An extensible approach to high-quality multilingual typesetting

John Plaice, Yannis Haralambous, Chris Rowley

An extensible approach to high-quality multilingual typesetting

John Plaice
School of Computer Science and Engineering
The University of New South Wales
UNSW SYDNEY NSW 2052, Australia
plaice@cse.unsw.edu.au

Yannis Haralambous
Département Informatique
École Nationale Supérieure des Télécommunications de Bretagne
BP832, F-29285 Brest Cedex, France
Yannis.Haralambous@enst-bretagne.fr

Chris Rowley
Faculty of Mathematics and Computing
The Open University, UK
Milton Keynes MK7 6AA, United Kingdom
C.A.Rowley@open.ac.uk

Abstract— **We propose to create and study a new model for the micro-typography part of automated multilingual typesetting. This new model will support quality typesetting for a number of modern and ancient scripts.**

The major innovations in the proposal are: the process is refined into four phases, each dependent on a multidimensional tree-structured context summarizing the current linguistic and cultural environment. The four phases are: preparing the input stream for typesetting; segmenting the stream into clusters (words); typesetting these clusters; and then recombining the clusters into a typeset text stream. The context is pervasive throughout the process; the algorithms used in each phase are context-dependent, as are the meanings of fundamental entities such as language, script, font and character.

I. Introduction

We present in this paper the outline of a new approach to the automation of some aspects of typesetting. Traditionally, typesetting was defined thus “The production of printed matter by computer, usually by producing a master copy for offset reproduction” [3] but we include a far greater range of output media in our view. One of the key innovations outlined here is to consider each of language, script, font, and character as multidimensional entities, as opposed to the current view, reiterated at length in the Unicode standard [17], that they are discrete and unchanging. As a result, typesetting will be undertaken in a *multidimensional context* — formally, a point in a multidimensional space — that summarizes the current linguistic and cultural environment. This point of view, consistent with the intensional programming approach explained in Sections II-B and IV below, will efficiently support much greater variation in the behavior of the typesetting engine. Moreover, this approach will allow the typesetter to be integrated with more general text processing tools, such as spell-checkers, style-checkers, content-checkers, transliterators, or translators (see also Mittelbach and Rowley [8]).

II. Background

A. Computer Typesetting, TEX and Ω

The first steps in computer typesetting took place in the 1950s, but it was not until 1982, when Donald Knuth introduced TEX [5], that it became possible to use computer software for high-quality typesetting of, at least, English and mathematics, as in his *The Art of Computer Programming* series.

In TEX’s very speedy character-level typesetter, *characters* in the input file are transformed almost directly into *glyphs* in the current font, and these glyphs are positioned side-by-side ‘on the *baseline*’; the only small refinement of this transformation process is that a font-specific finite-state automaton can be used to change the glyphs used (typically by using *ligatures*) and their horizontal placement (by *kerning*). The ‘words’ thus typeset are then separated by a font-specific amount of stretchable inter-word space (*glue*) to form a stream (a *horizontal list*) that is typically passed to TEX’s paragraph maker. In the TEX model, each glyph is an object that has only width, height, and depth; a similar box-and-glue model is also used for higher-level layout.

The Ω system [10], developed by the first two authors, is a series of extensions to the TEX system that facilitate multilingual typesetting. The Ω system has been used for typesetting languages in the following scripts: Latin (including Gothic and Gaelic), Greek, Cyrillic, Armenian, Georgian, Arabic, Hebrew, Syriac, Tifinagh, Japanese, Thai, Khmer, Devanagari (for Hindi, Sanskrit), Malayalam and Tamil.

The major difference between the TeX model and the current Ω model is that, before glyph selection, the character stream to be typeset is segmented and processed by a series of filters, each reading from standard input and writing to standard output. Once all of the filters are applied, the stream is passed to the standard TeX character-level typesetter. Filters have been written for character set conversion, transliteration, morphological analysis, spell-checking, and contextual analysis. In addition, a number of filters have been written for what we call *1.5-dimensional layout*, used for scripts that are written horizontally, but for which there is substantial vertical displacement for proper placement of glyphs: examples where this is necessary are typesetting Arabic, Indic and South-East Asian scripts, typesetting mathematics, and stacking International Phonetic Alphabet (IPA) diacritics.

