Acharya - A Text Editor and Framework for working with Indic Scripts

Krishnakumar V
Software Developer,
A-10/11 DMC New Colony,
Salem-636012
v.krishnakumar@gmail.com

Indrani Roy
Fellow,
Central Institute of Indian Languages,
Manasagangotri, Mysore-570006
indraniroy@gmail.com

Abstract

This paper discusses an open source project[1] which provides a framework for working with Indian language scripts using a uniform syllable based text encoding scheme. It also discusses the design and implementation of a multi-platform text editor for 9 Indian languages which was built based on this encoding scheme.

Keywords: Syllabic Encoding, Text Editor implementation, Transliteration

1 Introduction

1.1 Background

Back in 2004, ETV (Eenadu Television), Hyderabad, felt a need for a text editor to prepare news scripts for its regional news channels. A news production environment has its unique set of requirements including speed, efficiency, robustness etc. The software that was in use had various technical limitations including high CPU usage, lack of portability across the diverse set of platforms that were in use in ETV. Using UNICODE editors were unsuitable as the correctness of the output largely depended on the quality of the shaping engine in use and back then it produced inconsistent results. Apart from that ETV's real-time graphics engines had trouble shaping UNICODE text in Indic scripts.

A multilingual editor for Indic scripts had been developed at IIT Madras[2]. The team at IIT Madras favoured further development under an open source project. As a result an Open Source Project was started. The immediate aim of the project was to rewrite the editor, remove its limitations and redesign it for use in a News Production environment using modern design and development tools.

1.2 Acharya Text Editor

Acharya is a multi-platform text editor that supports Asamiya, Bangla, Devanagari, Gujarati, Kannada, Malayalam, Oriya, Punjabi, Tamil and Telugu. In addition to these scripts, it can also display text in Braille and RomanTrans using transliteration. It achieves this functionality by storing Indic text in syllabic units instead of characters as most other editors do. Although it uses a custom encoding, the editor supports conversion of text to standard encodings like ISCII (ISCII, 1993) and UNICODE (UTF-8). It can export documents as RTF and PDF files. In the case of PDF documents the fonts are embedded within so that they can be exchanged freely without the need for local language fonts to be available on the viewing system. The editor supports editing multiple documents through a tabbed interface. It includes standard features like clipboard support, finding strings and interfacing with the platform's printing system. To assist text entry, it has a word completion mechanism based on a dynamic dictionary. Currently, it runs on all major platforms including Windows, Mac OS X and various Linux distributions.

The editor consists of a small but extensible library for processing syllables, a text editing component and the rest of the user interface. Section 2 of this paper describes the library. The syllabic encoding along with its features is described in section 2.1. Section 3 describes the text editing component. Conclusion is offered in section 4 along with some

[1]http://imli.sourceforge.net
[2]http://acharya.iitm.ac.in

information on related work-in-progress.

2 Syllable Library

The syllable library provides an implementation of the syllabic encoding (described in the Section 2.1) which allows text to be represented directly as syllables instead of as characters. The library implements the rules of syllable composition, provides input methods, and routines for conversion of syllables to/from other encodings like ISCII and UNICODE. All of this functionality is exposed through opaque data types and a small API operating on them.

2.1 Encoding

As mentioned above, text is encoded directly as syllables. The encoding used is a modified version of the syllabic encoding scheme (Kalyanakrishnan, 1994) developed by Prof. R. Kalyana Krishnan at the Systems Development Lab, IIT Madras. This encoding tries to capture the syllabic nature of Indic scripts. In this encoding, each syllable can be specified as

$$C_{m=0..4}V_{n=0..1}$$

Where C is the consonant and V is the vowel. This means that each syllable can be one of V, C, CV, CCV, CCCV and CCCCV combinations. The initial C is the base consonant and the subsequent Cs represent conjunct combinations. The memory representation of each syllable is a 16-bit value with the following bit distribution[3]:

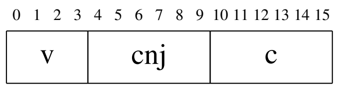

Figure 1: syllabic encoding

With this arrangement, it is possible to have upto 64 consonants with 16 vowels each. The bits 4-9 indicated by the *cnj* field hold the index into the base consonant's conjunct table. This table holds the values of the constituent consonants OR'ed into a 32-bit integer. For example:

The syllable *ndrA* is stored in the following way.

