Practical journal and proceedings publication on paper and on the web

Péter Szabó
Budapest University of Technology and Economics,
Dept. of Computer Science and Information Theory,
H-1117 Hungary, Budapest, Magyar tudósok körútja 2.
pts (at) cs dot bme dot hu
http://www.inf.bme.hu/~pts/

Abstract

Although TeX is a reliable, high-quality and well-understood tool for authors writing their conference and journal articles, editors and typesetters face a much more difficult task when they want to compose articles for actual journal publication or preprints. We present organisational and software solutions for problems editors of journals and proceedings might face. As case studies we present issues and some conclusions regarding the production of the proceedings for two conferences we organised (EuroTeX 2006 and the non-TeX-related LME 2006 conference).

1 Introduction

We address problems during typesetting a collection of articles — usually a conference proceedings or a journal issue, from now on referred to as a "collection". There are three parties cooperating: the authors, the editors and the printshop. Using our terms, an "editor" is someone who accepts articles from the authors, reviews articles, proofreads articles, typesets articles, or compiles a list of articles into a collection. We assume that editors work on LaTeX article-like documents, and they convert any document they receive to this format. We also assume that the document class has already been designed by the typographer. We discuss converting articles to LaTeX format, editing individual articles, and compiling a collection to be printed by the printshop, and also publishing it on the web as a set of PDF files.

We assume that time to be spent on editing is short, there are only a few editors, and not all the editors have a complete understanding of the whole publication process — some of them only review articles, others deal only with web pages, etc. We assume that there is a chief editor who would be able to do the whole job (except for peer review) if there was enough time.

We use two conferences we organised in 2006 as case studies. One of them is EuroTeX 2006, an installment of the annual conference of the European TeX community. Authors usually submit articles as TeX source (most of them writing LaTeX source using the document class the editors proposed), and the submitted material is of high typographic quality. That is, paragraphs, pages, tables and graphics look nice; graphics are in a scalable (vector) file format; extensive bibliographies arrive in a BibTeX format; and the layout is reasonably separated from the text so that editors can change the layout easily.

The other conference is LME GNU/Linux Conference 2006, known as LME 2006. It is one of the annual conferences of the Hungarian Linux and Unix community. Articles submitted are of varying linguistic and typographic quality. Most authors have never heard of TeX; many of them haven't ever written an article before. They use plain text editors or OpenOffice (or an equivalent word processor) when writing documents. Editors have a lot of work to do with each article: file format conversion (from OpenOffice to LaTeX), and proofreading and typesetting are slow. Some authors send graphics of extremely low quality or with unreadable captions — editors have to ask for a better version. They usually forget the bibliography or submit incomplete or incorrect entries — editors have to correct and supplement it.

We present the technology we found useful and best practices we have developed as a list of practical suggestions, some of them in imperative style. This is not meant to imply, however, that our solution is the only one feasible.

2 Organising work

Because of time pressure it is important that editors can work in a software environment most comfortable for them, and that they always have access to all the information they need. It is also important that document compilation works in a reasonably uniform way, so that e.g. line breaks don't depend on the computer the document was compiled on.

It is recommended that all work be done as soon as possible. For example, the mailing list and the repository can be created, mail client, chat client, repository client software and TEX distributions and companion programs (like Ghostscript, OpenOffice and *sam2p*) can be installed way before the first article is submitted. The same applies to creating a document class (possibly from a layout designed by a typographer), collecting mail and chat addresses of each editor, providing access to the repository for the editors, and a little planning about the workflow.

2.1 The repository

The repository is a shared file store used by the editors. In a simplest case it is a shared folder on a server to which all editors have read and write access. However, using a version control system (such as Subversion [1]) is strongly recommended, because of these advantages:[1]

