Encoding script-specific writing rules based on the Unicode character set

Malek Boualem & Mark Leisher

CRL (Computing Research Laboratory), New Mexico State University,
Box 30001, Dept 3CRL, Las Cruces, NM 88003, USA
E-mail: malek@crl.nmsu.edu, mleisher@crl.nmsu.edu
http://crl.nmsu.edu

Abstract:

The World Wide Web is now the primary means for information interchange that is mainly represented in textual format. However programs that create and view these texts generally do not adequately support texts using non-Latin scripts, particularly right-to-left scripts. Unicode as a universal character set solves encoding problems of multilingual texts. It provides abstract character codes but does not offer methods for rendering text on screen or paper. An abstract character such "ARABIC LETTER BEH" which has the U+0628 code value can have different visual representations (called shapes or glyphs) on screen or paper, depending on context. Different scripts which are part of Unicode can have different rules for rendering glyphs, composite characters, ligatures, and other script-specific features. In this paper we present a general approach to encoding script-specific rendering rules based on the Unicode character set and using finite state transducer. The proposed formalism for character classification and writing rules is modular and easy to read and to modify by average users. In addition it is based on the most stable font structure defined in the Unicode Standard, thus it should be reusable by other environments supporting fonts from the Unicode Standard. Moreover the associated program is written in JAVA which makes it portable in many environments. This approach will be demonstrated with writing rules for some languages that use the Arabic script.

I. Introduction

The Unicode Standard does not define glyph images. It defines how characters are interpreted, not how glyphs are rendered. The software or hardware-rendering engine of a computer is responsible for the appearance of the characters on the screen. Context analysis for rendering correct character glyphs is necessary for many scripts. The objective of the work presented here is designed to address multiple problems. Until the Unicode Standard is stable, most of the text editing software use (and still use) character fonts relative with other character coding standards or dependent upon local specific hardware environment. So the context analysis programs are often also designed dependent on the used character font formats. Sometimes to simplify the program processing, some character fonts are modified and the physical locations of the characters do not correspond to their logical locations. Consequently, it becomes difficult or even impossible to re-use these programs with other text editors using character fonts structured in a different way (unless the same character fonts are used).

Another problem which also is important is the fact that the formalisms used for the context analysis (character classes and writing rules) must preferably be modular and easily readable and modifiable by users. Indeed these language-specific characteristics may be defined by users who have knowledge in the specified languages and who are not necessary computer programers. However the compromise of effectiveness of processing often pushes the developers to design too technical formalisms that only they are able to read and to modify.

The objective of this work is to present a model for the context analysis which is on the one hand modular and compatible with the Unicode standard, and on the other hand expressed in a formalism that is easily readable and accessible to users. Thus, it will be possible to re-use this analyzer in other environments supporting character fonts from the Unicode Standard. This work was completed to improve the context analyzer for the MUTT multilingual text editor described below (that is why we called it MUTTCA: Multilingual Unicode Text Toolkit Context Analyzer). In addition the associated program is written in JAVA which makes it portable in many environments. Lastly, the context analyzer we present here has been designed on the basis of the context analyzer we designed in the framework of the development of MtScript[1]: the Multext [MULT.97] multilingual text editor [BOUA.97].

II. The Unicode Standard

2.1. Definition

The Unicode Worldwide Character Standard is a character coding system designed to support the interchange, processing, and display of the written texts of the diverse languages of the modern world. In addition, it supports classical and historical texts of many written languages. The Unicode Standard is a 16-bit character and is code-for-code identical to the International Standard ISO/IEC 10646. To keep character coding simple and efficient, the Unicode Standard assigns each character a unique 16-bit value, and does not use complex modes or escape codes. The Unicode Standard defines codes for characters used in the major languages written today. Some of the scripts included are Latin, Greek, Cyrillic, Armenian, Hebrew, Arabic, Devanagari, Bengali, Gurmukhi, Gujarati, Oriya, Tamil, Telugu, Kannada, Malayalam, Thai, Lao, Georgian, Tibetan, Japanese Kana, the complete set of modern Korean Hangul, and a unified set of Chinese/Japanese/Korean (CJK) ideographs. More scripts and characters are to be added shortly [UNIC.98].

2.2. Text Processing

Computer text handling involves processing and encoding. The Unicode Standard directly addresses only the encoding and semantics of text. It addresses no other action performed on the text (character glyphs, misspelled words, ...). An important principle of the Unicode Standard is that it does not specify how to carry out these processes as long as the character encoding and decoding is performed properly [UNST.96].

[1] MtScript.1.1 for Solaris and Linux systems is freely available and can be downloaded at the URL: ***http://www.lpl.univ-aix.fr/projects/multext/MtScript/***

III. Multilingual text editing

3.1. Difficulties in multilingual text editing design

Multilingual text-editing design difficulties occur on several levels: inputting, coding, editing, printing and data exchange (see figure 1).

