Digital Typography in the New Millennium: Flexible Documents by a Flexible Engine

Christos K.K. Loverdos[1] and Apostolos Syropoulos[2]

[1] Department of Informatics and Telecommunications, University of Athens
TYPA Buildings, Panepistimiopolis
GR-157 84 Athens, Greece
loverdos@di.uoa.gr
http://www.di.uoa.gr/~loverdos
[2] Greek TEX Friends Group
366, 28th October Str.
GR-671 00 Xanthi, Greece
apostolo@obelix.ee.duth.gr
http://obelix.ee.duth.gr/~apostolo

Abstract. The TEX family of electronic typesetters contains the primary typesetting tools for the preparation of demanding documents, and have been in use for many years. However, our era is characterized, among others, by Unicode, XML and the introduction of interactive documents. In addition, the Open Source movement, which is breaking new ground in the areas of project support and development, enables masses of programmers to work simultaneously. As a direct consequence, it is reasonable to demand the incorporation of certain facilities to a highly modular implementation of a TEX-like system. Facilities such as the ability to extend the engine using common scripting languages (e.g., Perl, Python, Ruby, etc.) will help in reaching a greater level of overall architectural modularity. Obviously, in order to achieve such a goal, it is mandatory to attract a greater programming audience and leverage the Open Source programming community. We argue that the successful TEX-successor should be built around a microkernel/exokernel architecture. Thus, services such as client-side scripting, font selection and use, output routines and the design and implementation of formats can be programmed as extension modules. In order to leverage the huge amount of existing code, and keep document source compatibility, the existing programming interface is demonstrated to be just another service/module.

1 Introduction

The first steps towards computer typesetting took place in the 1950s, but it was not until Donald E. Knuth introduced TEX in 1978 [16] that true quality was brought to software-based typesetting. The history of TEX is well-known and the interested reader is referred to [16] for more details.

Today, the original TEX is a closed project in the sense that its creator has decided to freeze its development. As a direct consequence no other programs

A. Syropoulos et al. (Eds.): TUG 2004, LNCS 3130, pp. 1–16, 2004.

are allowed to be called TEX. In addition, the freely available source code of the system was a major step on the road towards the formation of the Open Source movement, which, in turn, borrowed ideas and practices from the Unix world. Furthemore, the development of TEX and its companion system, METAFONT, had made obvious the need for properly documented programs. This, in turn, initiated Knuth's creation of the *literate programming* program development methodology. This methodology advances the idea that the program code and documentation should be intermixed and developed simultaneously.

The source code of TEX and METAFONT being freely available has had enormous consequences. Anyone can not only inspect the source code, but also experiment freely with it. Combined with TEX's (primitive, we should note, but quite effective for the time) ability to extend itself, this led to such success stories as LATEX and its enormous supporting codebase, in the form of packages. As a direct consequence of the fact that the source code is frozen, stability was brought forth. Note that this was exactly the intention Knuth had when developing his systems. A common referred-to core, unchanged in the passing of time and almost free of bugs, offered a "secure" environment to produce with and even experiment with.

However, in an everchanging world, especially in the fast-paced field of computer science, almost anything must eventually be surpassed. And it is the emerging needs of each era that dictate possible future directions. TEX has undoubtedly served its purpose well. Its Turing-completeness has been a most powerful asset/weapon in the battles for and of evolution. Yet, the desired abstraction level, needed to cope with increasing complexity, has not been reached. Unfortunately, with TEX being bound to a fixed core, *it cannot be reached*.

Furthermore, the now widely accepted user-unfriendliness of TEX as a language poses another obstacle to TEX's evolution. It has created the myth of those few, very special and quite extraordinary "creatures"[1] able to decrypt and produce code fragments such as the following[2]:

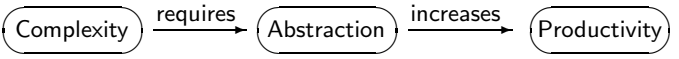
```
\def\s@vig{{\EO@m=\EO@n
  \divide\EO@n by20 \relax
  \ifnum\EO@n>0\s@vig\fi
  \EO@k=\EO@n\relax
  \multiply\EO@k by-20\relax
  \advance\EO@m by \EO@k\relax
  \global\advance\EO@l by \@ne
  \expandafter\xdef\csname EO@d\@roman{\EO@l}\endcsname{%
       \ifnum\EO@m=0\noexpand\noexpand\EOzero
       \else\expandafter\noexpand
       \expandafter\csname EO\@roman{\EO@m}\endcsname\fi}
  \expandafter\@rightappend
    \csname EO@d\@roman{\EO@l}\endcsname
          \t@\epi@lmecDigits}}
```

Of course, to be fair, programmers in several languages (C and Perl among others) are often accused of producing ununderstandable code and the well-known *obfuscated code* contests just prove it. On the other hand, with the ad-

[1] The second author may be regarded as one of Gandalf's famuli, while the first author is just a Hobbit, wishing to have been an Elf.

