Analysis of the different methods to encode SignWriting in Unicode

Guylhem Aznar, Patrice Dalle and Clément Ballabriga
Équipe TCI and Master Camsi
IRIT-UPS, 118 route de Narbonne, 31062 Toulouse Cedex 4
E-mail: aznar@irit.fr, dalle@irit.fr, ballabriga@7un.net

This paper lists, evaluates and discuss the solutions to encode SW in Unicode.

Abstract: SignWriting is the most complex and popular writing formalism for sign languages. Unicode is the most popular encoding of characters aimed at unifying the various language-oriented encodings into a single format supporting every human language. This paper focuses on the first functional layer, which gives a correspondence between a SignWriting sign and a series of bytes. This is one of the prerequisites to represent a sign language electronically. The different possibilities to encode a given SignWriting sign are evaluated and compared on different criteria : the Unicode space requirements, the number of bytes the storage will require, the mathematical complexity and the side advantages offered. Keeping as much as possible of the information on how signs are written and entered, and offering capabilities to easily compare the symbols that compose these signs is also considered, so that the encoding can serve to study and compare how SignWriting is written. A reference encoding is then proposed, to serve as a basis for the next layers. Other bi-dimensional writing formalisms, currently not supported by Unicode, are considered to extend the presented work.

1. SIGNWRITING

A sign language *sign*, corresponding to a meaning, is transcribed in a SignWriting (SW) *sign*, composed of symbols, positioned on a 2D canvas called a *signbox* (Sutton, 1995).

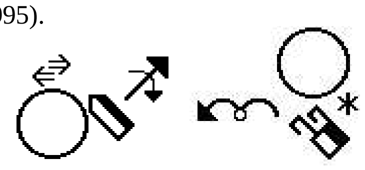


Figure 1 : two SW signs

Symbols correspond to static or dynamic positions or movements of the human body, and are described in the SymbolBank norm from the IMWA (Sutton, 2004). An analysis of SSS-2004 shows 25 973 symbols, divided into 8 categories, 10 groups, 50 elements, 5 variations, 6 fillings and 16 rotations.

Choosing a list of symbols from the SSS, and positioning them into a 2D signbox, results in a infinite number of combinations.

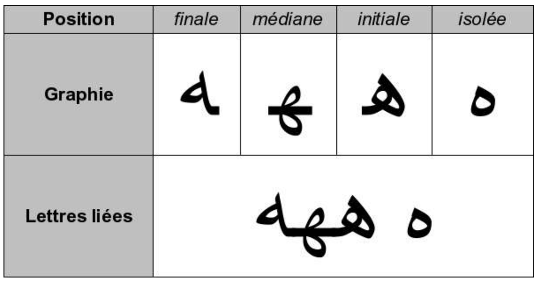
2. UNICODE

Unicode is simply an assignment of characters into code points. Unicode currently offers $2^{20}+2^{16}$=1 114 112 codes, split into 17 planes of 2^{16} = 65 536 codes. Only 100 000 characters have been assigned so far, i,e, 10% of the available code space. The first plane, also called Plane 0 , is used for existing encodings, to allow direct compatibility. It features a private area, a concept inherited from Asiatic encodings, used by systems or applications which must encode non standard characters. Plane 1 is used for ancient languages, mathematical and numerical symbols, and Plane 2 for rare, mostly historic, Chinese characters. Plane 14 currently contains non-recommended language tag characters and variation selection characters. Plane 15 and Plane 16 are fully reserved for private use. Unicode simply assigns a unique number to each character. But file storage, transfer and processing require handling these numbers following a mapping method. Unicode offers different ways to do such mappings, depending on constraints such as available storage space, compatibility requirements and interoperability, through various UTF and UCS. mappings. UTF-32 is the best choice when storage space or compatibility are less important than software uniformisation, and will be used by default in this paper in an hexadecimal transliteration U+$X_1X_2X_3X_4$, where X_n is the nth byte of an hexadecimal value X.