- All past versions of files are available. If something goes wrong today, one can check out yesterday's state from the repository, and continue from there. We can also easily see what has changed, so there is a good starting point for finding what went wrong. Once the latest working version is identified, it is possible to revert to it easily.
- Each editor has their own (partial) copy of the repository. If the repository is lost in a server crash, editors can combine their copies and start a new repository. (This is quite inconvenient, but still a lot better than having to rewrite the whole collection from scratch.)
- Each write to the repository (called "commit") is logged (who did it, when it happened, what files were affected and how). Thus if something goes wrong, we can find out who is capable of fixing the problem (usually the editor who introduced the problem is capable of fixing it, or he can provide the most relevant information for somebody else to fix it).
- If there are two different versions of the same file, there is no confusion as to which one is relevant (or more recent). The version control system automatically takes care of propagating changes in the right direction, without the need for manual review. This is a lot better than having several copies of the same file in a shared folder without knowing how they derived from each other and which one is relevant for future work.
- Synchronising working copies is easy. If an editor makes a change, he executes a commit operation (which copies all local changes back to the repository), notifies others (usually on the mailing list) to update, and the other editors execute an update operation (which copies changes from the repository to their local copy). This works even if two or more editors are making changes on the same *text* file. If a conflict arises (i.e. two people made changes on the same line of a file), it has to be resolved by hand. Conflict resolution is distributed: the editor who was slower to make his change has to resolve the conflict. The chief editor is freed from the work of comparing different versions of the same file received in e-mail.
- If an editor uses several computers, a version control system provides seamless synchronisation between each local copy.
- Most version control systems provide a read-only web view. (We used `SVN::Web` for Subversion.) This is useful to allow the world to know the progress of the editing process. Authors or other organisers can be given access to the repository's web view, so that they can download recent and old versions of the files, they can view the differences with the file versions, and they can see a history of changes by examining the commit log.

Editors mustn't be allowed to share files in any other way than using the repository. The most common objection is that they haven't used such a system before, and there is no time to learn it now. However, if the chief editor writes a short tutorial about the version control system and the repository, and he helps other editors to install it (preferably via phone or voice chat), the learning time can be reduced to one or two hours. Using a version control system really pays off in both time and reliability. The size of the project being small isn't a valid argument against it either, because advantages are present even for single-file projects.

The chief editor must design the repository tree structure, and enforce it by moving files. It is not a problem that editors don't fully understand the structure, because with a good version control system (such as Subversion), files and folders can be moved and renamed easily. Some rules we used with EuroTEX 2006: all filenames must be lower case English (with some additional restrictions on the allowed characters); file name length is not limited; all files received from the authors must be put

[1] The same advantages apply to software development — an area where version control systems have been used for decades with great success.

in the folder `art/00from_author/articlename`, all files needed by the article must be copied to `art/01recompiled/articlename`, and compiled there with only minimal modifications (in case of compilation problems, the author must be notified), and all compiled articles must be copied to and typeset in `art/02typeset/articlename` (with possibly a lot of modifications); the local texmf tree is in `texmf`, all necessary packages and fonts must be added there.

Everything possibly needed by editors should be added to the repository. This includes scripts, libraries, fonts and TEX packages used, and also tutorials and guidelines. Software which is easy to install from packages (e.g. MiKTEX and Ghostscript) should not be added, however, but should be mentioned in a guideline along with the recommended version of each package. Files that can be regenerated (such as temporary files like `.aux` files and output files like `.dvi` and `.pdf`) shouldn't be added, except for milestone versions of output files (e.g. the `.ps` file sent to the printshop or the `.pdf` file sent to the proofreader).

Some version control systems distinguish between text and binary files. The difference must be understood, and files must be added in the proper mode. Both file types have advantages.

Editors should be encouraged to immediately correct each mistake they find in the repository. If they are not sure whether their correction is good, an easy solution is to ask them to contact the chief editor via chat, commit the change, and let the chief editor review it immediately. (The web view can be used to quickly get an overview on the changes.) The downside is that a wrong change might be in a repository for a few minutes. To avoid that, version control systems offer *branches*, but branches are usually too complicated to learn and use for newbies.

Sometimes editors forget to add a few files to the repository (for example, they add a nonstandard document class, but they forget to add the nonstandard packages loaded by the document class). This mistake can be prevented by asking the editors to have two working copies, and if they add a file in one working copy, they should recompile in the other one. Under Linux using *strace* is an alternative solution: running `strace -e open latex foo` prints all the files opened by *latex* when compiling *foo.tex*.

2.2 Mailing list

There should be a mailing list to which authors, editors and organisers can post; and editors and organisers can read the posted messages. (Multiple mailing lists can be created if a large traffic volume is expected.) Authors should be encouraged to upload their articles to the web and post URLs to the mailing list. Alternatively, somebody should be made responsible for receiving articles from authors, adding them to the repository, and notifying editors about the article. It is generally a bad idea to receive articles on the mailing list, mostly because articles might be several dozen megabytes long.

