The description language chosen for FDNTEX

Benjamin Bayart
10, rue du Croissant
75 002 Paris
France
bayartb@edgard.fdn.fr

Abstract

In this paper I will introduce the package description language that has been chosen to be used in FDNTEX. In short, this language is XML with a dedicated DTD. First, I'll introduce FDNTEX, to obtain a good representation of its requirements. Then, I'll introduce the data that have to be contained in the description files. Finally, I'll introduce briefly the DTD itself.

This paper has to be understood for what it is: first thoughts on how to obtain the right language. What has to be expressed by this language is something definite, but the DTD itself is in an early stage development at the time of writing. Thus, if the goals described in the firsts sections and the DTD described later are in conflict, consider the DTD is wrong.

Vocabulary

The word "package" will be used with several meanings in this paper, this can be troublesome in certain places. A *package* can be, depending on the context:

- a LATEX style file, like `array.sty`;
- a part of the distribution, *e.g.*, the package containing TEX itself, or the one containing Ω;
- a binary version, in a given flavour, of the previous one, *e.g.*, the `rpm` file containing Ω, or TEX.

It's sometimes hard to distinguish between the three meanings since they often represent three stages in the life of the same object: `array` which is a package in the first meaning, as any LATEX user knows, is also a package according to the second meaning, since FDNTEX has a description of it, with dependencies, methods to build and install it, and so on; and will also be a package in the third meaning since the `rpm` flavour of the distribution will contain an `rpm` package called `array`.

FDNTEX is designed to exist under several "versions", *e.g.*, one for FreeBSD, one for Linux/RPM, one for Linux/Debian, one for HP-UX 9, and so on. Those versions will be named "flavours" in this paper, to avoid confusing the reader between "versions" of FDNTEX and versions of the packages that are part of it.

According to the context, "I" will refer either to the author, or to an hypothetical user's thoughts.

Introduction

FDNTEX is a new distribution of TEX, based upon different ideas from the previous ones.

Before teTEX appeared, a "TEX distribution" was, de facto, a pure distribution of "TEX, the program" and the strictly required files to build it and make it work in a standalone way. Any other tools, like fonts or formats or extensions, had to be installed by hand.

Since teTEX appeared, we have lived in a more user-friendly world: one can install the whole thing and obtain a rather complete TEX-based system including LATEX and lots of useful extensions.

But going on straight in the same way will lead us to a really heavy system, providing any available font, say Japanese fonts, to any user, even Russian-speaking ones. Thus distributing a hundreds-megabytes system, 85% of it being useless for each given end-user.

To avoid this problem, two ways can be studied: restrict distributions to a good subset of what is possible, and hope users will be successful in installing the missing parts by themselves; or use a different approach of the distribution problem. FDNTEX is an attempt to fulfill the second solution.

The basic ideas underlying FDNTEX design are briefly described as follow:

- Fully modular, and fine grained. *I don't want to use* `patgen`*, thus I don't install it.*
- Easy to upgrade a part without reinstalling the whole thing. *I upgrade my LATEX kernel every*

six months or so, and I don't want to change my Computer Modern fonts that often.

- Easy to install on the target system. *If I use a distribution instead of the root-source, it's just because I don't want to install a useless C compiler.*

The original idea was even simpler. We use TEX, in the leading team of FDN[1], for administrative purposes, and thus we all need to have it installed on our computers in a satisfactory way. Most of us are not good at using TEX outside of this restricted use (in fact, I'm the only one in the team who knows about the internals of TEX). teTEX didn't contain the packages we needed. Thus I started developing a TEX distribution that could satisfy those requirements. And this is also why this distribution is called "FDNTEX", originally, it was a TEX distribution to be used by FDN.

Let's have a look at some instructive examples to have a more precise view of what is standing behind those three simple ideas.

Fully modular By stating that "FDNTEX is to be modular", several problems are addressed.

The first problem has already been discussed: if I don't need `patgen`, since I don't want to generate hyphenation patterns for a new language, then I don't want it on my system. Simmilarly, if I don't use PostScript fonts at all, I don't want to have their metrics and the related software. It's useless, will slow down my system, and will obstruct the use of TEX on an old computer with a small disk.

The second problem is harder to understand. As I'm a French native, Ω is of some help to me. If I decide to use *only* Ω and Ω-based formats, then I don't want to install TEX itself, but Ω instead. It means that the minimal subset needed to start FDNTEX needn't contain TEX itself.

The third problem is that I want to be allowed to install only the minimal subset of the whole TEXware required to build the book I'm writing. That means that I don't want to install large things like "all the PS metrics" if I don't use PostScript fonts at all in my book. And, even more, if I only use Times for some titles, I don't want to install something too large on the poor old laptop that I have to use to write this book. Thus, something as large as "all the PS metrics" will not be a valid package for FDNTEX. It will have to be split into several parts, probably one per font.

All that leads us to a system with hundreds of packages. Just by splitting down the `web2c` bundle into distinct software units leads to several dozen packages. Each of the hundreds of extensions of LATEX is also an autonomous package, at least, and sometimes several, for large parts. This makes hundreds, or perhaps thousands, of packages as part of the final distribution.

[1] A non profit organization, which is an Internet access provider.

One cannot afford to know all of them in enough details to be able to choose. Thus, there must be a reliable description of the dependencies between packages (who will ever remember that `tabularx` requires `array` to be installed, or that `concmath` uses `url`?), and there must be a way to choose a reasonable subset for a given use. *E.g.*, if I want to use TEX to typeset a paper about electronics, I need a way to say "everything related to electronics is of interest to me", and a second way to give more precise instructions later to add or remove packages by hand.

Easy to upgrade This point is easier to understand, and easy to automate. The only really hard thing is to handle strange cases.

Let's have a look at a hypothetical example. Let's say Mr. X wrote a few packages, `a`, `b` and `c` which are really small ones and thus are distributed as a single one. As those packages are small, and distributed as a single thing, they are in the same bundle in FDNTEX. But, a few months later, our good old Mr. X has worked a lot, and his packages have gained hundreds of features, and are now really large ones, each being composed of dozens of small independent parts. We would then need to re-bundle them separately. Thus, how to explain to the system that upgrading from the first `a&b&c` bundle means installing the three separate bundles, and that from this point each part can be upgraded separately?

Of course the symmetrical example is also troublesome. If two separate things are now unified in a single bundle, how can we explain this to the system?

Even more, one can mix the problems: three elements are replaced by two in the new version. How to upgrade easily?

The 18 months spent working on FDNTEX have not yet led to any general solution to this ugly problem.

Easy to install on the target This point is an interesting one since it's one of those which led to lots of discussions about the design of FDNTEX.

The main idea can be explained so: as I run a RedHat system, I want each FDNTEX package to be an `rpm` file, so that I can install it easily, using my usual system tools.