[2] Taken from the documentation of the epiolmec package by the second author.

vent of quite sophisticated assemblers, today one can even write well-structured assembly language, adhering even to "advanced" techniques/paradigms, such as object-oriented programming. Naturally, this should not lead to the conclusion that we should start writing in assembly (again)! In our opinion, software complexity should be tackled with an emphasis on abstraction that will eventually lead to increased productivity, as is shown in the following figure:



TEX's programming language is more or less an "assembly language" for electronic typesetting. It is true that higher level constructs can be made – macros and macro packages built on top of that. But the essence remains the same. Although it is true that TEX is essentially bug free and its macro expansion facility behaves the way it is specified (i.e., as defined in [9]), it still remains a fact that it takes a non-specialist quite some time to fully understand the macro expansion rules in spite of Knuth's initial intentions [12, page 6].

The fact that one should program in the language of his/her choice is just another reason for moving away from a low-level language. And it is true that we envision an environment where as many programmers as possible can – and the most important, wish to – contribute. In the era of the Open Source revolution, we would like to attract the Open Source community and not just a few dedicated low-level developers. Open Source should also mean, in our opinion, "open possibilities" to evolve the source. This is one of our major motivations for reengineering the most successful typesetting engine.

Richard Palais, the founding chairman of TUG, pointed out back in 1992 [12, page 7] that when developing TEX, Knuth

> *. . . had NSF grant support that not only provided him with the time and equipment he needed, but also supported a team of devoted and brilliant graduate students who did an enormous amount of work helping design and write the large quantity of ancillary software needed to make the TEX system work . . .*

and immediately after this, he poses the fundamental question:

> *Where will the resources come from for what will have to be at least an equally massive effort? And will the provider of those resources be willing, at the end of the project, to put the fruits of all his effort in the Public Domain?*

The answer seems obvious now. The way has been paved by the GNU/Linux/-BSD revolutionary development model, as has been explained crystal clearly in *The Cathedral and the Bazaar* [15].

This paper is an attempt to define a service-oriented architecture for a future typesetting engine, which will be capable of modular evolution. We take a layered approach of designing some core functionality and then define extensible services on top of the core. The engine is not restricted to a specific programming language either for its basic/bootstrapping implementation or, even more

important, for its future enhancement. At the same time, we are bound to provide a 100% TEX-compatible environment, as the only means of supporting the vast quantity of existing TEX-based documents. We intend to achieve such a goal by leveraging the proposed architecture's own flexibility. Specifically, a TEX compatibility mode is to be supported and it should give complete "trip-test" compliance. Later on, we shall see that this compatibility is divided into two parts: source code compatibility and internal core compatibility. Both are provided by pluggable modules.

Structure of the Paper. In the following sections we briefly review the most important and influential approaches to extending or reengineering TEX, including TEX's inherent abilities to evolve. Then we discuss a few desired characteristics for any next generation typesetting engine. We advance by proposing an architecture to support these emerging needs. Finally, we conclude by discussing further and future work.

2 A Better TEX?

2.1 TEX the Program

TEX supports a Turing-complete programming language. Simply, this means that if it lacks a feature, it can be programmed. It contains only a few concepts and belongs to the LISP family of languages. In particular, it is a list-based macro-language with late binding [5, Sec. 3.3]:

> *Its data constructs are simpler than in Common Lisp: 'token list' is the only first order type. Glue, boxes, numbers, etc., are engine concepts; instances of them are described by token lists. Its lexical analysis is simpler than CL: One cannot program it. One can only configure it. Its control constructs are simpler than in CL: Only macros, no functions. And the macros are only simple ones, one can't compute in them.*

Further analysis of TEX's notions and inner workings such as *category codes*, TEX's *mouth* and *stomach* is beyond the scope of this paper and the interested reader is referred to the classic [9] or the excellent [3].

TEX the program is written in the WEB system of literate programming. Thus, its source code is self-documented. The programs tangle and weave are used to extract the Pascal code and the documentation, respectively, from the WEB code. The documentation is of course specified in the TEX notation. Although the TEX source is structured in a monolithic style, its architecture provides for some kind of future evolution.

First, TEX can be "extended" by the construction of large collections of macros that are simply called *formats*. Each format can be transformed to a quickly loadable binary form, which can be thought of as a primitive form of the module concept.