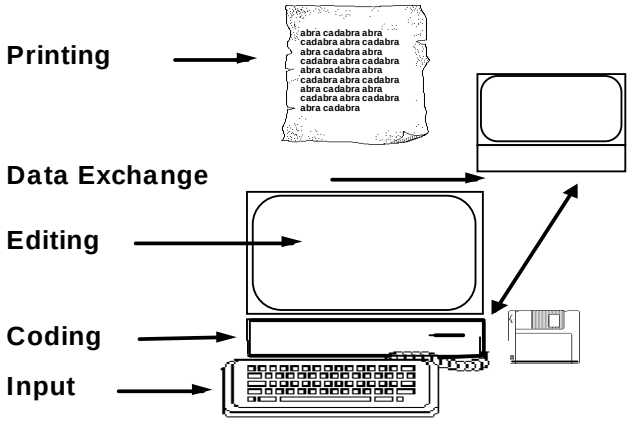


Figure 1.

3.1.1. Text inputting

Many keyboards represent only ASCII characters (or ISO 646), but certain localized keyboards may also include keys for special characters (French accented characters, etc.). In a broad multilingual context, the inclusion of languages such as Chinese (more than 6000 ideograms) or Arabic (approximately 4 sets of 28 letters and 10 vowels) requires the definition of specific keyboard input programs. Solutions proposed by computer manufacturers are very heterogeneous. Theoretically there exists a standard input method for keyboards with 48 keys (ISO/IEC 9995-3), at least for the Roman alphabet, but it is rarely used. A number of keyboard input methods are being proposed especially for the ISO 10646 [LABO.95] but is necessary to develop intuitive keyboard methods and, if possible, reduce the number of key presses.

3.1.2. Character coding

Computer manufacturers and software developers use numerous specific and non-compatible character codes. Meanwhile character coding norms have been standardized on an international level and are already used in some environments (ISO 8859 series). More recently the joined ISO 10646 and Unicode is proposed as a universal character set using a 16-bit code extensible to 32-bit in future editions. However, most of the existing environments strongly based on the ASCII code are not yet ready to implement character sets on multiple-octet code. Moreover SGML entities have been defined for encoding the characters of many different languages, and SGML is being used as a standard for the multilingual document interchange.

3.1.3. Text editing

Most of the languages are written horizontally from left to right, but some languages, such as Arabic or Hebrew, are written from right to left. As a consequence, the co-existence of languages in the same document, and particularly on the same line of the text, poses huge problems when inserting or deleting text zones. The example in figure 2 shows that it is often necessary to rearrange words to maintain the semantic coherence of a sentence.

The sequence "The fables of كليلة و دمنة" is stored:

T h e f a b l e s o f ك ل ي ل ة و د م ن ة

If we replace the Arabic word "و" by its English equivalent "and,"

the stored sequence becomes:

T h e f a b l e s o f ك ل ي ل ة and د م ن ة

but the displayed sequence should be "The fables of كليلة and دمنة"

Figure 2. Word rearrangements in a multilingual text

3.1.4. Printing

Printing multilingual texts suffers most obviously from the lack of printer fonts (essentially PostScript fonts). Many PostScript fonts are now available (freely or not) for Roman characters, but only a few fonts have been developed for the other character sets. Significant new efforts in this area include the OMEGA project activities [YHJP.95] for multilingual TeX and the works of C.Bigelow and K.Holmes [CBKH.95] in designing a UNICODE font *Unicode Lucida Sans* for editing and printing multilingual electronic documents.

3.1.5. Data exchange

With the rapid growth in the use of the Internet and the Wold Wide Web, the electronic transfer of multilingual documents is becoming more and more necessary. Until recently, only one part of the standard invariant characters of the ISO 646-IRV (ASCII) could allow a non-corrupted electronic text exchange, and multilingual documents could be transmitted safely only with the assistance of coding utilities such as *UUENCODE* and *BINHEX*. However the situation is improving: standards have been adopted on the Internet which allow the transfer of 8-bit characters without corruption in the TCP/IP protocol (for example, applications such *TELNET* and *FTP* are "8-bit clean"). In addition, the *MIME* norm allows uninterrupted data transfer in any circumstance by compressing and decompressing the files. However the emerging UNICODE standard still needs transformation formats (UTF-8) for information exchange. One must point out that the current guarantee for data transfer without corruption does not extend to the transfer of multilingual data.