[3]shown in little-endian byte order

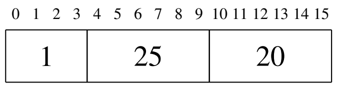

Figure 2: ndrA syllable

Comparing with figure 1, the vowel code is 1 which stands for the vowel *aa*. Similarly, the base consonant code is 20 and represents the consonant *na*. The conjunct code 25 is an index into the conjunct table of the consonant *na*. The value that will be stored at index 25 is shown in figure 3:

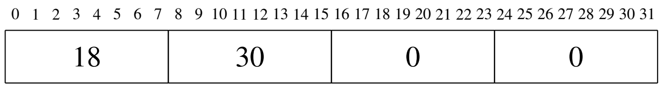

Figure 3: value at index 25 of the conjunct table of *na*

Bits 0-7 contain the consonant code of the first level conjunct, bits 8-15 of the second level conjunct and bits 16-23 of the third level. Bits 23-31 are reserved for future expansion. In this case, there are 2 consonants in addition to the base consonant *na*. The values 18 and 30 represent the consonants *da* and *ra* respectively.

These codes are specified in the files generic.vow[4], generic.con, generic.spl for vowels, consonants and special characters respectively. The conjunct combinations are specified in the file generic.cnj in this fashion:

```
ra:  ta (ta (ya))
```

Here *ra* is the base consonant and the line defines three conjuncts namely *rt(ra + ta)*, *rtt(ra + ta + ta)* and *rtty(ra + ta + ta + ya)*. The last conjunct is an example of a conjunct where all the four levels are used *ra + ta + ta + ya*. It occurs for example in the Oriya word *marttya*. Each pair of parenthesis stands for a level of conjunct. More complex conjuncts can be added by nesting them within parentheses. However, the current implementation supports only up to three levels of nesting. The generic.cnj file as it stands now defines 1240 conjuncts. When the 16 vowels are taken into account, we get a total of 19840 syllables. An additional set of 32 syllables for

[4]These files are named generic because the values they define are common to all scripts supported by the framework

local language numbers, punctuation and other special characters increases the total number of valid syllables to 19872.

This scheme also accommodates English text in the ASCII encoding by using a consonant code of 62 and the lower eight bits representing the ASCII code. It also has a few special syllable codes for switching scripts to be embedded within the data stream although they are not used within the editor.

2.1.1 Compactness

The 16-bit syllable value stands on its own and does not correspond to UNICODE or ISCII or any other encoding for that matter. One particular feature of this scheme is the compactness and the inherent compression. For example,

मर्त्त्य

the above word – *marttya (m + r + halant + t + halant + t + halant + y)*, in UNICODE UTF-16 encoding will be encoded as 8 16-bit values. UTF-8 requires 26 bytes to encode the word. In ISCII, it can be encoded in 8 bytes whereas in this encoding, the above word requires just 2 syllables of 16 bits each.

2.1.2 Rendering

One other aspect of this encoding is that there is a separation of content and its visual representation. On one hand, this means that text processing applications need not worry about dealing with display related issues like glyph reordering and proper placement of glyphs using the various zero width space characters as is the case with character based encoding schemes including UNICODE. On the other hand, this separation means that to display a syllable some kind of map is required between the syllable and its visual representation (glyphs). This mapping is font dependent when non standard fonts using the ISO8859-1 encoding are used but UNICODE fonts can also be used. Currently static tables are used to provide a one to one mapping between the syllables and its corresponding glyphs. This is a trade-off where memory is traded for quick display of glyphs. There is no need for cluster identification as the information is already there in the form of syllables. This lookup is $O(1)$ whereas in shaping engines like Pango, this operation is $O(n)$. These static tables can also be useful in environments where shaping engines like Uniscribe and Pango are not available or cannot be used.

2.2 Input Methods

The library provides routines for input methods which can be used in conjunction with the platform specific keyboard processing functions to support direct key-in of syllables. Currently, it includes input methods for INSCRIPT (ISCII, 1993) and Phonetic keyboards. However, the mechanism is general enough to add additional keyboard layouts including ones that work only with specific scripts. In the current implementation, the input methods load their respective data and delegate the bulk of the work to the syllable processing routine.