Also, by the prescient inclusion of the `\special` primitive command, TEX provides the means to express things beyond its built-in "comprehension". For

example, TEX knows absolutely nothing about PostScript graphics, yet by using `\special` and with the appropriate driver program (e.g., `dvips`), PostScript graphics can be easily incorporated into documents. Color is handled in the same way. In all cases, all that TEX does is to expand the `\special` command arguments and transfer the command to its normal output, that is, the DVI file (a file format that contains only page description commands).

Last, but not least, there is the notion of *change file* [3, page 243]:

> *A change file is a list of changes to be made to the* WEB *file; a bit like a stream editor script. These changes can comprise both adaptations of the* WEB *file to the particular Pascal compiler that will be used and bug fixes to TEX. Thus the* `TeX.web` *file needs never to be edited.*

Thus, change files provide a form of incremental modification. This is similar to the `patch` mechanism of Unix.

Yet, no matter how foresighted these methods may be, twenty years after its conception TEX has started to show its age. Today's trends, and more importantly the programming community's continuing demand for even more flexible techniques and systems, call for new modes of expressiveness.

2.2 The LATEX Format

LATEX [10], which was released around 1985, is the most widely known TEX format. Nowadays, it seems that LATEX is the de facto standard for the communication and publication of scientific documents (i.e., documents that contain a lot of mathematical notation). LATEX "programs" have a Pascal-like structure and the basic functionality is augmented with the incorporation of independently developed collections of macro packages. In addition, classes are used to define major document characteristics and are in essence document types, such as *book*, *article*, etc. Thus, each LATEX "program" is characterized by the document class to which it belongs, by the packages it utilizes, and any new macro commands it may provide.

The current version of LATEX is called LATEX 2_ε. Work is in progress to produce and widely distribute the next major version, LATEX3 [11]. Among the several enhancements that the new system will bring forth, are:

- Overall robustness
- Extensibility, relating to the package interface
- Better specification and inclusion of graphical material
- Better layout specification and handling
- Inclusion of requirements of hypertext systems

The LATEX3 core team expects that a major reimplementation of LATEX is needed in order to support the above goals.

The ConTEXt [13] format, developed by Hans Hagen, is monolithic when compared to LATEX. As a result, the lessons learned from its development are not of great interest to our study.

2.3 $\mathcal{NTS}$: The New Typesetting System

The $\mathcal{NTS}$ project [14] was established in 1992 as an attempt to extend TEX's typesetting capabilities and at the same time to propose a new underlying programmatic model. Its originators recognised that TEX lacked user-friendliness and as a consequence it attracted many fewer users than it could (or should). Moreover, TEX (both as a name and a program) was frozen by Knuth, so any enhancements should be implemented in a completely new system.

$\mathcal{NTS}$ was the first attempt to recognize that TEX's monolithic structure and implementation in an obsolete language (i.e., the Pascal programming language) are characteristics that could only impede its evolution. The techniques used to implement TEX, particularly its "tight", static and memory conservative data structures have no (good) reason to exist today (or even when $\mathcal{NTS}$ was conceived, in 1992), when we have had a paradigm shift to flexible programming techniques.

After considering and evaluating several programming paradigms [19] including functional, procedural and logic programming, the $\mathcal{NTS}$ project team decided to proceed with a Java-based implementation. Java's object-oriented features and its network awareness were the main reasons for adopting Java, as $\mathcal{NTS}$ was envisioned as a network-based program, able to download and combine elements from the network.

Today, there is a Java codebase, which has deconstructed the several functional pieces of TEX and reconstructed them in a more object-oriented way with cleaner interfaces, a property that the original TEX source clearly lacks. In spite of the promising nature of $\mathcal{NTS}$, the directory listing at CTAN[3] shows that the project is inactive since 2001[4]. It seems that the main focus is now the development of ε-TEX, which is presented in the following section.

2.4 ε-TEX

ε-TEX [17] was released by the $\mathcal{NTS}$ team as soon as it was recognized that $\mathcal{NTS}$ itself was very ambitious and that a more immediate and more easily conceivable goal should be set. So, it was decided that the first step towards a new typesetting system was to start with a reimplemented but 100% TEX compatible program.

ε-TEX was released in 1996, after three years of development and testing. It adds about thirty new primitives to the standard TEX core, including handling of bidirectional text (right-to-left typesetting). It can operate in three distinct modes:

1. "compatibility" mode, where it behaves exactly like standard TEX.
2. "extended" mode, where its new primitives are enabled. Full compatibility with TEX is not actually sought and the primary concern is to make typesetting easier through its new primitives.
3. "enhanced" mode, where bidirectional text is also supported. This mode is taken to be a radical departure from standard TEX.