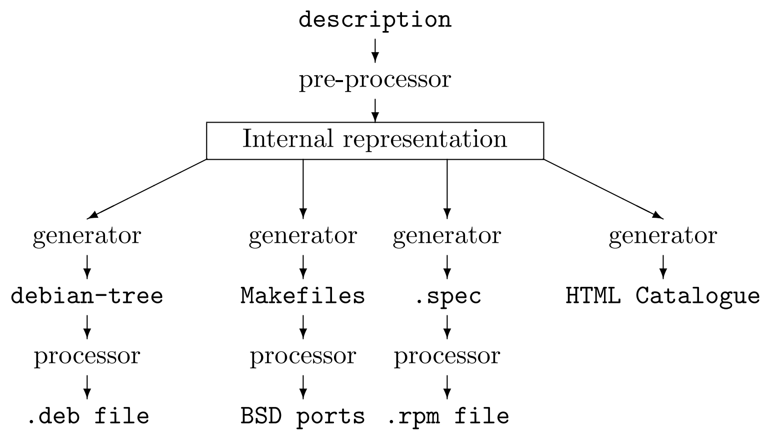

Figure 1: In this diagram, each teletype text identifies a physical representation of the information, and each roman text identifies a process to go from one representation to the other. The "Internal representation" is different since it only exists in the memory of a program, and not in a real file.

This means creating several flavours of the distribution, something like one per target system, and sometimes more, if several flavours of a system exist (as for Linux where Debian-based systems have almost no common point with RedHat-based ones).

Another way would be to develop a really standalone system, that would provide everything from the description language to the final binary format, including the building tools and all the other things needed. Such a system would be really easier for me (read "the distribution maker" here, instead of just "me") but would lead to something of less interest for most people. On the other hand, it would permit development of a reasonably complete distribution in a short period of time. The prototype was developed in only 6 months. And it would avoid developing all the general system that we are discussing here.

The ideas developed here are meaningful as long as there is a team working on the distribution, with people dedicated to each generator, but are meaningless if this is not the case, at least because I will never have all the variety of operating systems, and all the knowledge that is required to develop all of the flavours.

Several technical problems arise from this intention to be close to the target system; we will expose them later on, in the section "Developers' tools".

Design, implementation, tools

In this section, we will discuss the way FDNTEX is to be developed, and what will be the development tools. That is to say, not what the resulting binary packages should be, nor which packages will be part of the distribution, but *how* packages will be created for any flavour of the distribution.

The system design The prototype of FDNTEX[2] was written directly in an `rpm` representation (technically, it is a large bundle of files in `.spec` format, which is that used by `rpm`), and from that automatically translated into the right bundle of files for FreeBSD ports. This approach is of course wrong, but at least has shown that writing such a distribution is feasible.

The right choice is, of course, as shown on figure 1, to have a complete and precise description of the package and to consider its representation in a given system (say `rpm`, `swtools`, or things like that) as a projection. Thus this description has to be a superset of what can be expressed in the various target languages.

Developers' tools The tools used to create the distribution itself are clearly shown in figure 1. The first one is the pre-processor (mainly a parser) that understands the original description of the distribution. This one is, in fact, strongly linked to the language used to describe the packages.

Discussions about TPM[3] led to the conclusion that writing a brand new language from scratch seems to be a bad choice, and that it is better to use XML as the representation of the knowledge

[2] This prototype can be downloaded from `ftp://ftp.lip6.fr/pub/TeX/FDNTeX/Prototype`. It's rather old but good enough to give a more precise idea of the goal to reach.

[3] The TEX Package Manager, an idea proposed by Simon Cozens, Sebastian Rahtz and Fabrice Popineau, grew independently of FDNTEX but leads to a really similar system. It was widely discussed, first on `comp.text.tex`, and then on a separate mailing list.

database underlying the distribution. Thus, the pre-processor in figure 1 is only a suitable XML-parser with some knowledge of the system.

The most interesting points of XML for this purpose is that good XML-parsers exist on most systems, written in various languages, so that one can use a Perl-based system on Unix systems, or an anything-else-based one on MacIntoshes, without troubles related to the description of the distribution.

The various processors are, of course, based on the tools dedicated to the various flavours of the distribution, but not only that. They will include tools to automatically manage the rebuilding, so each one might look like a `Makefile` which uses the system dedicated tools.

The most complicated part is the generators, since they know about the data in the description of the package, and are able to create the projection for a given flavour. This part is hard to achieve, due to the large number of flavours.

Thus, the evidence seems to indicate that the only way to obtain good results is to have people with good knowledge of the target systems writing those generators.

Users' tools As stated before in the introduction, there will be some needs for users' tools, some being provided by the target system, like the ones to install or upgrade a package, others not, like the ones that manage configuration of the whole thing. Those tools will probably be re-used from previous distributions, like the ones used in teTEX or TEXlive.

More information The main aim of this paper is not to describe the internals of FDNTEX in a full extent, but to expose how the description language used by it was designed, and to discuss the points that have led to this design. Thus, if one wishes to have more information about other points related to FDNTEX, the reference documents, like the *FDNTEX manifest* or the current version of the DTD, can be retrieved from `ftp` sites like `ftp://ftp.lip6.fr/pub/TeX/FDNTeX`

At the time of writing, the *FDNTEX manifest* is no longer up-to-date but will probably be updated before you read it.

Describing a package

The description of a package should contain several parts.

First, its full identification, providing the version number, information about its author, a short description in various languages whenever possible, and other things of the like. All this information can be taken from the well-known Catalogue[4].

Second, information to locate it, like where it is located, in source form, on CTAN sites, or any other source-location for non-CTAN packages. This is different from the list of source files, and is to be used, *e.g.*, to show the packages in a tree for download. Other kinds of locations are of interest. A good one will be to place the package in a tree based on the functionality it provides instead of its location on CTAN, in such a tree, algebra would be a subset of mathematics, and any package related to typesetting algebra would be in this subtree, ignoring whether it's a package for plain TEX or for LATEX.

Third, its content, *i.e.*, the list of files contained in the package, and their position in a TDS conforming structure.

Fourth, information on how to build it, that is, going from the source available on the Internet to a representation that can be directly used.

Fifth, information on how it's linked to other packages, that is, dependencies.

Sixth, information on how to manage it on the target system (procedures for installation, deinstallation, upgrade, and so on).

Contents of the package The only technical choice is to decide if it only lists files relative to a supposedly well-known root, or if it lists them in a more parametric way. For example, in the first case, the font metric of `cmr10` can be listed as `texmf/fonts/tfm/public/cm/cmr10.tfm`, while in the parametric form it might be expressed as `%TEXTFMS%/%MYDIR%/cmr10.tfm` so that it can be automatically adjusted on the fly to a given target.

Since this kind of re-mapping of trees seems easy to obtain from real paths, the first choice has been retained. If one needs to re-map a TDS conforming tree to another one, it will be done in the "generator" rather than in the "pre-processor".

Building a package During the discussions about this description language, two different ways of describing the building have been studied.

The first one, the easier one from the developer point of view, and the more powerful, is to use a powerful language like a scripting one (Bourne Shell would be a perfect candidate) and to write the script in a well parameterized form, so that it can be used on a large variety of systems. This is simple, since the tools just have to use the script as is, and powerful, since one can express in this language exactly

[4] The Catalogue can be retrieved from any CTAN site or mirror at `CTAN:/help/Catalogue` and is written with XML, which will help re-using its contents.

all of what is feasible on a real system. But a burden is created for the person who writes the description of the package, and it creates a lack of portability: Bourne Shell would be of no interest on non-Unix systems. And moreover, there are large differences between different flavours of this language.

The second one, harder to achieve from the developer's point of view, and less powerful, is to express it in a high-level language. To build the `tools` bundle, it can be something like:

```
LATEX tools.ins
DTXTODVI afterpage IND GLO
DTXTODVI array IND GLO
...
```

stating that the `.ins` file to be processed to build the packages is `tools.ins`, and that each item of documentation has to be compiled twice, then has a run of `makeindex` to produce the index, then another one to produce the history of changes (a kind of glossary, technically), and another last run through LaTeX.

