Emaxml

An Emacs mode for editing XML

Paolo Debetto

Supervisor: Dr. Joe Wells

CS4 Dissertation

Deliverable 2

Contents

Chapter 1

Introduction

Emaxml is an extension of Emacs, written in Emacs Lisp, to edit XML documents. Major Emacs modes for editing SGML and XML already exist; this is different in that it allows viewing the document as a tree structure, both visually and logically.

1.1 Aims and objectives of the project

An XML document is often generated automatically by an application. Nevertheless, in many occasions XML code is edited directly by a human author. When a normal text editor (i.e. one with no XML-specific editing facilities) is used to this end, the author's creativity has to deal with the XML document at three levels:

1. At the **contents level**, the author is concerned with what the document is about, the actual information or concepts.

2. At the **structure level**, the author organises the document hierarchically, according to the rules set by the DTD for that particular class of documents.

   For non-trivial documents the overhead activity involved with keeping the structure in order or with changing the current structure can be very expensive.

   Moreover, the author has to be concerned with indentation or some other means to visually see the structure of the document.

   However, this activity is related to the conceptual contents of the document.

3. At the **syntactic level**, the author is concerned with getting the XML syntactic sugar right. This activity is strictly XML-related and has nothing to do with the topic of

the document. It is an error-prone activity and the overhead involved can be very expensive.

Obviously most of the work mentioned in the previous section can be automated to various degrees by an editor with XML editing facilities, to the purpose of letting the author concentrating on the contents and the structure of the document abstractly.

The approach of Emaxml is that of taking care of the XML syntax and providing means of seeing and manipulating the structure of the document effectively, by displaying the document in a tree-like fashion.

Figure 1.1 shows Emaxml at work.

The final concrete objective is to implement a fully functional Emacs mode, with a limited number of functionalities, but designed so that it can easily be limitlessly improved by anyone who might possibly want to work on it later.

1.2 Fundamental concepts of Emaxml

This chapter is a brief reminder of the concepts central to Emaxml specification, detailed in Deliverable 1[1].

Figure 1.1 shows an XML document displayed by Emaxml. The parts of the display are called **logical units**; their properties influence the response of Emaxml to the user's input. A list of the types of logical units is in Appendix A.

Not all locations in the display belong to a logical unit. For instance, the colored spaces at the left of an Element name or the colon following an Attribute name are not part of any logical unit. All such locations and their contents are said to be **automatic**, because they are managed by Emaxml and cannot be reached by the user. Non-automatic locations of the buffer form the **user space** of the buffer. The user may place the cursor on some automatic locations, namely where such a location is immediately on the right of a user location. These locations are therefore not completely automatic, are called **ubiquitous**, and allow the insertion of new text. For example, the colon in an Attribute is ubiquitous and when the cursor is over it any text inserted is appended to the relative Attribute name.

A **logical line** is either one line (up to a newline) of a multiline logical unit or an entire monoline logical unit.

The **previous** logical line with respect to a logical line is the first logical line that is encountered by going left and up in the display. The **following** logical line with respect to

[1] The terms in **bold** are specific to Emaxml.

a logical line is the logical line that is encountered by going right and down.

The topmost element is called the **seed element**[2], and does not correspond to an actual XML element; it contains information relative to the document, such as that contained in the XML declaration. From the user's point of view, the header of the seed element is treated as a normal header, whose attributes are the said information.

An element (and the relative subtree rooted at it) can be displayed **outline** or **inline** (that is, vertically or horizontally) and **expanded** or **collapsed** (that is, completely visible or displayed as the element name only).

These characteristics are independent so there are four ways of displaying a subtree (see Fig. 1.2), called **display modes**: outline-expanded, outline-collapsed, inline-expanded, inline-collapsed.

The **display state** of a subtree is defined by the display mode of all the elements it is formed by.

1.3 Changes to the original project plan

The most important change to the specification of Emaxml as described in Deliverable 1 regards the set of logical units which has been modified as follows:

- CDATA logical units have been eliminated.

  A CDATA section in an XML document serves the purpose of letting the author write any sequence of characters without having to "escape" them as entity references.

  In the previous specification, Emaxml had a logical unit for CDATA sections. The change is that now the author writes whatever s/he wants as character data, and the Writer will codify the character data optimally using combinations of CDATA sections, entity references, character references and plain character data.

- Entity reference logical units have been introduced.

  Entity references are a sort of macro facility used in XML. They are substituted with the text they refer to when the specific application processes the information contained in the XML file.

  An Entity reference logical unit is an elementary, special, monoline, primary logical unit, according to the definitions of these properties given in Appendix A.

[2] Because it comes before the root element...

This changes have affected the design of the Parser, the Writer and the Emaxml mode; in particular, the hierarchy of types (the XD data model) that governs the main data structure (the Etree) has been changed accordingly.

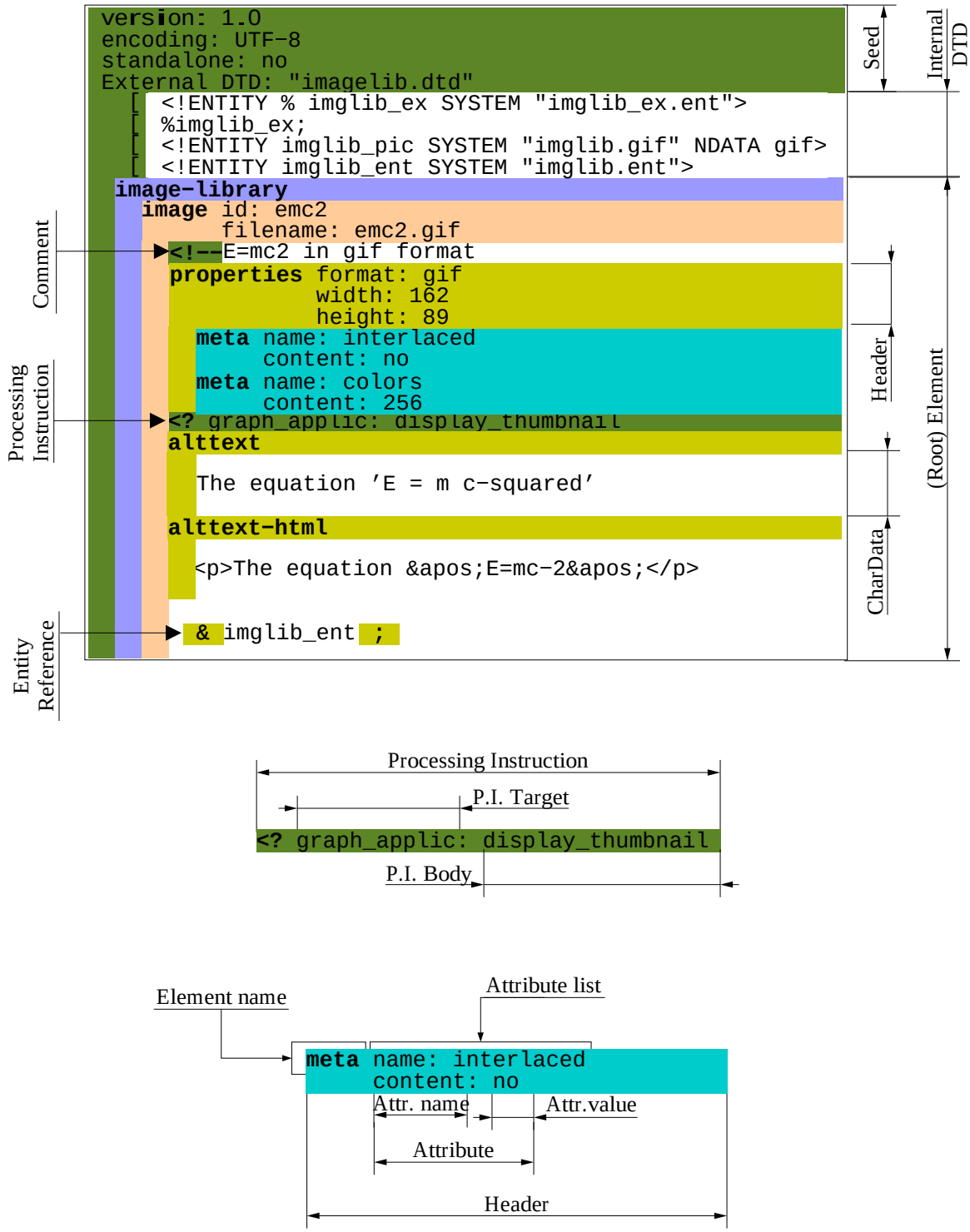
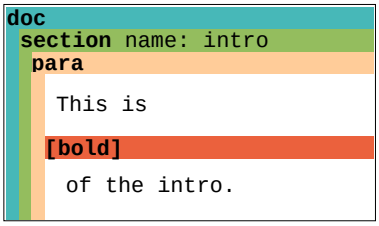
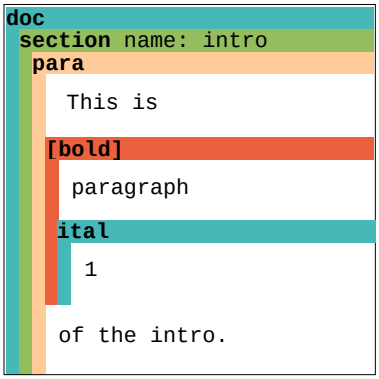