2.3 Unicode Conversion

UNICODE is the de-facto standard for storing and rendering text so conversion to/from UNICODE is essential for integration with other tools. UNICODE integration can be achieved either by having a static syllable-to-glyphs map with UNICODE fonts or a separate text codec to do the syllable to UNICODE conversion. In the current implementation, the text codec strategy is used to convert the syllables to its corresponding UTF-8

3 Editor Implementation

3.1 Text Storage

The most important data structure in a text editor is the one that stores text sequences. A poor choice will directly affect the performance of the editor as almost all editing operations work on these text sequences. A survey of popular data structures for text sequences is presented in (Crowley, 1998). The two most popular choices are *gap buffer* and *piece table*. A gap buffer is basically an array that has a gap which is moved to the point of edit so that the text that is entered is copied to the gap without further allocation of storage. The gap shrinks and expands on insertion and deletion of text respectively. Gap buffers have the advantages of being simple to implement and offer direct access to the text. The downside is they incur a copying overhead when the gap is not at the point of editing operations as text needs to be copied to either side of the gap. Also gap buffers are not suitable if the text has attributes and runs (run is a chunk of text that belongs to the same script) of text need to be stored. A multilingual text editor has both these requirements. To implement this in a gap buffer would require a parallel style or script buffer (Gillam, 2002) to track and demarcate the runs and its corresponding font changes. Whenever the gap is moved and text added or deleted, the style buffer would need to be updated as well. This can quickly get cumbersome when multiple scripts are used in the same document.

A piece table is an alternative to the gap buffer that does not suffer from these problems. In a piece table, the text is immutable and is always appended to the sequence. However, the logical order that is shown in the view is maintained by a separate list of *piece* descriptors. A piece includes information such as the script, the start and end positions within the sequence etc. So, when the user copies/deletes the text, it is the piece descriptors that are moved around and not the actual text. By introducing this level of indirection, the piece table solves the problem of copying overhead when text is moved around. However, the drawback is that the text is no longer accessible directly. To locate a position in the text sequence the editor has to traverse the piece table and locate the piece which contains the position. Despite this drawback, the piece table data structure offers a number of advantages – it is a persistent data structure and because the original text is never destroyed operations like undo and redo lend to a straightforward implementation by restoring the links between the removed pieces from the undo and redo stacks respectively. The other advantage of piece tables is that there is a direct mapping from script runs to pieces.

The piece table in this editor is implemented as a *piece chain* (Brown, 2006) – a circular linked list with a sentinel node. Since the piece chain is a linked list, the problem of linear addressing is pronounced ($O(n)$). To deal with this problem, the piece chain caches the last accessed (piece, position) pair to utilize the locality of reference (Wirth and Gutknecht, 1992). This small optimization has so far worked out well in practice as there is a strong locality of reference in text editing. To store the syllables itself, the deque class from the standard C++ library is used. It is a scalable data structure that guarantees efficient insertion of large amounts of text at the tail position. Another important issue is that of cursor movement. In the editor, syllables are displayed using a variable number of glyphs. Allowing the cursor to be positioned in the middle of a syllable would make it possible to delete that particular syllable partially which would make the data inconsistent. Therefore all cursor related operations including selection should be limited to syllable boundaries. This is achieved by using a separate deque object for storing the *width* of each syllable where width is the number of glyphs that the syllable is represented by visually. This additional information is used when mapping the syllable position in the text storage to its corresponding glyph position in the view and vice-versa.

3.2 File Format

As mentioned in section 2, the editor works in terms of syllables and not characters. While syllables can be stored to disk files directly, to retain compatibility with other Indian language applications, the editor stores the text to files in the 7-bit ISCII encoding. 7-bit ISCII is a simple and efficient format where English text in ASCII is stored as is and the text in Indic scripts are stored using code points from the upper half of the character set (128-255). Like the syllabic encoding and unlike UNICODE, ISCII uses a uniform representation for all the Indic scripts. Each script run starts off with a code that identifies the language of the run. This makes run detection very simple to implement. When the editor saves a document, all the syllables are broken down to their constituent ISCII characters and written to disk. Similarly, when a file is opened, the ISCII data is converted to the syllabic representation using the ISCII codec routines from the syllable library and from then on only the syllables are used.