3.2. Context analysis

In a text editor the context analyzer is a procedure which analyzes character codes provided through the keyboard or through a text's internal structure to generate individual or combined graphic forms (glyphs) according to specific and predefined rules. Most of the existing context analysis methods are local to specific text editors or to specific script fonts. Even though theoretical algorithms are general, the designed programs always depend on local environments, making the context analysis programs not modular and not portable. To make the context analysis programs easy one could modify the location of the glyphs in a font or duplicate them (see figure 3). But these costumized programs can not run with any other font [BOUA.90].

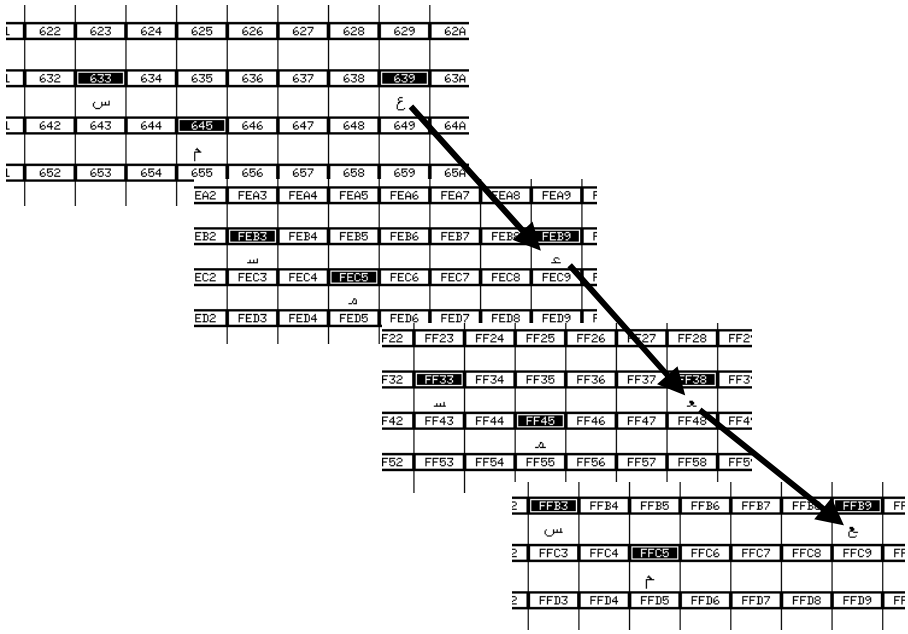


Figure 3[2]*. In this costumized font the glyphs are reached through regular translations*

Generally the context analyzer is associated with a set of character classes and a set of writing rules. It is preferable that this formalism should be easy to read and to understand by any user. Moreover the user may be able to modify the character classes or the writing rules. For example, the following set of Perl instructions (from the Arabjoin filter for rendering Arabic text [ARAJ.98]), even though they are effective, they are not easy to read or to modify by an average user:

```
$uchar[$i] =  $a && $final{$c} && $medial{$b}
|| $final{$c} && $initial{$b}
|| $a && $final{$b}
|| $isolated{$b}
|| $b;
```

[2] *This example has been designed using the **XMBDFED** font editor designed by Mark Leisher.*

3.2. Existing Unicode UNIX based multilingual text editors

Multilingual text editor design has frequently been carried out under the form of independent experimental studies, often leading to incompatible products which are difficult to use and do not conform to coding norms. However some Unicode-based multilingual text editors are in development or in improvement stages especially for the Unix system. Among this work, the OMEGA project includes a number of TEX extensions designed to improve multilingual text processing. It uses the ISO-10646/Unicode standard code with conversion mechanisms for other standard codes. Another text editor is MULE, a multilingual plain-text editor based on GNU Emacs. But Mule does not support Unicode and uses non-standard internal encoding. Otherwise there are some improvements for character conversion to Unicode (UTF-8) and to make Emacs 21 compatible with Unicode [MULE.97]. Another Unicode-based text editor on Unix is the YUDIT multilingual text editor for the X window system developed by G.Sinai [YUDI.98]. Lastly, the Unicode-based multilingual text editor we present here is UCEDIT (Unicode Editor) from the MUTT toolkit (Multilingual Unicode Text Toolkit) developed by Mark Leisher [MUTT.98]. MUTT is a Motif-based tool suite with many important features.

IV. The Multilingual Unicode Text Toolkit (MUTT)

4.1. CRL PANGEA Project

Begun in September of 1997, the Pangea project was supposed to be a three-year effort to create a standards based infrastructure for multilingual computing in general as well as natural language processing. The project lasted for one year and resulted in the MUTT (Multilingual Unicode Text Toolkit) for application developers (see figure 4). The project goal was to use a single character set internally along with low-level text handling facilities for strings and files, text entry and editing capabilities for as many languages as possible, multilingual searching and sorting capabilities, printing of multilingual text, available with Motif/X11 interface components for text display and entry and provide some basic multilingual capabilities for Java.

