
</P>
</DIV1>
</DIV0>
</BODY>
</TEXT>
```

The SGML extractor (*mtsgmlql*--see Deliverable ???), given an appropriate query, extracts the elements to be processed through the chain of annotation tools; e.g., to get all the paragraphs of our example, the query is

---

[3] Note that this example is drawn from the current Multext corpus. This corpus utilizes an interim encoding scheme which does not conform to the specifications in deliverable 1.5.1. This example conforms to the current format of the corpus, which has not yet been made consistent with the final markup specifications.

```
select
   <chunk>
       every PCDATA descendant_from p
   </chunk>
from p in every P descendant_from number( file( example ) )
```

The output of the extractor is also a text in nSGML format. For the example above, the result has the following form[4]:

```
<chunk>
<pcdata loc="2.1.1.1.2.1">
L'usine, qui devrait être implantée à Eloyes (Vosges) représente un investissement
d'environ 3,7 milliards de yens. Elle fabriquera des pièces détachées pour la filiale
de Minolta en RFA.
</pcdata>
</chunk>
<chunk>
<pcdata loc="2.1.1.2.2.1">
A Montpellier, après que le maire J. Besse ait annoncé la création du port fluvial,
se pourrait-il qu'aujourd'hui le projet Port-Mariane soit définitif ?
</pcdata>
</chunk>
```

The results of applying the segmentation tools can be translated back into SGML, yielding something like the following:

