How MlBibTeX's Documentation Is Organised*

Jean-Michel HUFFLEN
LIFC (EA CNRS 4157)
University of Franche-Comté
16, route de Gray
25030 BESANÇON CEDEX
FRANCE
jmhufflen@lifc.univ-fcomte.fr
http://lifc.univ-fcomte.fr/home/~jmhufflen

Abstract

MlBibTeX's documentation is planned to be multilingual — that is, written in several languages — and to be able to share as many examples as possible. Different people can write translations of the original English documentation in parallel. Besides, we show how the translations of this documentation can be updated if need be. This documentation can be used as a printed text or as an on-line document. The functionalities managing this documentation can be reused by another program. In a first part, we explain in detail what our requirements are. Then we show how they are implemented.
Keywords MlBibTeX, LaTeX, mlbdoc package, Scheme, multilingual documentation, configuration management.

Streszczenie

Planujemy, aby dokumentacja MlBibTeXa była wielojęzyczna, tj. napisana w wielu językach i aby można było współdzielić możliwie wiele przykładów. Różni tłumacze mogą jednocześnie pisać tłumaczenia oryginalnej dokumentacji anglojęzycznej. Oprócz tego chcemy pokazać, jak można w razie potrzeby tę dokumentację aktualizować. Dokumentacja ta będzie mogła być używana albo w postaci drukowanej albo *on-line*. Mechanizmy zarządzania nią mogą zostać użyte do innych zastowań. W pierwszej części wyjaśnimy w szczegółach jakie stawiamy wymagania, a następnie omówimy sposób, w jaki zostały one zaimplementowane.
Słowa kluczowe MlBibTeX, LaTeX, pakiet mlbdoc, Scheme, dokumentacja wielojęzyczna, zarządzanie konfiguracją.

0 Introduction

Software documentation is an important part, and it is well-known that maintaining such documentation causes some difficulty [20], especially if the program evolves in successive versions. Updating a documentation is often done late, in comparison with program update. There are different kinds of documentation: requirements, source code documentation, installation manual, documentation about tests, user manual, ... In this article, we will only focus on installation and user manuals. In addition, that is good for such manuals to be provided in different natural languages, but may complicate updates, if precise conventions have not been defined. Here we will explain our conventions for the documentation of MlBibTeX[1]. Let us recall that this programs aims to be a 'better and extended BibTeX' [18] — the bibliography processor usually associated with the LaTeX word processor [12] — with particular focus on multilingual features [6]. As explained in [7], MlBibTeX allows BibTeX's bibliography styles to be run in a compatibility mode, but its new features are based on paradigms related to XML[2]. Last but not least, MlBibTeX has been written in Scheme [10], as we explained in [8]. We have begun to write MlBibTeX's documentation in both English and French. This led us to the definition of a pre-

* Title in Polish: *Jak jest zorganizowana dokumentacja MlBibTeXa*.

[1] **M**ulti**L**ingual BibTeX.

[2] e**X**tensible **M**arkup **L**anguage. Readers interested in a general introductory book to this formalism can refer to [19].

cise framework for a multilingual documentation. Later, we also plan to give a documentation in German. Anyway, we think that other people could write other translations of this documentation. In a first section, we list our requirements about a good multilingual documentation. Then Section 2 shows that the tools used to install and maintain software do not exactly meet our requirements. Section 3 explains how our documentation system is implemented. Reading this article only requires basic knwoledge about programming as well as software installation and maintenance, as described in [20] from a general point of view, or more specifically in [13], that introduces the languages and programs doing such tasks within the GNU[3] project.

1 Requirements

Here are our requirements for MlBibTeX's installation and user manual. Of course, the word 'requirements' is too strong, since there is no 'official' contract between a developer and a customer. However, the list above clearly explains what our framework is, what we aim to do, and why.