The mailing list should be used only for notification and discussion, not for data transfer. All data to be worked on should be added to the repository, and others should be notified on the mailing list to update their working copy and do the appropriate action on the file. If there is a consistent proposal during a discussion, it also should be added to the repository instead of the mailing list.

2.3 Phone

Using the phone is the most efficient way that two distant parties can cooperate in real time. A phone call is extremely useful when one of the coworkers needs help (e.g. the commit resulted in a conflict, and the other party doesn't know how to resolve it), or when actions have to be synchronised (e.g. an editor commits a change he is not sure about, and the chief editor reviews it immediately).

When working on a computer connected to the Internet, one can make voice calls for free. Using Internet voice calls also gives the benefit of having free hands, so one can use his ears, eyes, mouth and fingers at the same time to solve a problem. Laptop users shouldn't rely on the built-in microphone of their laptop because of the terrible sound quality and the echo experienced on the other side of line. An external headset or a multimedia earphone (even as cheap as 5 euros) is a minimum.

2.4 Chat

Sometimes it might be feasible to use some chat (instant messaging) application instead of making a phone call. It is recommended that each editor have a chat account, and be online while working. However, we note that cooperation can be much more successful using the phone, because on the phone parties have each other's exclusive attention, with only a very few possible events to interrupt or suspend the conversation.

The chief editor should be registered in as many instant messaging networks as needed, and should use a multi-protocol client such as Gaim. Editors should use a client that beeps or pops up a window when a new message arrives, so they notice the message immediately. Web-based clients are thus out, because they don't notify the receiver.

2.5 Software

It is important to have software recommendations (including version numbers) for editors, so if the compilation output on two machines differs, it might be solved easily by switching to the recommended software.

On Unix we used teTeX 2 and 3 with some packages downloaded from CTAN to our local texmf tree. For TeX source editing one could use any text editor; we recommended Kile and Kate. On Windows we used MiKTeX as a TeX distribution and TeXnicCenter (and even Textpad) for editing.

We experienced font rendering problems and other bugs with Ghostscript 8.1x, so we recommended to upgrade to Ghostscript ≥ 8.53.

As additional tools, we used the latest *sam2p* for raster image conversion, the latest *pdfconcat* for PDF concatenation and the *pdftops* tool from the Xpdf distribution for PDF to PostScript conversion.

We had our Subversion repository on a Unix server. For security, we allowed read-write access using `svn+ssh://` only. Users were authenticated using SSH public keys. We forced the *svnserve* command for these users in the `authorized_keys` file of SSH, with the parameters `-tunnel-user=... -t -r ....` We also used an `authz-db` file in `svnserve.conf` to further tune access. `SVN::Web` was our read-only web frontend to Subversion. We patched it a little so that it could display commit log messages and files in a character set other than UTF-8. As a Windows client we recommended TortoiseSVN with PuTTY's *pageant* utility to avoid typing the passphrase for the public key again and again. We also prepared a tutorial on generating an SSH public key and setting up TortoiseSVN on Windows.

Our chief editor relied on the common scripting facilities of Unix (shell scripts, GNU Make and Perl), which helped his work a lot. However, other editors could work without those scripts if they wanted to, and they were in no way forced to understand the scripts. The recommended use of scripting was documented for them in a tutorial.

3 Tasks of the editors

Once papers start arriving, editors can start working on them. Although version control systems allow parallel modifications to the same file, this might result in conflicts, so we recommend that editors announce on the mailing list when they start or stop working on an article.

Usually one editor is able to typeset an article perfectly, except for proofreading, which should be done by as many people as possible. For LME 2006 each paper was reviewed by two experts, and checked for spelling and linguistic mistakes by two proofreaders, and we found errors even after that.