The MUTT toolkit is widely used, supporting hundreds of users in a variety of languages. Some of its strengths include a wide variety of easily extensible input methods, full Unicode text editing in the Motif/X11-based environment and initial Java support. MUTT currently supports approximately 40 languages with 95 input methods and conversion between Unicode and over 100 encodings and transliterations.

4.2. The Current MUTT Context Analyzer

The current context analyzer used for MUTT depends on a dictionary that maps groups of characters to glyphs. No script-specific information is used except for the few simple cases in Greek and Hebrew. The context analyzer also has builtin rules for common situations like handling non-spacing marks. The problem with this approach is that the context analysis character classifications and rules are very inflexible and changing them is a difficult process that requires that programs be recompiled.

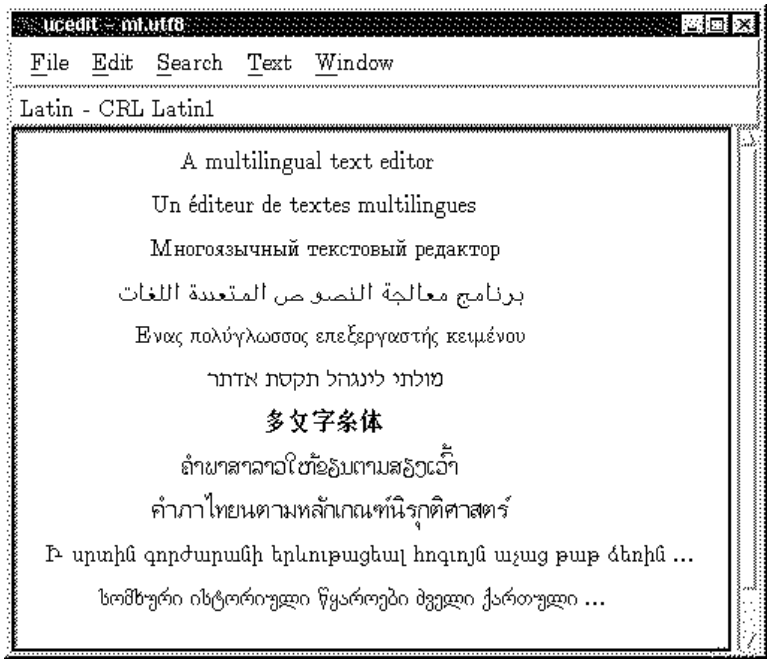


Figure 4. A screen of UCEDIT (MUTT)

V. The MUTTCA context analyzer

5.1. The MUTTCA architecture

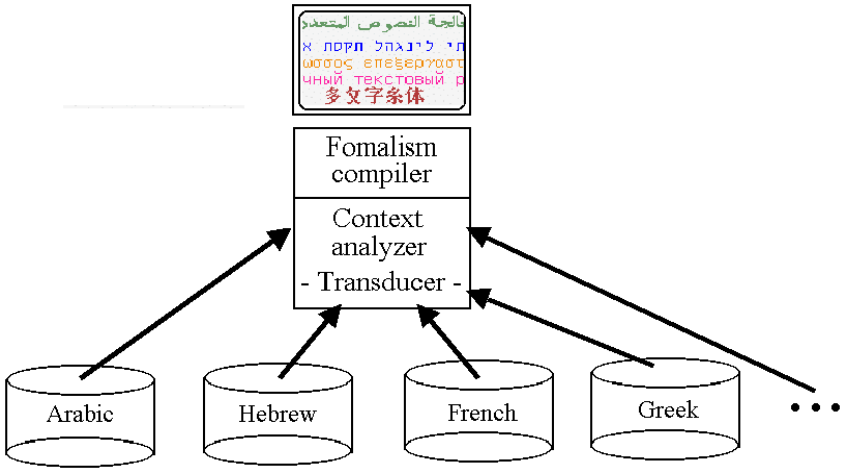


Figure 5. MUTTCA context analyzer architecture

The MUTTCA context analyzer is composed in two main parts:

- Language formalisms
- Programs

5.1.1. Language formalisms

Each language is represented on the basis of the following data:

- Language writing rules formalism (character classes and writing rules).
- Transliteration tables (for ASCII keyboards).
- Code conversion tables (for character code conversion between the different coding systems).

The "Transliteration tables" are simply classical tables containing correspondences between the ASCII keys and the characters of considered languages. The "Code conversion tables" are classical tables too containing correspondences between the different character coding systems.