- The basis of MlBibTeX's installation and user manual is the English version. Some translations may be provided, but any change related to the services provided — as well as any error fixing — must be applied to the English version at first, and then translations can be updated.
- MlBibTeX's original installation and user manual — as well as all its translations — must be processed by LaTeX or pdfLaTeX. Some translations or additional parts may be typeset by another TeX-based engine — e.g., LuaTeX — but the word processor must be a TeX-based engine or format. This last point holds for languages using a special TeX engine, e.g., pTeX [16] for Japanese.
- On-line versions of MlBibTeX's installation and user manual can be available, they must be build by means of converters from source texts for TeX-based engines to (X)HTML[4], such as TeX4ht or LaTeX2HTML [4, Ch. 3–4].
- Each translation may be written and updated by different people, possibly working at different sites. Of course, a team cannot be in charge of all the translations, but error-fixing should be supported by the people that have realised previous versions.
- If some points are added in a translation, that should be clearly stated within the source text.
- If some parts are missing within a translation, they must be replaced by a text in another language, preferably the original English version.
- Examples illustrating manuals should be shared among versions, as far as possible. If an example is replaced within a translation, each update of the original example must cause the changed example to be re-examined. In other words, examples could be translated for better understanding, but examples' translations should be updated as soon as original examples are.
- A translation should be done from the original English text, as far as possible. If a translation has been done from another translation, this information must be kept. In other words, translating a translation into a close language might be easier, but if any source text of a translation is updated, the translation's target must be updated, too.

2 State of the art

2.1 Using LaTeX documentation tools?

The items of our requirements related to TeX-based engines are easy to put into action. However, the packages described in [14, Ch. 14] seem to us to be unsuitable for our purpose. First, they have been developed to document new functionalities added to LaTeX; if you use MlBibTeX, only few macros are to be defined in LaTeX, in comparison with other points of the documentation: installing MlBibTeX, calling it with suitable options, extended syntax for bibliography (.bib files), use of bibliography styles written in nbst[5] [6]. Second, the only way to build multilingual documentation by means of these packages seems the use of tags [14, § 14. 1.5] such as:

```
%<*english>
    ... (A text in English.)
%</english>
%<*polish>
    ... (This text's translation in Polish.)
%</polish>
%<*english|polish>
    ... (An example, given verbatim, common to
    both the English and Polish versions.)
%</english|polish>
```

Concerning our purpose, this tag system has severe drawbacks. First, even if we can order the insertion of a file between a pair of opening and closing

[3] Recursive acronym for '**G**NU's **N**ot UNIX'.

[4] (e**X**tensible) **H**yper**T**ext **M**arkup **L**anguage. XHTML is a reformulation of HTML — the original language of *Web* pages — using XML conventions. [15] is a good introduction to these languages.

[5] **N**ew **B**ibliography **ST**yles. This language is close to XSLT (e**X**tensible **S**tylesheet **L**anguage **T**ransformations), the language of transformations used for XML texts [25].

tags, this complicates the parallel writing of translated documentations[6]. Second, if the master file is changed, it is up to users to generate only files impacted by this change. In other words, the only 'sure' solution it to generate again all the files of all the versions, what is exaggerated after just fixing a typographical mistake. This second point is related to dependency management, as we will show now.

2.2 Dependency management

The points of our requirements not related to the use of TeX-based engines concern dependency management. In Software Engineering, there are utility tools that automatically rebuild executable files to be updated after some change of source code files. The most well-known program to do that is make, present since Unix's first versions. The configuration is described in a file called Makefile, e.g.:

```
example.o: example.c macros.h
    gcc -c -o example.o example.c
```

expresses that if the example.o file is either non-existing, or older[7] than the example.c or macros.h file, this example.c file is compiled by means of the gcc[8] compiler. In other words, if a source file has been modified since the last compilation order, this source file has to be compiled again. So, the command '`make example.o`' may or may not issue a compilation order. In practice, this make program is used both in maintenance and installation. In the first case, we can rebuild only executable files impacted by changing source files. In the second case, the whole process of installing a software is launched by compilation orders issued by the make command. More details about this program can be found in [17]. However, this make program has three main drawbacks. First, Makefile's syntax is not user-friendly. A partial solution to this problem is the use of a generator of Makefiles, e.g., imake [3], this preprocessor being particularly useful for software using graphical capabilities. Second, the specification of the commands launched by the make program assumes that some tools and libraries — e.g., a C compiler or some graphical libraries — are available. The imake program may localise such tools and libraries, but most often a configure command is provided in the distribution of the software to be installed and performs such checks [13, App. B]. So, the 'standard' way to install software on Unix-based systems, — e.g., Linux — is:

```
./configure    # Looking for tools needed for
       # software installation. Absolute paths to
       # reach them are put in Makefiles files.
make   # Building the software.
make install    # Installing it in public places.
```