There are two current limitations to the use of Ω. First, because Ω is so versatile, it is difficult to define good high-level interfaces that can be used without in-depth understanding of the low-level system. Second, the output from applying several filters is simply too low-level; the relationship between the original input and the typeset output is simply too distant. If one is only interested in the final visual layout, this is not so much of a problem. However, we are increasingly interested in the ability to be able to *search* through documents to find information; we therefore need to be able to retain the link between the original text and the typeset output, so that this information can be placed in all generated documents.

B. Intensional Programming

Intensional programming [12] is an approach to computing that supposes that there is a multidimensional context, and that all programs are capable of adapting themselves to this context. The context is pervasive, and can simultaneously affect the behavior of a program at the lowest, highest and middle layers.

When an intensional program is running, there is a *current context*. This context is initialized upon launching the program from the values of environment variables, from explicit parameters, and possibly from active context servers. The current context can be modified during execution, either explicitly through the program's actions, or implicitly, through changes at an active context server.

A context is a point in a multidimensional space, i.e., given a dimension, the context will return a value for that dimension. The simplest contexts are dictionaries (lists of attribute-value pairs). A natural generalization is what will be used in this paper: the values themselves can be versions, resulting in a tree-structured context. The set of contexts is furnished with a partial order $\sqsubseteq$ called a *refinement relation*.

For example, to describe Australian English, we could use the context:

```
<script:<Latin>+
 lang:<English;dialect:<Australian>>>
```

where `script` and `lang` are called *dimensions*, and `lang:dialect` a *compound dimension*. See Section IV for more details.

During execution, the current context can be queried, dimension by dimension, and the program can adapt its behavior accordingly. In addition, if the programming language supports it, then contextual conditional expressions and blocks can be defined, in which the *most relevant* case, with respect to the current context and according to the partial order, is chosen among the different possibilities.

In addition, any entity can be defined in multiple *versions*. A version is of the same structure as a context. Whenever an identifier designating an entity appears in an expression or a statement, then the most relevant version of that entity, with respect to the current context, is chosen. This is called the *variant substructure principle*. The general approach is called *intensional versioning* [14].

The ISE programming language [13] was the first language combining both intensional programming and versioning. It is based on the procedural scripting language Perl, and it has greatly facilitated the creation of multidimensional Web pages. Similar experimental work has been undertaken under the supervision of the first author with C, C++, Java, and Eiffel. And, when combined with a context server (see Paul Swoboda's PhD thesis [15]), it becomes possible for several documents or programs to be immersed in the same context.

III. Significance

The significance of high-quality highly automated multilingual typesetting cannot be overestimated. We know from Marshall McLuhan's work [6] just how important was the introduction of metal type to European society. Typesetting was, in some sense, the first industrial process, upon which all others were based. It was also the process that enabled the others, since it allowed knowledge to spread rapidly across Europe. It also facilitated the rise of national vernaculars and the subsequent creation of nation-states.

Today, with the development of the Internet and even more so the Web, something different is occurring. We now have *access* to online documents in hundreds of languages, using a multitude of scripts. At the same time, grandiose endeavors such as the Million Book Project [7] (scanning of about 4% of the books ever written) are being undertaken. Bit by bit, the world's collected writings are being made available, to everyone. And, with miniaturization of storage, these writings will be available not just online, but on our personal portable devices.

However, making these works available is not sufficient. They still need to be printed, whether it be on a screen, in a bound paper volume, or on some future substrate. But we are not yet at a point where we can automatically reproduce the quality of books typeset in the nineteenth century, particularly for the non-Latin scripts. In fact, the problem is harder, because we now need real-time printing of documents from the Web.

In India, this problem is of utmost importance. India has two national languages (Hindi and English), one recognized mother language (Sanskrit), and 14 official languages, each with its own script. In addition, there are approximately 200 minority languages. Clearly, a better, more general approach to multilingual typesetting is needed, one that promises ease of use with high quality.