UNICODE ENGINES

Unicode does not deal with fonts : it simply matches first bits and codes, following a mapping like UTF-32, then codes and characters, following the standard arrangement of Planes. There is no bijection between the code and its graphical representation called “glyph”, unlike in traditional encodings such as ISO 8859-15 “Latin 9”: there are many ways to display a similar glyph. Matching one or more characters with a glyph is the job of the Unicode engine. For example, the French glyph “è” can be obtained through a single character called “LATIN SMALL LETTER E WITH GRAVE” which is given code U+00E8. Yet the same glyph can be displayed with the two characters “LATIN SMALL LETTER E” and “MODIFIER LETTER LOW GRAVE ACCENT”, respectively U+0065 and U+02CE. The latter could even be replaced by “COMBINING GRAVE ACCENT” U+0300 ! Such grammar, required to compose a glyph from characters, is called an Unicode engine (Fanton, 1998). There are many existing Unicode engines. The engine uses a font to represent the glyphs. There are currently less than ten “pan-Unicode” fonts, i.e. capable of supporting most of the glyphs Unicode can offer.

Some languages such as Arabic or Devanagari require a specific treatment of the glyphs. For example, in the Arabic alphabet, most glyphs have four allographs, depending on the position of the letter in a word : isolated, initial, median, final. Such post-treatment of the glyphs into graphs is done by the Unicode engine.



Position	*finale*	*médiane*	*initiale*	*isolée*
Graphie	ـه	ـهـ	هـ	ه
Lettres liées	ه ههه			

Figure 5 : the Arabic letter “hâ”, the words “hâ hâhâhâ”

3. USING UNICODE FOR SW ENCODING

SW is currently encoded in SWML (Da Rocha & coll, 2001). Unicode encoding can take advantage of the previously presented properties of Unicode : symbols encoding, and symbols positioning can be studied as separate problems, with the reconstruction left to the Unicode engine, which will require a grammar. In each approach, the following criteria must be considered : integration into the operating systems, minimization of the storage space required, minimization of the mathematical cost of the Unicode engine algorithm in CPU time, respect of the Unicode standard.

ENCODING THE SYMBOLS

The first problem is matching each symbol into a unique number. From that code, as explained before, various mappings will be available to encode the number into bytes. Only two solutions are possible, depending on the importance of the aforementioned criteria: a) minimization of the storage space : “sequential” approach, where symbols are not sorted into groups with special meanings, but simply follow a sequence without any given order b) minimization's of the CPU time : “bitwise” approach, where symbols are grouped. Each group corresponds to a given parameter of the symbol such as rotation, filling, etc. Each group corresponds to a bit field.

SEQUENTIAL APPROACH

Obviously, the easiest way to match a code to each of the 25 973 symbols is to proceed in sequence. This approach presents however a major problem : while SSS evolves on a yearly basis, inserting new symbols could logically only occur at the end of the block. Since symbols are organized following a given structure (categories, groups, elements, variations, fillings, rotation), this problem would be much more important than for other languages, especially for the software treatment of the given space: determining the parameter of a given code would require a case-by-case analysis for symbols outside the given space. And even in the given space, without the addition of any symbols, determining parameters would in the best case require modular arithmetic to perform range comparison and check whether a symbol belongs to a given group/category/etc.

BITWISE APPROACH

In order to minimize the CPU time, a bitwise approach could allow to easily find a symbol though a simple bit masking. However, this approach would increase the space used to encode the symbols. Let us start with an example tree of depth n, where for each level, each node can have the same number of leaf. In SW case, each level represents a parameter (c: category, g: group, e: element, v: variation, f: fillings, r: rotation). Following this simplification, each parameters can be represented as a set, function of the level i, callen Ei. Encoding requires b bits, rounded to the next integer:

$$b_b=\sum_{i=0}^{n-1}(\ln_2(card(Ei)))bits$$

For example, the 6th level representing the rotation parameter corresponds to set E5 since we are starting at E0. Because there are 16 rotations, this set has 16 parts, and thus requires 4 bits. This bitwise approach thus has a mean costs of n-1 bits compared to the sequential approach. However, each of the 6 parameters c,g,e,v,f,r does not take an uniform amount of space : two different symbols can have different numbers of variations for example. Therefore, when encoding the variations into a fixed-length bit field, we must consider the worst case. Some space is wasted: it can be understood as identically sized boxes, which are as big as one of the box is filled, but globally are as empty as this case is rare.