[3] `http://www.ctan.org/tex-archive/systems/nts/`
[4] We have last accessed the above URL in March 2004.

Today, ε-TEX is part of all widely used TEX distributions and has proven to be very stable. Indeed, in 2003 the LATEX team requested that future distributions use ε-TEX by default for LATEX commands, which has since been implemented in TEX Live and other distributions.

2.5 Ω

Ω [16], which was first released in 1996, is primarily the work of two people: Yannis Haralambous and John Plaice. It extends TEX in order to support the typesetting of multilingual documents. Ω provides new primitives and new facilities for this reason. Ω's default character encoding is the Unicode UCS-2 encoding, while it can easily process files in almost any imaginable character encoding. In addition to that, Ω supports the parameterization of paragraph and page direction, thus allowing the typesetting of text in almost any imaginable writing method[5].

Much of its power comes from its new notion of ΩTPs (Ω Translation Processes). In general, an ΩTP is normally used to transform a document from a particular character encoding to another. Obviously, an ΩTP can be used to transform text from one character set to another. An ΩTP is actually a finite state automaton and, thus, it can easily handle cases where the typesetting of particular characters are context dependent. For example, in traditional Greek typography, there are two forms of the small letter theta, which are supported by Unicode [namely ϑ (03D1) and θ (03B8)]. The first form is used at the beginning of a word, while the second in the middle of a word. The following code borrowed from [16] implements exactly this feature:

```
input: 2; output: 2;
aliases:
LETTER  = (@"03AC-@"03D1 | @"03D5 | @"03D6 |
           @"03F0-@"03F3 | @"1F00-@"1FFF) ;
expressions:
^({LETTER})@"03B8({LETTER} | @"0027)
  => \1 @"3D1 \3;
. => \1;
```

For performance reasons, ΩTPs are compiled into ΩCPs (Ω Compiled Processes).

External ΩTPs are programs in any programming language that can handle problems that cannot be handled by ordinary ΩTPs. For example, one can prepare a Perl script that can insert spaces in a Thai language document. Technically, external ΩTPs are programs that read from the standard input and write to the standard output. Thus, Ω is forking a new process to allow the use of an external ΩTP. In [16] there are a number of examples (some of them were borrowed from [7]).

We should note that the field of multilingual typesetting is an active research field, which is the main reason why Ω is still an experimental system. We should also note that ε-Ω [4], by Giuseppe Bilotta, is an extension of Ω that tries to incorporate the best features of ε-TEX and Ω in a new typesetting engine.

[5] Currently the boustrophedon writing method is the only one not supported.

2.6 pdfTEX

pdfTEX [18] is yet another TEX extension that can directly produce a file in Adobe's PDF format. Recently, pdf-ε-TEX was introduced, merging the capabilities of both pdfTEX and ε-TEX.

3 Towards a Universal Typesetting Engine

From the discussion above, it is obvious that there is a trend to create new typesetting engines that provide the best features of different existing typesetting engines. Therefore, a Universal Typesetting Engine should incorporate all the novelties that the various TEX-like derivatives have presented so far. In addition, such a system should be designed by taking into serious consideration all aspects of modern software development and maintenance. However, our departure should not be too radical, in order to be able to use the existing codebase. Let us now examine all these issues in turn.

3.1 Discussion of Features

Data Structures. TEX's inherent limitations are due to the fact that it was developed in a time when computer resources were quite scarce. In addition, TEX was developed using the now outdated structured programming program development methodology.

Nowadays, hardware imposes virtually no limits in design and development of software. Also, new programming paradigms (e.g., aspect-oriented programming [8], generative programming [2], etc.) and techniques (e.g., extreme programming [1]) have emerged, which have substantially changed the way software is designed and developed.

These remarks suggest that a new typesetting engine should be free of "artificial" limitations. Naturally, this is not enough as we have to leave behind the outdated programming techniques and make use of modern techniques to ensure the future of the Universal Typesetting Engine. Certainly, $\mathcal{NTS}$ was a step in the right direction, but in the light of current developments in the area of software engineering it is now a rather outdated piece of software.

New Primitive Commands. Modern document manipulation demands new capabilities that could not have been foreseen at the time TEX was created. A modern typesetting engine should provide a number of new primitive commands to meet the new challenges imposed by modern document preparation. Although the new primitives introduced by ε-TEX and Ω solve certain problems (e.g., bidirectional or, more generally, multidirectional typesetting), they are still unable to tackle other issues, such as the inclusion of audio and/or animation.