In such a case, the distribution includes *template files*, called Makefile.in. They are processed by the configure command to create final outputs, as 'definitive' Makefile. Most often, configure files are generated by means of the autoconf program [24]. The third drawback of make also exists within Makefile.in files and is related to *information redundancy*. If we consider programs written using the C programming language [11], the fact that an example.c program relies upon a macros.h file — see the example above — is put at the beginning of the example.c file:

```
#include "macros.h"
```

because the macros.h file contains preprocessing directives and type definitions that must be known by the C compiler when it operates on the example.c file. As a consequence, this information related to dependency is put twice: a first time within the source file, a second one within Makefile. The same drawback holds with the Ant program [23]: it can be viewed as a 'modern' version of make, using XML-like syntax for its configuration files[9], but end users have to put down dependency relations in these files. To cope with this problem, a workaround consists of using an option of the gcc compiler:

```
gcc -MM example.c
(answers:)  example.o: example.c macros.h
```

and adding dependency information to Makefiles dynamically. Another solution is provided by some tools usable above the make command, the most well-known tool being automake [24], an alternative one being CMake[10] [1]. However, these tools are language-dependent. The make program can deal with source files written in any programming language, provided that the tool used to compile them is identified. On the contrary, automake and CMake only deal with some programming languages, such as C or C++ [22] — or Lua [9] for CMake — but are presently unusable with (LA)TeX source files. Last but not least, our dependency relation may be *semantic*: if an example file e is replaced by another, say e_0, we want e_0's author to be warned if e has changed, but e_0 may or may not have to be updated:

[6] Of course, there are version management tools that can perform a merge operation among several releases developed in parallel from the same files — the most recent and efficient being undoubtedly Subversion [2] — but this merge operation is still difficult and error-prone. Besides, it has to be guided by users.

[7] Last modification times are compared.

[8] **G**nu **C** **C**ompiler. See [21] for details about it.

[9] ... generally called build.xml.

[10] **C**ross platform **MAKE**.

this relation is called *weak dependency*, that is, this relation can be ignored. To sum up, the existing tools do not meet our requirements, that is why we have developed our own system.

3 MlBibTeX's documentation system

3.1 Package and commands

The documentation's files use the mlbdoc package, that will be added to MlBibTeX's distribution. This package requires the fancyvrb package [14, § 3.4.3], used to insert example files *verbatim*. To avoid name conflicts, all the names of the command provided by this package are prefixed by '`mlbdoc`'. Table 1 gives them. In this table, '*cf*' is for the current file being processed by LaTeX, '*f*' and 'f_0' are for any file names, '$f \leftarrow f_0$' (resp. '$f \leftharpoonup f_0$') denotes that *f* depends (resp. *weakly* depends) on f_0. When *cf* is processed, an additional auxiliary file, *cf*.dep.scm, is built. Such a file is devoted to be read by a Scheme interpreter. More precisely, all the .dep.scm files are supposed to be stored in the same directory, and the dependency graph should be acyclic. Table 2 gives all the Scheme functions used to manage MlBibTeX's documentation. In this table, '*d*' is a string denoting a directory name — use `"."` for the current directory.

3.2 Discussion

Using LaTeX to generate files processed by Scheme might seem strange. Of course, this is related to the fact that Scheme is MlBibTeX's implementation language. In addition, an important advantage, related to Lisp[11]-like languages, is that data and programs use the same format. We take advantage of this feature: when .dep.scm files are processed (cf. Fig. 2), the expressions they include are evaluated. On another point, our functions are portable, we just had to develop an interface between operating systems and Scheme interpreters. Besides, our interaction between LaTeX and Scheme cannot be compared by what is done within LuaTeX [5]. In both cases, there is some cooperation between a TeX-based engine and a more 'classical' programming language. LuaTeX allows fragments written using Lua to be called as procedures, whereas our Scheme functions 'intelligently' pilots calls of LaTeX or pdfLaTeX.