3.1 File format conversion

Recent versions of OpenOffice 2.0 contain a LaTeX export filter, which can be used to convert word processor documents to LaTeX. The filter handles paragraph breaks, bold and italic, emits simple Latin-1 and Latin-2 accented characters properly (without the *inputenc* package), and can export math formulas (we didn't test this feature thoroughly, because for LME 2006 we had only simple math formulas in documents). Since our documents had a lot of display verbatim material, and the export filter emitted it line-by-line, escaping each special character differently, we wrote a Perl script *postootex.pl* which post-processes the output of the export filter, that is, converts consecutive typewriter lines to a *verbatim* environment. The export filter was also quite loose on exporting font changes, it emitted superfluous `\rmfamily`, `\mdseries`, and font size change etc. commands even when there was no change at all. So we added code to the Perl script to remove these. We wanted a LaTeX document that is easy to read and edit for humans, so we converted the markup input to Latin-2 (e.g. `\'a` to `á`). We also made the script remove the multitude of unnecessary braces inserted randomly by the export filter. Lists and enumerations were emitted almost properly, but the export filter insisted on reproducing the exact list formatting (margins, item width, etc.), so we removed this too, but with that we lost the list depths, so we had to check each nested list by hand. Exporting of tables, figures and floats was so preliminary that we decided to retype these elements by hand.

The usefulness of a custom Perl script to convert TeX sources might sound questionable. We decided to write a script after frustration during the manual cleanup of the export filter's output on a 5-page document. We wrote the script so it tries to follow the LaTeX syntax closely enough that it doesn't get confused e.g. by nested braces in an undelimited macro argument, and thus can clean up the source file reasonably well. The script didn't try to fix rare problems — its sole purpose was to save the editor the time of manually cleaning up the most common export glitches.

Raster images in OpenOffice documents didn't get converted (the `\includegraphics` command got exported, but it pointed to a nonexistent file). Fortunately, the OpenOffice document was a ZIP file, which contained the images as PNG and JPEG files, which we could convert with *sam2p* to EPS and PDF. We didn't even try to export vector graphics, because authors sent such ugly figures that we decided to redraw them. We used Dia to redraw the figures, but we weren't satisfied with its formatting capabilities. It was a nightmare to change the visual appearance of the elements from the default.

None of the authors using OpenOffice supplied a structured bibliography, so we had to create the corresponding BibTeX source files by hand. The most tedious part of this task was to convert all URLs within the document to citations, and add fairly verbose entries to the bibliography database, looking up more information about the cited work on the Internet.

For EuroTeX 2006, most authors followed the guidelines and used the LaTeX document class we proposed, so no file format conversion was needed. Unfortunately, the final column width and font differed from those in the class we proposed earlier, so we got quite a number of overfull hboxes when recompiling articles. We also received articles in plain TeX (!) and ConTeXt, which we converted to LaTeX by hand, heavily using the search and replace functionality in our text editor.

For EuroTeX 2006 one of the authors sent a beautifully typeset article in PDF format, which we decided to include in the collection as is. Since the fonts and the column sizes were correct, we only had to add the running header and footer. We did this by importing the pages of the PDF file one-by-one as boxes with the *pdfpages* LaTeX package.

3.2 Article compilation

We prepared shell scripts for Unix which set environment variables, run *mktexlsr* in the local texmf tree, and build TeX formats with the necessary hyphenation patterns. This way it is easy to ensure that all editors work in the same environment. Should any difference arise (e.g. two editors have a different version of a LaTeX package installed, and they get different output), it can be resolved by adding the file to the local texmf tree.

It is important that all documents be compilable automatically. If an editor manages to compile a document, he should immediately write a shell script to perform the compilation. E.g. if LaTeX has to be run at most five times with a couple of BibTeX and *makeindex* runs in between, the shell script should contain the relevant commands in the proper order. It is not important to optimize for the number of LaTeX runs — a possibly badly compiled document is a lot worse than a slowly but correctly compiled one. For clarity, another shell script should be written that cleans up any temporary and output files. A Makefile can be used instead of shell scripts, but dependencies must not be indicated — a compilation should recompile the whole document from scratch. All scripts must share the same interface, so they can be called in a batch when the whole collection is recompiled, e.g. like this:

```
for DIR in *; do
  (cd "$DIR" && ./recompile.sh)
done
```

If the document contains raster images, they should be converted to both EPS and PDF, these files added to the repository, and the filename specified without extension in the parameter of `\includegraphics`. This way the document is compilable with both LaTeX and pdfLaTeX. We recommend *sam2p* for raster image conversion.

We decided that the recompilation of external graphics should not be part of the document compilation process. That is, when the document contains a figure drawn in Xfig, the Fig to EPS conversion isn't run when the document is compiled automatically. This gives us the advantage that even those editors can compile the document (and correct errors in the text) who don't have the appropriate graphics editors or converters installed. Asking them to install that extra software is not always feasible, because some graphics software needs specific operating systems or libraries.