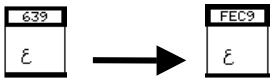
The structure of the writing rules formalism is as follows:

```
#
# COMMENTS SECTION
#
#

BEGIN TRANSCODING
 U+NNNN:U+NNNN
 U+NNNN:U+NNNN
 U+NNNN:U+NNNN
 ...
END TRANSCODING

# Comments
#

BEGIN CLASSES
 C1={U+NNNN,U+NNNN,U+NNNN,...}      /* comments */
 C2={U+NNNN,U+NNNN,U+NNNN,...}      /* comments */
 C3={U+NNNN,U+NNNN,U+NNNN,...}      /* comments */
 ...
END CLASSES

# Comments
#

BEGIN RULES
 Cn,Cm:n,"ap,bq";      /* comments */
 Cn,Cm:n,"ap,bq";      /* comments */
 Cn,Cm:n,"ap,bq";      /* comments */
 ...
END RULES
```

a. <u>TRANSCODING SECTION:</u>

This section includes, if they exist for the considered language, trancoding correspondences between the character codes and their corresponding glyph codes having the same glyph image. For example, the Arabic character "ARABIC LETTER AIN" which has the U+0639 code in the U+0600 to U+06FF block has a corresponding equivalent glyph image coded U+FEC9 in the Arabic Presentation Forms-B (U+FE70 to U+FEFF). In the "TRANSCODING SECTION", this correspondence is written:

U+0639:U+FEC9



b. <u>CHARACTER CLASSES SECTION:</u>

This section includes a classification of the characters of the considered language according to their common characteristics. For example, the Arabic characters can be classified as belonging to the following sets:

- Class of non-joining characters
- Class of dual joining characters
- Class of right joining characters
- Class of digits
- ...

c. <u>WRITING RULES SECTION:</u>

This section includes the different writing rules for each character class of the considered language, thus according to the scriptural rules commonly used for the language. The set of rules is built on the basis of a finite state transducer formalism. Each rule has the following syntax:

$$\mathbf{C_n\ ,\ C_m : n,\ "a_p\ ,\ b_q"\ ;}$$

Where:

- $\mathbf{C_n}$: Contextual Class, is the class containing the character (contextual character) near the cursor on the edited text.
- $\mathbf{C_m}$: Input Class, is the class containing the character entered on the keyboard.
- **n** : Flag indicating which character near the cursor to be deleted for replacing it by another glyph (0: no deletion, 1: deletion of the right character, -1: deletion of the left character).
- $\mathbf{a_p}$: Arithmetic step indicating the distance between the transcoded contextual character and the corresponding glyph in the compatibility area to be displayed.
- $\mathbf{b_q}$: Arithmetic step indicating the distance between the transcoded entered character and the corresponding glyph in the compatibility area to be displayed.

Even if this formalism seems a little complicated but it can be easily mastered by any user having knowledge of the considered language and who wants to modify or to improve the language formalism. Moreover this formalism makes the languages completely independent of the programs. Thus the modification of the language formalisms does not require the re-compilation of the programs.

5.1.2. The context analysis programs

The MUTTCA programs are composed in two types of programs:
- The formalism compiler program.
- The context analyzer program.

The formalism compiler program has the role of checking the structure of the formalism asked by the user when typing the text and loading it to be used by the context analyzer program. The context analyzer program has the role of expecting any entered character and analyzing it according to the corresponding language writing rules.

These programs are being written in JAVA which makes them portable in many software and hardware environments.

5.2. Context analyzer applications

5.2.1. Dynamic composition of accented characters

The Unicode standard allows for the dynamic composition of accented forms. The combining characters are encoded following the base characters to which they apply. For example inputting and encoding the French composed character "**ê**" through the sequence "**e**+^" is naturally logical and easier for both the user and the computer than sequences such "**compose_key**+**e**+^" or "^+**e**", etc. The MUTTCA context analyzer includes rules for composing accented characters with an escape mechanism to eventually avoid accentuation. For example, accented characters in French can be composed using the following rules:
- **e** + ' => **é** ;
- **a** + ` => **à** ;
- **e** + **ESC** + ' => **e'** ;
- etc.

5.2.2. Interpreting Characters and Rendering Glyphs

The Unicode Standard does not define glyph images, it defines how characters are interpreted, not how glyphs are rendered. In this case, the context analyzer program is responsible for rendering glyphs on the screen. An abstract character such "ARABIC LETTER AIN" which has the U+0639 code value can have different visual representations (called shapes or glyphs) on screen or paper, depending on context (see figure 6). Different scripts which are part of Unicode can have different writing rules for rendering glyphs and also composite characters, ligatures, and other script-specific features. The results on screen or paper can differ considerably from the prototypical shape of a letter or character. For example:

- Greek: **σ** (beginning and middle of a word), **ς** (word ending)
- German: **s** + **s** => **ß** ; **s** + **ESC** + **s** => **ss** ; etc.
- Arabic:

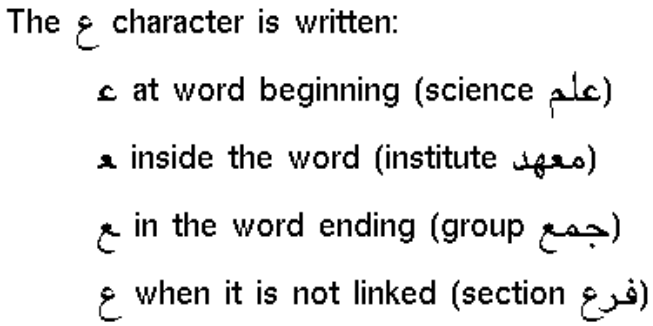


Figure 6. Character associated glyphs

5.2.3. Arabic vowels

The Arabic vowels are super and subscript non-spacing marks combined with the characters (consonants). The Unicode Standard does not specify a sequence order in case of multiple vowels applied to the same character since there is no possible ambiguity of interpretation. But the problem is related to the number of vowels for the same character. The vowel ARABIC SHADDA (U+0651) is the only one which can be combined with only one of the other vowels and the vowel ARABIC SUKUN (U+0652) can not be combined with any other vowel. This exceptions can be taken in consideration by the MUTTCA context analyzer through specific rules which are being added.

5.3. Example for rendering Arabic script-based glyphs

5.3.1. Arabic script and Arabic block in Unicode

The Arabic script is extended for representing a number of other languages than Arabic, such Persian, Urdu, Pashto, Sindhi and Kurdish. It is written from right to left and it is cursive. The same letter may be written in different forms depending on its position in the word. The Unicode standard encodes the basic Arabic characters in the same relative positions as in **ISO-8859-6**. The different glyphs of characters are represented in the Compatibility area of Unicode.

5.3.2. Compatibility area and Arabic script-based glyphs in Unicode

Compatibility characters are those that would not have been encoded except for compatibility because they are variants of characters already coded. Most of the compatibility characters are located in the "Compatibility area" (Arabic contextual form glyphs, Arabic ligatures, etc.). The principle of using compatibility characters in Unicode coded texts is to maintain the main information in the text. Replacing a character by its compatibility equivalent character may change the coding information in the text but it may not change the information which is necessary for text processing such as sorting or searching.

5.3.3. Standard Arabic script-based glyphs adopted to demonstrate MUTTCA

The Arabic script-based classes and writing rules used in MUTTCA correspond to the **"Joining Classes"** and the **"Joining Rules"** defined in the Unicode Standard. The Arabic rendering rules are applied on the range of the basic Arabic characters of Unicode 2.1 (U+0600 to U+06FF or exactly U+060C to U+06F9) and uses the contextual glyphs of the Arabic Presentation Forms-A (U+FB50 to U+FDFF) and the Arabic Presentation Forms-B (U+FE70 to U+FEFF). But in our example we use the range of the Arabic glyphs (U+FE80 to U+FEFF), see figure 7.

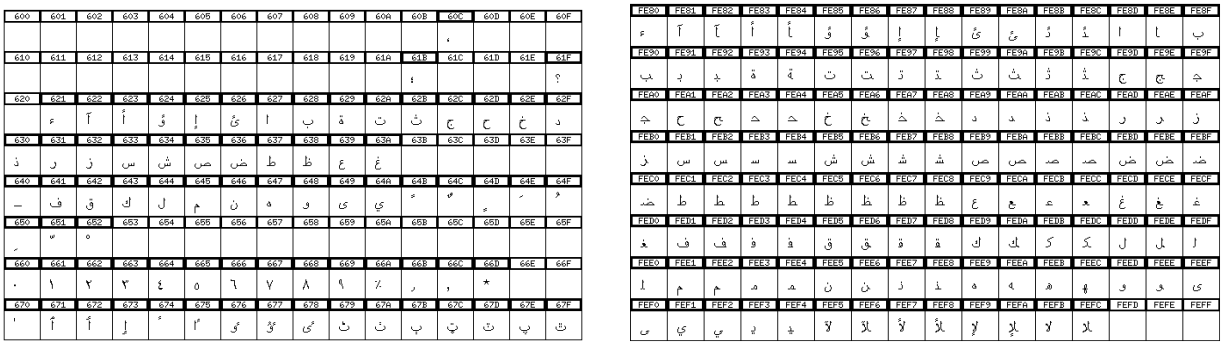


Figure 7. Arabic characters and glyphs from Unicode adopted to demonstrate MUTTCA

We may mention that our goal in this paper is to present a modular and standard formalism for the Unicode-based context analysis. The example that we propose is limited but the analysis could be easily generalized by simply adding new character classes and new writing rules.

