Cahiers GUTenberg

NOUVELLES PISTES POUR UNE DISTRIBUTION DE TEX

Benjamin Bayart

Cahiers GUTenberg, nº 35-36 (2000), p. 121-132.

<http://cahiers.gutenberg.eu.org/fitem?id=CG_2000___35-36_121_0>

© Association GUTenberg, 2000, tous droits réservés.

L’accès aux articles des *Cahiers GUTenberg*
(http://cahiers.gutenberg.eu.org/),
implique l’accord avec les conditions générales
d’utilisation (http://cahiers.gutenberg.eu.org/legal.html).
Toute utilisation commerciale ou impression systématique
est constitutive d’une infraction pénale. Toute copie ou impression
de ce fichier doit contenir la présente mention de copyright.

Nouvelles pistes pour une distribution de TEX

Benjamin Bayart

Résumé. On exposera sommairement la problématique dans laquelle est apparue l'idée d'une nouvelle distribution de TEX. Une discussion nettement postérieure, et partant de problèmes différents, pour la définition d'un TPM[1] a amené rapidement à des conclusions extrêmement similaires. Les principes de base de FDNTEX tels qu'ils ont été imaginés au début, puis modifiés lors des discussions à propos de TPM, seront exposés et expliqués en détails.

1. The name of the game

Pour sacrifier à la tradition : pourquoi ce nom là ? Simplement parce que FDN (French Data Network, association loi 1901, fournisseur d'accès et de services Internet) héberge les serveurs pour le projet (CVS, Web, Ftp, etc) et qu'à l'origine du projet il s'agissait de produire une distribution installable par tous les membres du bureau de la dite association.

Au chapitre du vocabulaire, dans cet article, on entendra par *package* une extension de LATEX 2ε telle qu'attendue par la commande `\usepackage` ; et par *module* un élément logiciel unitaire installable, par exemple un fichier `.rpm`, ou `.deb` ou de tout autre format similaire.

2. Pourquoi une distribution ?

L'apparition de distributions de TEX cohérentes, simples à installer, et disponibles sur une grande variété de systèmes, tient une place de choix dans les améliorations récentes connues par les systèmes basés sur TEX.

Bien entendu, dans les distributions que l'on rencontre de nos jours, teTEX et fpTEX (sa transposition dans le monde Windows) sont l'aboutissement de cette évolution.

1. TEX Package Manager.

Un problème rencontré lors de travaux relativement récents à mis au jour une défaillance du système de distribution actuel : le système en cours d'écriture exigeait une installation particulière de TEX, certes basée sur teTEX, mais comprenant également nombre d'extensions qui n'y avaient pas encore été intégrées. Comment reproduire rapidement et simplement une installation de ce type, en la tenant à jour si possible ?

Faire croître la distribution de base en contribuant à son développement n'est pas une solution tout à fait satisfaisante : le résultat de plusieurs collaborations de ce type serait, à terme, une distribution d'une taille bien trop imposante, et qui, pour un usage donné, comporterait une majorité de composants inutiles. De surcroît, des problèmes de cohérence et de maintenance se poseraient probablement : comment maintenir tel composant intégré par une personne qui depuis ne s'intéresse plus au projet ?

De là naquit l'idée d'une distribution plus souple, plus modulaire, ayant une bonne gestion des dépendances internes. C'était la fin de l'été 1998.

3. Les objectifs visés

Des 18 mois de travaux sur la distribution, des développements des différents prototypes, et des nombreuses discussions qui ont eu lieu autour, dans le monde TEX et avec des administrateurs systèmes, il est ressorti plusieurs idées fortes qui sont le fondement du projet actuel.

Cette distribution doit donc satisfaire aux objectifs suivants, parfois contradictoires, qui seront expliqués plus avant :
- modulaire ;
- la plus complète possible ;
- minimale une fois installée ;
- portable sur plusieurs systèmes, avec un ordre de préférence ;
- dédiée au système sur lequel on l'installe ;
- documentée.