All documents should be compiled to PDF with a fixed name (we used `compiled.pdf` for intermediate compilations and `final.pdf` for milestones). The reason why we are using PDF instead of PostScript is that PDF files are easier to manipulate (e.g. concatenate, add hooks) and they are also easier to preview in the web environment, so even visitors of the web view of our version control system can view milestones of typeset articles in PDF format. Using pdfLaTeX is recommended (because it can break a line in the middle of a hyperlink, and it has some nice typographic add-ons), but if the document compiles with LaTeX only, it should be converted to PostScript with *dvips*, and then converted to PDF.

Bitmap fonts must be avoided — a PDF with bitmap fonts looks ugly in Acrobat Reader, it is large and it renders slowly. Most TeX fonts are available in vectorised (usually Type 1) format today (either in distributions or from CTAN). For example, the Bluesky fonts are the Type 1 outlines of Computer Modern (CM), the base TeX font family. If the article uses the EC fonts, then the CM Super Type 1 outlines can be embedded to PDF, or the Latin Modern (LM) fonts can be used instead — but be aware of the slightly different metrics and character shapes (such as the letter "ő") between EC and LM.

Font installation and use can be cumbersome even if the font files are there in the proper folder of the local texmf tree. To avoid this problem, we used a custom Perl script *dff.pl* which wraps execution of all tools which embed fonts (currently *pdflatex*, *dvips* and *dvipdfm*) and provides the proper environment variables, command line arguments and font map files to these tools so that the right fonts will be found and used. The script also ensures that *pdflatex* and *dvips* use the same font map file.

The error and warning messages LaTeX emits are useful, because they identify possible problems in the article. We decided to abort automatic compilation when a LaTeX error is encountered, and thus force the editor to fix the error. We were quite permissive with warnings (including overfull box indications): we allowed compilation to continue, but wrote a Perl script which looks for warnings in the article log files, and we checked all these warnings after each milestone compilation. Finally we managed to get rid of all warnings. At some points we had to cheat, for example with long URLs in the bibliography it is quite hard to avoid the underfull hbox warning, so we just disabled this warning there by setting `\hbadness=10000`.

We used log analysis not only to find overfull boxes, but also badly embedded or missing fonts, and even articles accidentally omitted from the table of contents.

3.3 Editing

When reaching this point, the document is a valid LaTeX article, with all its graphics converted to embeddable formats; the source markup is cleaned up enough for humans to edit; and there is a shell script that recompiles the article to PDF from scratch.

Simple editing is a straightforward task, which we took advantage of in LME 2006: we had a lot of volunteers for proofreading, so we quickly set up a tutorial for them on using the version control system, told them which files to start editing, and they could start contributing their changes.

We used standard tools for proofreading and typesetting corrections: the output-to-source navigation feature of the DVI previewer, and the big black `\overfullrule` to spot overfull boxes. For pages with complicated graphics or transformations, we previewed the PDF file instead of the DVI file. Xpdf was our preferred choice for PDF previewing, because it doesn't have unnecessary GUI elements in its window, and it allows reloading the PDF file with a single keypress.

3.4 Concatenation

A collection is just a concatenation of the articles — except for the need for continuous page numbering to be maintained, a table of contents has to be generated, and there are some extra pages at the beginning and at the end.

We added the extra pages by introducing two special articles: `01Begin` and `99End`. The cover pages (two pages at the beginning and two others and the end) were part of these articles, but we had to strip these pages and send them separately to the printshop. Since we had to convert the document to PostScript anyway, the page range options to *pdftops* solved the problem.

We didn't generate the table of contents automatically; instead, we wrote a driver file which listed all the articles (with author, title and starting page number) in the order we wanted them to appear in the collection, and we typeset the driver file during the compilation of `01Begin`.

Automatic recompilation of the collection can work only if individual articles are already compiled automatically. We wrote a shell script which recompiled the whole collection. It also took care of propagating page numbers between articles. After each article compilation it counted the number of pages, modified the starting page number of the next article in both the driver file and in a helper file which would be `\input` by the document class. It took care of inserting an empty page so that each article began on an odd page. At the end it recompiled the two special articles in order to get the table of contents right. (No further compilation was necessary since we designed the TOC in such a way that the number of pages it occupied was constant.)