3.3 Utilities

3.3.1 Transliteration

Because of the uniformity of the encoding all the supported scripts have a means of displaying the same set of syllables hence transliteration in this encoding is basically changing the script code for the user-selected piece of text and notifying the view that is displaying the text to re-render the selected text using the font of the target script. What this means is that transliteration as supported by this encoding will survive a round-trip conversion without any loss of data. An example to illustrate the last point:

Supposing in a multilingual document, the user selects the character ग (*ga* in Hindi) and transliterates to RomanTrans, the editor will display `ga`. Internally, the *ga* syllable is stored in the following way:

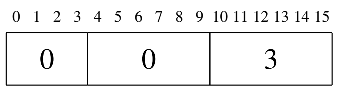

Figure 4: ga syllable

The text storage tracks the script code for every syllable. When the above syllable is converted to RomanTrans, the text storage object does not modify the syllable but changes only the script code to RomanTrans and notifies the view displaying the text. The view upon receiving the notification from the storage object then re-renders the *ga* syllable using RomanTrans's font map. Similarly, when the user once again changes to Tamil, the editor correctly displays க (*ka* in Tamil) this time using Tamil's font map which specifies that *ga* should be mapped to the same glyph as *ka*. If the user once again changes the script back to Hindi, the letter ग (*ga* in Hindi) is displayed correctly.

The above scheme is possible because the text content is kept separate from the actual display of text and more importantly the text content itself is stored as syllables which are the fundamental units of transliteration.

3.3.2 Word Completion

Word completion, also known as *auto-completion* under certain applications, is a handy feature to have specially for typing lengthy and frequently used words fast. In its current implementation, this editor does not automatically complete words. The user needs to trigger it explicitly. This is mainly to keep the editor less disruptive (in terms of the typing flow) and also to keep the implementation simple. When typing long words, the user after typing the first few characters can trigger the pop-up with possible completions by means of the designated keyboard shortcut. The list of words that appear in the completion box is obtained by doing a prefix search (of what the user had typed so far) on a dynamic dictionary. This dictionary is implemented using *ternary search trees* (Bentley and Sedgewick, 1998). A ternary search tree (henceforth TST) is a versatile data structure that combines the time efficiency of tries and the space efficiency of binary search trees. TSTs are generally faster than hashtables and offer much more functionality than simple key look-ups because they maintain the ordering of the data stored within. When augmented with additional information, TSTs can also be used for implementing spell checking and by using a fixed edit distance, alternative word suggestions as well. A full description is beyond the scope of this short paper. However, (Bentley and Sedgewick, 1997) provide all the details.

4 Conclusion & Future Work

Inside ETV, this editor has been in production use since 2005. It serves as the primary tool for document preparation in Indian languages. The fact that it is being used in a news production environment is a testament to its stability and the overall soundness of the syllabic encoding scheme.

At the time of writing, support for speech output of text is being worked on. Since the text is stored in terms of syllables, speech output is obtained by breaking the syllables into phonemes and sending them to a concatenative speech synthesis engine (currently we are using Mbrola). The editor already has support for Braille output using transliteration and this output can be fed to a braille printer after minor post processing the tools for which are being worked on. Work is on for incorporating tools like morphological analyzers into this framework for building advanced linguistic applications.

This is an ongoing effort in the form of an open source project. The full source code for the entire system is provided on the website and help is available on the mailing list.

Acknowledgements

We are grateful to Prof. R.Kalyana Krishnan, Systems Development Lab, IIT Madras for guidance throughout this project, Mr. B.Ganesh ex-CTO of Technical Services Department of ETV for initiating this project and contributing to it, Mr. Anirban Sam of ETV for coordinating the testing of this software and providing detailed bug reports, Mr. G.Venugopal, Systems Manager, ETV for the administrative support that facilitated distributed development. Finally, ETV deserves a special mention for supporting the development of this open source project.

References

Charles Crowley. 1998. *Data Structures for Text Sequences*. http://www.cs.unm.edu/~crowley/papers/sds.pdf

1993. Indian Script Code for Information Interchange. In *Bureau of Indian Standards*.

James Brown. 2006. Editing Text with Piece Chains. http://catch22.net/tuts/editor17.asp

James Brown. 2006. Unicode Text Editing. http://catch22.net/tuts/editor18.asp

Jon Bentley and Robert Sedgewick. 1998. Ternary Search Trees In *Dr. Dobbs Journal*. http://www.ddj.com/windows/184410528

Jon Bentley and Robert Sedgewick. 1997. Fast Algorithms for Sorting and Searching Strings. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, New Orleans, USA.

Niclaus Wirth and Jurg Gutknecht. 1992. *Project Oberon - The Design of an Operating System and Compiler*. ACM Press/Addison-Wesley Publishing Co. New York, USA.

R.Kalyanakrishnan. 1994. Syllable level coding for Indian languages. http://acharya.iitm.ac.in/software/docs/scheme.php.

Richard Gillam. 2002. *Unicode Demystified - A Practical Programmer's Guide to the Encoding Standard*. Addison-Wesley Longman Publishing Co., Inc., Boston, USA.