More prosaic areas, such as the formatting of legal documents and business forms, also have a need for high quality typographic design in a range of languages and here high levels of automation are often paramount due to the high volume of material and the essential need for clarity and accuracy. Less conventionally, safety critical systems need very high quality typographic designs as has been shown by the screen fonts and layout requirements for the new British air traffic control system.

IV. Structuring the Context

As was stated in Section II-B, we use the same notation to designate versions of entities and to designate contexts. This section has three subsections. First, we define versions and the refinement relation. Then, we define *version binders*, which hold versioned entities. Finally, we define *version operators*, which are used to change from version to version. In the following section, we will show how all of these are to be used.

A. Versions and Refinement

Let $\{(\mathbb{S}_i, \sqsubseteq_i)\}_i$ be a collection of sets of ground values, each with its own partial order. Let $\mathbb{S} = \cup_i \mathbb{S}_i$. Then the set of versions $\mathbb{V}$ ($\ni V$) over $\mathbb{S}$ is given by the following syntax:

$$V ::= \emptyset \mid A \mid \Omega \mid \langle B; L\rangle \qquad (1)$$
$$B ::= \epsilon \mid \alpha \mid \omega \mid v \qquad (2)$$
$$L ::= \emptyset_L \mid d{:}V + L \qquad (3)$$

where $d, v \in \mathbb{S}$.

There are three special versions:

- $\emptyset$ is the *empty version* (also called *vanilla*);
- A is the *minimally defined version*, just more defined than the empty one;
- Ω is the *maximally defined version*, more defined than all other versions.

The normal case is that there is a *base value* B, along with a *version list* (L for short), which is a set of *dimension-version* pairs. We write δL for the set of dimensions of L.

A sequence of dimensions is called a *compound dimension*. It can be used as a path into a version. Formally:

$$D = \cdot \mid d{:}D \qquad (4)$$

If V is a version, $V(D)$ is the subtree of V whose root is reached by following the path D from the root of V:

$$V(\cdot) = V \qquad (5)$$
$$\langle B; d{:}V' + L\rangle\,(d{:}D) = V'(D) \qquad (6)$$

As with versions, there are three special base values:

- ϵ is the *empty base value*;
- α is the *minimally defined base value*, just more defined than the empty base value;
- ω is the *maximally defined base value*, more defined than all others.

The normal case is that a base value is simply a scalar.

To the set $\mathbb{V}$, we add an *equivalence* relation $\equiv$, and a *refinement* relation $\sqsubseteq$. We begin with the equivalence relation:

$$\emptyset \equiv \langle\epsilon; \emptyset_L\rangle \qquad (7)$$
$$A \equiv \langle\alpha; \emptyset_L\rangle \qquad (8)$$
$$\Omega \equiv \left\langle \omega; \sum_{d\in\mathbb{S}} d{:}\Omega \right\rangle \qquad (9)$$
$$d{:}\emptyset \equiv \emptyset_L \qquad (10)$$
$$d{:}V + d{:}V' \equiv d{:}(V+V') \qquad (11)$$
$$\langle B; L\rangle + \langle B; L'\rangle \equiv \langle B; L+L'\rangle \qquad (12)$$
$$L + \emptyset_L \equiv L \qquad (13)$$
$$L + L \equiv L \qquad (14)$$
$$L + L' \equiv L' + L \qquad (15)$$
$$L + (L' + L'') \equiv (L+L') + L'' \qquad (16)$$

Therefore $\emptyset$ and A are notational conveniences, while Ω cannot be reduced. The $+$ operator is idempotent, commutative, and associative.

We now give the partial order over the base values:

$$\epsilon \sqsubseteq B \qquad (17)$$

$$B \sqsubseteq B \qquad (18)$$

$$B \sqsubseteq \omega \qquad (19)$$

$$\frac{B \neq \epsilon}{\alpha \sqsubseteq B} \qquad (20)$$

$$\frac{v_0, v_1 \in \mathbb{S}_i \quad v_0 \sqsubseteq_i v_1}{v_0 \sqsubseteq v_1} \qquad (21)$$

The last rule states that if v_0 and v_1 belong to the same set $\mathbb{S}_i$ and are comparable according to the partial order $\sqsubseteq_i$, then that order is subsumed for refinement purposes.