Input Formats. For reasons of compatibility, the current input format must be supported. At the same time the proliferation of XML and its applications makes it more than mandatory to provide support for XML content. Currently,

XMLTEX is a TEX format that can be used to typeset validated XML files[6]. In addition, XLATEX [6] is an effort to reconcile the TEX world with the XML world. In particular, XLATEX is an XML Document Type Definition (DTD) designed to provide an XMLized syntax for LATEX. However, we should learn from the mistakes of the past and make the system quite adaptable. This means that as new document formats emerge, the system should be easily reconfigurable to "comprehend" these new formats.

Output Formats. The pdfLATEX variant has become quite widespread, due to its ability to directly produce output in a very popular document format (namely Adobe's Portable Document Format). Commercial versions of TEX are capable of directly generating PostScript files without the need of any driver programs. However, as in the case of the input formats, it is quite possible that new document formats will appear. Thus, we need to make sure that these document formats will find their way into TEX sooner or later.

In addition, XML initiatives such as MathML and SVG (Scalable Vector Graphics) are increasingly common in electronic publishing of scientific documents (i.e., quite demanding documents from a typographical point of view). Thus, it is absolutely necessary to be able to choose the output format(s) from a reasonable list of options. For example, when one makes a drawing using LATEX's `picture` environment, it would be quite useful to have SVG output in addition to the "standard" output. Currently, Ω can produce XML content, but it cannot generate PDF files.

Innovative Ideas. The assorted typesetting engines that follow TEX's spirit are not mere extensions of TEX. They have introduced a number of useful features and/or capabilities. For example, Ω's ΩTPs and its ability to handle Unicode input by default should certainly make their way into a new typesetting engine. In addition, ε-TEX's new conditional primitives are quite useful in macro programming.

Typesetting Algorithms. The paragraph breaking and hyphenation algorithms in TEX make the difference when it comes to typographic quality. Robust and adaptable as they are, these algorithms may still not produce satisfactory results for all possible cases. Thus, it is obvious that we need a mechanism that will adapt the algorithms so they can successfully handle such difficult cases.

Fonts. Typesetting means to put type (i.e., font glyphs) on paper. Currently, only METAFONT fonts and PostScript Type 1 fonts can be used with all different TEX derivatives. Although Ω is Unicode aware, still it cannot handle TrueType fonts in a satisfactory degree (one has to resort to programs like `ttf2tfm` in order to make use of these fonts). In addition, for new font formats such as

[6] Validation should be handled by an external utility. After all, there are a number of excellent tools that can accomplish this task and thus it is too demanding to ask for the incorporation of this feature in a typesetting engine.

OpenType and SVG fonts there is only experimental support, or none at all. A new typesetting engine should provide font support in the form of plug-ins so that support for new font formats could be easily provided.

Scripting. Scripting is widely accepted as a means of producing a larger software product from smaller components by "gluing" them together. It plays a significant role in producing flexible and open systems. Its realization is made through the so-called "scripting languages", which usually are different from the language used to implement the individual software components.

One could advance the idea that scripting in TEX is possible by using TEX the language itself. This is true to some extent, since TEX works in a form of "interpretive mode" where expressions can be created and evaluated dynamically at runtime – a feature providing the desired flexibility of scripting languages. But TEX itself is a closed system, in that almost everything needs to be programmed within TEX itself. This clearly does not lead to the desired openness.

A next generation typesetting engine should be made of components that can be "glued" together using any popular scripting language. To be able to program in one's language of choice is a highly wanted feature. In fact, we believe it is the only way to attract as many contributors as possible.

Development Method. Those software engineering techniques which have proven successful in the development of real-world applications should form the core of the program methodology which will be eventually used for the design and implementation of a next generation typesetting engine. Obviously, generic programming and extreme programming as well as aspect-oriented programming should be closely examined in order to devise a suitable development method.

All the features mentioned above as well as the desired ones are summarized in Table 1.

Table 1. Summary of features of TEX and its extensions.

	TEX	$\mathcal{NTS}$	ε-TEX	Ω	LATEX(3)	**Desired**
implementation language	traditional	Java	traditional	traditional	traditional	perhaps scripting
architecture	monolithic	modular?	monolithic	monolithic	monolithic	modular
TEX compatibility	100%	yes	100%	100%	100%	via module
input transformations				ΩTPs		pluggable
Unicode	(Babel)	(Java)	(Babel)	true		true
XML				yes	via package	yes
typesetting algorithms	TEX	TEX-like	TEX-like	TEX-like	TEX-like	pluggable
scripting language	TEX	$\mathcal{NTS}$ (?)	ε-TEX	Ω	TEX	any
output drivers	dvi(ps,pdf)	dvi(?)	dvi(ps,pdf)	dvi(ps,pdf)	dvi(ps,pdf)	any
`TRIP`-compatible	yes	almost	ε-`TRIP`	yes	yes	yes (via module)
library mode	no	no	no	no	no	yes
daemon (server) mode	no	no	no	no	no	yes
programming community	< LATEX	1 person?	< TEX	very small	big	> LATEX

3.2 Architectural Abstractions

Roughly speaking, the *Universal Typesetting Engine* we are proposing in this paper, is a project to design and, later, to implement a new system that will support all the "good features" incorporated in various TEX derivatives plus some novel ideas, which have not found their way in any existing TEX derivative.