```
<tok from="2.1.1.1.2.1\1">L'</tok>
<tok from="2.1.1.1.2.1\3">usine</tok>
<tok from="2.1.1.1.2.1\8">,</tok>
...
```

2.2.2. Storage and interchange formats

1.2.2.1. Hypertext approach

As described above, data can be exported to SGML at any point during the annotation chain. Annotation information could be merged into the original document, or maintained separately. In MULTEXT, we have adopted a strategy whereby annotation information is not merged with

[4] Note that #PCDATA are considered here as pseudo-elements in the SGML tree, which is consistent with the HyTime philosophy. Therefore the extractor allows #PCDATA pseudo-elements to be wrapped in a **<pcdata>** tag and treated as regular elements.

the original, but rather retained in separate SGML documents and linked to the original or other annotation documents using the HyTime/SGML standard for hypertextual linkage (DeRose and Durand, 1994). The original or "hub" document remains stable and is not modified in any way by any process. This strategy has the following advantages:

it avoids the creation of potentially unwieldy documents--envision, in a worst case, a single document containing segmentation and part of speech markup, plus markup for alignment with translations in the eleven EU languages, plus alignment with the speech recording, plus variant part of speech taggings from several taggers, etc.!

it avoids problems with markup containing overlapping hierarchies, which are not allowable in SGML. For example, annotation added by a given tool (e.g., sentence boundaries markup) could conflict with markup in the original (e.g., markup for emphasis). Or, conflicting markup could be produced by different versions of the same tool (e.g., two different word segmenters could produce two different sets of word boundaries).

different versions of the same kind of annotation (e.g., different POS annotation) can be associated with the text.

it simplifies the input to the tools, since very few tools need access to all of the original information plus all of the annotation information, and in some cases would not be able to use all of this information (e.g., variant encodings produced by different versions of the same tool).

it is trivial to map the annotation information in tabular format, which contains information about the location of elements in the input SGML document, into a HyTime/SGML file containing this information and links to the input document. In this way, annotation information is added to the SGML document at its appropriate location.

it is very much in line with what is evolving in the SGML world, and it is likely that SGML/HyTime software which handles complex linkages will be available in the near future.

annotation can be accomplished by associating the SGML original or other annotation documents with other, pre-existing documents--e.g., instead of generating a document containing POS markup and linking it to the original, links could be made directly with lexicon entries.

it gives easy access to the original SGML document (or, as mentioned above, any among several versions annotated for certain features) for use by other applications.

2.2.3. Document linkage architecture

1.2.3.1. Overview

The hyper-document comprising each text in the MULTEXT corpus and its annotations will consist of several documents, each described below.

The base or "hub" document is the unannotated document which serves as input to the segmenter, containing only markup of types 0 and 1 (defined in Deliverable 1.5.1). The hub document is in a normalized SGML format (nSGML), as defined in the appendix. The hub document is "read only" and is not modified in the annotation process.

As mentioned above, the output of any tool in the chain can be exported to SGML format. Each exported document is a proper SGML document with a DTD, containing annotation information linked to its appropriate location in the hub document. Thus the annotation document contains what can be seen as "remote markup", virtually added to the hub. Some annotation documents will be permanent and retained for distribution, while others will be retained only temporarily (possibly, to use as input later on to additional annotation tools), at the user's option. Below we discuss the important annotation documents that can be created from the output of the tools.

Links between documents can be one-way or two-way. A one-way link points ***from*** a given location--i.e., the location at which the pointer appears in an SGML document--***to*** another location in the same or a different SGML document.[5] Two-way links associate two (or more) locations by pointing to both; the location of the two-way link itself is not dependent upon the

[5] Note the distinction between files, which are storage units, SGML documents, which may be composed of a number of files by means of external entity references, and HyTime hyper-documents, which consist of either one SGML document that uses HyTime links to connect parts of that document, or several SGML documents connected by HyTime links.

location of either of its ends. In MULTEXT, all annotation documents will be linked to the hub or other annotation documents using one way links. The exception is output of the aligner for parallel texts, which will consist of an SGML document containing ***only*** two-way links associating locations in two documents in different languages. The two linked documents are two annotation documents containing segmentation information, such as sentence boundaries. The overall architecture is described by figure 4:

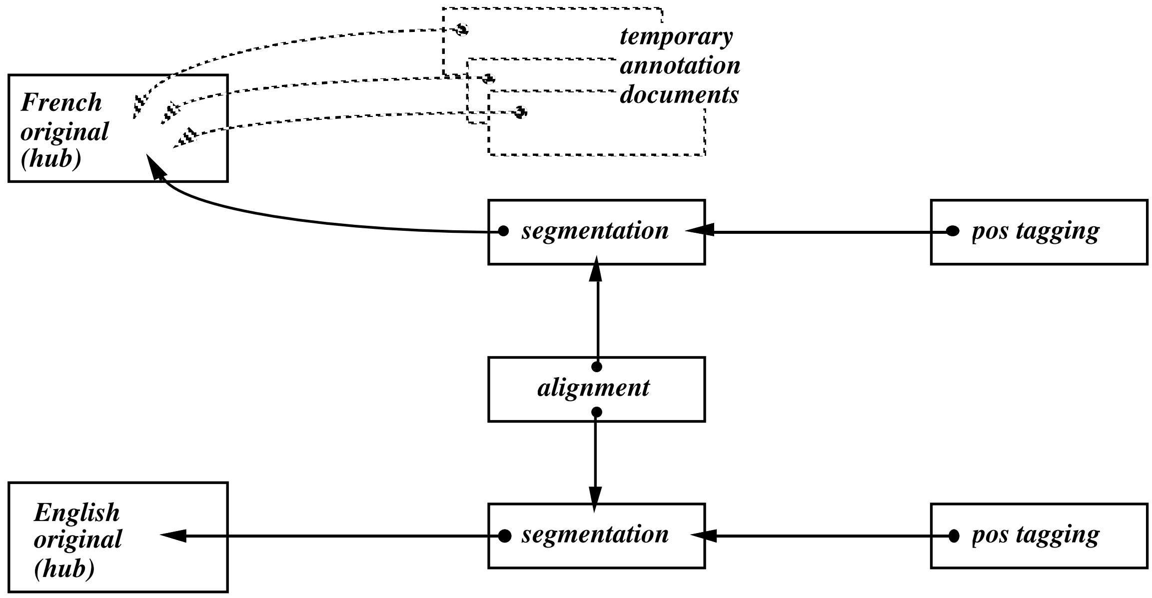

Figure 4. MULTEXT document linkage architecture

1.2.3.2. Segmenter output

The output of the segmenter consists of markup signaling sentence and word boundaries. Retaining segmentation markup in a separate SGML document with links to the original avoids problems of overlapping hierarchies--that is, invalid nesting of SGML elements--that are likely to arise when markup for sentence and word boundaries, which in itself forms a well-formed hierarchy, is superimposed on markup recording typographic phenomena recorded in the hub document, such as emphasis. For example, if the hub document contains the following markup:

```
It certainly wasn't <hi r=it>him. He</hi> would never do such a ...
```

segmentation could produce