Now we can define the partial order over entire versions:

$$\emptyset \sqsubseteq V \qquad (22)$$

$$V \sqsubseteq \Omega \qquad (23)$$

$$\frac{V \neq \emptyset}{A \sqsubseteq V} \qquad (24)$$

$$\frac{V_0 \equiv V_1}{V_0 \sqsubseteq V_1} \qquad (25)$$

$$\frac{V_0 \sqsubseteq V_1}{d{:}V_0 \sqsubseteq d{:}V_1} \qquad (26)$$

$$\emptyset_L \sqsubseteq L \qquad (27)$$

$$\frac{L_0 \sqsubseteq L_1 \quad L'_0 \sqsubseteq L'_1}{L_0 + L'_0 \sqsubseteq L_1 + L'_1} \qquad (28)$$

$$\frac{B_0 \sqsubseteq B_1 \quad L_0 \sqsubseteq L_1}{\langle B_0; L_0\rangle \sqsubseteq \langle B_1; L_1\rangle} \qquad (29)$$

Rule 28 ensures that the $+$ operator defines the least upper bound of two versions.

B. Version Domains and Version Binders

When doing intensional programming, we work with *sets* of versions, called *version domains*, written $\mathcal{V}$. There is one operation on version domains, namely the *best-fit*. Given a version domain $\mathcal{V}$ of existing versions and a requested version V_{req}, the best-fit version is defined by:

$$best_V(\mathcal{V}, V_{\text{req}}) = \max\{V \in \mathcal{V} \mid V \sqsubseteq V_{\text{req}}\} \qquad (30)$$

If the maximum does not exist, there is no best-fit version.

Typically, we will be versioning *something*, an object of some type. This is done using *version binders*, simply $(V, object)$ pairs. Version binder domains $\mathcal{V}_b$ then become functions mapping versions to objects. The *best-fit object* in a version binder domain is given by:

$$best_O(\mathcal{V}_b, V_{\text{req}}) = \mathcal{V}_b(best_V(\text{dom}\ \mathcal{V}_b, V_{\text{req}})) \qquad (31)$$

C. Version Operators

Version operators allow one to selectively *modify* versions. Their syntax is similar to that of versions.

$$\begin{array}{rcll} V_{\text{op}} & ::= & V \mid [P_{\text{op}}; B_{\text{op}}; L_{\text{op}}] & (32) \\ P_{\text{op}} & ::= & -- \mid \epsilon & (33) \\ B_{\text{op}} & ::= & - \mid B & (34) \\ L_{\text{op}} & ::= & \emptyset_{L_{\text{op}}} \mid d\!:\!V_{\text{op}} + L_{\text{op}} & (35) \end{array}$$

A version operator is applied to a version to transform it into another version. (It can also be used to transform a version operator into another; see below.) The $-$ operator removes the current base value, while the $--$ operator in P_{op} is used to clear all dimensions not explicitly listed at that level.

Now we give the semantics for $V\ V_{\text{op}}$, the application of version operator V_{op} to version V:

$$\begin{array}{rcll} V_0\ V_1 & = & V_1 & (36) \\ \Omega\ V_{\text{op}} & = & \text{error} & (37) \\ \langle B; L\rangle\ [--; B_{\text{op}}; L_{\text{op}}] & = & & (38) \\ & & \left\langle B; L\backslash(\delta L - \delta L_{\text{op}})\right\rangle\ [\epsilon; B_{\text{op}}; L_{\text{op}}] & \\ \langle B; L\rangle\ [\epsilon; B_{\text{op}}; L_{\text{op}}] & = & \left\langle (B\ B_{\text{op}}); (L\ L_{\text{op}})\right\rangle & (39) \end{array}$$

The general case consists of replacing the base value and replacing the version list. First, the base value:

$$\begin{array}{rcll} B\ - & = & \epsilon & (40) \\ B_0\ B_1 & = & B_1 & (41) \end{array}$$