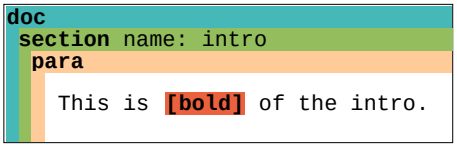
Figure 1.1: Screenshot of Emaxml, with the indication of the Logical Units.



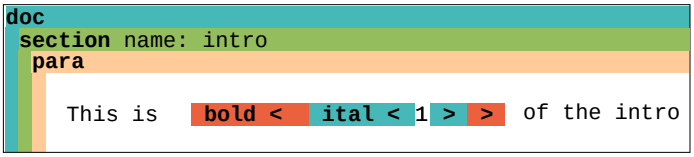
Outline Collapsed



Outline Expanded (default)



Inline Collapsed



Inline Expanded

Figure 1.2: Display modes

Chapter 2

Functional specification of the system

The expected functionality of the system has been set in Deliverable 1; what follows is a revision of it, according to the feedback from the readers and discussion with the project supervisor.

The functionalities are divided in two main categories: standard Emacs operations revisited for Emaxml, and Emaxml-specific operations.

2.1 Standard Emacs editing operations *à la* Emaxml

The typical Emacs user will expect a number of standard editing facilities to be available in Emaxml mode, and their behavior to parallel that of other modes.

It is not as important to implement a large number of functionalities immediately as it is to design the code in a modular fashion and document it properly, and choose at least a few operations from each of the following categories:

- **PM**: point movement;
- **ID**: insertion and deletion;
- **MK**: mark operations;
- **RE**: region setting;
- **SR**: search & replacement;
- **GE**: general common operations (e.g. 'undo').

Appendix B lists a set of commands that may be implemented for Emaxml as part of this project.

2.1.1 Visiting a file

This section has not changed from Deliverable 1.

2.1.2 Saving a file

When a file needs saving, the Writer is invoked to produce the XML code from the internal representation of the document.

Also, the Internal DTD is syntactically checked before the file is saved, because it may contain sequences of characters that would make the written file illegal[1]. In the case that the Internal DTD is recognised as incorrect, the file will be saved without it, and the user will be advised so that s/he can take whichever action s/he thinks appropriate.

2.1.3 Highlighting the region

When using the mouse to highlight the region, or in Transient Mark mode[2], only the locations of user space included in the region will be highlighted.

However, when a primary logical unit is completely included in the highlighted region, all of its physical space will be colored, including the non-user space, otherwise only the included user space will be colored.

Two useful commands are implemented in Emaxml: mark-more and mark-less.

Mark more is to expand the region to the next bigger logical unit containing the region (or containing point if mark is undefined).

Mark less does the opposite. Calls to mark-less always select the first child when there is more than one.

For example, if the region is currently a string in an Attribute value, mark-more sets region to the containing Attribute. Further calls to mark-more set region to the containing Attribute list, then to the Header, then to the Element, and so on up to the entire buffer.

[1] Consider for example an Internal DTD containing `...]]> <root-element>...`. If this is written without being checked, the file will be subsequently unparsable.

[2] In Transient Mark mode, when the mark is active, the region is highlighted.

2.1.4 Text search and replacement

Text search and replacement in Emaxml will operate on the user space only. Both string and regular expression search and replacement will be implemented.

2.1.5 Moving point

This section has not changed from Deliverable 1.

2.1.6 Mark and the mark ring

The mark ring will be implemented in Emaxml, consistently with its standard definition and functionality. Obviously, the internals relative to such implementation will be specific to Emaxml; markers have to be objects of a different data structure, since they must describe a location in terms of the tree.

An important property of a marker that must be preserved is that it "moves" with the position in the buffer to which it is pointing, i.e. if some text is added or deleted before that position, the marker always points to the same character.

2.1.7 Killing and yanking

Killing and *yanking* in Emacs jargon mean *cutting* and *pasting* respectively.

In standard Emacs operation when a portion of the buffer is killed it is deleted from the buffer and stored in the kill ring for later use. One property (I shall call it **yanking transparency**) of an Emacs buffer is that if some text is killed and immediately yanked, the buffer does not change. Point may be in a different location, though.

Yanking is an insertion operation; the portion of buffer being yanked is inserted before point.

Killing in Emaxml

Many Emacs commands exist for killing. The ones considered here are: kill-line, kill-region, kill-word, kill-sexp, kill-buffer. Their names are self-explanatory, (a part from kill-sexp, maybe, which in Emaxml kills the current element). Kill-line kills a logical line instead of a normal line.

Since the object of the editing in Emaxml is a tree, the killing and yanking operations must be redefined in terms of logical units.

Let us consider a portion p of an Emaxml buffer being killed that starts from location l_0 in primary logical unit u_0 and ends at location l_1 in primary logical unit u_1.

When p is killed, it is stored in the kill ring.

If p is a string, it is stored as such, whith no information regarding wich logical unit it was part of. On the other hand, Emaxml stores killed logical units and illogical units as *trees*.

This stored tree is:

- If p is an entire logical unit, the tree rooted at u_0, including all its children.
- If p is an illogical unit, the tree rooted at the smallest logical unit s that include all the logical units totally or partially in p, and that does not include the children of s which are not in p and the portions of user space of s not in p.

  This implies that the stored tree may have some blank logical units in it.

Killing p has the following effects:

(1) If p is a string, point is left at l_0.

(2) If p is a primary logical unit then point is left:

  - if p has a peer q below it, at the first character of q's user space;
  - otherwise if p has a peer q above it, at the first character of q's user space;
  - otherwise at the first character of p's parent's user space.

(3) If p is a secondary logical unit, then it must be one of the elementary logical units in an Element or in a Processing Instruction; a blank logical unit of the same type as p is inserted in place of p with point at its first location.

(4) If p is an illogical unit, then point is left as in policies (1), (2), (3) but substituting "the portion of u_0 in p" for p.

  When killing an illogical unit, Emaxml rebuilds the tree as follows, in order:

  (i) all primary logical units entirely included in p are pruned;

  (ii) all secondary logical units entirely included in p are blanked and possibly removed;

  (iii) all characters in p are eliminated.

Yanking in Emaxml

The following are the main general properties of yanking in Emaxml:

- The object p to be yanked is either a string or a subtree in the kill ring.
- Yanking leaves point after the last location of the user space of the yanked unit, i.e. the cursor is left under the first character of the user space after the yanked unit.
- If p is a string, it is yanked at point. This involves applying low-level control[3].