3.1. Une distribution modulaire

C'est là la clef de voûte du système : une distribution de TEX moderne se doit d'être modulaire, bien plus que ce qui fut utilisé jusqu'à présent. Pourquoi ce choix, et pourquoi le pousser presque à l'extrême ? Pour répondre à ces deux exemples classiques.

Le premier exemple est un cas simple. Qui utilise, ne serait-ce qu'occasionellement `patgen` ? Qui a ce programme installé par sa distribution de TEX ? À

la première question, une infime minorité aura répondu utiliser ce logiciel[2], à la seconde (vérifiez sur votre teTEX habituelle), une immense majorité aura répondu disposer du logiciel. Une majorité d'utilisateurs a donc installé un logiciel inutile. Il s'agit là d'un cas précis, et peu alarmant étant donné la place prise par ce logiciel, mais il est loin d'être isolé.

Le second exemple est un cas moins simple, mais qui sera amené à devenir de plus en plus fréquent. Quelqu'un qui aurait besoin d'utiliser Ω pour composer en Unicode et bénéficier des extensions apportées par ce moteur, et également d'utiliser Λ, devrait pouvoir ne pas installer TEX. Bien que cela puisse sembler paradoxal pour une distribution de TEX. Les versions alternatives au moteur TEX classique étant de plus en plus nombreuses, et de plus en plus utilisées, ce type de choix va se multiplier.

La seule réponse possible à ces deux problèmes est d'avoir une distribution qui ne présuppose que très peu de chose voire, si c'était faisable, rien.

De plus, certaines segmentations qui peuvent sembler de prime abord contre-nature ont été faites. Par exemple, le fichier `modes.mf` a été isolé dans un module pour pouvoir être mis à jour très simplement, et à moindre coût, par exemple pour adapter une installation à une nouvelle imprimante.

3.2. Prévoir le maximum de modules

Une telle distribution, étant modulaire à un niveau fin (puisqu'on doit pouvoir installer TEX et e-plain, sans pour autant installer plain), pourra, et donc devra, intégrer sans scrupules le maximum d'extensions classiques ou plus inhabituelles.

En effet, il serait dommageable que l'on puisse installer des extensions pour le japonais, mais que l'on doive installer certaines extensions mathématiques extravagantes à la main. Un des buts doit donc être de mettre à disposition de l'utilisateur final un maximum de fonctionnalités dans lesquelles il fera un choix, quitte à revenir sur son choix plus tard, quand il aura besoin d'ajouter ou de supprimer des fonctionnalités.

3.3. Le minimum de perte à l'installation

Le système d'installation de la distribution devra installer le strict minimum pour mettre en œuvre les fonctionnalités demandées par l'utilisateur, afin d'éviter le cas cité plus haut de `patgen`.

---

2. Pour les autres, ceux, innombrables, qui ne s'en sont jamais servi : `patgen` sert à génerer la description des motifs de césure pour une langue donnée dans un format lisible par TEX.

L'ensemble des modules installés devra donc être la « fermeture transitive » des modules demandés par l'utilisateur et de ceux qui leur sont nécessaire pour fonctionner.

Pour atteindre cet objectif, une gestion fine et rigoureuse des dépendances entre modules sera nécessaire.

Un système idéal devrait même être capable, lorsque l'utilisateur déclare ne plus avoir besoin d'une des fonctionnalités qu'il avait demandé au départ, de supprimer tous les modules qui ne sont plus requis. Un tel système n'existe pas encore, et pourrait avoir quelques effets de bord inattendus et parfois désastreux : que l'utilisateur demande initialement un module A, qui requiert B qui n'a pas été spécifiquement demandé, puis s'habitue à utiliser B, puis demande la désinstallation de A qu'il n'utilise plus, si B n'est plus requis par personne il sera supprimé, et la disparition de B risque d'être déroutante.