Now, the version list:

$$\begin{array}{rcll} L\ \emptyset_{L_{\text{op}}} & = & L & (42) \\ (d\!:\!V + L)\ (d\!:\!V_{\text{op}} + L_{\text{op}}) & = & & (43) \\ & & d\!:\!(V\ V_{\text{op}}) + (L\ L_{\text{op}}) & \\ L\ (d\!:\!V_{\text{op}} + L_{\text{op}}) & = & & (44) \\ & & d\!:\!(\emptyset\ V_{\text{op}}) + (L\ L_{\text{op}}), \quad d \notin \delta L & \end{array}$$

Version operators can also be applied to version operators. There are two cases:

$$\begin{array}{rcll} [P_{\text{op}}; B_{\text{op}0}; L_{\text{op}0}]\ [\epsilon; B_{\text{op}1}; L_{\text{op}1}] & = & & (45) \\ & & \left[P_{\text{op}}; (B_{\text{op}0}\ B_{\text{op}1}); (L_{\text{op}0}\ L_{\text{op}1})\right] & \end{array}$$

$$\begin{array}{rcll} [P_{\text{op}}; B_{\text{op}0}; L_{\text{op}0}]\ [--; B_{\text{op}1}; L_{\text{op}1}] & = & & (46) \end{array}$$
$$\left[--; (B_{\text{op}0}\ B_{\text{op}1}); \left((L_{\text{op}0}\backslash(\delta L_{\text{op}0} - \delta L_{\text{op}1}))\ L_{\text{op}1}\right)\right]$$

Now that we have given the formal syntax and semantics of versions, version binders, and version operations, we can move on to typesetting.

V. The Four Phases of Typesetting

At its most basic level, a micro-typesetter is a function that transforms a stream of characters to a stream of positioned glyphs. In our new model, micro-typesetting is split into four separate *phases*: *preparation*, *segmentation*, *micro-typesetting* and *recombination*. Since each of these phases is dependent on the context, we can write the process, using C++ syntax, as:

```
stream<Glyph>
micro_typeset(stream<Char> input,
              Version context) {
  stream<Char> prepared =
    input.apply(otp_list.best(context));
  stream<Cluster> segmented =
    segmenter.best(context)(prepared);
  stream<TypesetCluster> typeset =
    clusterset.best(context)(segmented);
  stream<Glyph> recombined =
    recombine.best(context)(typeset);
  return recombined;
}
```

where *function*`.best(`*context*`)` means that the most relevant version of *function*, with respect to *context*, is selected. Below, we examine each of the phases in detail.

A. Preparation

```
stream<Char> prepared =
  input.apply(otp_list.best(context));
```

The preparation phase in this new approach is similar to the current situation in the Ω system. At all times, there is an active Ω Translation Processing List (ΩTP-list). This list consists of individual Ω Translation Processes (ΩTP's), each of which is a filter reading from standard input to standard output. What is new is that the whole process will become context-dependent. First, the most relevant ΩTP-list, with respect to the context and using the refinement relation over contexts, will be the one that is active. Second, once chosen, it can test the current context and adapt its behavior, by selectively turning on or off, or even replacing, individual ΩTP's.

The preparation phase will work entirely on *characters*, i.e. at the *information exchange* level but it will allow additional typographic information to be added to the character stream, so that the following phases can use the extra have information to produce better typography.

B. *Segmentation*

```
stream<Cluster> segmented =
  segmenter.best(context)(prepared);
```

The segmentation phase splits the stream of characters into clusters of characters; typically, segmentation is used for word detection. In English, word detection is a trivial problem, and segmentation just means recognizing 'white space' such as the blank character, Unicode `U+020`. By contrast, in Thai, where there is normally no word-delimiter in the character stream (blanks are traditionally only used as sentence-delimiters), it is impossible to do any form of automatic processing unless a sophisticated morphological analyzer is being used to calculate word and syllable boundaries. In many Germanic and Slavic languages, it is also necessary to find the division of compound words into their building blocks. These processes are closely related to finding word-division points so this should be incorporated into this part of the process (a very different approach to that of TeX). The choice of segmenter is thus clearly seen to be context-dependent.