First we tried concatenating the PDFs using the *pdfconcat* tool. Unfortunately it doesn't support PDF outlines (i.e. the table of contents tree) properly, so we switched to Ghostscript. Although Ghostscript 8.5x has support for concatenating outlines, the support had other glitches which prevented it from working with PDFs generated by TeX. We prepared a small fix for that (which modified some of the PDF-writing operators such as *linkdest*), which also made hyperlinks work within the article. We also needed hyperlinks from the TOC to the article, but this was easy because we could use page-based links instead of symbolic ones, since we already knew the starting page number of the article.

We also tried the *pdfpages* LaTeX package for concatenation, but this package didn't support outlines or hyperlinks in source PDF files.

It was a key design principle in our workflow to have automatic compilation. Since the work of the editors is judged based on the quality of the final output (both in print and the web), and humans tend to make mistakes (especially if they try to rush when the deadline is approaching), we wanted to have a document compilation policy which allows as few mistakes as possible in the final compilation. The more special cases the editors have to remember, and the more steps they do manually, the more mistakes they make. Automatic compilation minimizes these mistakes. It also increases the reliability of the editing process since if the computer of the chief editor gets broken during the editing, he can check out all the articles from the repository, and recompile the whole collection on any other computer with a single command. Unix shell scripts, Perl scripts and Makefiles helped us a lot for automating the compilation process.

3.5 Preparing for print

[3] gives a good technical introduction to the problems editors face when sending the work to the printshop, and it also gives several solutions for each problem (with both free and proprietary tools).

Printshops usually expect the text as colour-separated PostScript files. The cover pages have to be sent separately. The *psselect* tool can be used to select and reorder pages from a PostScript document, and options can be specified for *pdftops* to emit only a certain page range when converting from PDF to PostScript.

For high quality colour output one can use spot colours with the *xcolor* LaTeX package. As a simple alternative solution, one can create a PostScript document with colour, and later separate it. Separation means creating four copies of each PostScript page, each of these being grayscale, and the brightness values are used as C, M, Y and K components in the CMYK colour space. Aurora [2] is an old but working free tool which can do this conversion in pure PostScript. Using Aurora one processes the PostScript (or PDF) document four times, with settings for the individual component. Aurora wraps the *setgray*, *setrgbcolor*, etc. PostScript operators so that they will activate only one component of the specified colour. It also modifies the *image* and *colorimage* operators that draw raster images, but unfortunately it doesn't understand the image dictionary syntax introduced in PostScript Language Level 2. To overcome this, we implemented it in PostScript code which we load right after Aurora. Our code converts a PostScript image dictionary to a non-dictionary call of *image* or *colorimage*, and it also decodes indexed images manually.

The solution is quite slow (partly because of Aurora and partly because of our code); it processes a page with a colour image in about 10 seconds — but at least it is correct, because it hooks all affected operators at the proper place. Fortunately, we experience the slowdown only for raster images — colour text and vector graphics are rendered as quickly as without separation.

To make the job of the printshop easier, we prepared a script which separates the pages of a PostScript file to grayscale and non-grayscale. We took care of colour raster images manually, and we autodetected non-grayscale colours everywhere by looking at the colour-changing operators in the output of *pdftops*. Since this PostScript output has a quite simple syntax, we could find colour changes using regular expressions. Once the non-grayscale pages were found, we selected them with *psselect*, and renumbered the pages (back to the original) with a Perl script.

Printshops expect crop marks on each page. The *crop* LaTeX package can generate those marks. A few test pages should be sent to the printshop in advance so they can confirm that they get the crop marks where they expect them. In our case study projects we didn't use the *crop* package because it was more convenient for us to add the simple crop marks with a Perl script to the PostScript output of *pdftops*. The script also took care of enlarging the paper size. This way we could use our DVI and PDF previewers without having to see the enlarged page with the crop marks, and marks were added only to the PostScript file sent to the printshop.

3.6 Publishing on the web

We didn't want to have an HTML version of the articles, because converting LaTeX markup to high quality HTML is a difficult and time-consuming task which is hard to automate unless HTML export was in mind from the very beginning. We provided a HTML page with all the articles (with author, title, abstract and citations as BibTeX source) and a PDF file for each article. We also provided a big, concatenated PDF.