COMPARING THE USE OF UNICODE SPACE

The cost of the sequential approach is known and fixed, and could easily fit in the Plane 2. The cost of the bitwise approach can be calculated with the previous formula. An analysis of SSS-2004 to calculate the card Ei for each of the 6 parameters reveals 23 bits are required. This means the bitwise encoding will require more than one Unicode per symbol since 2^{23} is 8 times greater than the total space offered by Unicode. The only possible solution is to use a sequence of 2 unicodes. It can come from a) an artificial extension of the Unicode space, which is contrary to the logic of the Unicode standard where each character must have its own code within the Unicode space or b) the use of modifier characters.

A MIXED BITWISE APPROACH WITH MODIFIERS

In the latter case, symbols are decomposed into a combination of parameters like for the “è” example : it will turn the chosen parameters into Unicode modifier characters. This would of course reduce the required Unicode space, but would let in exchange the storage space bear the equivalent cost of this simplification. At least two unicodes will be required anyway: if one parameter is made into a modifier, say variation for example, one Unicode is required for the $2^{23-\ln2(\text{card Ev})}=2^{17}$ codes corresponding to the remaining 5 parameters, and another Unicode is required for this modifier. Therefore, removing n parameters approximatively results in a storage space requirements of 1+n unicodes. The approximation is due to the possible cases where “small” parameters could fit within a single code as cumulative modifiers, like ”rotated left with front face exposed”. The main interest of a mixed bitwise approach with modifiers is using a single unicode for the main part, to follow the Unicode logic of one code per character. For example, variation and rotation requiring respectively 6 and 4 bits could be made as a single modifier requiring 10 bits. The main part would then require a 13 bits unicode. Deciding which parameters will become modifiers will require a linguistical analysis of SW. Then, deciding whether modifiers will be encoded following a sequential or a bitwise approach will require another comparison, on speed, space and side benefits criterias.

COMPARING THE SPEED

A good example is calculating the speed to extract a parameter of the code of a given symbol such as the variation : text search operation, given the inter and intra personal variabilities, will frequently have to extract many parameters – and that for each symbol. Deciding to extract the “variation” parameter follows a simplification hypothesis, which will minimize the advantage of the best method, because there is always the same number of possibilities for the following parameters (rotation and filling). Extracting the variation of a given code when a sequential encoding is used could be done as :

```
int extract_variation_s(int code) {return ((c/nf*nr)%nv);}
```

Here *c* is the code, *nf* the maximal amount of fillings, *nr* the maximal amount of rotations, and *nv* the maximal amount of variations : *nf*=6, *nr*=16, *nv*=5. The most costly operations are 2 modular divisions, in 16 bits since we have less than 65 536 symbols. In the case of a bitwise encoding, the same function would be:

```
int extract_variation_b(int c){return ((c>>(br+bf))&7);}
```

where *br* is the amount of bits required to encode the rotation, *bf* the amount of bits required to encode the filling, and 7 is the decimal value of 111 binary, which is used to mask the 3 bits of variation. Here the most costly operations are a bit shifting on 32 bits and a bitwise "and" on 32 bits. A practical experimentation of an AMD Athlon XP 2400, 50 million operations take 1744 ms in the sequential approach, versus 292 ms in the bitwise approach. The bitwise approach with modifiers would represent an intermediate case where extraction of the parameters which are modifiers could require the use of modular operations if the modifiers are encoded in a sequential approach, while the other operations will be as fast as the full bitwise approach. These approaches should now be compared to SWML. The implementations may vary, but can be simplified to the minimal operation which will always be present when the variation will have to be extracted from a SWML-formatted symbol. This minimal operation is matching the pattern where the variation parameter is stored. The fastest possible way to perform that operation in C is with regular expression. Supposing the regexp is already compiled, to give a speed advantage to this approach:

```
regcomp(&preg,"<symbol[^>]*>[0-9]+-[0-9]+-[0-9]+-([0-9]+)-[0-9]+-
 [0-9]+</symbol>",REG_EXTENDED); regexec (&preg,SWML,2,tab, 0);
```

This instruction is evaluated like the previous approaches on a AMD Athlon XP 2400. However, due to its low speed, it is only realized 50 000 times – it then takes 1788 ms. The Unicode sequential and bitwise approaches have been put in their worst possible configuration, and the SWML minimal step in its best possible configuration.

The Unicode approaches still respectively perform 1025 times faster for the sequential approach, and 6123 times faster for the bitwise approach.