The second way is far better for people who have to describe a package, and is easily translated on any system by the "generator". But if one wants to have a complete enough language to handle any and every case, one will create a language as complex as a traditional scripting one. Such a complex high-level language would then be useless, since it would be too hard to understand.

The good choice is, then, to have both. If the high level language can express what is needed to build the package for most flavours, then just use it, and, if a given flavour needs to express it in a more dedicated form, then just override the first high-level description.

Moreover, if a given package is too hard to express its building in the high level language, then just use the low-level one, and discard the generic description of the building.

This model sounds good, since 95% of the packages will be described in the easy way, and only the the most problematic 5% will be described in the hard one. Thus porting the distribution to a new system will be: 1) providing the right generator, 2) porting those 5%.

Some other information will be of use to build the package, like a knowledge of what is required to be installed before starting to build anything.

An example can be "one cannot build `web2c` without `make` and a C compiler", but this is an easy one, and external[5] so that we can avoid saying it.

Another strange example is that, to build Ω one needs LaTeX, which is strange since LaTeX can be built on top of Ω. In fact, LaTeX is required only during the *building* stage, to produce the documentation in a suitable `dvi` form; but LaTeX is not required to *install* Ω on a target system.

Thus, in the dependencies section of the description, one will have to pay attention to the building stage. Of course, conflicts can arise (*e.g.*, one cannot build the documentation if a given flavour of a package is installed).

Another point is that a single "building" can produce several packages, *e.g.*, compiling the `web2c` bundle produces dozens of packages. Thus, there must be stated in some way what building is required for which package.

Links between packages As we have already seen, several kinds of links can exist between packages. The next few sections will list them, and will examine the level of complexity required.

Installation dependency This kind of link is the most evident one: listing in a package description all the packages that need to be installed for this one to work correctly.

A first hard point is to determine this list, by reading the package documentation, by reading its source, by examining its behaviour closely, and so on. It is easy to state these dependencies in the description file, even if the information is hard to obtain. Just stating that `tabularx` requires `array` is easy.

The second hard point, and the really hard one, is that not all dependencies are that easy to give. Let's study two interesting examples.

The first example is the "functionality" one.

It seems clear that `array` has no meaning if LaTeX is not installed, and that LaTeX has no meaning without TeX. Now, consider that `array` *is* of interest if Ω and Λ are installed[6]. Such a case can be solved in two ways — either put the burden on the side of `array`, by stating "this requires LaTeX or Λ to be installed" (which places burdens on lots of package descriptions), or put the burden on LaTeX and Λ to state "this provides a LaTeX-like format" (so that the `array` description may say "this requires a LaTeX-like format").

In fact, the second alternative requires that we have a list of defined *functionalities* that packages

[5] Indeed, the requirements expressed here are not in the scope of this distribution. We will summarise those requirements as "one needs a complete and working operating system to build/install FDNTeX".

[6] Ω is an evolution of TeX adding features useful for internationalisation, like Unicode; and Λ is the name of LaTeX 2ε when built on top of Ω.

may provide (or require), and to be sure to express functional dependencies only according to this list, if there is a way to do so.

On targets that already use this kind of thing, it's easy to realise the projection of this information by direct translation. On other targets, it will be a little harder — we will need to express "A or B or C or…" by listing all the alternatives for a given functionality. Fortunately the job can be done by the generator, since it has a knowledge of all the packages before it starts to generate anything.

The second example is "soft" dependencies.

Imagine a document class that has an option `psfonts` that can be called to adapt the layout to PostScript fonts instead of traditional Computer Modern ones, and another option `concrete` that uses beton instead of Computer Modern. Does this class depend on Computer Modern, beton, and PostScript fonts?

The conundrum is, that the class really depends only on Computer Modern *or* beton *or* PostScript, yet if only one of these sets of fonts is installed, the class is not fully usable. Dependency checking should not report "everything good" if in fact a part of the system cannot be used for lack of another.

We can describe this situation by introducing the concept of *soft dependencies* — that is "this package prefers this other one to be installed, but can be used without it, at your own risk".

It seems useful to define several levels of softness for this situation, *e.g.*, strong if the default behaviour, or the one the most used, needs the dependency, and weak if the dependency is needed for a weird use of the package, or by an option of little interest.

If the softness is expressed on a range from 0 to 10, 0 representing "not required at all" and 10 "strictly required" (like `array` for `tabularx`), the system could be controlled by specifying a single value n, to express "install all dependencies higher than n". The system may set a default value $n = 3$, so that the end-user can care only about the points he wants to, or drop to $n = 1$, if he doesn't want to bother at all about choosing, or raise n to 10 if he wants a *really minimal* system.

Same source bundle Two packages produced from the same source bundle, such as `patgen` and `gftopk` which are both produced from `web2c`, do have a link between them; this link has to appear clearly in the description.

One way to describe the situation is to use a purely object oriented representation. One has a source-bundle `web2c`, a building method `build-web2c` and a package `patgen`, then one states that the method `build-web2c` has to be applied to the source `web2c` in order to produce the package `patgen`[7].

A second way is to consider that the real object is the source-bundle and that the packages that are created from it are pieces of information related to it, and only to it, that is, in a structural way, the package description is a part of its parent source-bundle description.

The first technique sounds really powerful, but will quickly become hard to handle, and will create a heavy burden for a lot of people, only to handle extremely rare cases (no such cases have arisen in the 472 packages in the Prototype that would require such a complex system, even if some extremely rare ones have been met in the real world).

Thus, currently, the second technique is used.

Stage dependencies A case of "stage dependency" that was considered in the previous section is the "installation dependency". Installation dependencies are complex, and are probably the only ones that need the idea of soft dependencies.

Some other cases have to be handled.

One, also considered earlier, is the "building dependency", which states that one cannot build a given package unless another is installed. We can insist that building dependencies should all be strict, though one can find funny cases when soft dependency might be useful (the best one is METAFONT which requires X11, but can be built without windowing support). Forcing strict dependencies for the building stage is a strategic choice: FDNTEX has to behave everywhere in the same way, independently of the system on which it is built. Thus if one wants to rebuild the METAFONT binary package, X11 libraries *have to* be installed first since it have been decided that METAFONT has windowing support in FDNTEX.

Another interesting case would be to list documentation dependencies in a separate list, so that the system could ask the user "`XXX` is required in order to read the documentation of the package you're installing, do you want to install it?". This is of importance, since on a minimal system, the user can decide not to install a large set of fonts that is only required by the documentation of a tiny package.

[7] The way it's expressed is of no importance here, it can be `patgen` that states how it wants to be built, or the building method which states what it's able to build, but both methods are equivalent, given that when a generator runs, all the descriptions of all the packages have been loaded to permit consistency checking.

Conflicts The last kind of relationship between two packages is *conflict*: that is, a package cannot be installed if another one is already. The most evident case is the `tree` LATEX-package, since three different implementations exist, have the same name (`tree.sty`), have different syntaxes and different behaviours, and installing more than one would lead to non-predictable results (one cannot say which one would be called during the building of a document). Those packages clearly conflict. (The difficulty is not in finding packages that conflict, that's rather easy, but in describing the conflict.)