3.4. Ne pas perdre la portabilité actuelle

La portabilité des distributions courantes de TEX est en grande partie liée à la portabilité de TEX lui-même, et des programmes qui ont été développés autour. Il ne faudrait pas que cette grande qualité disparaisse.

3.5. Ajustement fin au système cible

Cet objectif là ne découle pas directement des problèmes exposés précedemment, mais plutôt de la grande modularité à laquelle on arrive. Pour gérer tous ces modules, leurs dépendances, leur installation, leur mise à jour, leur configuration, etc, il faudra un système qui ne sera ni simple à écrire, ni simple à manipuler.

L'utilisateur final de la distribution aura, a priori, été amené à utiliser des outils similaires pour installer son système d'exploitation, et il serait donc judicieux que les modules de la distribution finales soient manipulables par les mêmes outils. Cela permet de transformer ce qui aurait été une connaissance spécifique à TEX en une connaissance spécifique au système d'exploitation utilisé.

Toutefois, pour les systèmes ne prévoyant pas ce type d'outils, ou pour les personnes qui ne souhaitent ou ne peuvent pas les utiliser, il sera nécessaire de prévoir une distribution basée sur des outils plus traditionnels comme ceux d'archivage.

Ainsi, le but est d'obtenir, par exemple, une variante de la distribution manipulable avec `swtools` pour les utilisateurs de HP-UX, une variante manipulable avec `dpkg` pour les inconditionnels de Debian, etc.

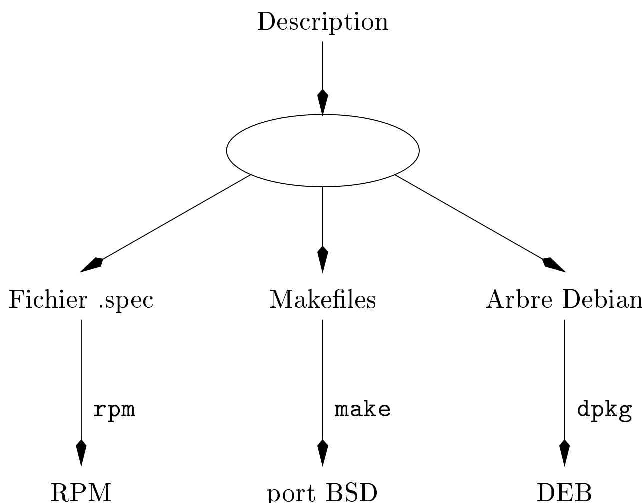

FIG. 1 – Représentation du processus de génération des différentes variantes de FDNTEX depuis une description unique de chaque module.

Une conséquence directe, pour garantir la cohérence de l'ensemble entre les différentes variantes, est qu'il est indispensable d'automatiser au maximum la création de ces variantes.

3.6. Documentation

Si chaque module doit, comme c'est devenu la tradition, être diffusé avec sa documentation, il sera nécessaire que la distribution elle-même, ainsi que ses outils propres s'il y en a, soit documentée comme il est de coutume dans les projets ayant trait à TEX.

4. Représentation de l'information

Obtenir automatiquement toutes les variantes de chaque module suppose qu'il existe une unique description, dans un langage donné, du module en question, et qu'un utilitaire viendra exprimer cette information dans un dialecte plus aisément compréhensible pour les outils de manipulation de la variante voulue.

La description de la majorité des modules peut se résumer à très peu d'informations :
– les modules dont il a besoin pour fonctionner ;

- une description si possible complète (auteur, date, version, etc) ;
- une description des quelques tâches simples à accomplir pour produire le module et
- une liste des fichiers contenus dans le module.