```
<s>It certainly wasn't <hi r=it>him.</s> <s>He</hi> would never do such a ...
```

which is invalid SGML since the **<hi>** and **<s>** elements are not properly nested. See the discussion in Deliverable 1.5.1.

The information in the segmentation document will be associated with the relevant locations in the hub via HyTime and/or TEI links. Both HyTime and the TEI provide several different means to specify the location of the endpoints of a link. One of the most common is to point to a unique identifier, specified in the ID attribute on the target tag. However, since segmentation documents are pointing into the hub document, which is read-only, it may be necessary to modify the original by adding ID attributes to each tag, as a prior step. Another method is to use locations in the SGML tree to point to specific elements (e.g., the third child of the second child of the first child of the root). Using this method solves the read-only problem, eliminates the indexing needed for IDs, and provides a semantically meaningful location that might be useful for the tools.

For segmentation purposes, we also need to point to locations that are not the entire content of an SGML element--for example, to a specific character that is the start of a sentence. Both HyTime and TEI allow for mixing pointing mechanisms in the same location indicator. Therefore, segmentation markup will be linked to the hub document via one-way links, specified in two steps:

¶ tree addressing to point to the nearest enclosing tag for the location in question;

• reference to the position of a specific character from the location given in step ¶.

The segmentation document could consist of only the remote markup identifying word and sentence boundaries, punctuation, abbreviations, etc. with no tag content. For example, consider the following text:

```
...
<P>
L'usine, qui devrait être implantée à Eloyes (Vosges) représente un investissement
d'environ 3,7 milliards de yens. Elle fabriquera des pièces détachées pour la filiale
de Minolta en RFA.
</P>
...
```

The following two tags point to the first two words inside the **<p>** element:

```
<tok from="2.1.1.1.2.1\1" to="2.1.1.1.2.1\2">
<tok from="2.1.1.1.2.1\3" to="2.1.1.1.2.1\6">
...
```

We use here a short notation to specify the target location in the hub document. The notation "2.1.1.1.2.1\1" is equivalent to the TEI notation "CHILD (2) (1) (1) (1) (2) (1) STRLOC (1)". See Deliverable 1.5.1 for specification of addressing mechanisms.

The above markup is straightforward, but for the time being we recommend retaining the actual token string as the tag content, as follows:

```
<tok from="2.1.1.1.2.1\1">L'</tok>
<tok from="2.1.1.1.2.1\3">usine</tok>
...
```

This method duplicates some of the content of the hub document in the segmentation document; however, the segmentation document becomes self-standing. Since many tools will use this document as input, this has processing advantages. This method also allows eliminating the location of the end of the token, since this information is implicit in the string itself.

Another option would be to retain sentence markup in a document separate from the document containing token markup; however, we recommend retaining them in the same document to avoid the multiplication of documents and to provide a self-standing document containing both kinds of information, since sentence and token boundaries are very often used together. Since markup for sentences and tokens forms a well-formed tree, this presents no problem for a straightforward DTD.

```
<s from="2.1.1.1.2.1\1">
<tok from="2.1.1.1.2.1\1">L'</tok>
<tok from="2.1.1.1.2.1\3">usine</tok>
...
</s>
...
```

1.2.3.3. Tagger output

The output of the Tagger consists of a separate SGML document containing the annotation information generated by the tool--i.e., lemma and part of speech tags for the tokens in the text. We see two main options for retaining the annotation information generated by the tagger:

¶ the output of the tagger can be stored in a separate document and linked to the segmentation document with one-way links, as follows :

Segmentation document :

```
<s from="2.1.1.1.2.1\1">
<tok from="2.1.1.1.2.1\1">L'</tok>
<tok from="2.1.1.1.2.1\3">usine</tok>
...
</s>
...
```

Tagger document :

```
<gram from="1.1"><lem>le</lem><pos>DFS</pos></gram>
<gram from="1.2"><lem>usine</lem><pos>NCFS</pos></gram>
...
```

- since the tagger has all segmentation information available to it during processing, another option is to create a new annotation document containing the segmentation information as well as the POS category information. This new document is linked to the hub with one-way links. If a segmentation document already exists, this document could be discarded if desired.

```
<s from="2.1.1.1.2.1\1">
<tok from="2.1.1.1.2.1\1"><form>L'</form><lem>le</lem><pos>DFS</pos></tok>
<tok from="2.1.1.1.2.1\3"><form>usine</form><lem>usine</lem><pos>NCFS</pos></tok>
...
</s>
...
```

This second method avoids problems that may arise due to updating of documents. We can imagine, for example, that segmentation boundaries may be changed after part of speech tagging (e.g., at the post-editing stage).

1.2.3.4. Aligner output

The output of the aligner consists of a series of links between two documents, identifying corresponding segments (sentence, word) in two texts. Sentences, and words are easily adressable by their location in the SGML tree in the segmentation documents under analysis. The links between the two segmentation documents are stored in a third document :