4 Conclusion

For other purposes, we built LaTeX documents whose typesetting was controlled by the make program, and we were not fully satisfied by the result. We think our system — presently running on Linux — is better, even if it cannot be used for any documentation. At the time of writing, we have written alone all the parts of the present documentation. The 'baptism of fire' will arise when other people participate in this documentation.

[11] **LIS**t **P**rocessor.Scheme belongs to this language family.

`\mlbdocrunswith{s}{s0}` expresses that *cf* is to be processed with the *s* (resp. s_0) TeX-based engine to produce a .dvi[a] (resp. .pdf[b]) file; if *s* and s_0 are not made precise, they default to LaTeX and pdfLaTeX;

`\mlbdoctranslates{f}` expresses that *cf* is a translation of *f*, so $cf \leftharpoonup f$;

`\mlbdocexf{f}` inserts the contents of *f* *verbatim* and expresses that $cf \leftarrow f$;

`\mlbdocinput{f}` tells LaTeX to process *f* and expresses that $cf \leftarrow f$; the `\input` command is redefined, too, and has the same effect[c];

`\mlbdocinclude{f}` tells LaTeX to process *f* if it appears within the `\includeonly` command's arguments[d] and expresses that $cf \leftarrow f$; the `\include` command is redefined, too, and has the same effect;

`\mlbdocreplaces{f}{f0}` inserts *f* by means of the `\mlbdocinput` command; expresses that *f* replaces f_0 within the current translation, that is, $cf \leftarrow f$ and $f \leftharpoonup f_0$;

`\mlbdocincludegraphics[opt][opt0]{f}` if the package graphics or graphicx[e] has been loaded, inserts *f* as the original `\includegraphics` would do with optional arguments *opt* and opt_0 and expresses that $cf \leftarrow f$; the `\includegraphics` command is redefined and has the same effect;

`\mlbdocincludegraphics*[opt][opt0]{f}` like above, but the original command interfaced and redefined is `\includegraphics*`.

[a] **De**vice-**I**ndependent.
[b] **P**ortable **D**ocument **F**ormat.
[c] However, we think that using the `\mlbdocinput` command is better, for sake of clarity. The same remark holds about the other three commands `\mlbdocinclude`, `\mldocincludegraphics`, and `\mlbdocincludegraphics*`.
[d] See [14, § 2.1.2] for more details about this feature of LaTeX.
[e] See [14, § 10.2.2 & 10.2.3] for more details about these packages allowing the insertion of graphical files.

Table 1: Commands provided by the mlbdoc package.

5 Acknowledgements

Many thanks to Jerzy B. Ludwichowski, who has translated the abstract and keywords in Polish very quickly.

References

[1] *CMake*. February 2009. `http://www.cmake.org/`.

[2] Ben Collins-Sussman, Brian W. Fitzpatrick

`((mlbdoc-dependencies 'get) d)` reads and processes all the .dep.scm files present in the `d` directory; returns `#t` — the 'true' value — if such files exist and the dependency graph is acyclic, `#f` — the 'false' value — otherwise;

`((mlbdoc-dependencies 'display-for) f)` displays the names of all the files `f` depends on, returns `#t` if `f` exists, `#f` otherwise;

`((mlbdoc-dependencies 'display-all))` displays all the dependencies and returns `#t`;

`((mlbdoc-dependencies 'remove-wd) f` f_0`)` removes the weak dependency relation $f \leftharpoondown f_0$; returns `#t` except if there is a non-weak dependency relation between these two files, in which case `#f` is returned;

`((mlbdoc-dependencies 'make) target)` causes `target` to be built if need be; `target` is a string denoting a .dvi or .pdf file name; if you want to rebuild the whole, bind `target` to the `all` symbol; returns `#f` if `target` is an unsuitable file name, otherwise returns `#t`;

`((mlbdoc-dependencies 'reset-all))` cleans up the dependency graph and returns `#t`.

Table 2: Using the Scheme functions to manage MlBibTeX's documentation.

and C. Michael Pilato: *Version Control with Subversion*. O'Reilly. June 2004.

[3] Paul DuBois: *Software Portability with imake*. 2nd edition. O'Reilly & Associates, Inc. September 1996.