Ce langage serait utilisé comme indiqué au diagramme de la figure 1. L'idée ressemble un peu à ce qui est utilisé actuellement par certains ML, ou qui fut utilisé par KNUTH pour `web` : un seul source, qui, selon le traitement qu'on lui fait subir, produira plusieurs résultats radicalement différents. Dans notre cas, selon le format de sortie retenu, à partir de la description d'un module, on doit être capable de produire une description qui convienne à `rpm` (donc un fichier `.spec`), ou au mécanisme de ports de BSD (une série de fichiers architecturés autour d'un `Makefile`), ou...

Dans le même esprit, produire un descriptif de la distribution, à mettre en ligne, ou sur papier, ou en PDF, ou encore contribuer au catalogue qui existe déjà, peut se résumer à ajouter un format de sortie au système.

C'est ce format de description des modules qui fut radicalement changé par l'approche de TPM. Il est en effet apparu que le langage rapidement improvisé qui était utilisé dans les premiers développements n'était pas, de loin, suffisant. De l'avis général, XML semblait le meilleur choix possible. À l'heure où cet article est écrit, la DTD exacte n'est pas arrêtée, on décrira donc la nature et la qualité des informations qui devront être représentées plutôt que leur formalisme exact.

Schématiquement, la description d'un module peut se décomposer en quatre parties, ou plus exactement, trois parties, et des informations permettant de combiner le tout.

4.1. Préliminaires

Il s'agit là de toutes les informations classiques d'identification du module : son nom, le nom de l'auteur, la date de dernière modification, le numéro de version, des descriptions variées (plus ou moins longues, en plusieurs langues, etc).

Toutes ces informations sont sans particularités intéressantes par rapport à une distribution de n'importe quel autre logiciel, on pourra, par exemple, se baser sur le catalogue existant pour les trouver, dans un premier temps, avant d'envisager de produire le-dit catalogue à partir des informations.

Bien que ces informations nécessitent une grande rigueur et soient fondamentales, elles ne posent pas de problèmes techniques de modélisation.

4.2. Dépendances et liens

La problématique des dépendances entre modules (quel module requiert quel autre pour fonctionner convenablement) est beaucoup plus complexe. Le modèle retenu devrait être capable d'exprimer des choses assez fines. Voyons les cas les plus inhabituels, c'est-à-dire ceux qui sortent d'une simple relation de dépendance telle que « `tabularx` requiert `array` ».

Les fonctionalités — pour reprendre une terminologie déjà utilisée par d'autres distributions d'autres logiciels — qui permettent de dire, par exemple, qu'un module requiert *un* TEX et non pas *le* TEX. En effet, les fontes `EC` ont peu de sens sans un moteur TEX pour réaliser la mise en page, mais, qu'il s'agisse de TEX, d'Ω ou d'ε-TEX, peu importe.

Cette notion de base devra certainement être raffinée, par exemple pour indiquer que l'on a besoin, pour tirer parti des fontes PostScript d'un pilote capable d'utiliser de telles fontes (en effet, `dvips` n'est pas nécessairement le seul), et pas simplement d'un pilote quelconque.

De nombreux formats de description de modules d'autres distributions fournissent un mécanisme similaire, sa réalisation et son formalisme, quel qu'ils soient dans la DTD finalement retenue seront fonctionnels et efficaces.

Certains problèmes pourraient cependant rester sans solution avec les approches classiques, par exemple : le package `french` a besoin pour fonctionner convenablement, soit de fontes accentuées, soit d'un moteur MlTEX. Deux fonctionnalités « remplaçables » l'une par l'autre, dans cet usage précis, mais trop différentes pour être considérées identiques.

Dépendances strictes et dépendances larges sont nécessaires pour l'immense majorité des cas, si l'on souhaite permettre une sélection précise de ce qui est installé sur un sytème. L'exemple classique est un package dont le *mode d'emploi* utilise une fonte inhabituelle qui n'est aucunement requise par le package lui-même. Pour fonctionner de manière optimale, le module devrait disposer de la fonte, pour que l'on puisse lire le mode d'emploi dans de bonnes conditions, mais c'est un luxe qui est bien inutile pour quelqu'un qui dispose déjà de ce mode d'emploi sous une autre forme ou n'en a simplement pas besoin.