Yanking a non-string unit involves somehow tree manipulation. The only limitation posed by Emaxml is that secondary logical units (such as Attribute name or Processing Instruction Target) cannot be yanked other than in an appropriate compound logical unit[4].

Since both logical and illogical units are complete subtrees, there is no difference in yanking them.

Let us consider a unit p stored by Emaxml in the kill ring.

If p is a primary logical unit, then yanking transparency does not hold [5], so p must be explicitly yanked as a peer or as a child. If yanked as a peer it is inserted before the current primary logical unit, if yanked as a child it is inserted as its last child.

The Emaxml policies for yanking p into some logical unit u_2 are described in Fig. 2.1.

p is a	**u_2 is a**	**Yanking p into u_2**
string	elementary logical unit	p can be yanked anywhere in u_2
primary logical unit	Header	p can be yanked either as child or as peer of the element which u_2 belongs to
primary logical unit	primary logical unit	p yanked as a peer of u_2
Attribute list or Attribute	Header	p inserted at end of u_2's Attribute list

Figure 2.1: Yanking criteria for p not illogical.

2.2 Emaxml specialized editing operations

Appendix C lists the minimum set of new commands specific to Emaxml to be implemented as part of this project. They are described in the following sections.

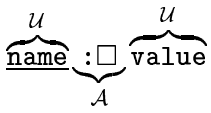
[3] See section 2.3.1 for definition of low-level control.

[4] This could be improved in future by giving a meaning to such operations. For example, yanking a secondary logical unit s of type t into an unappropriate primary logical unit u may be defined as inserting a new appropriate primary logical unit as a sibling of p, blank but with the field t equal to s.

[5] See Deliverable 1 for a discussion of this.

2.2.1 Creating a new instance of a logical unit

Creating a new instance inserts a **blank logical unit**. A blank logical unit is an instance of a logical unit with the strings referring to the user space empty. For example, an Attribute is of the form:



where '□' indicates a space, '$\mathcal{U}$' indicates user space and '$\mathcal{A}$' indicates automatic/ubiquitous space. A blank Attribute is therefore displayed as a colon followed by a space (which cannot be seen) and represented internally[6] as

```
(attribute (attName "") (attValue ""))
```

In the example, the user is able to move with the cursor over the colon and insert characters which are taken as part of the attribute name, or to place the cursor after the space to insert characters of the attribute value.

Specifically, the blank logical units are described in terms of both displaying and internal representation in Fig. 2.2.

logical unit	**Display**	**Internal representation**
Attribute name		`(attName "")`
Attribute value		`(attValue "")`
Attribute	:□	`(attribute (attName "") (attValue ""))`
Attribute list	:□	`(attList (attribute (attName "") (attValue "")))`
Element name	□	`(eleName "")`
Header	□_ _ _	`(header (eleName ""))`
Element	□_ _ _	`(element (header (eleName "")))`
CharData	Three blank lines (one as top "margin", one for the text and one as bottom "margin")	`(charData "")`
Processing Instruction Target		`(PITarget "")`
Processing Instruction Body		`(PIBody "")`
Processing Instruction	<?□:□_ _ _	`(PI (PITarget "") (PIBody ""))`
Comment	<!--□	`(comment "")`
Internal DTD	□[□	`(intDTD "")`
Entity reference	□&□□;□	`(entRef "")`

Figure 2.2: Blank logical units. '□' indicates a space, underlined text indicates a colored background.

When a new logical unit is created, point is placed at its first ubiquitous location, since there are no available user locations in a blank logical unit.

[6]See section 4.1 for details on internal representation

Inserting a new logical unit does not always make sense, for example creating a new Attribute name alone, or creating a Processing Instruction Body when the cursor is on the middle of an Entity reference. The meaning of such commands must be interpreted guessing what the user may be wanting when s/he issues them. That depends on where point is and what type of logical unit the user wants to create. The following is the set of rules that governs the insertion of new logical units.

- A primary logical unit can be created as a child of an element or as a sibling of the current primary logical unit. Different commands are provided for these two operations.
- Creating a Attribute list, a Attribute name or a Attribute value is equivalent to creating an Attribute.
- Creating a Processing Instruction Target or a Processing Instruction Body is equivalent to creating a Processing Instruction.

2.2.2 Adjusting the displaying of the tree structure

This section has not changed from Deliverable 1.

2.3 Control issues

Emaxml, at the stage of development I set as target for my project, will not enforce control over the tree structure created by the user or read from an XML file in terms of the DTD.

Emaxml will see the document "simply" as a syntactic structure that must comply with the grammar set out by the BNF rules in [1]. Note that the internal DTD is *not* parsed. The user will be able to manipulate the tree as s/he wants without being bothered with indentation. The "less-thans", quotes, and other syntactic sugar of XML will be hidden. This should hopefully let the user concentrate more on the contents, but, on the other end, the user will also be able to create documents that are not well-formed, or invalid.

However, the actual code implementing the editing mode must be designed so that adding semantic awareness and control should be easy.

Examples of events that may trigger a contents check or other semantics-related operations are:

- the contents of a secondary elementary logical unit (i.e. Element name, Attribute name, Attribute value) is changed at the character level;

- point leaves an elementary logical unit.
- the tree is changed; this includes creation of instances of any logical unit, killing and yanking at the logical and illogical levels,
- the contents of the seed element are changed;
- the contents of the Internal DTD are changed.

These and other such events must be easily recognizable and exploitable from the programmer's point of view.

2.3.1 Low-level control

Some logical units, by their nature, have limitations on what can be inserted in them. For instance, no spaces may be inserted in a Processing Instruction Target, since the target of a processing instruction can only be one word.

Emaxml enforces low-level control on the text inserted by the user, by matching the contents of the current elementary logical unit with a suitable regular expression.

By *low-level control* I mean that only the wellformedness is checked, as opposed to checking the validity as well, which would involve ensuring that what the user inserts is also consistent with the DTD declarations (for instance, that a referenced entity has been declared in the DTD).

Chapter 3

Background: Emacs buffer-related technologies

A buffer in Emacs is a data structure that contains the information needed to display something in a window, according to the mode the buffer is set to. For example, a buffer in Text mode contains simply a sequence of ASCII[1] characters, while a buffer in Enriched mode contains also information about characteristics of portion of text, such as the color, the weight, the justification, etc. A buffer in Enriched mode, when saved, may look like this:

```
Content-Type: text/enriched
Text-Width: 70

This text is normal.

<x-color><param>red</param>This text is red.</x-color>
<bold>This text is bold.</bold>
```

This will produce a line of normal text, one of red text and one of bold text. Information is in this case stored as markup *in the buffer*, and it is saved in the file.

Enriched mode is the simplest way of controlling the layout of text in a window. In sections more powerful approaches are examined. These sections are mainly a resume from the Emacs Lisp manual ([4]).

3.1 Text properties

Each character position in a buffer or a string can be associated to a *text property list*. Each property has a name and a value, and there exist Lisp functions to access and modify them.

[1] Or Unicode, or ...

Copying text between strings and buffers preserves the properties along with the characters; this includes such diverse functions as 'substring', 'insert', and 'buffer-substring'.

There are many predefined text properties, not only related to layout, but also to the behavior of Emacs related to a character position. Fig. H.1 is a list of some text properties that may be useful in designing Emaxml.

New text properties can be created and managed with the relative built-in functions. For example, in Emaxml may be useful to attach to a portion of the buffer a property called 'subtree' in which to store the corresponding portion of the Etree, in order to associate the logical and the physical view of the document.

By default, inserted characters take on the same properties as the preceding character. This is called *inheritance* of properties. Which properties are inherited, and from where, depends on which properties are *sticky*. Insertion after a character inherits those of its properties that are *rear-sticky*. Insertion before a character inherits those of its properties that are *front-sticky*. When both sides offer different sticky values for the same property, the previous character's value takes precedence.