[4] Michel Goossens and Sebastian Rahtz, with Eitan M. Gurari, Ross Moore and Robert S. Sutor: *The LaTeX Web Companion*. Addison-Wesley Longmann, Inc., Reading, Massachusetts. May 1999.

[5] Hans Hagen: "The Luafication of TeX and ConTeXt". In: *Proc. BachoTeX 2008 Conference*, pp. 114–123. April 2008.

[6] Jean-Michel Hufflen: "MlBibTeX's Version 1.3". TUG*boat*, Vol. 24, no. 2, pp. 249–262. July 2003.

[7] Jean-Michel Hufflen: "BibTeX, MlBibTeX and Bibliography Styles". *Biuletyn* GUST, Vol. 23, pp. 76–80. In *BachoTeX 2006 conference*. April 2006.

[8] Jean-Michel Hufflen: "MlBibTeX: Reporting the Experience". TUG*boat*, Vol. 29, no. 1, pp. 157–162. EuroBachoTeX 2007 proceedings. 2007.

[9] Roberto Ierusalimschy: *Programming in Lua*. 2nd edition. Lua.org. March 2006.

[10] Richard Kelsey, William D. Clinger, Jonathan A. Rees, Harold Abelson, Norman I. Adams iv, David H. Bartley, Gary Brooks, R. Kent Dybvig, Daniel P. Friedman, Robert Halstead, Chris Hanson, Christopher T. Haynes, Eugene Edmund Kohlbecker, Jr, Donald Oxley, Kent M. Pitman, Guillermo J. Rozas, Guy Lewis Steele, Jr, Gerald Jay Sussman and Mitchell Wand: "Revised^5 Report on the Algorithmic Language Scheme". HOSC, Vol. 11, no. 1, pp. 7–105. August 1998.

[11] Brian W. Kernighan and Denis M. Ritchie: *The C Programming Language*. 2nd edition. Prentice Hall. 1988.

[12] Leslie Lamport: *LaTeX: A Document Preparation System. User's Guide and Reference Manual*. Addison-Wesley Publishing Company, Reading, Massachusetts. 1994.

[13] Miki Loukides and Andy Oram: *Programming with* GNU *Software*. O'Reilly & Associates, Inc. December 1996.

[14] Frank Mittelbach and Michel Goossens, with Johannes Braams, David Carlisle, Chris A. Rowley, Christine Detig and Joachim Schrod: *The LaTeX Companion*. 2nd edition. Addison-Wesley Publishing Company, Reading, Massachusetts. August 2004.

[15] Chuck Musciano and Bill Kennedy: HTML *&* XHTML*: The Definitive Guide*. 5th edition. O'Reilly & Associates, Inc. August 2002.

[16] Okumura Haruhiko: "pTeX and Japanese Typesetting". *The Asian Journal of TeX*, Vol. 2, no. 1, pp. 43–51. April 2008.

[17] Andrew Oram and Steve Talbott: *Managing Projects with make*. 2nd edition. O'Reilly & Associates, Inc. October 1991.

[18] Oren Patashnik: *BibTeXing*. February 1988. Part of the BibTeX distribution.

[19] Erik T. Ray: *Learning* XML. O'Reilly & Associates, Inc. January 2001.

[20] Ian Sommerville: *Software Engineering*. 8th edition. Addison-Wesley. June 2006.

[21] Richard Matthew Stallman and the GCC developer community: *Using the* GNU *Compiler Collection*. October 2003. `http://gcc.gnu.org/onlinedocs/gcc.pdf`.

[22] Bjarne Stroustrup: *The C++ Programming Language*. 2nd edition. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts. 1991.

[23] Jesse Tilly and Eric M. Burke: *Ant: the Definitive Guide*. O'Reilly & Associates, Inc. May 2002.

[24] Gary V. Vaughn, Ben Ellison, Tom Tromey and Ian Lance Taylor: GNU *Autoconf, Automake, and Libtool*. Sams. October 2000.

[25] W3C: XSL *Transformations (*XSLT*). Version 1.0*. W3C Recommendation. Edited by James Clark. November 1999. `http://www.w3.org/TR/1999/REC-xslt-19991116`.