The technique used in other similar languages is to have each package state which others would conflict, *e.g.*, if `a` and `b` are in conflict, then `a` states in its description "There is a conflict with `b`", and vice versa. There is a serious drawback to this technique: if a third package (say `c`) arises that conflicts with the two previous ones (this is the most probable case, if it conflicts with one, there are great probabilities it will also conflict with the other), then it has to state "There is a conflict with `a` and with `b`", and then the descriptions of `a` and `b` have to be corrected to indicate this new conflict. This is clearly troublesome, not least because `a` and `b` have to be rebuilt and to change their version numbers while their descriptions otherwise remain the same.

Another technique would be to have a separate list of mutual exclusions that is the only document that has to be corrected to handle those cases, given that the generator will easily translate the information provided by this list into the form described in the previous paragraph.

At the time of writing, the traditional way is used in FDNTEX; but we plan to switch to the other technique when a robust solution to the release number automatic increment is found.

Automatic behaviour from dependency and conflict information As already discussed, we aim to have tools, driving the system processor, that create the final package if the system is not able to do the job fully automatically.

The easy part is to use the building dependencies to automate this process. Whenever the automated system wants to build a given package, it first checks that all the building dependencies are satisfied.

The hard part may be explained with an example. Let's say that a package `a` needs the first flavour of `tree` to build its documentation, and that `b` needs the second flavour. When attempting to build `a`, the system will build and install the first flavour of `tree` if it's not already present. But, when the system tries to build `b`, it will notice the lack of the second flavour of `tree` and try to install it, which will fail due to the conflict.

The right behaviour would be to remove the conflicting flavour, then to install the required one, taking care of the dependencies while doing it (that is, remove everything that requires the conflicting flavour).

This is not linked to the way the information is provided, but to the way it is used.

Management information The management information is that which needs to be bundled with each package to allow good management of the whole system in a consistent way (like checking the dependencies, configuring the various elements, allowing one part to use another if both are installed on the target, and so on); and to allow management of the package itself (when installing, uninstalling, upgrading, and so on).

Most packages have minimal and recurrent requirements of management: rebuilding the `ls-R` database; handling the configuration files (a dozen or so cover a large majority of the packages); adding the right symbolic links at the right place; rebuilding the formats; and so on. Building stage information will be given in a generic fashion, using a high-level language, but can be given in fully user-controlled fashion too, if required.

Optimizations can almost certainly be described here too, like the fact that, if one installs 6 packages, it will probably be enough to rebuild the `ls-R` database only once, after the last package. This optimization is important, since repetitive rebuilding of the database is time consuming, and can be handled easily: the rebuilding is delayed until either the end of the installations, or an instruction that an up-to-date database is needed immediately. Then, the rebuilding will take place only when strictly needed and at the end of all the installations.

Some subtle cases can arise, because the actions to be accomplished are complex to describe. For example, in the first stages of an upgrade, the actions relate to the previous version of the package, and thus should be provided by it, and during the final stages, the actions relate to the new version. Describing the upgrade of a package requires description in each version how to install and how to un-install it for an upgrade. (These actions might be slightly different from a normal (un)installation procedure.)

Further subtleties can arise when packages evolve strangely, as in the previous case of 3 packages that are replaced by 2. We must consider how packages can describe an evolution that has not been planned;

moreover, the description of the last stages may have to care about the previously installed versions of the packages to decide how to handle the whole operation smoothly.

A good model to be reused, at least for the functionalities it provides, is that used by Debian since it takes into account all the subtleties outlined above. Please refer to the "Debian packaging manual" to have a more precise idea of what can be of interest.

A DTD to store this information

We will now introduce the first draft of the DTD that will be used by FDNTEX to store the descriptions of the packages. It's not yet, at the time of writing, used to produce any package. It will evolve quickly to a first release version, used to produce the first flavours, and then, probably, evolve again to a more mature version when new flavours appear.

Of course, any comments, improvements or suggestions are welcome, as far as they improve the DTD and make it closer to the description given in the previous sections, as this description is more mature than the DTD itself.

General structure A full document, validated according to this DTD, is a series of `Author`, `License` and `BPackage`. The description of the whole distribution can be seen, for convenience, as a single very large document, since one will need the descriptions of all the packages to generate the description in a given flavour (it's required for some flavours, like the FreeBSD one, and harmless for others, like the RPM one). Physically, there will more probably be one external document for each entity in the document: one per `Author`, per `License` and per `BPackage`, and maybe even one per `Package` (an internal element of `BPackage`).

`Author` and `License` are top-level objects just because there are relatively few of them, and there is a need for consistency, thus instead of "describing" dozens of times who is David P. Carlisle, it seems more efficient to describe him in a unique entity, and then give a reference wherever it's needed, in the description of all the packages he has written. Similarly for licenses: since there are only a dozen or so of them (according to the Catalogue), it seems useless to describe them separately for every package.

Thus, if considered as a single document, the description of the whole distribution might look like figure 2.

If we want to consider it as more modular, then we have to choose a method to aggregate all the information. One approach is to believe in XML even more strongly and use it to aggregate the whole document, using entities; this would lead us to a document which looks like the one in figure 3. An advantage is that the XML-parser can perform some consistency checking, such as one on the ids of the objects (*e.g.*, check that `Author` has been defined when referred to in a package description). The disadvantage is that the aggregating document will change each time a new package or author or license is added to the system.

```
<?xml version="1.0" standalone='yes'?>
<!DOCTYPE doc SYSTEM "./fdntex.dtd">

<doc>
 <Author Id="Carlisle">
  <Name>David P. Carlisle</Name>
  <EMail>david@latex-project.org</EMail>
 </Author>

 <License Id="LPPL">
  LaTeX Project Public License
  ...
 </License>

 <BPackage>
  <BPIdentification>
   ...
  </BPIdentification>
  ...
 </BPackage>
 ...
</doc>
```

Figure 2: The whole description seen as a single file: this is the way it will be seen by the parser and by the various "generators", not the way it has to be written.

Another approach is to use a full standalone document for each part, each having its own DTD, and then to rely on the top-system to parse all the needed documents in the right order. In such a case, when the system needs a reference to the `Author`-id "Carlisle" it looks for a file named `./authors/carlisle.xml`, parses it, and then performs the consistency checking. This does not use the XML-based mechanism, and needs no top-file to handle the list of all other files, but it requires that we develop another parser on top of the XML one, and also requires more work when porting the distribution to a new target.

The third solution, while being more complex, seems the most interesting and will probably be used in the final system. In the DTD presented here, only

```
<?xml version="1.0" standalone='yes'?>
<!DOCTYPE doc SYSTEM "./fdntex.dtd" [
 <!ENTITY carlisle
       SYSTEM "./authors/carlisle.xml">
 <!ENTITY bayart
       SYSTEM "./authors/bayart.xml">
 ...
 <!ENTITY lppl
       SYSTEM "./licenses/lppl.xml">
 ...
 <!ENTITY tools
       SYSTEM "./packages/tools.xml">
 ...
] >

<doc>
 &carlisle;
 &bayart;
 ...
 &lppl;
 ...
 &tools;
 ...
</doc>
```

Figure 3: A multi-part document based upon entities: this allows the XML-parser to check the ids and id-references in the whole system.

the first two alternatives (which are equivalent for the XML-parser) are available.