COMPARING THE STORAGE REQUIREMENTS

For the sequential approach, one Unicode will be necessary for each symbol. For the bitwise approach, two unicodes will at least be necessary for each symbol – regardless whether modifiers are used or not. SWML requires 18 characters per symbol. In conclusion, the proposed methods will use from 6 to 18 times less space.

COMPARING ADDITIONAL BENEFITS

From a video recognition perspective, a bitwise approach would also offer the additional advantage of fuzzy completion : in the case where the specific symbol is not fully recognized, setting the bits to identify which parameters were recognized (ex: rotation, element, etc.) would be a first step – other parameters could be prompted to the user, or guessed depending on the context (signs previously used, etc). From a linguistic perspective, a bitwise approach would also ease lexicographic treatment of sign languages : for a new unknown symbol, the recognized parameters would be filled in the fuzzy completion, while the missing parameters (ex: a new element) could be temporarily assigned a code in one of the private use areas, until a linguist can review it. From a standardization perspective, leaving some empty space to add future SSS symbols would cost no more than the space being wasted by a bitwise approach, following the assumption that "empty" groups and categories are the most likely to be completed in the future.

ENCODING THE SYMBOLS POSITION

SWML currently does not save any order in which the symbols are entered to create a sign, the symbols are simply positioned in a 128x128 area. Yet saving the order of symbols entry could be used in lexical analysis of SW. Unicode only features composition methods, ie grammatical ways to create glyphs from characters which are composed. However, Everson (2002) estimated that 8% of the remaining writing formalisms not yet supported by Unicode would require innovative rendering methods – such as 2D positioning for Mayan and Egyptian hieroglyphs. Therefore, we consider preserving the order of symbols, and offering and extensible 2D positioning.

A POSITION AND NO RELATION

The positioning problem can be subdivided into 2 problems : positioning the symbols on a 2D signbox, and describing the relation between the symbols. SWML currently does not describe the relation between the symbols, while their relation can have various meanings such as an ordered sequence of movements, temporal co-occurrence, contact between body segments, etc. This relation is simply described by adding additional symbols, which are also positioned on the canvas. This simplifies the problem, removing the “relation” feature, but also removes information which could be used later on. For example, contact between symbols is not defined. Should it be defined as a relation, this property (contact or the lack of) between symbols could be preserved even during magnification or minimization of the sign. Likewise, manipulation of the symbols linked in a spatial sequence could take advantage of that property to automatically reposition the other symbols when the symbol initiating the sequence has been moved. Such relations could also be used to simplify the Unicode engine grammar.

POLAR OR CARTESIAN COORDINATES

Following SWML approach and using a dedicated code to position a symbol in the 128x128 signbox would only require 16 384 codes, from a partially used Unicode plane or a personal use area in the worst case. Reserved planes, offering 2^{16} coordinates, can even be used for a finer positioning in 4 096x4 096 with two unicodes. If no precision is necessary, a single reserved plane can be used, offering $2^{16/2}$ ie 256x256 scale, with a single Unicode. The simplest solution is to decide on a center, and give coordinates from that center. This is the solution currently used by SWML. It could be made to keep the sequence of entered symbols. A variation of that method is using a dichotomies positioning, which can save space depending on the precision needed. Yet since at least one Unicode will be used with any method, it has no interest and artificially complicates this solution.

RELATIVE COORDINATES

This solution removes any signbox size limit, while also keeping the starting symbol and the order of the sequence as entered by the user. However, the algorithm is complex, since it needs a step-by-step reconstruction taking into account the preceding step to construct the sequence of symbols.

COORDINATES GIVEN BY A FUNCTION

A parameter of the coordinates could be given to a function, encoded along, which would return the other position. The algorithm would be as complex as the function required to position each symbol, which could be following the sequence under which they where entered. This encoding would be best used with image recognition, to track body trajectory movements. However this would be the most complex solution, since it would at least require a fitting function. A simplified version of this approach could be used for relation operators, which would then be considered as functions.

COMPARISON

The speed costs are too complex to be calculated. But obviously, every proposed solution could be used to position symbols on the signbox, to preserve the order of the symbols, etc. In any case, the minimal cost will be 1 Unicode. With so many similarities and very little advantages, it seems evident that the simplest method should be chosen depending on the needs. The polar coordinates were initially favored, before inter and intra-personal variations had been identified. Its interest now seems very limited. The relative positioning requires a step-by-step reconstruction, which brings unneeded complexity. No approach will provide significant advantages in the positioning method, except in very specific cases. Therefore, the Cartesian coordinates, already used by SWML must be recommended. The function based positioning method should be limited to a) image and video interpretation, to trace trajectories and b) relation operators, should they be implemented.