Obviously, it is not enough to just propose the general features the new system should have – we need to lay down the concrete design principles that will govern the development of the system. A reasonable way to accomplish this task is to identify the various concepts that are involved. These concepts will make up the upper abstraction layer. By following a top-down analysis, eventually, we will be in position to have a complete picture of what is needed in order to proceed with the design of the system.

The next step in the design process is to choose a particular system architecture. TEX and its derivatives are definitely monolithic systems. Other commonly used system architectures include the microkernel and exokernel architectures, both well-known from operating system research.

Microkernel Architecture. A microkernel-based design has a number of advantages. First, it is potentially more reliable than a conventional monolithic architecture, as it allows for moving the major part of system functionality to other components, which make use of the microkernel. Second, a microkernel implements a flexible set of primitives, providing high level of abstraction, while imposing little or no limitations on system architecture. Therefore, building a system on top of an existing microkernel is significantly easier than developing it from scratch.

Exokernel Architecture. Exokernels follow a radically different approach. As with microkernels, they take as much out of the kernel as possible, but rather than placing that code into external programs (mostly user-space servers) as microkernels do, they place it into shared libraries that can be directly linked into application code. Exokernels are extremely small, since they arbitrarily limit their functionality to the protection and multiplexing of resources.

Both approaches have their pros and cons. We believe that a mixed approach is the best solution. For example, we can have libraries capable of handling the various font formats (e.g., Type 1, TrueType, OpenType, etc.) that will be utilized by external programs that implement various aspects of the typesetting process (e.g., generation of PostScript or PDF files). Let us now elaborate on the architecture we are proposing. The underlying components are given in Figure 1.

The *Typesetting Kernel* (TK) is one of the two core components at the first layer. It can be viewed as a "stripped-down" version of TEX, meaning that its role as a piece of software is the orchestration of several typesetting activities. A number of basic algorithms are included in this kernel both as abstract notions – necessary for a general-purpose typesetting engine – and concrete implementations. So, TK incorporates the notions of paragraph and page breaking, mathematical typesetting and is Unicode-aware. It must be emphasized that TK "knows" the concept of paragraph breaking and the role it plays in typeset-

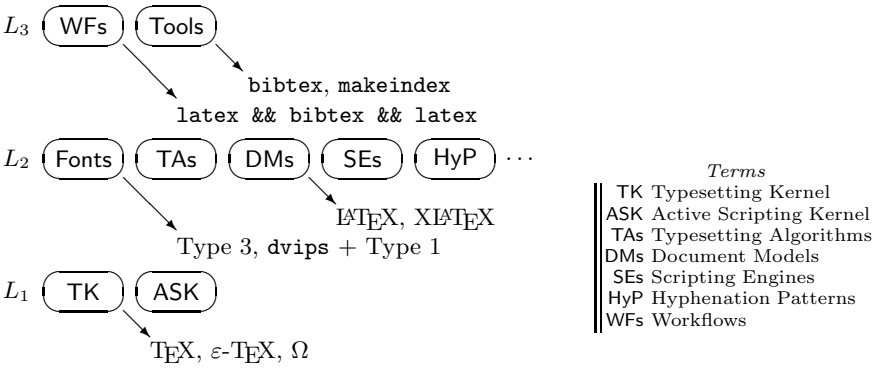


Fig. 1. The proposed microkernel-based layered architecture. The arrows show rough correspondence between the several architectural abstractions and their counterparts in existing monolithic typesetting engines.

ting but it is not bound to a specific paragraph breaking algorithm. The same principle applies to all needed algorithms.

The *Active Scripting Kernel* (ASK) is the second of the core components and the one that allows scripting at various levels, using a programming (scripting) language of one's choice. It is in essence a standardized way of communicating between several languages (TEX, Perl, Python), achieved by providing a consistent Application Programming Interface (API). The most interesting property of ASK is its *activeness*. This simply means that any extension programmed in some language is visible to any other available languages, as long as they adhere to the standard Active Scripting Kernel API. For example, an external module/service written in Perl that provides a new page breaking algorithm is not only visible but also available for immediate use from Python, C, etc.