```
<link targets="1.1;1.1">
<link targets="1.2,1.3;1.2"
<link targets="1.4;">
```

The first link shows a one-to-one mapping of two sentences; the second shows the mapping of two sentences in the first document on to one sentence in the second; and the third shows a sentence in the first document with no corresponding translation in the second. The exact mechanisms for specifying links are discussed in Deliverable ???.

3. Final testing and refinement

3.1. Corpus encoding standard

MULTEXT's Corpus Encoding Standard (CES) includes DTDs for primary data as well as annotation documents and provides guidance for the encoding of the various elements in linguistic corpora. A full description of the CES is contained in deliverable ????.

In order to fully test and refine the CES, a cycle of test and feedback was implemented to gain information about the encoding needs for primary data within the various projects. In addition to receiving input from projects with which MULTEXT explicitly collaborated, the CES was publicly disseminated via the World Wide Web over the past year in order to gain feedback from the general user community. Throughout this period, the CES has been modified and extended to accommodate user needs as they have become apparent.

3.1.1. Testing of the primary data specifications

The CES specifications for primary data have been tested an a variety of different text types, both within the MULTEXT project and in other projects:

within the MULTEXT project, it has been used to encode the Journal of the Commission Written Questions in five languages, including markup down to the paragraph level as well as some sub-paragraph markup for names, dates, etc.;

within the MULTEXT-EAST project, it has been used to encode data in the six Central and Easter European languages of the project (Czech, Romanian, Slovenian, Hungarian, Bulgarian, Estonian), for the following text types:

- fiction containing markup down to the paragraph level and in some cases substantial sub-paragraph level markup;
- newspapers

within the PAROLE project, the CES is being used to encode, down to the paragraph level, a wide variety of text types, including fiction, newspapers, poetry, etc.

3.1.2. Testing of the annotation data specifications

The CES includes DTDs for storing annotation information generated by the MULTEXT chain of annotation tools, including:

annotation documents containing sentence and token boundaries

annotation documents containing part of speech annotation

alignment information

The CES DTDs for annotation data were necessarily developed in step with the MULTEXT annotation tools, in order to ensure a clean mapping between tool formats and the SGML data formats. Development of the linkage mechanisms for linking the various documents in the SGML database was substantially influenced by the kinds of information available to and from the tools at various points in the annotation process.

Note that the CES specifications also provide means to handle data which have not been processed using the MULTEXT tool chain, or where standard SGML linkage mechanisms can be used. In particular, the CES DTD for primary data provides optional "escape mechanisms" which enable representing annotation information in the primary data, where it is necessary or desirable to do so. It also allows for the use of the SGML ID mechanism for inter-document reference, in cases where annotation documents and primary data are bundled together as a corpus in a common SGML document and this mechanism is therefore available.

3.2. Tool testing

All of the MULTEXT annotation tools have been extensively tested on texts of a variety of genres, including:

technical reports
transcripts of question/answer dialogues
fiction
newspapers
poetry

In addition, the tools have been applied to texts in the following languages:

English
French
German
Italian
Spanish
Swedish
Estonian
Hungarian
Bulgarian
Czech
Slovenian
Romanian

Tool testing has been accomplished in part via collaboration with other projects, including MULTEXT-EAST...???. In such instances the tools were applied to data and the results examined by language or other experts, and feedback was sent to the tool developers. The tools and/or resources were modified in order to treat reported problems, errors, omissions, etc.

All MULTEXT tools have been made available in pre-release version via the World Wide Web, freely accessible for non-commercial and non-military use. This has allowed for testing and feedback from the general user community.

3.3. Portability

The tools comprising the MULTEXT annotation tool chain have been installed at several sites, including UAB (Barcelona), Vassar College (USA), CNRS/Avignon (France), and the Josef

Stefan Institute (Slovenia), and on a number of different platforms: Solaris + BSD or System V, Linux, HP, and Silicon Graphics.

Intensive interaction between the installation sites and the MULTEXT tool developers enabled rapid modification and adjustment in order to enable successful installation in this variety of platforms, thus ensuring the robustness of installation for the final deliverables.

The project has utilized the Autoconf software from GNU (version 2.10) in order to enable the porting of MULTEXT tools to different Unix platforms. Autoconf is a tool for producing shell scripts that automatically configure software source code packages to adapt to many kinds of UNIX-like systems. The configuration scripts produced by Autoconf are independent of Autoconf when they are run, so their users do not need to have Autoconf.

The use of Autoconf demands an additional prior step before the compilation and installation of a given piece of MULTEXT software, involving the following commands:

```
$ ./configure
$ make all
$ make install
```

The first command (./configure) automatically configures the source code and the compilation procedure according to the particular system upon which the software is being compiled. The Makefile file (Makefile.env) containing the compilation command sequence is similarly adapted to the particular machine. This is accomplished by generating a header file (src/mt_config.h) that is included in the source code, and which includes all required system definitions. Once this is accomplished, the compilation (make all) and installation (make install) of any tool continues as under usual circumstances.

The configuration scripts produced by Autoconf require no manual user intervention when run; they do not normally even need an argument specifying the system type. Instead, they test for the presence of each feature that the software package they are for might need individually. As a result, they deal well with systems that are hybrids or customized from the more common UNIX variants.

Autoconf creates a configuration script from a template file that lists the system features that the package needs or can use. After the shell code to recognize and respond to a system feature has been written, Autoconf allows it to be shared by many software packages that can use (or need) that feature. If it later turns out that the shell code needs adjustment for some reason, it needs to be changed in only one place; all of the configuration scripts can be regenerated automatically to take advantage of the updated code.

3.4. Graphic interface

To make the use of the MULTEXT annotation tool chain simpler for the user, we have developed a minimal user interface, *Mtxchain*.

This tool enables the user to quickly establish a specific sequence of tools to be invoked, and provides means to examine the results of each process as the tools in the chain are applied.

Mtxchain allows access to four general MULTEXT tools:

pre-segmentation tools
segmentation tools
lexical access
tagger tools

Each of these tools is composed of a sequence of subtools, following the MULTEXT "software-lego" approach.

3.4.1. *Mtxchain* overview

Mtxchain provides the following general facilities:

invocation of a process, given a chosen input file and a user-specified chain of tools, from the following set:

- SGML : invokes *mtsgmlql* for extracting parts of an SGML document to be processed
- SEGMENTATION : invokes the *MtSeg* tool set
- LEXICAL_LOOKUP : invokes the *MtLex* tool

display of the results of the last tool applied
choice of language (default: French)
display of tool resources required by the tools

user specification of the following:

- the *data directory,* where tool resources are stored (default: /usr/multext/mttools/data)
- the *results directory,* where output files are placed (default: current directory)
- output file basename : names of output files are composed of a *basename* followed by "." and the name of the last tool that processed it (for example, text.mtsegsent) (default: 'text')

display of help information
display the input file
side-by-side display of tool input and output, and facility to display differences between them

Because a graphic interface was not specifically provided for in the MULTEXT deliverables, this interface is necessarily minimal and requires continued work in order to make it completely robust.

References

Abiteboul, S. and Bidoit, N. (1986) Nonfirst normal form relations: an algebra allowing data restructuring. *J. Comput. Sys. Sci.* 33, 3, 361-393.

Liberman, M., Marcus, M. (1992). *Tutorial on Text Corpora,* presented at the Association for Computational Linguistics Annual Conference.

McEnery, T., Daille, B. (1993) *Database design for corpus storage: the ET10-63 data model.* ET10-63 report.

DeRose, S.J., Durand, D.G. (1994) *Making HyperMedia work: A users's guide to HyTime.* Kluwer Academic. Boston.

Roth, M. A., Korth, H. F., Silberschatz, A. (1988). Extended algebra and calculus for nested relational databases. *ACM Transactions on Database Systems,* 13, 4, 389-417.

Sampson, G. (1992). *The SUZANNE corpus.* University of Sussex, Bringhton.

Appendix

Normalized SGML format (nSGML)

SGML is a very complex language that enables the encoding of the same facts in many synonymous ways. Some SGML constructs are extremely difficult to process (entity replacement, minimization, etc.). MULTEXT has therefore adopted the approach that all SGML documents be rendered into a normalized SGML format (nSGML), which is valid SGML with the following additional constraints:

the defined character set is ISO-8859-1;

tags and corresponding attributes appear on the same line;

each line starts at the first position (i.e., no leading blanks or tabs);

no carriage returns should appear in #PCDATA (note that carriage returns in legacy documents should have been interpreted as paragraph breaks, etc.);

no markup minimization is used (in particular, all closing tags are present);

all entities are expanded except for character entities;

no character present in character set should be represented by an entity (e.g. "é" must not be represented by "é").

Note that this specification for nSGML is preliminary and is subject to further refinement.