5.3.4. Example of a simple Arabic formalism

Here is an extract of a simple formalism for the context analysis of Arabic:

```
#
# The Multilingual Unicode Text Toolkit (MUTT)
# Computing Research Laboratory, New Mexico State University
#
# Arabic writing rules
# Copyright (c) 1998 - CRL
#
# This set of rules is compatible with Unicode.
# The rendering rules are applied on the range of the basic
# Arabic characters of Unicode (U+0600D to U+06FF / U+060C to U+06F9)
# and use the contextual glyphs of the Arabic Presentation
# Forms-B (U+FE70 to U+FEFF).
#

# TRANSCODING
```

```
BEGIN TRANSCODING

U+0621:U+FE80
U+0626:U+FE89
U+0628:U+FE8F
U+062A:U+FE95
U+062B:U+FE99
U+062C:U+FE9D
U+062D:U+FEA1
U+062E:U+FEA5
U+0633:U+FEB1
U+0634:U+FEB5
U+0635:U+FEB9
U+0636:U+FEBD
U+0637:U+FEC1
U+0638:U+FEC5
U+0639:U+FEC9
U+063A:U+FECD
U+0641:U+FED1
U+0642:U+FED5
U+0643:U+FED9
U+0644:U+FEDD
U+0645:U+FEE1
U+0646:U+FEE5
U+0647:U+FEE9
U+064A:U+FEF1
U+0622:U+FE81
U+0623:U+FE83
U+0624:U+FE85
U+0625:U+FE87
U+0627:U+FE8D
U+0629:U+FE93
U+062F:U+FEA9
U+0630:U+FEAB
U+0631:U+FEAD
U+0632:U+FEAF
U+0648:U+FEED
U+0649:U+FEEF

END TRANSCODING

BEGIN CLASSES

# Control characters
# ------------------
C1={U+0000-U+0007,U+0009-U+001A,U+001C-U+001F}  /* control */
C110={U+0008}                                   /* backspace */
C111={U+007F}                                   /* delete */
C5={U+0020} /* space */

# Symbols
# -------
C2={U+0021-U+002F,U+003A-U+003E,U+0040,U+005B-U+0060,U+007B-U+007E)
C200={U+061B,U+061F,U+066A-U+066D}  /* Arabic symbols */
```

```
# Digits
# ------
C3={U+0660-U+0669}

# Non-joining characters
# ----------------------
C7={U+FE80} /* hamza */

# Dual joining characters
# -----------------------

# Independent forms of dual joining characters

C60={U+FE89,U+FE8F,U+FE95,U+FE99,U+FE9D,U+FEA1,U+FEA5,U+FEB1,U+FEB5,
     U+FEB9,U+FEBD,U+FEC1,U+FEC5,U+FEC9,U+FECD,U+FED1,U+FED5,U+FED9,
     U+FEDD,U+FEE1,U+FEE5,U+FEE9,U+FEF1}

# Initial forms of dual joining characters

C61={U+FE8B,U+FE91,U+FE97,U+FE9B,U+FE9F,U+FEA3,U+FEA7,U+FEB3,U+FEB7,
     U+FEBB,U+FEBF,U+FEC3,U+FEC7,U+FECB,U+FECF,U+FED3,U+FED7,U+FEDB,
     U+FEDF,U+FEE3,U+FEE7,U+FEEB,U+FEF3}

# Medial forms of dual joining characters

C62={U+FE8C,U+FE92,U+FE98,U+FE9C,U+FEA0,U+FEA4,U+FEA8,U+FEB4,U+FEB8,
     U+FEBC,U+FEC0,U+FEC4,U+FEC8,U+FECC,U+FED0,U+FED4,U+FED8,U+FEDC,
     U+FEE0,U+FEE4,U+FEE8,U+FEEC,U+FEF4}

# Final forms of dual joining characters

C63={U+FE8A,U+FE90,U+FE96,U+FE9A,U+FE9E,U+FEA2,U+FEA6,U+FEB2,U+FEB6,
     U+FEBA,U+FEBE,U+FEC2,U+FEC6,U+FECA,U+FECE,U+FED2,U+FED6,U+FEDC,
     U+FEDE,U+FEE2,U+FEE6,U+FEEA,U+FEF2}

# Right joining characters
# ------------------------

# Independent-initial forms of right joining characters

C70={U+FE81,U+FE83,U+FE85,U+FE87,U+FE8D,U+FE93,U+FEA9,U+FEAB,U+FEAD,
     U+FEAF,U+FEED,U+FEEF}

# Medial-final forms of right joining characters

C71={U+FE82,U+FE84,U+FE86,U+FE88,U+FE8E,U+FE94,U+FEAA,U+FEAC,U+FEAE,
     U+FEB0,U+FEEE,U+FEF0}

END CLASSES

BEGIN RULES

C*,C*:0,"b";      /* non particular characters: not changed */
C*,C110:-1,"";          /* backspace */
C*,C111:1,"";           /* delete */
```

```
C3,C3:+1,"a,b";   /* Numbers */
C*,C5:0,"b";            /* space */
C*,C7:0,"b";            /* hamza */
C*,C60:0,"b";           /* dual joining characters: independent form */
C*,C70:0,"b";     /* right joining characters: independent-initial form */

# Rules for dual joining characters
# ---------------------------------

# state: dual joining character in independent form
C60,C60:+1,"b+1,a+2";         /* dual joining: final */
C60,C70:+1,"b+1,a+2";         /* right joining: final */

# state: dual joining character in initial form
C61,C*:+1,"b,a-2";      /* dual joining independent + any */
C61,C110:-1,"";         /* backspace */
C61,C111:1,"";               /* delete */
C61,C60:0,"b+3";        /* dual joining: medial */
C61,C70:0,"b+1";        /* right joining: medial */

# state: dual joining character in medial form
C62,C*:+1,"b,a-2";      /* dual joining final + any */
C62,C110:-1,"";         /* backspace */
C62,C111:1,"";               /* delete */
C62,C60:0,"b+3";        /* dual joining: medial */
C62,C70:0,"b+1";        /* right joining: medial */

# state: dual joining character in final form
C63,C60:+1,"b+1,a+2";   /* dual joining: final */
C63,C70:+1,"b+1,a+2";   /* non regular: final */

END RULES
#
```