Above TK and ASK, at the second layer, we find a collection of typesetting abstractions.

Fonts are at the heart of any typesetting engine. It is evident that font architectures change with the passing of time, and the only way to allow for flexibility in this part is to be open. Although there many different font formats, all are used to define glyphs and their properties. So instead of directly supporting all possible font formats, we propose the use of an abstract font format (much like all font editors have their own internal font format). With the use of external libraries that provide access to popular font formats (e.g., a Free Type library, a Type 1 font library, etc.), it should be straightforward to support any existing or future font format.

The various *Typesetting Algorithms* (TAs) – algorithms that implement a particular typographic feature – should be coded using the Active Scripting Kernel API. In a system providing the high degree of flexibility we are proposing, it will be possible to exhibit, *in the same document*, the result of applying several paragraph and page breaking algorithms. By simply changing a few runtime parameters it will be possible to produce different typographic "flavors" of the same document.

A *Scripting Engine* (SE) is the realization of the ASK APIs for a particular scripting language. For reasons of uniformity, the TEX programming language will be provided as a Scripting Engine, along with engines for Perl, Ruby and Python. This will make all the existing TEX codebase available for immediate use and it will provide for cooperation between existing LATEX packages and future enhancements in other languages. Thus, a level of 100% TEX compatibility will be achieved, merely as a "side-effect" of the provided flexibility.

The idea of a *Document Model* (DM) concerns two specific points: The document *external* representation, as it is "edited" for example in an editor, or "saved" on a hard disk, and its *internal* representation, used by the typesetting engine itself. It is clear that under this distinction, current LATEX documents follow the (fictional) "LATEX Document Model", XLATEX documents follow the "XLATEX document model" and an XML document with its corresponding DTD follows an analogous "XML+DTD Document Model".

We strongly believe that how a document is written should be separated from its processing. For the last part, an *internal* representation like the *Abstract Syntax Trees* (ASTs) used in compiler technology is highly beneficial. One way to think of DM is as the typographic equivalent of the Document Object Model (DOM). That is, it will be a platform-neutral and language-neutral representation allowing scripts to dynamically access and update the content, structure and style of documents.

Several *Document Processors* (DPs) may be applied to a specific document before actual typesetting takes place. DPs are the analog of ΩTPs. By leveraging the scripting power of ASK, the representation expressiveness of DPs is increased – as opposed to algorithmic expressiveness (Turing-completeness), which is evident, e.g., in Ω, but is not the sole issue.

The *Workflows* (WF) and *Tools* are at the highest architectural layer. Currently, there are a number of tools that may not produce a final typeset result, but are important for the proper preparation of a document. For example, such tools include bibliography, index and glossary generation tools. In the proposed architecture, all these programs will take advantage of other architectural abstractions – such as the Document Model or the Scripting Engines – in order to be more closely integrated in the typesetting engine as a whole.

Of particular importance is the introduction of the *Workflows* notion. A workflow is closely related to the operation or, to be more precise, *co*operation of several tools and the typesetting engine in the course of producing a typeset document. In effect, a workflow specifies the series of execution (probably conditional) steps and the respective inputs/outputs during the "preparation" of a document. By introducing a workflow specification for each tool, we relieve the user from manually specifying all the necessary actions in order to get a "final" `.pdf` (or whatever output format has been requested). Instead, the user will declaratively specify that the services of a tool are needed and the engine will load the respective workflows, compose them and execute them.

We shall give a workflow example concerning a BibTeX-like tool. What we do here is to transform our experience of using `bibtex` into declarations specifying its behaviour in cooperation with `latex`:

```
WORKFLOW DEFINITION bibtex

SERVICE bibtex NEEDS latex
SERVICE bibtex INTRODUCES latex
```

In effect, this translates a hypothetical `Makefile`:

```
all:
    latex mydoc
    bibtex mydoc
    latex mydoc
```

for the preparation of the fictitious `mydoc.tex` document into a declarative specification that is given only *once* as part of the `bibtex` tool!

3.3 On Design and Evolution

Recent advances in software engineering advocate the use of multidimensional separation of concerns as a guiding design principle. Different concerns should be handled at different parts of code and ideally should be separated. For example, the representation of a document and its processing are two separate concerns and should be treated as such. Their interaction is better specified out of their individual specifications. Thus, we have introduced the Document Models notion to cope with the existing TeX/LaTeX base as well as any future document representation.

Several architectural abstractions of Figure 1 are candidates to be specified as "services" at different granularities. For example, any *Tool* of the third layer can be thought of as a service that is registered with a naming authority and discovered dynamically, for immediate use on demand. A TrueType Font Service, regarding the second layer *Font* abstraction, is another example, this time more of a fine-grained nature, in the sense that a *Tool* (coarse-grained service) utilizes a *Font* (fine-grained service).