It is possible to save text properties along with the text in a buffer, by exploiting the hook 'write-region-annotate-functions'.

3.2 Overlays

Overlays are used to alter the appearance of a buffer's text on the screen, for the sake of presentation features. An overlay is an object that belongs to a particular buffer, and has a specified beginning and end. It also has properties that can be examined and set; these affect the display of the text within the overlay.

Overlay properties are like text properties in that the properties that alter how a character is displayed can come from either source. But in most respects they are different. Text properties are considered a part of the text; overlays are specifically considered not to be part of the text. Thus, copying text between various buffers and strings preserves text properties, but does not try to preserve overlays. Changing a buffer's text properties marks the buffer as modified, while moving an overlay or changing its properties does not. Unlike text property changes, overlay changes are not recorded in the buffer's undo list.

Many overlay properties are common with text properties. Figure I.1 is a list of some peculiar overlay properties that may be useful in Emaxml design.

Chapter 4

Implementation design

The project is to be implemented as an extension to Emacs, i.e. as an Emacs mode. Thus, it is to be coded in Emacs Lisp.

The following priorities have to be kept in mind during coding:

- Consistency with the XML specification given in [1], in particular with the BNF definitions numbered in square brackets (**BNF-defs** for short).
- Modularity, so that functions and constants can be re-used in different contexts and extended easily.
- Readability of the code, to facilitate future possible improvement.

4.1 Data structures

The object of the editing, an XML file, will be seen by Emaxml in two ways at the same time: as an Emacs buffer (the **Ebuffer**), used to display the visual representation of the tree, and as a list (the **Etree**), structured as to allow an abstract, logical view of the document and appropriate manipulation.

An interface to manipulate the Etree and its components is provided as the *XD data model* described below.

4.1.1 The XD data model

The **XML Document data model** (**XD**) is composed of:

- a hierarchy of types (**XD-types**) that reflect Emaxml logical units;

- a set of functions (**XD-functions**) to manipulate the objects in the data model (**XD-objects**).

- a set of constants (**XDRE Toolkit**) that reflect the BNF-defs;

The XD-types

An XD-object belongs to one of the XD-types listed in Fig. E.1, and represents an instance of a logical unit in the display.

In concrete terms an XD-object p is a list $(C\ s_1\ [s_2 \cdots])$ whose first element C is a symbol denoting the XD-type of p and whose other element(s) s_i may be branches of the tree generating from p (that is, p's **children**, which are XD-objects themselves) or a string which refers to the part of the user space connected with p.

An example of an XD-object of type 'attribute' may be:

```
(attribute (attName "length")
           (attValue "25.52cm"))
```

The XD-type are listed in Fig. E.1 together with the children they may have.

A 'seed' object is a special kind of 'element' object: it may have only four attributes in its header (namely "version", "encoding", "standalone" and "external DTD"), does not have an element name and its children are limited as defined in Fig. E.1.

The XD-functions

The XD data model provides functions for the manipulation of its objects. Their names start with `XD-` and follow these naming conventions:

- `XD-<...>` refers to a function that performs an operation on a child, for example `(XD-<get> etree 'header)` returns the entire object of type *header* of the object `etree`;

- `XD->...<` refers to a function that performs an operation on the contents of a child, for example `(XD->get< etree 'seed 'header 'eleName)` returns the string associated with the element name in the header of the seed of the object `etree`;

- `XD-{...}` refers to a function that returns a list of objects, for example `(XD-{getall} elt 'PI 'comment)` returns a list of all the comments and processing instructions contained in the element `elt`;

- `XD-...-p` indicates a predicate function, as for standard Lisp convention, i.e. a function that checks some condition and returns `nil` or `t`.

The XDRE toolkit

The **XDRE** toolkit is a set of string constants which are regular expressions that match some of the basic building blocks of XML, defined by the BNF-defs[1]. Their purpose is to be used in the parsing functions instead of literal regexps, for readability.

Each constant's name is of the form `XDRE-component` , where `component` reflects the name of a rule in the BNF-defs.

In constructing the regexp, the symbols '<<', '>>', '||', '**', '++', '--' are used in place of '\\(', '\\)', '\\|', '*', '+', '?' respectively.

The table in Appendix D describes what each XDRE represents.

4.1.2 The Etree

The Etree is the structural representation of the file being edited. It is maintained and manipulated through the facilities provided by the XD data model.

The Etree is practically a 'seed' XD-object, that is, a list of objects which are lists themselves. A simple example of an Etree object may be:

```
(seed (header (eleName "")
              (attList (attribute (attName "version")
                                  (attValue "1.0"))
                       (attribute (attName "encoding")
                                  (attValue "UTF-8"))
                       (attribute (attName "standalone")
                                  (attValue "no"))
                       (attribute (attName "extDTD")
                                  (attValue "SYSTEM \"dtdfile.dtd\"")))))
      (comment "Simple document")
      (element (header (eleName "root")
                       (attList (attribute (attName "att1")
                                           (attValue "val1"))))
               (charData "This is some character data")
               (element (header (eleName "child")))
               (PI (PITarget "aTarget")
                   (PIBody "aBody"))))
```

This may be extracted from an XML document that looks like:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE root SYSTEM "dtdfile.dtd">
```

[1]Not all the BNF-defs can be translated into regular expressions, mostly because there is no trivial way of translating a BNF difference construct, such as in '(Char - ']')*', which indicates a sequence of zero or more instances of the BNF production 'Char' which are not ']'.