VI. Conclusion

The method we presented in this paper has previously been demonstrated for different languages. The language formalism is easy to understand and to modify by the user. Indeed some users have improved some language support simply by adding new character classes or new writing rules or modifying them. For instance the previous example for Arabic rules can be easily improved for rendering the different forms of the LAM ALIF Arabic ligature or the Arabic vowels. However at the time of writing this paper, the writing rules syntax <**C_n,C_m:n,"a_p,b_q";>** is only bi-dimensional (one contextual class and one input class). This syntax is being improved since an entered character between two characters may change the glyphs for both the right and the left characters. In addition, character classes can also be extended to "string classes" to use this formalism for applications processing character strings or words (morphological or syntax analysis, etc.). Some other improvements will be made on the JAVA programs to make the MUTTCA context analyzer portable and reusable in many different hardware and software Unicode based environments.

VII. References

[ARAJ.98] Roman Czyborra, The Arabjoin script for rendering Arabic text.
http://czyborra.com/unicode/arabjoin

[BOUA.97] Boualem A.M., Harié S., “MtScript: a multilingual text editor”, Computers and the Humanities, Volume 31, No. 2, Kluwer Academic Publishers, 135-151, 1997.

[BOUA.90] A.M. Boualem, "The multilingual terminal", research report, INRIA Sophia Antipolis, January 1990, 1-4.

[CBKH.95] C. Bigelow, K. Holmes, “The design of a UNICODE font”, version française dans le Cahier GUTenberg n°20, May 1995, 81-102.

[LABO.95] A. Labonté, "Input methods to enter characters from the repertoire of ISO/IEC 10646 with a keyboard or other input devices". ISO/CEI JTC1/SC18/GT9 Working Draft, February 1995. ftp://ftp.funet.fi/pub/doc/charsets/ucs-input-methods

[LIST.98] Archives of the Unicode mail list <unicode@unicode.org>

[MULE.97] The MULtilingual Enhancement to GNU Emacs
http://www.etl.go.jp/~mule/MulePage.html

[MULT.97] MULTEXT project was coordinated by the CNRS "Parole et Langage" Laboratory for building standard methods for linguistic data representation and developing language processing tools for about fifteen languages.
http://www.lpl.univ-aix.fr/projects/multext/

[MUTT.98] The Multilingual Unicode Text Toolkit, CRL, New Mexico State University.
ftp://crl.nmsu.edu/CLR/multiling/unicode/

[UNIC.98] The Unicode Web site
<http://www.unicode.org>

[UNST.96] The Unicode Standard, Version 2.0, The Unicode Consortium, Addison-Wesley Developers Press, 1996.

[YHJP.95] Y. Haralambous, J. Plaice, "Ω, une extension de TEX incluant UNICODE et des filtres de type Lex", Cahier GUTenberg n°20, May 1995, 55-79.

[YUDI.98] Yudit Unicode editor for X window
ftp://sunsite.unc.edu/pub/Linux/apps/editors/X/

VIII. Biographies

Malek Boualem received a PhD in computer science from Nice university, France (1993). He has been an associate professor at the Provence University and worked with the French CNRS Research Center. M.Boualem is specialized in multilingual software and is one of the authors of MtScript (a multilingual text editor) which won the 1996' CNRS/ANVIE prize. Now he is working with the CRL in designing a Unicode-based multilingual toolset.

Mark Leisher is a programmer with the Computing Research Lab of New Mexico State University specializing in the development of multilingual text processing software, particularly based on Unicode.