C. *Cluster typesetting*

```
stream<TypesetCluster> typeset =
  clusterset.best(context)(segmented);
```

During the typesetting phase, a *cluster engine* processes a character cluster, taking into account the current context including language and font information, and produces the typeset output — a sequence of positioned glyphs. In many cases, such as when hyphenation or some other form of cluster-breaking is allowed, there will be multiple possible typeset results, and all of these possibilities must be output. When dealing with complex scripts or with fonts allowing great versatility (as with Adobe Type 3 fonts), numerous different cluster engines will be needed: these will be selected and their behaviour will be fine-tuned according to the context.

D. *Recombination*

```
stream<Glyph> recombined =
  recombine.best(context)(typeset);
```

The final phase, before calling a higher-level formatting process such as a paragrapher, is the recombination phase. Here, the typeset clusters are placed next to each other. For simple text, such as the English in this proposal, this simply means placing a fixed stretchable space between typeset words. In situations such as Thai and some styles of Arabic typesetting, kerning would take place between words. Once again, the recombiner's behavior is context-dependent.

VI. Examples

Given the sophistication of the four-phase process, and that the choice of segmenter, cluster engine and recombiner are all context-dependent, and that the actions of each of these, once they are chosen, also depends on the context, this new model of typesetting engine is potentially *much* more powerful than anything previously proposed or implemented. We intend to test and validate it on, at least, the following scripts:

- *Latin, Greek and Cyrillic, IPA*: left-to-right, discrete glyphs, numerous diacritics, stacked vertically, above or below the base letters, widespread hyphenation;
- *Hebrew*: right-to-left, discrete glyphs, optional use of diacritics (vowels and breathing marks), which are stacked horizontally below base letter;
- *Arabic, Naskh style*: right-to-left, contiguous glyphs, contextually shaped, numerous ligatures, optional use of diacritics (vowels and breathing marks), placed in 1.5-dimensions, above and below;
- *Indic scripts*: left-to-right, 1.5-dimensional layout of clusters, numerous ligatures, applied selectively according to linguistic and stylistic criteria;
- *Chinese, Japanese*: vertical or left-to-right, often on fixed grid, with annotations to the right or above the main sequence of text, automatic word recognition — Chinese and Japanese words use one or more characters, but these are not visually apparent — needed for any form of analysis;
- *Egyptian hieroglyphics*: mixed left-to-right and right-to-left, 1.5-dimensional layout.

Once the basic typesetting is validated, then further experiments, viewing language as a multidimensional entity, will be undertaken. Already with Ω, we have typeset Spanish with both the Hebrew and Latin scripts; Berber with the Tifinagh, Arabic and Latin scripts; Arabic with Arabic, Hebrew, Syriac, Latin and even *Arabized Latin* (Latin script with a few additional glyphs reminiscent of the Arabic script). The Arabic script can be rendered in Naskh or Nastaliq or many other styles. Japanese can be typeset with or without *furigana*, little annotations above the *kanji* (the Chinese characters) to facilitate pronunciation.

The objective is to incorporate solutions to all such problems, currently solved in an *ad hoc* manner, into our framework; each time, the key is to correctly summarize the context. With this key, then the choice of segmenters, clusters engines and recombiners to build, and of how they are built, is clarified; nevertheless, these algorithms may remain complex, because of the inherent complexity of the problems they are solving.

VII. Conclusions

If the model that we propose to develop is successful, then we will be able to produce, with relative ease, high-quality documents in many different languages and scripts.

Furthermore, this new approach of contexts can be used to improve macro-typesetting as well as micro-typesetting. The third author, in his role as a leader of the LaTeX3 Project, has worked with closely related ideas in the context of Mittelbach's *templates* for higher-level formatting processes [2]. Here the particular instance of a template object that is used to format a document element will depend on a context that is derived from both the logical position of that element in the structured document and from the formatting of the physically surrounding objects in the formatted document. Collaboration between the current authors and other members of the LaTeX3 team will lead to many new interfaces that give access to the new functionality.