Authors and licenses Top-level objects `Author` and `License` are both identified by a unique id; the *id* system provided by XML is used for this purpose.

An `Author` refers to a physical person, or to a well-defined group. It might be the author of a package, or of a package description, depending on when it's referred to, or even of both. Up to now, this contains little information, the aim being not to collect personal data about people, but merely to cite them to allow anybody to contact them in case of troubles like bug reports, or license problems.

Currently, the three basic elements used here, namely `Name`, `EMail` and `License` are, in the XML typing system 'ANY', that is any non-parsed text. In the near future it will become a bit more structured with markups for the license text, and a more formal way to give an e-mail address. An example of the current structure can be seen in figure 2. The `EMail` markup in the `Author` entity is optional and might be repeated if needed.

About BPackage A `BPackage` is a bundle of packages (or big-package) which are all built from the same sources, as TEX and METAFONT are both built from the same `web2c` source-tree. `web2c` will then likely be a `BPackage`, producing several packages, including `tex` for TEX and `metafont` for METAFONT.

A `BPackage` contains a `BPIdentification` tag that identifies it in the distribution, a `SourceBundle` to help retrieve the sources, a `Building` section saying how to build the packages from it, and a non-empty list of `Packages`.

The `BPIdentification` contains a mandatory `BPName`, which is used to identify the bundle when producing a packaged version of sources in a given flavour[8], an optional version number, two textual descriptions: a `ShortDescription` (maybe several, in several languages), a `LongDescription` (may also be in several languages), a mandatory `LicenseId` and a non-empty list of `AuthorId`.

The `LicenseId` is the license under which the bundle is distributed, independently from FDNTEX. If the packages are bundled together only for convenience in the distribution, but are not when referring to the original sources, then the license will be the same as the one for the whole distribution. The license under which the description file is distributed is the one of FDNTEX, if the author of the description wants to claim that it's part of FDNTEX, and then it doesn't need to be exposed there.

The `AuthorId` can be the author of the bundle, that is the people who put all those things together, or the author of the description. There is currently no way to distinguish between them.

The descriptions are supposed to be identical, when several are provided, but in various languages. That is, if four `ShortDescriptions` are provided, they are supposed to be four times the same text, but in different languages, the language being specified as usual in an XML document, and defaulting to English, *e.g.*:

```
<ShortDescription xml:lang='fr-FR'>
 Description en franais
</ShortDescription>

<ShortDescription>
 English description
</ShortDescription>
```

A bundle of sources A `SourceBundle`, as required by a `BPackage`, is a non-empty list of `SourceFile` and a `Prepare` directive that describes how to unpack all of those sources in a suitable tree for the building stage.

[8] Some flavours, like RPM and Debian, have their own source distribution format, and thus will need a name for the source package; others, like FreeBSD, give references to the real-world source, and then might not need all of that information. Since it's required by some, it's provided to all.

```
<SourceBundle>
 <SourceFile FileId="esieecv-1">
  CTAN:/macros/latex/contrib/supported/ESIEEcv.tar.gz
 </SourceFile>
 <Prepare UnpackTo="./ESIEEcv/">
  <untgz FileId="esieecv-1""/>
 </Prepare>
</SourceBundle>
```

Figure 4: Example of a `SourceBundle` tag for a simple package with sources from CTAN.

A `SourceFile` needs an id, to be referred to by the `Prepare` directive, and is supposed to be URL-like, that's either a URL, or something like `CTAN:/systems/knuth/web.tar.gz`, which is not a true URL. Several pseudo-protocols, which actually are default generic locations, will be defined. The need for two of them (`CTAN` to refer to any CTAN mirror, and `FDNTEX` to refer to any mirror of the whole distribution sources[9]) is already plain.

'Composite' generic locations may also be defined: for example, a package may be derived from `CTAN`, but a patch (maybe as simple as a Makefile) to facilitate its handling within FDNTEX may come from `FDNTEX`.

The `Prepare` directive is used for the usual sorts of files (tar, zip, and so on archives, patch files, etc.). Its variant `PrepareCust` (customize) is not yet well defined; its intent is to provide extensions for system-specific requirements, as when a particular flavour requires special treatment for unpacking the sources.

The preparation directive has an attribute that gives the place where it will unpack the sources relative to a supposed well-known root directory. *E.g.*, for the RPM flavour, the sources are supposed to be unpacked somewhere under `/usr/src/redhat/BUILD`, usually in a directory called `source` if the source archive is `source.tgz`, but maybe elsewhere if several sources are provided. In the default case, for this example, the attribute will be `"source"`. The directive is a list of actions, in the right order, that are to be accomplished to obtain a full source tree.

An example is shown at figure 4.

When retrieving the source file, the system is supposed to issue FTP commands like:

```
[ whatever is required to connect to the
  local CTAN mirror and go to the root
  of the mirror ]
cd macros/latex/contrib/supported
get ESIEEcv.tar.gz
```

[9] Even if `FDNTEX` will probably become equivalent to something like `CTAN:/systems/unix/fdntex`.

This is of importance, since, when a tarball is built on the fly, like this one, it will most probably have the same structure as it has on the archive disk.

At the time of writing, several actions are defined, but others will probably be added:

untar expands a tarball, has a mandatory attribute which is the `FileId`, and an optional one named `Offset`, used if an archive has to be expanded somewhere in the tree provided by a previous tarball (like `xdvik` within `web2c`);

untgz which behaves exactly in the same way with archives compressed by `gzip` or by the traditional `compress`;

untbz which behaves in the same way with archives compressed by `bzip2`;

patch which applies a patch, as provided by a "unified diff" to a source tree; it takes at most 4 attributes: the `FileId` (mandatory), an `Offset` (optional) which allows a patch to be applied to a subtree, a `Depth` (optional) which is used as the `-p` argument to the `patch` command that is issued, and a `Compression` (optional, default to `none`) which specifies how the patch was compressed, can be `gz`, `bz2`, `Z` or `none`.

The `PrepareCust` directive will be used to provide full control to the author of a description for a given flavour of the distribution. For example, it's not clear how a `zoo` archive can be unpacked on a Windows-like system, so it would be better to provide full control to the author, instead of a too fragile action in the generic directive. A way to mix both the `Prepare` and the `PrepareCust` might be provided in a future version.

After all the actions have been accomplished, the sources have to be ready in the directory specified by `UnpackTo` relative to the well-known root defined by the flavour of the distribution.

Description of the building stage As for the preparation stage, the building stage may be written in a generic way, using pre-defined actions, in a `Building` directive. Otherwise (when the predefined actions are not suitable), it may be written in a system specific way with any kind of scripting language, using a `BuildingCust` directive. The kind of scripting language used will be chosen according to the target system.

There must be, at most, one `Building` directive, and there may be several `BuildingCust` ones. If so, they all have to have different targets. The `Building` directive is optional since some packages might not be built in any generic way. *E.g.*, a `dvi` viewer is strongly system-dependent and then has no generic building description, since it's not to be built on unknown targets.