PDF for the web differs from the printed document in:

- PDF for the web has outlines (structured table of contents) and in-document hyperlinks. All of these can be generated with the *hyperref* LaTeX package.
- PDF for the web has different margins, usually equal inner and outer side margins.
- PDF for the web doesn't contain crop marks.

- PDF for the web should contain scalable Type 1 fonts, because Acrobat Reader renders these fonts faster and nicer than bitmap fonts. Scalable fonts also reduce the file size.
- The size of PDF files for the web does matter. Raster images should be small. Fonts should be subsetted. Concatenated PDFs shouldn't contain the same font twice.
- PDF for the web can contain more pages with colour.

The first thing we did was add a "compilation mode" parameter to the compilation scripts. The document class also received this parameter, so it could generate slightly different output based on the mode (e.g. it could decide whether to load the *hyperref* package or not). With compilation modes we could also control if we need the overfull box indicator.

Ghostscript was smart enough to create a concatenated PDF with all fonts subsetted, except that pdfLaTeX had already subsetted fonts in the individual articles. So we turned font subsetting off in pdfLaTeX (changing all `<` signs to `<<` in the font map file). This increased the size of intermediate PDFs substantially, but the final PDF became small.

We also wanted to have all fonts, including the base 14 fonts (like `/Times-Roman`) embedded, since we otherwise experienced accent positioning problems (e.g. with letter "ő"), since PDF viewers use different glyphs in standard fonts. To achieve this, we had to call Ghostscript with these parameters:

```
-dCompatibilityLevel=1.3
-dPDFSETTINGS=/prepress
-dEmbedAllFonts=true.
```

The sizes of raster images emitted by *sam2p* were small enough, but unfortunately Ghostscript insisted on recompressing the images (usually with suboptimal parameters). We solved this by writing the Perl script *pdfdelimg.pl* which extracted images from a PDF, and replaced them with dummy images. We run Ghostscript on these replaced PDFs, and we used *pdfdelimg.pl* again to replace images back in Ghostscript's PDF output. Our script distinguished dummy images by their dimensions.

In all other respects, Ghostscript produced small PDF output.

4 Conclusion

High quality text and math output is the most common reason why people like TeX. Editors also appreciate the freedom they have when they design their workflow. They have several tools to choose from (many version control systems, many TeX engines, many printer drivers, many converters), and they can customize the tools. Having the source of the document in text files makes it possible to use a version control system for parallel file editing. Since there are multiple stages of compilation, there are multiple ways to hook in changes. Scripts can be written to automate compilation and generate both the printable and the web version from the same sources, with a single command. As far as we know, this set of features is unique to the TeX editing workflow.

It is up to the chief editor precisely how to design the workflow and to what extent document compilation is automated. We tend to use a lot of custom scripts in our workflow, because we found that using scripts pays off in speed, quality of output and reliability, even when the script is run only once or twice; and we can also reuse our scripts in future projects. We admit that designing and setting up a good workflow needs quite a lot of software experience: the chief editor has to understand not only TeX-, font- and PDF-related file formats and tools, but also version control systems (on both client and server side), web application installation, web page editing, mailing list management, and script programming. We believe that it is worth learning these and to improve the workflow gradually.

Communication between the editors is also important. The version control system ensures that editors have the relevant versions of all files they need, and also that they can make corrections to any file they want to. The mailing list and other communication channels can be used to distribute and synchronise work.

This article has presented some tools and techniques which can make collection preparation more productive and less painful. Since TeX and its related tools are free software, there is a good chance that editors can find even better tools for their needs on the net. As tools and techniques continue to improve, working with TeX becomes even more fun.

References

[1] Ben Collins-Sussman, Brian W. Fitzpatrick, and Michael C. Pilato. *Version Control with Subversion*. O'Reilly, June 22 2004. `http://svnbook.red-bean.com/nightly/en/`.

[2] T. Graham Freeman. Aurora: Colour separation with PostScript devices. Technical report, Australian Defence Force Academy, July 1994. `http://www.ctan.org/tex-archive/support/aurora/aurora.pdf`.

[3] Siep Kroonenberg. TeX and prepress. *TUGboat*, 25(2), 2004. `http://www.tug.org/TUGboat/Articles/tb25-2/tb81kroonenberg.pdf`.