Other examples of the importance of such a structured context in document processing can be found in work by the third author with Frank Mittelbach [9].

Another example of dependence on this visual context occurs in the use of Adobe Type 3 fonts, which are designed so that glyphs can be generated differently upon each rendering (see [1] for a discussion of a number of effects). On another level, the Open-Type standard [11], jointly developed by Adobe and Microsoft, allows for many different kinds of parameters — beyond the basic three of width, height, and depth —, multiple baselines, and a much richer notion of ligature. Our new engine for micro-typography will provide new capabilities, adaptable to new kinds of parameters, and increased control. Thus we shall be able to provide a simple high-level interface that takes advantage of new developments in font technologies.

In addition, the full-scale introduction of context will even allow reconsideration of the very contexts of glyph and character. In the second author's article on the relationship between the two [4], it is clear that glyphs and characters are not absolutes, but, rather, fluid from one context to another.

At another level, the existing Ω and LaTeX systems have already influenced the specifications of XML [18] (how to deal with multiple character sets), SVG [16] (the text model) and XSL [19] (the model for printing in multiple-directions and the concept of formatting objects). The success of this project and of our further research in typesetting will lead directly to additional enhancements to XSL and SVG, by providing, for example, specifications for XSL formatting objects to support high-quality typography and a text model that better supports glyph specification.

Finally, this proposed model should be understood as the preparation for a much more ambitious project, that will deal not just with low-level typesetting but also general problems of document structuring and layout for demanding typographic designs in a highly automated environment. Detailed discussion along these lines has already been initiated between the Ω and LaTeX3 projects, which look forward to these wider horizons.

REFERENCES

[1] Jacques André. *Création de fontes en typographie numérique* [Creating fonts for digital typography]. Documents d'habilitation, IRISA+IFSIC, Rennes, 1993.

[2] David Carlisle, Frank Mittelbach and Chris Rowley. New interfaces for LaTeX class design, 1999. `http://www.latex-project.org/papers/tug99.pdf`

[3] Computer Typesetting. `http://www.xrefer.com/entry/441575`

[4] Yannis Haralambous. Unicode et typographie : un amour impossible [Unicode and typography: an impossible couple]. *Documents numériques* 1:1, 2002.

[5] Donald Knuth. *Computers and Typesetting*. 5 volumes, Addison-Wesley, 1986.

[6] Marshall McLuhan. *The Gutenberg Galaxy: The Making of Typographic Man*. University of Toronto Press, 1962.

[7] Million Book Project. `http://zeeb.library.cmu.edu/Libraries/LIT/Projects/1MBooks.html`

[8] Frank Mittelbach and Chris Rowley, 1996. Application-independent representation of text for document processing. `http://www.latex-project.org/papers/unicode5.pdf`

[9] Frank Mittelbach and Chris Rowley. Language information in structured documents, 1997. `http://www.latex-project.org/papers/language-tug97-paper-revised.pdf`

[10] Omega Typesetting and Document Processing System. `http://omega.cse.unsw.edu.au`

[11] OpenType. `http://www.opentype.org`

[12] John Plaice and Joey Paquet. Introduction to intensional programming. In *Intensional Programming I*, World-Scientific, Singapore, 1996.

[13] John Plaice, Paul Swoboda and Ammar Alammar. Building intensional communities using shared contexts. In *Distributed Communities on the Web, LNCS* 1830:55–64, Springer-Verlag, 2000.

[14] John Plaice and William W. Wadge. A new approach to version control. *IEEE-TSE* 19(3):268–276, 1993.

[15] Paul Swoboda. *A Formalization and Implementation of Distributed Intensional Programming*. PhD Thesis, The University of New South Wales, Sydney, Australia, 2003.

[16] Extensible Markup Language (XML). `http://www.w3c.org/Graphics/SVG`

[17] Unicode Home Page. `http://www.unicode.org`

[18] Extensible Markup Language (XML). `http://www.w3c.org/XML`

[19] The Extensible Stylesheet Language (XSL). `http://www.w3c.org/Style/XSL`