Up to now, only 6 actions have been provided for building a package, but, of course others will be in the first non-alpha release of the DTD.

tex to call TEX on a file, which has two attributes, `file` (mandatory) is the name of the file to be processed, and `format` (optional, default to `plain`) is the name of the format to be used.

latex to call LATEX on a file, which has a `file` mandatory argument.

dtxtodvi provides a high level interface to build the documentations of LATEX packages (which is, *de facto*, most of the work when building the distribution). It has a mandatory `file` attribute which is supposed to be a suffix-less file name (suffix has to be `dtx`), and 5 optional attributes. `idx` (default to no) saying if there is an index to process. `glo` (default to no) saying if there is a glossary (history of changes, usually) to process. `bib` saying if a bibliography requires a BIBTEX run. Those 3 attributes can be either `yes` or `no`. `pre-runs` (default 2) says how many times LATEX has to be run before the index, glossary and bibliography are processed. `post-runs` (default 1) says how many times LATEX has to be run after that.

mktexlsr which has an optional attribute named `mandatory` which states if the action can be delayed or not, and rebuilds the `ls-R` (or equivalent on the target) database. Today, it is useless, but will be useful in a near future when the necessary actions are created to install a font while building a package. This is required for packages which are composed of a font itself and the LATEX package to handle it, since they usually have to be installed before building the documentation.

move takes two mandatory attributes, `from` and `to`, and is used to move a file or a directory from a place to another. No wildcard is allowed here. It's not to be used here for the installation of the package, but only for moving files while building the package.

cd takes a mandatory argument `to` and is used to move into the tree while building. Building is supposed to start in the directory specified in the `UnpackTo` attribute of the preparation stage.

A call to

```
<tex file="myfile.tex" format="latex"/>
```

is of course equivalent to

```
<latex file="myfile.tex"/>
```

as far as the target system handles the two following commands in the same way:

```
tex '&latex' myfile.tex
latex myfile.tex
```

which is 'not always'.

An example of such a `Building` directive is shown at figure 5.

Description of the package itself As one can guess, this is the most complex part of the description, or at least the really interesting one.

A package description is composed of 6 parts: `Identification`, `Installation`, `UnInstallation`, `FileList`, `Methods`, and `Dependencies` directives. The definition of none of these is final, but we will discuss what we believe is a good prototype.

A `Package` has a mandatory `Id` attribute that will be used when one needs to refer to it in dependencies of other packages. The `Id` should be the name of the package, but it's permissible to use anything else.

Identifying a package Just as a `BPackage` has a `BPIdentification`, a `Package` has an `Identification`, which is to be systematically used (the `BPIdentification` will only be used by some flavours of FDNTEX).

The `Identification` has to provide a `Version`, which is the one provided by the main file of the package. If there are several important files which all have their version number, then the version number provided here can be an aggregate of those, or a date. *E.g.*, if the two important files are numbered 1.2 and 5.6, then the resulting package can be numbered 1.2.5.6 or 5.6.1.2, both being valid. When a date is provided, *e.g.*, for the LATEX kernel, it should be like 20000403 to ease the comparison of two version numbers when upgrading the system.

If a version number is provided for the bundle (in the `BPIdentification` of the corresponding `BPackage`) then it's appended to the version

```
<Building>
 <latex file="ESIEEcv.ins"/>
 <dtxtodvi file="ESIEEcv" idx="yes" glo="yes" />
 <latex file="test.tex"/>
</Building>
```

Figure 5: Example of a `Building` directive for a simple LaTeX package with a full documentation including index and history of changes.

number of the package itself, so that TeX 3.14159, bundled in `web2c` 7.3a will be in a package numbered 3.14159.7.3a in the final binary version of each flavour of the distribution.

A `Release` is mandatory to indicate if the package has evolved in its FDNTeX port but not in its source version, as when a forgotten dependency is added to the description. Currently, the release is a single integer. The release number can by increased either by the author of the description, when it evolves, or by the system, in cases of automatic dependency handling (see section “Conflicts”, above, for an example of such a case).

In future releases of the DTD the `Release` will probably be more informative, perhaps in a 2- or 3-integer system, like 1.0.0 for the first release, then, increasing the first one if the description of the package has evolved in an important way (*e.g.*, mending broken building directives), increasing the second digit if it evolved in a harmless way (*e.g.*, added a dependency) which means there is probably no need for upgrading, or the third digit if it evolved in a minor way (*e.g.*, to fix a typo in a `LongDescription`) in which case there is absolutely no reason to upgrade.

Textual descriptions use exactly the same structure as those for `BPackage`s, as do the `LicenseId` (which is the license under which the package itself is distributed) and `AuthorId` (which is the author of the package). Here, there is no confusion between ‘author of the package’ and ‘author of the description’: It’s systematically the author of the package; the author of the description has already been cited in the identification of the bundle.

Installing a package Some target systems may not be able to deal correctly with the case where a single source-bundle provides several packages. On such systems, the `BPackage` acts as a ‘virtual’ package which has no real existence, and which installs no files on the system. All the related packages will require the virtual package to be built and installed (through the dependencies mechanism) before building themselves. The ‘building’ stage of such packages will be empty, and their ‘installation’ phase will install the part of the virtual package that is required.

Of course, such a subtlety need only be used when there are several `Packages` in a `BPackage`. In such cases, the building and installation stages will be system-specific only for those targets that experience difficulties, and for that class of packages.

In most cases, when the bundle has to be installed in a single run, the installation stage will be handled within the building stage, and the per-package installation stage will be empty.

1. When we build `web2c` on a system that can handle multiple packages, the building stage builds and installs the whole thing, and the installation stage does nothing.
2. When we build `web2c` on a system that cannot handle multiple packages, the building stage only builds the bundle, and the per-package installation system installs that package’s part (*e.g.*, the `dvitype` binary for the `dvitype` package).
3. The `tools` bundle of packages for LaTeX can be handled in its entirety by generic directives, but is inherently a multi-package bundle; for such bundles the building stage will only build, and the per-package installation will install each package, so that the directives have to be written only once.
4. When we build a single small package (such as `ESIEEcv`) for any system, the building stage builds and the installation stage installs the package. This is the most frequent case.

The case of really complex bundles like `web2c` is handled like this because re-writing the installation stage for systems which cannot handle multi-packages is *really* hard, and error-prone. Using this method, the errors will appear only for truly minimalist systems, and not for all.

So, just like the building stage, the installation stage will use a high level language to describe things to be done, and this generic description can be overloaded when a target needs some special things to be performed that cannot be described by the generic language.

It should also be noted that the first installation — the one performed just after the building stage has been completed — is likely to differ from a 'normal' one — installing the final package on the target system. Thus, two tags are provided: `WhileBuild` to describe the installation while we are building the binary package, and `OnTarget` for the other one. Both of them have a "cust" variant, to allow overriding.

The high level language, at the time of writing, is quite poor and will evolve a lot. Some information is given as attributes of the `Installation` tag:

bindir for the directory where the binary executables have to be installed, this is supposed to be relative to the root of the target system, or at least to another root than the one used for the other directories; in fact it will often be `bin`, which will be concatenated to any prefix given while installing the real thing, *e.g.*, `/usr/local`.

libdir is the same thing for binary system libraries (mostly `libkpathsea`).

incdir is for the system include files (mostly the `.h` files related to `libkpathsea`).

docdir is the directory where the documentation for this package should stand, relative to the root-directory of the TEX system, usually something like `doc/latex/ESIEEcv` to be concatenated with *e.g.*, `/usr/local/share/texmf`, where `/usr/local` is the prefix specified during the installation and `share/texmf` is the "well-known" root.

stydir is the directory for `.sty` files provided by the package.

bstdir same thing for BibTEX styles;

bibdir for bibliographic databases;

tfmdir for `tfm` files;

mfdir for METAFONT sources;

mapdir for files related to the `map` system for PS fonts;

istdir for makeindex styles.