The proposed architecture makes special provisions for evolution by keeping rigid design decisions to a minimum. Built-in Unicode awareness is such a notable rigid design decision, but we feel that its incorporation is mandatory. Besides that, the ideas of pluggable algorithms and scripting are ubiquitious and help maintain the desired high degree of flexibility.

At the programming level, any style of design and development that promotes evolution can be applied. In the previous section we have actually demonstrated that the proposed architecture can even handle unanticipated evolution at the workflow level: the `bibtex` tool workflow specification causes the execution of an existing tool (`latex`) but we have neither altered any workflow for `latex` nor does `latex` need to know that "something new" is using it. In effect, we have *introduced* (the use of the keyword `INTRODUCE` was deliberate) a new *aspect* [8].

4 Conclusions and Future Work

In this paper we have reviewed the most widespread modern approaches to extending TEX, THE typesetting engine. After analyzing weaknesses of the approaches and the existing support for several features, we have presented our views on the architecture of an open and flexible typesetting engine.

We have laid down the basic architectural abstractions and discussed their need and purpose. Of course, the work is still at the beginning stages and we are now working on refining the ideas and evaluating design and implementation approaches.

The introduction of the Active Scripting Kernel is of prime importance and there is ongoing work to completely specify a) the form of a standard procedural API and b) support for other programming styles, including object-oriented and functional programming. This way, an *object* may for example take advantage of an algorithm that is better described in a *functional* form. There are parallel plans for transforming TEX into a Scripting Engine and at the same time providing Engines powered by Perl and Python.

We are also investigating the application of the workflow approach at several parts in the architecture other than the interaction among tools. This, in turn, may raise the need for the incorporation of a *Workflow Kernel* at the core layer, along with the Typesetting Kernel and the Active Scripting Kernel.

References

1. chromatic. *Extreme Programming Pocket Guide*. O'Reilly & Associates, Sebastopol, CA, USA, 2003.
2. Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison–Wesley Publ. Co., Reading, MA, USA, 2002.
3. Victor Eijkhout. TEX by Topic. `http://www.cs.utk.edu/~eijkhout/tbt`
4. ε-Ω Project home page. `http://www.ctan.org/tex-archive/systems/eomega/`
5. $\mathcal{NTS}$ FAQ. `http://www.ctan.org/tex-archive/info/NTS-FAQ`
6. Yannis Haralambous and John Plaice. Omega, OpenType and the XML World. *The 24th Annual Meeting and Conference of the TeX Users Group, TUG 2003*.
7. Yannis Haralambous and John Plaice. Traitement automatique des langues orientales et composition sous Omega. *Cahiers GUTenberg*, pages 139–166, 2001.
8. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *ECOOP '97 – Object-Oriented Programming: 11th European Conference, Jyväskylä, Finland, June 1997. Proceedings*, number 1241 in Lecture Notes in Computer Science, pages 220–242. Springer-Verlag, Berlin, 1997.
9. Donald Erwin Knuth. *The TEXbook*. Addison-Wesley, 1984.
10. Leslie Lamport. *LATEX: A Document Preparation System*. Addison–Wesley Publ. Co., Reading, MA, USA, 2nd edition, 1994.
11. LATEX3 Project home page. `http://www.latex-project.org/latex3.html`.
12. Richard Palais. Position Paper on the future of TEX. `http://www.loria.fr/services/tex/moteurs/nts-9207.dvi`, reached from `http://tex.loria.fr/english/moteurs.html`, October 1992.

13. PRAGMA Advanced Document Engineering. ConTEXt home page. http://www.pragma-ade.com/
14. $\mathcal{NTS}$ Project home page. http://www.dante.de/projects/nts/
15. Eric E. Raymond. The Cathedral and the Bazaar. http://www.catb.org/~esr/writings/cathedral-bazaar/
16. Apostolos Syropoulos, Antonis Tsolomitis, and Nick Sofroniou. *Digital Typography Using LaTeX*. Springer-Verlag, New York, NY, USA, 2003.
17. $\mathcal{NTS}$ Team and Peter Breitenlohner. The ε-TEX manual, Version 2. MAPS, (20):248–263, 1998.
18. Hàn Thế Thành, Sebastian Rahtz, and Hans Hagen. The pdfTEX users manual. MAPS, (22):94–114, 1999.
19. Jiri Zlatuska. $\mathcal{NTS}$: Programming Languages and Paradigms. EuroTEX 1999, http://www.uni-giessen.de/partosch/eurotex99/zlatuska.pdf