De même, le programme de génération automatique de fontes est prévu pour manipuler des formats très différents, et devrait donc, en toute rigueur, dépendre de tous les outils de génération de tous ces formats. Mais il n'est pas utile de produire l'image d'une fonte PostScript sur un système ou seules des fontes METAFONT sont utilisées, et réciproquement.

Dans ce cadre là, un niveau de dépendance entre 0 et un maximum M donné peut être une bonne approche : 0 quand deux modules sont totalement indé-

pendants, M quand il est inenvisageable d'utiliser l'un sans l'autre, et, entre les deux, un certain nombre de seuils codifiés : nécessaire au mode d'emploi, nécessaire pour des modes d'utilisation peu fréquents, nécessaire pour l'utilisation habituelle, etc.

Dépendances vis-à-vis du système cible , par exemple, la nécessité d'avoir affaire à une version donnée d'un système totalement hors de la distribution. Les exemples simples pouvant être des bibliothèques X11 (ou X10), un préprocesseur, un outil de développement comme `make`, etc. Si ces dépendances sont simples à exprimer dans le contexte d'une distribution pour *un* système, elles deviennent beaucoup plus complexes à formaliser.

Deux mécanismes complémentaires permettant de résoudre ce problème sont envisagés. L'un basé sur des exceptions : « le module dépend de `A` dans le cas général, mais de `A` et `B` pour telle variante ». L'autre basé sur des dépendances purement formelles, qui ne seront évaluées qu'au moment de produire la variante de la distribution correspondant à un système donné : « requiert la disponibilités des `threads` POSIX » qui sera ultérieurement évalué en « requiert une version du système postèrieure à 42.1 » ou « requiert la bibliothèque `xxx` ».

Liens entre deux modules — autre que de dépendance. Par exemple indiquer la relation entre `tangle` et `weave`, qui ne sont pas *stricto sensu* dépendants l'un de l'autre. Ou encore la relation qu'il peut y avoir entre deux fontes qui sont prévues pour être utilisées ensemble, comme par exemple deux fontes PostScript données, même si cela ne peut pas être vu comme une contrainte.

4.3. Actions

Le troisième type d'information nécessaire à la description d'un module est la série d'actions qui doivent être entreprises pour mener à bien les différentes étapes de sa vie (installation, suppression, remplacement par une version plus récente, etc).

De même, les directives de compilation devront être indiquées puisque les différentes variantes de la distribution devront être produites automatiquement.

Ces deux cas, qui peuvent sembler simples, ne le sont pas. En effet, si, pour une variante donnée de la distribution, ces informations sont le plus souvent très simples à écrire, elles deviennent beaucoup plus complexes dans le cadre d'une description générale de module. Une des différences fondamentales constatées, même entre deux variantes sur des systèmes Unix, est l'approche choisie par le système : certains utiliseront des scripts shell, et d'autres des Makefile, induisant ainsi une grosse différence de syntaxe.

La première approche retenue, qui semble la plus généraliste, était de choisir un langage puissant et expressif, et d'apprendre, au fur et à mesure, à traduire les fonctionnalités utilisées dans les autres languages. Par exemple, traduire le shell de `rpm` en Makefile pour les ports BSD. Cette approche, doublée d'un mécanisme d'exception pour les fonctionnalités trop complexes à traduire, avait l'avantage d'être un modèle ouvert et donc souple, mais le défaut de ne pas être stable, puisque le langage évolue, ainsi que les outils de traduction automatique, à mesure que la distribution se complète et que les cas difficiles apparaissent.