Instructions (defined so far) are as follows:

mkdir to create a directory, out of the ones specified previously in the attributes.

docfile is a file to be installed in `docdir`, and the same for `binfile`, `libfile`, `bstfile`, `bibfile`, `styfile`, `tfmfile`, `mffile`, `mapfile`, and `istfile`. The only one to have an attribute is `binfile`, which has an attribute `strip` which can be either `yes` or `no`, is optional, and defaults to `yes`, and says if the binary is to be stripped.

mktexlsr (see description in section "Description of the building stage", above).

format which is empty and has a mandatory attribute `name` giving the name of the format to rebuild. There is still no way to say that all the formats need to be rebuilt or that several of them have to be; this facility will be provided in future versions of the description language.

In practice, in most cases, the `WhileBuild` installation method will use those instructions, while the `OnTarget` will only state a single `mktexlsr`, since the system already takes care to move all the files listed in the `FileList` to the right place.

The default behaviour for uninstalling a package is to perform the converse of the the same actions as for installing, that is remove the file instead of installing, remove the directories if empty, and so on. Formats are also rebuilt as necessary.

The list of files The `FileList` contains a list of `docfile`, `cfgfile`, `file` and `dir`, each being part of the archive to create. Configuration files are isolated so that the uninstall and upgrade systems can handle them smoothly and save them. Directories are removed while uninstalling, if they are empty.

The dependencies The `Dependencies` tag contains a list of `BuildDep` which gives the name of a package that has to be installed in order to build this one, `Dep` which gives the name of a package that has to be installed for this one to be used properly, and `Conflict` giving the name of a package that creates a conflict if installed on the same system as this one.

The `Dep` tag has an integer attribute that gives the softness level (see section "Installation dependency" on page 164), whose default value is 10 (hard dependency). A value of 0 is legal but useless since it means no dependency at all.

A more fine-grained system will be used for future versions of the description language, at least for the conflicts since this model is really far from perfect. The new system will most probably be based on an external list of packages that are known to be in conflict. An analagous softness level will be used for conflicts, to say if it is legal to override a conflict directive or not.

A complete example

The package described here is a small one that permits the typesetting of a curriculum vitæ as French companies like to see them. It's a LATEX package, with a test file and the documentation included with the source in the `.dtx` file. The description given here has been validated against the DTD we have just defined, but not yet used to build a real package since currently there is still no generator written.

Some parts of the document have been deleted (like the empty description of the 4 packages listed in the dependencies list).

```
<?xml version="1.0" standalone='yes'?>
<!DOCTYPE doc SYSTEM "./fdntex.dtd" [
<!-- used to shorten the file name later
     since this XML document is typeset in
     two columns mode -->
<!ENTITY CTANmlcs
 "CTAN:/macros/latex/contrib/supported">
] >

<!--     maximum length of a line      -->

<doc>
 <Author Id="bayart">
  <Name>Benjamin B. Bayart</Name>
  <EMail>bayartb@edgerd.fdn.fr</EMail>
  <EMail>bayartb@guinness.domicile.fr
                              </EMail>
 </Author>
 <License Id="lppl">
  LaTeX Project Public License
 </License>
 <License Id="fdntexl">
  The FDNTeX license
 </License>

 <BPackage>
  <BPIdentification>
   <BPName>BP-ESIEEcv</BPName>
   <LicenseId Id="lppl"/>
   <AuthorId Id="bayart"/>
  </BPIdentification>

  <SourceBundle>
   <SourceFile FileId="esieecv-1">
    &CTANmlcs;/ESIEEcv.tar.gz
   </SourceFile>
   <Prepare UnpackTo="./ESIEEcv/">
    <untgz FileId="esieecv-1"
           Offset=".."/>
   </Prepare>
  </SourceBundle>

  <Building>
   <latex file="ESIEEcv.ins"/>
   <dtxtodvi file="ESIEEcv"
             idx="yes"
             glo="yes" />
   <latex file="test.tex"/>
  </Building>

<Package Name="ESIEEcv">
 <Identification>
  <Version>2.0a</Version>
  <Release>1</Release>
  <ShortDescription>
   ESIEEcv to typeset French
   curriculum vitae
  </ShortDescription>
  <ShortDescription xml:lang="fr-FR">
   ESIEEcv pour mettre en forme un
   curriculum vitae a la franaise
  </ShortDescription>
  <LongDescription>
   This package allows one to typeset
   a curriculum vitae as a French
   company would expect to receive it.
  </LongDescription>
  <LongDescription xml:lang="fr-FR">
   Ce package permet la mise en forme
   d'un curriculum vitae tel qu'une
   entreprise franaise s'attendra a
   le recevoir.
  </LongDescription>
  <LicenseId Id="lppl"/>
  <AuthorId Id="bayart"/>
 </Identification>
 <Location>
  macros/latex/contrib/supported
 </Location>
 <Location kind="func">
  lang/french
 </Location>
 <Installation
  stydir="tex/latex/ESIEEcv"
  docdir="doc/latex/ESIEEcv">
  <WhileBuild>
   <styfile>ESIEEcv.sty</styfile>
   <docfile>ESIEEcv.dvi</docfile>
   <docfile>test.tex</docfile>
   <docfile>test.dvi</docfile>
   <mktexlsr/>
  </WhileBuild>
  <OnTarget>
   <mktexlsr/>
  </OnTarget>
 </Installation>
 <FileList>
  <file>
   tex/latex/ESIEEcv/ESIEEcv.sty
  </file>
  <docfile>
   doc/latex/ESIEEcv/ESIEEcv.dvi
  </docfile>
```

```
    <docfile>
     doc/latex/ESIEEcv/test.tex
    </docfile>
    <docfile>
     doc/latex/ESIEEcv/test.dvi
    </docfile>
   </FileList>
   <Methods>
    Not yet defined
   </Methods>
   <Dependencies>
    <Dep name="a-LaTeX-format"/>
    <Dep name="tabularx"/>
    <BuildDep name="a-LaTeX-format"/>
    <BuildDep name="tabularx"/>
    <BuildDep name="babel"/>
    <BuildDep name="fnt-ec"/>
   </Dependencies>
  </Package>
 </BPackage>
</doc>
```