```
<!-- Simple document-->

<root att1="val1">
This is some character data
   <child/>
   <?aTarget aBody?>
</root>
```

4.1.3 The Ebuffer

The Ebuffer is an instance of an Emacs buffer, which is an internal data structure that Emacs is able to display[2]. It contains therefore the information that relates the Etree to its visual representation.

Some of this information is implicitly given in terms of the 'syntax table', which relates patterns to layout and behavioral characteristics. Emacs is made aware of these relations and takes care of displaying text that matches those patterns according to the relevant layout.

Other layout information is explicitly attached dynamically to portions of the Ebuffer using text properties and/or overlays, and it is maintained constantly consistent with the Etree.

At this point of the development I have not designed the details of the implementation of the Ebuffer. It will be mostly developed experimentally, since I am not familiar with the technologies involved. The background knowledge has been collected and looks very promising; the next steps in the design of the Ebuffer will be parallel to the development of the prototype of the Emaxml mode, and finally a proper specification will be produced for documentation purposes.

4.2 Whitespace handling

A piece of whitespace is a sequence of one or more spaces, tab characters, carriage return characters or linefeed characters. In the following discussion I refer to whitespace which is not part of markup, i.e. it is part of a piece of character data.

Whitespace which is between other non-whitespace characters is certainly part of the character data and must be preserved, while whitespace which is immediately before or after a piece of markup (from now on referred to as **boundary** whitespace), may be there for one of two reasons:

[2] See section 3 for details on Emacs buffers.

- Because it is integral part of the topic of the document (e.g. indentation such as in C code or poetry), and must be preserved.

- Because it is used to make the XML file more human-readable in raw XML format (e.g. blank lines, or tabs used for indentation). This whitespace does not affect the semantics of an XML file, and it should be up to the user whether to preserve it or not.

A sequence of whitespace characters only between two markup constructs corresponds in the Etree to a `charData` object whose string is whitespace only. Such an instance is called **void**.

In general, Emaxml is set to comply with one of the following policies for boundary whitespace, at the user's choice:

- **Allow-all**: do not perform any processing on the boundary whitespace.

- **Allow-none**: no boundary whitespace is preserved at all.

- **Allow-all-but-void**: preserve boundary whitespace but discard void `charData` objects. In practice, whitespace between markup that not contains any character data is considered to be for indentation purposes only. This is the default policy.

In practice, the software components apply the chosen policy as described in the below sections concerned with their definitions.

4.3 Software components

As said, the system is concretely an Emacs mode called Emaxml mode. This is the major infrastructure that manages the editing, using the Parser and the Writer (the other two components of the system) for specific tasks.

The Parser and the Writer are effectively two independent, reusable pieces of software, based upon a common data model, the XD data model.

4.3.1 The Parser

The **XMLDoc Parser** (**XDP** for short) takes as input an Emacs buffer in Fundamental mode (hence with no meta-information about text properties) containing an XML document

and extracts the relative Etree. The Parser checks the syntax of the document and gives an indication of the error in case it is not correct.

XDP has been implemented already, and tested informally, so the following is a description rather than a design specification. The idea behind XDP is to provide an independent, reusable tool for parsing XML files according to the XD data model. In some respect, XDP and the XD data model are quite limited, since they do not cover all the aspects of an XML document to a high degree of detail, but they can be useful for any application that, like Emaxml, needs to represent and manipulate the skeleton of an XML document for practical purposes.

Emaxml parses an XML document by moving point in the buffer which contains the XML file. At the current position of point, the parser expects to find a sequence of characters that corresponds to one of a series of possible XD-object, according to the BNF-defs. If such a sequence is found, the relative XD-object is built (**object extraction**), and point is advanced, otherwise the parsing is unsuccessful.

The parsing process is a recursive one, so at the end of the day it consists of placing point at the beginning of the buffer and trying to extract a 'seed' object.

XDP is coded in the `~ceepd1/tesi/prg/XDP/XDP.el` file and consists of:

- a set of auxiliary functions (the **XDP Toolkit**), that carry out general operations related to parsing;
- a set of extracting functions (the **XDC-functions**), each of which is concerned with parsing an XML component.

Whitespace in parsing

The Parser is responsible of filtering the whitespace present in the XML file according to the chosen whitespace policy.

If the policy is "Allow-none", all boundary whitespace is removed from the character data.

If the policy is "Allow-all", all whitespace is preserved in the character data.

If the policy is "Allow-all-but-void", all whitespace is preserved, but void 'charData' objects are not.

XDP Toolkit

Parsing and in particular object extraction involve some elementary operations, provided by the XDP toolkit, that fall in one of the following categories:

- **Matching and skipping**

  The parser often needs to check if the text starting at point matches a particular regexp. It may need to retrieve it or ignore it. Functions like `XDC-match-minus` or `XDC-skip` provide such operations.

- **BNF construct handling**

  The objects to be extracted derive from the BNF-defs, which are composed of *conjuctions* (sequences), *disjunctions* (selections, '|') and *repetitions* ('*', '+', '?'). Functions in this category (such as `XDP-and` or `XDP-*`) provide these features.

- **Object manipulation**

  Functions in this category provide operations that are object-specific such as translating a standard entity reference to the corresponding character, or extracting information from the prolog of the XML document, or building an object from its components.

Generally speaking, XDP functions try to match the contents of the buffer at point with something (a regular expression, the result of one or more other XDP functions, ...) and return what matched.

A return value of `t` means that the requested match was not found but the function is successful anyway. For example, when trying to match '0 or more instancies of something', a non-match is a success nonetheless.

All XDP functions are expected to leave point at the end of what they matched, or where it was if nothing was matched.

See Appendix F for the list and details of the XDP functions.

XDC parsing functions

n

Every XD-type has a corresponding XDC-function that parses the text at point and returns an object of that type if one was there, or `nil`.

Moreover, there are several XDC-functions that refer to some BNF-defs. The object returned by such a function is not in the XD data model, but is of the same structure of an XD-object. For example, `XDC-prolog` parses the prolog of an XML document as defined by the BNF-def number 22.

A return value of `nil` means that the object was not recognized at point.

Most of XDC functions are straight-forwardly constructed by reproducing the BNF-def using a combination of XDP functions, XDRE regexps and XDC functions themselves, e.g.:

```
01 ;; [28] doctypedecl ::= '<!DOCTYPE' S Name (S ExternalID)? S?
02 ;;                      ('[' (markupdecl | DeclSep)* ']' S?)? '>'
03 ;; doctypedecl -> Name ExternalID? InternalID?
04
05 (defun XDC-doctypedecl ()                                  ;
06   (XDP-build 'doctypedecl                                  ;
07              (XDP-skip "!DOCTYPE" XDRE-S)                  ; '<!DOCTYPE' S
08              (XDC-Name)                                    ; Name
09              (XDP-01 (XDP-and (XDP-skip XDRE-S)            ; (S
10                               (XDC-ExternalID)))           ; ExternalID)?
11              (XDP-01 (XDP-skip XDRE-S))                    ; S?
12              (XDP-01 (XDP-and (XDP-skip "\\[")             ; ('['
13                               (XDC-InternalID)             ; (markupdecl | DeclSep)*
14                               (XDP-skip "\\]")             ; ']'
15                               (XDP-01 (XDP-skip XDRE-S)))) ; S?)?
16              (XDP-skip ">")))                              ; '>'
```

Lines 1-2 contain the BNF-def as from [1].

Line 3 describes what the object is composed of, i.e. a Name object, possibly an ExternalId object, possibly an InternalID object.

Line 6 invokes the XDP-build function to build a 'doctypedecl' object as described by lines 7-16.

Line 7 skips over '<!DOCTYPE' and whitespace.

Line 8 extracts a Name object.

Lines 9 and 10 deal with an optional pair of whitespace followed by an ExternalID object, and extract the latter.

Line 11 skips over some optional white space.

...and so on.

4.3.2 The Writer

The Writer carries out the opposite of the Parser: it takes an Etree and produces an Emacs buffer, in Fundamental mode, whose contents are the XML document corresponding to that Etree.

The Etree received as input can be assumed to be always errorless (i.e. to be formed

of legal XD-objects as defined in Fig. 4.1.1), since it may only have been produced by the Parser or by the Emaxml mode, which both enforce syntactic control over the structure.

An XML document d is said to be **correctly produced** from an Etree e under a certain whitespace policy w if and only if the result of parsing d under w is equal to e.

A set of functions (**XWC-functions**[3] provide translation from an XD-object to an equivalent string in XML syntax with no whitespace added.

This string is then processed to add whitespace according to the current whitespace policy; in terms of whitespace, the Writer can produce an XML file optimized for:

- Storage (i.e. with no extra whitespace added), if the policy is 'Allow-none'.
- Human inspection (i.e. the markup is indented), if the policy is 'Allow-all' or 'Allow-all-but-void'. The indentation style is implemented very simply at this stage of development, and can be improved later.

In any case, the character data is written as it is (hence the Parser and the Emaxml mode are responsible for this), but in the second case the markup is also indented.

As said, the result of the Writer is a buffer containing plain text, which can be saved, or inspected and modified. Before saving it, Emaxml checks the possible internal DTD as described in section 2.1.2, using the Parser.

4.3.3 The Emaxml mode

Emaxml is to be implemented as a major Emacs mode. The design of this implementation is described in the following sections at a very high level, i.e. it is a set of steps that must be carried out rather than a detailed definition. A suitable specification will finally be produced for documentation purposes.

The main source of information about the features of an Emacs mode and the techniques used to set one up is the Emacs Lisp manual ([4]), to which one may refer for further detail. See also Appendix G for brief definitions of some Emacs concepts.

General description of an Emacs mode

A *mode* is a set of definitions that customize Emacs and can be turned on and off by the user. There are two varieties of modes: *major modes*, which are mutually exclusive and used

[3]The prefix "XWC-" is chosen to match that of the XDC-functions, because an XWC function is the inverse of the XDC-function for the same XD-type. For example, if p is an XD-object of type 'attList', then p=(`XDC-attList`(`XWC-attList`(p))).

for editing particular kinds of text, and *minor modes*, which provide features that users can enable individually.

An example of a mode is 'C' mode, whose purpose is to edit C code files. This mode is activated when loading a C file, or by calling the Emacs Lisp function 'c-mode'. Some of the many features available when a buffer is in C mode are:

- the syntactic constructs of C are highlighted in different colors; this provides also instantaneous syntactic check;
- when the user types a closing brace the corresponding opening brace blinks;
- the text can automatically be indented according to one of many styles;
- the program can be compiled in Emacs;
- point can be moved to next/previous function;
- there are tools for version control and debug.

Some of these functionalities are automatically managed by the mode, others are activated by a key sequence or by an item in a menu.

Most features of the mode can be finely tuned using Emacs customization system.

Implementing an Emacs mode

Implementing a mode consists basically of two phases[4]:

- writing the Lisp functions that perform the various operations;
- setting up the mode, that is, making Emacs aware of when and how to use those functions, and which way it should perform its common operations.

In general, a mode is set up by defining its components:

- The *keymap* maps key sequences to Lisp functions.
- The *syntax table* defines categories of characters in terms of the syntax;
- The *buffer-local variables* can be used to define the behavior of Emacs relative to some common functions. For example, C mode and Lisp mode both set the variable

[4]Please refer to Appendix G for the definition of the technical terms in this section.

'paragraph-start' to specify that only blank lines separate paragraphs. They do this by making the variable buffer-local in the buffer that is being put into C mode or Lisp mode, and then setting it to the new value for that mode.

- The *standard hooks* can be used to make Emacs perform some common operation in an appropriate way, peculiar to the new mode.

- *New hooks* are defined and set to default values. The new mode will call the functions listed in these hooks when performing particular operations, so allowing the user or a developer to customize the behavior of the mode by changing the values of the hooks.

4.4 Implementation of low-level control

Low-level control (see section 2.3.1) concerns the contents of elementary logical units, and can be achieved in two ways:

by constant monitoring: the check is triggered by a change in the logical unit, by means of hooking a function to the appropriate text property or overlay;

by checking on exit: the check is triggered by point moving out of the logical unit, in the functions that are mapped to the keys used for moving point.

In both cases the user should be adviced of the error, yet be let free of keeping it if s/he wants to. A logical unit containing an accepted error should be highlighted on the display.

The first policy offers a more sophisticated control, but involves matching a regular expression every time the user inserts/deletes a character or kills/yanks a string. This may prove very resource-consuming.

The second policy lets the user type anything and then advices him/her that something is wrong when s/he tries to leave the logical unit.

Therefore if "checking on exit" is implemented, Emaxml should reasonably also:

- advice on where exactly the error is;

- remember somehow which errors have been already accepted, and keep this information updated automatically if the error is corrected, otherwise the same error would be detected every time the user moves point through the incriminated elementary logical unit;

- save this information for successive sessions.

Which policy is best?

"Constant monitoring" is more resource-consuming, but resources are abundant these days (and more in the future).

"Checking on exit" relies on surplus information, whose maintenance increases the overall complexity of the system, hence decreasing its expandibility.

As a conclusion, "constant monitoring" is chosen as low-level control policy.

Chapter 5

Documentation design

Following the Emacs philosophy and spirit, Emaxml is meant to be an extendible system.

Useful and usable documentation is a key requirement. If only part of the target functionalities is implemented, but it is well documented, it will still be a partial success.

Although the most comprehensive source of documentation for future (possible) developers will be the Dissertation Deliverable, two more documentation source are required: code documentation (again for developers) and Emacs on line help (for the end[1] users).

The Emacs Lisp Manual ([4]) has a chapter ("Tips and Conventions") that covers the conventions to be used in writing, commenting and documenting code for Emacs. Emaxml documentation is to comply with those conventions.

5.1 Code documentation

The code is to be documented internally and as a set of tables listing the functions and included as appendices in the Dissertation Document. The latter is not required by the Emacs standard, but it is included for quick reference and a general view of the structure of the code.

5.1.1 Internal documentation

In Lisp a semicolon starts a comment, but, as a convention in Emacs Lisp, a comment is classified according to the number of semicolons put at the beginning. The indentation commands of the Lisp modes in Emacs, such as 'M-;' ('indent-for-comment') and >TAB> ('lisp-indent-line'), automatically indent comments according to these conventions, depend-

[1] ...possible...

ing on the number of semicolons.

These are the conventions recommended:

- Comments that start with a single semicolon, ';', usually explain how the code on the same line does its job. In Lisp mode and related modes, the 'M-;' ('indent-for-comment') command automatically inserts such a ';' in the right place, or aligns such a comment if it is already present.