4. PROPOSED UNICODE ENCODING

SYMBOL ENCODING

We propose to support both the sequential encoding and the mixed bitwise encoding. The pure bitwise encoding is not proposed because is does not follow the logic of Unicode standard, and therefore may not be accepted by the Unicode consortium. The sequential encoding could be immediately used to offer backward compatibility with existing SWML systems while offering a 1000 fold speed increase for parameter extractions and a 18 fold space saving. The mixed bitwise approach will only be evaluable when turning parameters into modifiers will be agreed. Following the example where the variation and the rotation are turned into modifiers, it will require two unicodes, but fit within one plane since $2^{13}+2^{10}<2^{16}$

This approach will also provide a 6000 fold speed increase for the remaining parameter extractions, and offer fuzzy editing capabilities for video recognition software. Even if the current Unicode policy is against giving codes to pre combined characters, such advantages could help the request.

INTEGRATIVE POSITIONNING APPROACH

A simple, non optimised, grammar, is proposed, with 3 elements: the symbol SYM, the operators OP, the parameters PAR, taken from a reserved plane to indicate the position. Since a sign is a set of positioned symbols, it is terminated by the TER special operator:

```
sign ->partialsign TER
 partialsign -> partialsign element | element
element -> SYM | OP
 OP -> PAR| PAR PAR | CONTACT | SEQUENCE
SYM -> (existing symbols)
```

This basic grammar could be further optimised. Yet it provides a very simple way to position 2D Unicode symbols at some coordinates P by default, without any operator, from 256x256 to 4 096x4 096. It also adds two sample operators previously suggested :CONTACT, to indicate whether two symbols are touching, and SEQUENCE, to indicate a sequence of movements. They could be used as symbols or as operators. For example, in a mixed bitwise approach with filling and rotation as modifiers, a sample “deaf” symbol coud be:

```
HEAD12 10 20 FINGER FILL2ROT3 10 25 CONTACTSYMBOL 10 22 T
 HEAD12 10 20 FINGER FILL2ROT3 10 25 CONTACT T
```

The first approach requires 11 codes, the second 9 codes. with a 4 096x4 096 signbox – this could be further optimised using a 256x256 signbox with a single PAR. Each approach would then take respectively 9 and 7 codes. The same sign in SWML requires 360 codes, with only a 128x128 signbox- between 40 and 50 times more.Additional operators could be added with the help of linguists, for SW or other languages needing specific spatial management - such as Mayan hieroglyphs.

5. CONCLUSION

Unicode will bring serious speed and size improvements. Moreover, the integrative positionning approach could be appliable to 8% of the languages requiring it. Giving a code to each symbol is not a complicated task for either approach – it can be fully automated, and use the private areas for quick prototyping until a dedicated area has been granted. But officially giving a code require describing the character (symbol) and its properties, in details, which will be a long and complex task. Transforming symbols into operators, thus expressing relations, will also be a challenge for linguists. But then Unicode will bring a real grammar to SW, and offer interesting relational information which will be usable in the user interface.

6. REFERENCES

Sutton V., Gleaves R. (1995), SignWriter - The world's first sign language processor. La Jolla, CA: Ed. Center for Sutton Movement Writing,

Sutton V. (2004), International Movement Writing Alphabet. La Jolla, CA: Ed. Center for Sutton Movement Writing

Da Rocha Costa A., Dimuro G. (2001) A SignWriting-Based Approach to Sign Language Processing. Gesture Workshap 2001, London

Fanton M. (1998) Finite State Automata and Arabic Writing. Proceedings of the Workshop on Computational Approaches to Semitic Languages (COLING-ACL'98), Montréal, PQ

Da Rocha A., Dimuro G., De Freitas J. (2004) A sign matching technique to support searches in sign language texts. Actes du Workshop RPSL, LREC 2004 (pp 32—34) Lisboa, Portugal

Everson M. (2002) Leaks in the Unicode Pipeline. 21st International Unicode conference, Dublin, Ireland