The full DTD

As has already been explained, this Document Type Definition (DTD) is not final. It is the DTD that this paper has described, and it has been used to validate the description. A more up to date version might be available at `ftp://ftp.fdn.fr/pub/FDNTeX/Develop/fdntex.dtd`, and a more up-to-date version of this paper (or at least something describing the corresponding version of the DTD) should be available at the same place.

```
<!ELEMENT doc (Author|BPackage|License)*>

<!ELEMENT Author (Name,EMail*)>
<!ATTLIST Author Id ID #REQUIRED>

<!ELEMENT Name ANY>
<!ELEMENT EMail ANY>

<!ELEMENT License ANY>
<!ATTLIST License
        Id ID #REQUIRED
        xml:lang NMTOKEN 'en'>

<!ELEMENT BPackage (BPIdentification,
                    SourceBundle,
                    Building,
                    BuildingCust*,
                    Package+)>

<!ELEMENT BPIdentification
        (BPName,
         Version?,
         ShortDescription*,
         LongDescription*,
         LicenseId,
         AuthorId+)>

<!ELEMENT BPName ANY>
<!ELEMENT Version ANY>
<!-- ShortDescription and LongDescription
     are defined later on,
     when defining Package -->
<!ELEMENT LicenseId EMPTY>
<!ATTLIST LicenseId Id IDREF #REQUIRED>
<!ELEMENT AuthorId EMPTY>
<!ATTLIST AuthorId Id IDREF #REQUIRED>

<!--
 Identification
  BPName
  Version?
  ShortDescription
  LongDescription
  License
  AuthorId
-->

<!ELEMENT SourceBundle
        (SourceFile+,
         (Prepare|PrepareCust))>
<!ELEMENT SourceFile ANY>
<!ATTLIST SourceFile
        FileId ID #REQUIRED>
<!ELEMENT Prepare
        (untar|
         untgz|
         patch)+>
<!ATTLIST Prepare
        UnpackTo CDATA #REQUIRED>
<!ELEMENT untar EMPTY>
<!ATTLIST untar
        Offset CDATA "."
        FileId IDREF #REQUIRED>
<!ELEMENT untgz EMPTY>
<!ATTLIST untgz
        Offset CDATA "."
        FileId IDREF #REQUIRED>
<!ELEMENT patch EMPTY>
<!ATTLIST patch
        Offset CDATA "."
        Depth CDATA "1"
        FileId IDREF #REQUIRED
        Compression (gz|bz2|Z|none)
                                    "none" >
<!ELEMENT PrepareCust ANY>
```

```
<!ATTLIST PrepareCust
        UnpackTo CDATA #REQUIRED
        >



<!--
 SourceBundle
  SourceFile*
  Prepare | PrepareCust
   (UnpackTo?) implied
-->
<!ELEMENT Building
        (tex|
         latex|
         dtxtodvi|
         mktexlsr|
         move|
         cd)+>
<!ELEMENT tex EMPTY>
<!ATTLIST tex
        file CDATA #REQUIRED
        format CDATA "plain">
<!ELEMENT latex EMPTY>
<!ATTLIST latex
        file CDATA #REQUIRED>
<!ELEMENT dtxtodvi EMPTY>
<!ATTLIST dtxtodvi
        file CDATA #REQUIRED
        idx (yes|no) "no"
        glo (yes|no) "no"
        bib (yes|no) "no"
        pre-runs CDATA "2"
        post-runs CDATA "1">
<!-- mktexlsr will be defined later -->
<!ELEMENT move EMPTY>
<!ATTLIST move
        from CDATA #REQUIRED
        to CDATA #REQUIRED>
<!ELEMENT cd EMPTY>
<!ATTLIST cd
        to CDATA #REQUIRED>

<!ELEMENT BuildingCust ANY>
<!ATTLIST BuildingCust
        Target (i386|ppc|sparc|alpha)
                                #REQUIRED
        System (linux|freebsd|solaris|
                hpux9|hpux10)   #REQUIRED>

<!ELEMENT Package (
        Identification,
        Location+,
        Installation,
        UnInstallation?,
        FileList,
        Methods,
        Dependencies?,
        Provides?)>
<!ATTLIST Package
        Name ID #REQUIRED>

<!ELEMENT Identification (
        Version,
        Release,
        ShortDescription+,
        LongDescription+,
        LicenseId,
        AuthorId+)>
<!ELEMENT Release ANY>
<!ELEMENT ShortDescription (#PCDATA)>
<!ATTLIST ShortDescription
        xml:lang NMTOKEN 'en'>
<!ELEMENT LongDescription (#PCDATA)>
<!ATTLIST LongDescription
        xml:lang NMTOKEN 'en'>
<!-- LicenseId and AuthorId are
     already defined -->

<!ELEMENT Location (#PCDATA)>
<!ATTLIST Location
        kind (ctan|func) "ctan">

<!ELEMENT Installation
        (WhileBuild,
         WhileBuiltCust*,
         OnTarget,
         OnTargetCust*)>
<!ATTLIST Installation
        bindir CDATA "."
        libdir CDATA "."
        incdir CDATA "."
        docdir CDATA "."
        stydir CDATA "."
        bstdir CDATA "."
        bibdir CDATA "."
        tfmdir CDATA "."
        mfdir  CDATA "."
        mapdir CDATA "."
        istdir CDATA ".">
<!ELEMENT WhileBuild
        (mkdir|
         docfile|
         binfile|
         libfile|
         bstfile|
         bibfile|
         styfile|
         tfmdile|
         mffile|
         mapfile|
         istfile|
         mktexlsr|
         format)+>
<!ELEMENT WhileBuildCust ANY>
<!ELEMENT OnTarget
        (mkdir|
         docfile|
         binfile|
         libfile|
         bstfile|
         bibfile|
         styfile|
         tfmdile|
         mffile|
         mapfile|
         istfile|
         mktexlsr|
         format)+>
<!ELEMENT OnTargetCust ANY>

<!ELEMENT mkdir ANY>
<!ELEMENT docfile ANY>
<!ELEMENT binfile ANY>
<!ELEMENT libfile ANY>
<!ELEMENT bstfile ANY>
<!ELEMENT bibfile ANY>
<!ELEMENT styfile ANY>
<!ELEMENT tfmfile ANY>
<!ELEMENT mffile  ANY>
<!ELEMENT mapfile ANY>
<!ELEMENT istfile ANY>
<!ELEMENT mktexlsr EMPTY>
<!ATTLIST mktexlsr
        mandatory (yex|no) "no">
<!ELEMENT format EMPTY>
<!ATTLIST format
        name CDATA #REQUIRED>
<!ELEMENT UnInstallation
        (WhileBuild,
         WhileBuiltCust*,
         OnTarget,
         OnTargetCust*)>

<!ELEMENT FileList
        (docfile|cfgfile|file|dir)*>
<!ELEMENT cfgfile ANY>
<!ELEMENT file ANY>
<!ELEMENT dir ANY>

<!ELEMENT Methods ANY>
<!ELEMENT Dependencies
        (BuildDep|
         Dep|
         Conflict)+>
<!ELEMENT BuildDep EMPTY>
<!ATTLIST BuildDep
        name IDREF #REQUIRED>
<!ELEMENT Dep EMPTY>
<!ATTLIST Dep
        name IDREF #REQUIRED
        level CDATA "10">
<!ELEMENT Conflict EMPTY>
<!ATTLIST Conflict
        name IDREF #REQUIRED>

<!ELEMENT Methodes EMPTY>
<!--
  FileList
  Methods
-->

<!ELEMENT Provides EMPTY>
<!ATTLIST Provides
        Name ID #REQUIRED>
```

Acknowledgments

FDNTEX, and thus this paper, is not the pure outcome of my ideas of what should constitute a better distribution of TEX, it's also the result of long talks with friends of mine, like Rémy Card and Olivier Gutknecht; and with people more involved in the TEX world and in writing distributions of TEX like Sebastian Rahtz who suggested the use of XML as a description language, Simon Cozens who gave me the idea of a high level language, Fabrice Popineau who worked with S. Rahtz on the system used in TEXlive, Taco Hoekwater who proposed great ideas for dependencies that have not been matured enough (at least by me) to be discussed in this paper, Robin Fairbairns who gave us good advice during the discussions about TPM, and probably others that I forgot.

Special thanks to Sylvia Pédron who spent a lot of hours helping me to improve my English and proof reading this paper.