Une seconde approche est celle d'un langage concis et minimaliste permettant de décrire les situations les plus fréquentes, et un mécanisme d'exception par module pour ceux qui demandent trop de travail. Par exemple, compiler la documentation d'un package est une chose simple à décrire dans un langage de haut niveau, les seules variables dans le processus étant le nombre de compilations requises, la présence ou non des divers index ou de la bibliographie, et les éventuels annexes classiques comme des graphiques METAPOST. Elle a l'avantage de mener à un système fermé, donc plus aisément diffusable, mais qui souffrira donc de graves lacunes dans ses premières versions.

4.4. Configuration et administration

La configuration (et/ou administration) de la distribution est un problème multiple qui, s'il n'est déjà pas évident à traiter sur les distributions actuelles, deviendra bien pire dans une distribution modulaire.

La première évidence est que, si on souhaite que le système soit utilisable, il faut que les outils de configuration soient eux-même modulaires, et que chaque module en ayant besoin contienne la partie de l'outil de configuration qui le concerne. En effet, on ne peut pas envisager un outil qui contiendrait déjà tout le nécessaire pour configurer tous les modules, et déciderait des actions à entreprendre en fonction de ce qui est installé : l'outil refléterai alors l'état de la distribution à un moment de son évolution, et non pas ce qui est réellement installé sur la machine. D'autre part, si chaque module fournissait son propre outil de configuration, l'ensemble serait très rapidement inutilisable, la cohérence des outils entre eux n'étant pas assurée.

Il conviendra donc, avant que la distribution puisse être considérée comme utilisable par le grand public, qu'un tel outil soit développé. Un autre intérêt de cette approche étant que, si on a un outil central assez générique, et des pièces ajoutées par chacun des modules installé, alors, l'outil central peut être adapté précisément au système sur lequel il sera utilisé, du moment qu'il respecte le même protocole.

Toutefois, certains cas simples et classiques de configuration seront modélisés dans un langage restreint. Par exemple, l'ajout d'un format pour certains moteurs TEX est une opération qui reste toujours plus ou moins la même et peut donc être décrite de manière générique.

4.5. Imbrications

Une dernière forme d'information devra être contenue dans la description des modules, sans pour autant être, a priori, rattachée à un module donné : la façon dont ceux-ci peuvent être imbriqués.

En effet, si on souhaite être en mesure de faire de TEX, METAFONT, et `patgen` trois modules autonomes, il faudra, à partir du source habituel de l'ensemble `web2c` être en mesure de produire plusieurs modules binaires directement installables.

Les modèles des outils de construction de distributions que l'on rencontre actuellement possèdent presque tous la notion de « source unique » qui produit plusieurs modules. Mais cette notion, pour puissante qu'elle soit, n'est pas suffisante dans le cas précis d'une distribution de TEX.

Le cas type qui pose problème dans ce modèle est celui d'Ω. En effet, ce logiciel, techniquement, est une modification de TEX. Il faut donc, si on veut produire une version exécutable d'Ω produire également une version binaire de TEX, ou, plus exactement, c'est le cheminement le plus naturel. Ainsi, il faut, à partir de deux « sources » (`web2c` pour tout les outils TEX habituels, et `omega` pour les extensions) produire un très grand nombre de modules : tous ceux contenus dans `web2c`, à savoir une vingtaine, plus tous ceux pour Ω. Le même problème se posera pour `oxdvi`[3] qui est distribué comme une modification de `xdvi`.

À ce jour, aucun modèle fiable et générique n'a été choisi pour indiquer que n sources produisent m modules. Le modèle le plus satisfaisant serait celui de RedHat, pour `rpm`, mais il reste trop simpliste pour certains cas.

5. État d'avancement

Le projet FDNTEX est encore en pleine évolution et ne pourra pas être considéré comme utilisable largement avant de longs mois. Cependant, le projet est suffisament avancé pour que des utilisateurs avertis puissent faire des tests, et l'utiliser de manière quotidienne.