- Comments that start with two semicolons, ';;', usually describe the purpose of the following lines or the state of the program at that point.
- Comments that start with three semicolons, ';;;' are used outside function definitions to make general statements explaining the design principles of the program.
- Comments that start with four semicolons, ';;;;', are used for headings of major sections of a program.

5.1.2 Function tables

Functions belonging to an Emacs mode are of two types: interactive functions, meant to be called by the end user (via a key combination, a menu, or by name) and internal functions, which perform the auxiliary computation.

All functions are listed in tables (put in appendices in the Dissertation Document), grouped by software component. One further table lists the interactive functions from all software components.

The purpose of these tables is to give a general, wide view of the structure of the entire program, its main components and the naming conventions adopted, and as quick reference in terms of number and type of arguments.

Each entry contains the name of the function, the name and type of its arguments, and a brief description.

5.2 Emacs on line documentation

The standard on line documentation for an Emacs mode includes:

Function documentation strings Every command, function, or variable intended for users to know about should have a documentation string. This is enclosed in the function or variable definition itself as the very first line of the code, and it is displayed

when information is asked about that particular function or variable using Emacs help facilities such as Apropos, 'C-h f' ('describe-function') or 'C-h v' ('describe-variable').

Documentation strings are described in the Emacs Lisp Manual ([4], nodes "Documentation Basics" and "Documentation Tips") and allow easy inclusion of hyperlinks to other parts of the documentation.

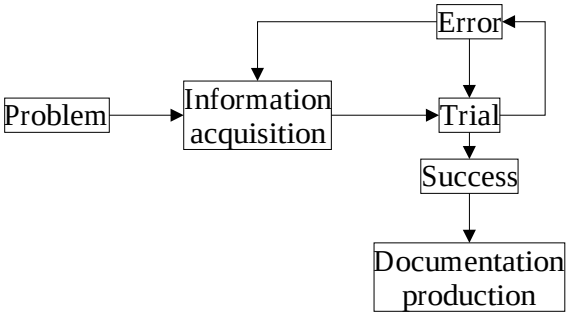
Manual An Emacs mode is documented in a manual for the end user, written in the Texinfo language and organized in terms of topics of discussion.

Chapter 6

Point of the situation and plan

So far, I have implemented the Parser, which works has described and has been tested informally on a set of XML files.

The work has been carried out in a sort of experimental fashion. By this I mean that I have been loosely following a cycle



This is due to my initial lack of experience with Emacs internals, Lisp, and XML.

What is missing is testing, which has been discussed in Deliverable 1 to some degree.

My plan for the future is to implement the Writer and test it with the Parser, then implement the Emaxml mode and test it. Finally, I will work on the Dissertation Document.

Bibliography

[1] W3C XML Core Working Group. *Extensible Markup Language (XML) 1.0 (Second Edition)*. http://www.w3.org/TR/2000/REC-xml-20001006, 2001.

This is the official specification of XML. Includes the BNF grammar describing the syntax of XML.

[2] E. Rusty Harold and E. Scott Means. *XML in a Nutshell*. O'Reilly, 2001.

Good discursive explanation of the basics of the various aspects of XML, plus a comprehensive coverage of all related topics and applications. I found it useful for initial documentation, and also as a quick reference.

[3] Free Software Foundation. *Emacs Info Manual*. Free Software Foundation, 1999.

Major source of information about the usage of Emacs. It is more than a help on-line; it can be searched in many ways and, as far as my experience is concerned, always answers one's questions. Moreover, it does not pop up unwanted saying that you are writing a letter.

[4] B. Lewis, D. LaLiberte and R. Stallman and the GNU Manual Group. *Emacs Lisp Manual*. http://www.gnu.org/manual/elisp-manual-20-2.5/elisp.html, 1993.

A book on Lisp, Emacs Lisp, Emacs internals, Emacs Lisp libraries. As readable as a novel, as useful as a quick reference. Available in a variety of formats including Info, which makes it embedded in Emacs.

[5] B. Glickstein. *Writing GNU Emacs Extensions*. O'Reilly, 1997.

Covers the customization of Emacs from the very basics of Lisp to a full major mode implementation. Very rich of practical examples paired with Lisp theory.

[6] H. Abelson, G. J. Sussman and J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, 1985.

An inspiring book, accidentally about Lisp, and purposefully about abstraction. It has been said that its footnotes alone are more interesting than most books around.

[7] M. Gankarz. *The UNIX Philosophy.* Digital Press, 1995.

Software Engineering is not just a waterfall.

Appendix A

Logical units and their categories

Fig. A.1 is a list of the types of logical units in Emaxml.

Element	A whole element, including its header and its children
Header	An element's name and its attributes
Element name	The first word of a Header
Attribute list	The set of attributes of an element and their values
Attribute	An attribute's name and its value
Attribute name	An attribute's name
Attribute value	An attribute's value
CharData	The plain text. In CharData logical units, illegal characters ('"<>&) are edited and displayed normally. However, they are codified to entity references when the XML file is written.
Processing Instruction	A processing instruction
Processing Instruction Target	The target of a processing instruction, i.e. the first word of a Processing Instruction
Processing Instruction Body	The command of a processing instruction
Comment	A comment text
Internal DTD	The text relative to the definition of the internal DTD
Entity reference	The text of an entity reference

Figure A.1: Logical Units

Subsets of Logical units are grouped into categories according to some properties, as described in Table A.2.

Category	Property	Logical units
Elementary	Composed of characters only.	Element name, Attribute name, Attribute value, CharData, Processing Instruction Target, Processing Instruction Body, Comment, Internal DTD, Entity reference.
Compound	Composed of elementary and compound logical units.	Element, Header, Attribute list, Attribute, Processing Instruction.
Primary	Logical units that are the main building blocks of the structure of the document at the *structure level* of document authoring, as opposed to the *syntactic level* (see 1.1).	Element, CharData, Processing Instruction, Comment, Entity reference, Internal DTD
Secondary	Logical units that are not really part of the structure of the document from a high level point of view, but which it is useful to identify on the display for editing purposes.	Header, Attribute list, Attribute, Element name, Attribute name, Attribute value, Processing Instruction Body, Processing Instruction Target.
Special	Primary logical units which cannot contain other primary logical units, i.e. are terminal nodes of the structure of the document.	CharData, Processing Instruction, Comment, Entity reference, Internal DTD.
Multiline	May contain *newline* characters.	Internal DTD, Comment, CharData, Attribute value.
Monoline	May not contain *newline* characters.	Element name, Attribute name, Attribute value, Processing Instruction Target, Processing Instruction Body, Entity reference.

Figure A.2: Categories of Logical units

Appendix B

Standard Emacs commands recoded for Emaxml

Notes:

- Editing operations are here referred to by their Emacs Lisp names, due to lack of space. For an exact definition, where these names are not self-explanatory, use 'F1 f <command>' in Emacs.
- The **Cat** column refers to the editing categories listed in section 2.1.

Cat	Shortcut	Emacs Lisp name	Action
ID	C-d	delete-char	Delete the following character. No action at end of an elementary logical unit.
ID	ESC k	kill-sentence	Performs kill-element.
ID	C-y	yank	Described in 2.1.7. Perform "yanking as a child".
ID	S-insert	yank	Described in 2.1.7. Perform "yanking as a sibling".
ID	BACKSPACE	delete-backward-char	In the middle of an elementary logical unit, behave as usual. At beginning, behave as C-b.
ID	RET	newline	Multiline logical unit: add a newline at point.
ID	insert	overwrite-mode	Usual behavior.
GE	C-l	recenter	Usual behavior
GE	C-_, C-/	undo	Usual behavior
KY	C-k	kill-line	Monoline: Kill the rest of the current logical line.Multiline: also, if no non-blanks there, kill thru newline.
MK	C-@, C-SPC	set-mark-command	Usual behavior.
MK	C-x C-x	exchange-point-and-mark	Usual behavior.
PM	C-a	beginning-of-line	Move point to beginning of current logical line.
PM	C-b, ←	backward-char	Slide backward.
PM	C-↓	forward-paragraph	Traverse down.
PM	C-e	end-of-line	Move point to end of current logical line.
PM	C-f, →	forward-char	Slide forward.
PM	C-←	backward-word	Usual behavior, through user space.
PM	C-n, ↓	next-line	Slide to next logical line, with climbing transparency.
PM	C-p, ↑	previous-line	Slide to previous logical line, with climbing transparency.
PM	C-→	forward-word	Usual behavior, through user space.
PM	C-↑	backward-paragraph	Traverse up.
PM	end	end-of-buffer	Usual behavior.
PM	home	beginning-of-buffer	Usual behavior.
RE	C-x h	mark-whole-buffer	Usual behavior.
RE	ESC @	mark-word	Usual behavior.
RE	double-mouse-1	mouse-set-point	Usual behavior, but highlighting the region as described in section 2.1.3.
RE	drag-mouse-1	mouse-set-region	Usual behavior. Region as described in section 2.1.3.
RE	mouse-1	mouse-set-point	Usual behavior, but highlighting the region as described in section 2.1.3.
RE	mouse-2	mouse-yank-at-click	Usual behavior, but yanking done a la Emaxml (see section 2.1.7.
RE	triple-mouse-1	mouse-set-point	Usual behavior, but region set to the whole logical unit, and highlighting as described in section 2.1.3.

Figure B.1: New meanings for old commands

Appendix C

Emaxml-specific commands

Lisp-like non-definitive name	**Action**
create-element-child	Create a blank Element as a child of the current smallest element. Described in section 2.2.1.
create-element-sibling	Create a blank Element as a sibling of the current smallest element. Described in section 2.2.1.
create-*logun*	Create a blank instance of a logical unit of type *logun*.
create-primary	Create a sibling of the current primary logical unit just before the current primary logical unit.