3. Version de `xdvi` adaptée aux extensions d'Ω, en particulier sur les fontes unicode.

5.1. Le prototype

Avant d'arrêter une définition précise de ce que devait recouvrir le projet, et de ce qu'il devait être à terme, un prototype complet, contenant à peu près tout ce qu'on est en droit d'attendre d'une bonne distribution de TEX a été réalisé. Il n'existe que deux variantes à ce jour : Linux (`rpm` sur processeurs Intel, Sparc, et PowerPC), et les ports FreeBSD 3.0 pour Intel.

Ce prototype est figé depuis juillet 1999 et n'évolue quasiment plus.

5.2. Le langage de description

Le langage de description de modules offre d'ores et déjà, à l'heure actuelle la majorité des fonctionalités attendues, au détail prêt qu'il n'est pas encore basé sur XML, *i.e.* les champs, leur sens et leur structure sont valides et exploités par plus d'une centaine de modules, mais n'ont pas encore le bon formalisme de rédaction.

Les fonctionnalités pour lesquelles une modélisation des données n'a pas encore été mise en place sont :
- modélisation des différents niveaux de dépendance (stricte, large, etc), peu de modèles de distributions utilisés par des systèmes d'exploitations permettent cette fonctionnalités, aussi le formalisme n'a pas encore été arrêté ;
- modélisation des « fonctionnalités » avancées (le cas MlTEX/fontes EC), parce que les contours réels de ce qui doit être modélisable ne sont pas clairs, le nombre de modules ayant besoin de telles dépendances étant encore trop faible ;
- les dépendances vis-à-vis du système cible ne sont pas modélisées mais reprises directement dans la variante voulue, simplement parce que ces dépendances sont, par définition, liées aux systèmes cibles et donc peu utiles à modéliser de manière générique, une modélisation générique se ramenant à une simple table de correspondance du type « Threads POSIX » se dit « `libpthread-1` » pour RPM, $\emptyset$ pour les ports BSD, etc ;
- les liens autres que la dépendance ;
- le langage de haut niveau modélisant les actions n'est pas formalisé ;
- la configuration des modules n'est pour le moment pas prise en compte ;
- les imbrications « n vers m » sont modélisées mais ne sont pas réalisées dans toutes les variantes.

5.3. Les variantes

Les variantes actuellement existantes ou en cours de développement sont Linux/RPM, avec recompilation automatisable sur différentes architectures ;

Linux/Debian, portable également sur différentes architectures ; et FreeBSD sur Intel, le portage vers les autres versions libres de BSD devant être facilité par le fait qu'OpenBSD, NetBSD et FreeBSD partagent le même système de « ports ».

Le choix d'unix libres pour les premières variantes est un choix induit par le fait, simple, qu'il faut disposer du système cible pour produire la variante correspondante. La seconde étape dans la liste des systèmes cibles sera, a priori, les unix non-libres, qui demanderont des volontaires disposant des systèmes pour le développement. Ensuite seront traités les systèmes basés sur des cœurs de type Unix (par exemple MacOS-X, NeXTStep et BeOS).

De plus, un port vers les systèmes Microsoft est envisagé, sur le même principe que fpTEX, si des développeurs capables de le réaliser souhaitent s'en occuper.

5.4. Échéancier

Bien entendu, sur ce type de projet, fonctionnant essentiellement sur du bénévolat, donner des dates ne peut être qu'indicatif, et ne peut concerner que les parties du projet pour lesquelles des volontaires sont déjà à la tâche.

De plus, cet échéancier, indicatif, est celui à la date de rédaction de cet article, soit deux mois avant GUT-2000.

À retenir donc :

avril 2000 language de description en XML, première DTD.
juin 2000 première variante en RPM.
août 2000 ports BSD et Debian.
décembre 2000 diffusion, dans ces trois variantes, d'une distribution, si possible au moins équivalente à teTEX, version de la DTD a priori définitive.