traverse-right	Traverse right. Described in section 2.1.5.
traverse-left	Traverse left. Described in section 2.1.5.
mark-more	Described in section 2.1.3
mark-less	Described in section 2.1.3

Figure C.1: New commands to be implemented

Appendix D

Details of XDRE constants

BNF-def	XDRE constant	Explanation
[2] Char	XDRE-Char	Unicode character range
[3] S	XDRE-S	White Space
[87] CombiningChar	XDRE-CombiningChar	Among others, this class contains most diacritics
[89] Extender	XDRE-Extender	Extenders
[85] BaseChar	XDRE-BaseChar	Among others, this class contains the Unicode alphabetic characters of the Latin alphabet
[86] Ideographic	XDRE-Ideographic	Unicode ideographic characters
[84] Letter	XDRE-Letter	BaseChar's + ideographic characters
[88] Digit	XDRE-Digit	Unicode digits
[4] NameChar	XDRE-NameChar	Characters allowed in Names
[5] Name	XDRE-Name	Matches a legal Name
[25] Eq	XDRE-Eq	Equality sign
[68] EntityRef	XDRE-EntityRef	Matches an entity reference (eg. '&cright;')
[66] CharRef	XDRE-CharRef	Matches a character reference (eg. 'Ћ')
[19] CDStart	XDRE-CDStart	Matches '<![CDATA['
[21] CDEnd	XDRE-CDEnd	Matches ']]>', the CDATA section terminator
[69] PEReference	XDRE-PEReference	Matches a Parameter Entity (eg. '%abc;')
[26] VersionNum	XDRE-VersionNum	Matches the version number declaration in an Xml Declaration
[81] EncName	XDRE-EncName	Matches the encoding name in an Encoding Declaration
[13] PubidChar	XDRE-PubidChar	Characters allowed in names of PubidLiteral's

Figure D.1: The XDRE toolkit

Appendix E

Details of the XD-types

Table E.1 is the list of the types belonging to the XD data model.

The notation used is:

→: "has as children";

*: "zero or more";

?: "zero or one";

+: "one or more";

|: indicates alternative children.

seed	→	header intDTD? (comment | PI)* element (comment | PI)*
intDTD	→	*string*
element	→	header (element | comment | PI | entRef | charRef | charData)*
header	→	eleName attList*
eleName	→	*string*
attList	→	attribute+
attribute	→	attName attValue
attName	→	*string*
attValue	→	(*string* | entRef | charRef)*
comment	→	*string*
PI	→	PITarget PIBody
PITarget	→	*string*
PIBody	→	*string*
charRef	→	*string*
entRef	→	*string*
charData	→	*string*

Figure E.1: Data types in the XD data model.

Appendix F

Details of XDP functions

XDP	XDP-*
Functionality performed	Checks if point is at 0 or more occurrencies of FORM.
BNF construct handled	Char*
Parameters	FORM - a Lisp form composed of XDP and XDC constructs.
Return	`t` if 0 occurrencies found. A list of the occurrencies found otherwise.

XDP	XDP-01
Functionality performed	Checks if point is at 0 or 1 occurrencies of FORM.
BNF construct handled	
Parameters	S?
Return	`t` if 0 occurrencies found. The occurrency found otherwise.

XDP	XDP-match
Functionality performed	Checks if point is looking-at RE.
BNF construct handled	[^<& ']
Parameters	RE - a regular expression.
Return	The match-string if matched. `nil` otherwise.

XDP	XDP-match-minus
Functionality performed	Checks if point is looking-at the difference regexp (RE1 - RE2).
BNF construct handled	Name - (('X' — 'x') ('M' — 'm') ('L' — 'l'))
Parameters	RE1 and RE2 - two regexps.
Return	The match-string if matched. `nil` otherwise.

XDP	XDP-match-until
Functionality performed	If looking-at 'RE1*TERMINATOR' return what matches 'RE1*' and set point at end of it.
BNF construct handled	((Char - '-') — ('-' (Char - '-')))*
Parameters	RE1 - a regexp. TERMINATOR - a string.
Return	What matches 'RE1*' if matched. - `nil` otherwise.

XDP	XDP-skip
Functionality performed	Skips over a regular expression. Used for portions of buffer that don't represent any object.
BNF construct handled	""
Parameters	RE - a list of one or more regexps.
Return	`t` if RE is matched. `nil` otherwise.

XDP	XDP-build
Functionality performed	Constructs an object by putting together the results of the forms in ITEMS. It is based on a call to XDP-and whose result is put in a one-element list.
BNF construct handled	[32] SDDecl ::= ...
Parameters	ITEMS - a list of Lisp forms.
Return	An object, if all of ITEMS matched. `nil` otherwise.

XDP	XDP-and
Functionality performed	Handles sequences. It also deals with forms that return `t` to mean a successful non-match, by not appending the `t` to the list returned.
BNF construct handled	STag content ETag
Parameters	FORMS - a list of Lisp forms.
Return	A list with the results of evaluating FORMS if all non-`nil`. `nil` otherwise.

XDP	XDP-or
Functionality performed	Handles selections.
BNF construct handled	Comment — PI — S
Parameters	FORMS - a list of Lisp forms.
Return	The first form in FORMS that matches what point is at, if any. `nil` otherwise.

Appendix G

Glossary of Emacs technologies

This chapter briefly defines some technologies related to the design of Emaxml. It is not intended to be exhaustive. For a complete and better explanation refer to the Emacs manual ([4]).

Face A face in Emacs jargon is a set of layout attributes, namely: font family, width, height, weight, slant, underline, overline, strike-through, box, inverse-video, foreground, background, stipple, inherit.

Echo Area The Echo Area is a line at the bottom of an Emacs frame, for displaying messages.

Keymap The keymap is the data structure that records the bindings of key sequences to the commands that they run. For example, the global keymap binds the character 'Ctrl-n' to the command function 'next-line', therefore when 'Ctrl-n' is pressed the cursor moves to the next line.

One of the characteristics of an Emacs mode is which key combinations trigger which operations. These are defined by the mode keymap.

Syntax Table A syntax table provides Emacs with the information that determines the syntactic use of each character in a buffer. This information is used by the parsing commands, the complex movement commands, and others to determine where words, symbols, and other syntactic constructs begin and end.

Each buffer has its own major mode, and each major mode has its own idea of the syntactic class of various characters. For example, in Lisp mode, the character ';' begins a comment, but in C mode, it terminates a statement. To support these

variations, Emacs makes the choice of syntax table local to each buffer. Typically, each major mode has its own syntax table and installs that table in each buffer that uses that mode.

Hook A hook is a variable where it is possible to store a function or functions to be called on a particular occasion by an existing program.

For instance, if the name of a function is added to the variable 'after-save-hook' (using function 'add-hook'), that function will be called after saving any file.

Appendix H

Text properties

The following is a list of some text properties that may be useful in designing Emaxml.

Property	Description
face	Face associated with the position.
mouse-face	Face used instead of 'face' when the mouse is on or near the character. This allows the text to change layout according to the mouse pointer position.
display	This property activates various features that change the way text is displayed. For example, it can make text appear taller or shorter, higher or lower, wider or narrow, or replaced with an image.
help-echo	If text has a string as its 'help-echo' property, then when the mouse is moved onto that text, Emacs displays that string in the echo area. The value of this property can also be a function to be executed.
local-map	Used for key lookup instead of the buffer's local map. *(Controlling movement of point only to user/ubiquitous space.)*
syntax-table	The 'syntax-table' property overrides what the syntax table says about this particular character. *(This affects many existing Emacs commands, especially at the character level of editing.)*
read-only	If a character has the property 'read-only', then modifying that character is not allowed. *(Ubiquitous space.)*
invisible	A non-'nil' 'invisible' property can make a character invisible on the screen. *(Storage of information related to Emaxml internals; implementation of display modes.)*
intangible	If a group of consecutive characters have equal and non-'nil' 'intangible' properties, then point cannot be placed between them. If the user tries to move point forward into the group, point actually moves to the end of the group. If the user tries to move point backward into the group, point actually moves to the start of the group. *(Automatic space.)*
modification-hooks	Functions to be executed when the character is modified. *(Controlling what the user writes.)*

Figure H.1: Some text properties and (in *italic*) examples of use in Emaxml. See appendix G for definitions of Emacs terminology.

Appendix I

Overlays

The following is a list of some peculiar overlay properties that may be useful in Emaxml design.

Property	**Description**
priority	When two or more overlays cover the same character and both specify a face for display, the face attributes of the one whose 'priority' value is larger override the face attributes of the lower priority overlay. Overlays take priority over text properties.
window	If the 'window' property is non-'nil', then the overlay applies only on that window. Useful to display the same buffer with different layouts in different windows.
insert-in-front-hooks	This property's value is a list of functions to be called before and after inserting text right at the beginning of the overlay.
insert-behind-hooks	This property's value is a list of functions to be called before and after inserting text right at the end of the overlay.
before-string	This property's value is a string to add to the display at the beginning of the overlay. The string does not appear in the buffer in any sense–only on the screen. *Displaying automatic space.*
after-string	This property's value is a string to add to the display at the end of the overlay. The string does not appear in the buffer in any sense–only on the screen. *Displaying automatic space.*
evaporate	If this property is non-'nil', the overlay is deleted automatically if it ever becomes empty (i.e., if it spans no characters).

Figure I.1: Some overlay properties and (in *italic*) examples of use in Emaxml. See appendix G for definitions of Emacs terminology.