Maintaining large software distributions: new challenges from the FOSS era

Roberto Di Cosmo *and Berke Durak **and Xavier Leroy **and Fabio Mancinelli *and Jérôme Vouillon *
*PPS, University of Paris 7, `Firstname.Lastname@pps.jussieu.fr`
**INRIA Rocquencourt, `Firstname.Lastname@inria.fr`

Abstract. *In the mainstream adoption of free and open source software (FOSS),* distribution editors *play a crucial role: they package, integrate and distribute a wide variety of software, written in a variety of languages, for a variety of purposes of unprecedented breadth.*
Ensuring the quality of a FOSS distribution is a technical and engineering challenge, owing to the size and complexity of these distributions (tens of thousands of software packages). A number of original topics for research arise from this challenge. This paper is a gentle introduction to this new research area, and strives to clearly and formally identify many of the desirable properties that must be enjoyed by these distributions to ensure an acceptable quality level.

Keywords: Open source software, dependency management, EDOS project

1 Introduction

Managing large software systems has always been a stimulating challenge for the research field in Computer Science known as Software Engineering. Many seminal advances by founding fathers of Comp. Sci. were prompted by this challenge (see the book "Software Pioneers", edited by M. Broy and E. Denert [BD02], for an overview). Concepts such as structured programming, abstract data types, modularization, object orientation, design patterns or modeling languages (unified or not) [Szy97, GHJV94], were all introduced with the clear objective of simplifying the task not only of the programmer, but of the software engineer as well.

Nevertheless, in the recent years, two related phenomena: the explosion of Internet connectivity and the mainstream adoption of free and open source software (FOSS), have deeply changed the scenarii that today's software engineers face. The traditional organized, safe world where software is developed from specifications in a fully centralized way is no longer the only game in town. We see more and more complex software systems that are assembled from loosely coupled sources developed by programming teams not belonging to any single company, cooperating only through fast Internet connections. The availability of code distributed under FOSS licences makes it possible to reuse such code without formal agreements among companies, and without any form of central authority that coordinates this burgeoning activity.

This has led to the appearance of the so-called *distribution editors*, who try to offer some kind of reference viewpoint over the breathtaking variety of FOSS software available today: they take care of packaging, integrating and distributing tens of thousands of software packages, very few being developed in-house and almost all coming from independent developers. We believe that the role of distribution editors is deeply novel: no comparable task can be found in the traditional software development and distribution model.

This unique position of a FOSS distribution editor means that many of the standard, often unstated assumptions made for other complex software systems no longer hold: there is no common programming language, no common object model, no common component model, no central authority, neither technical nor commercial[1].

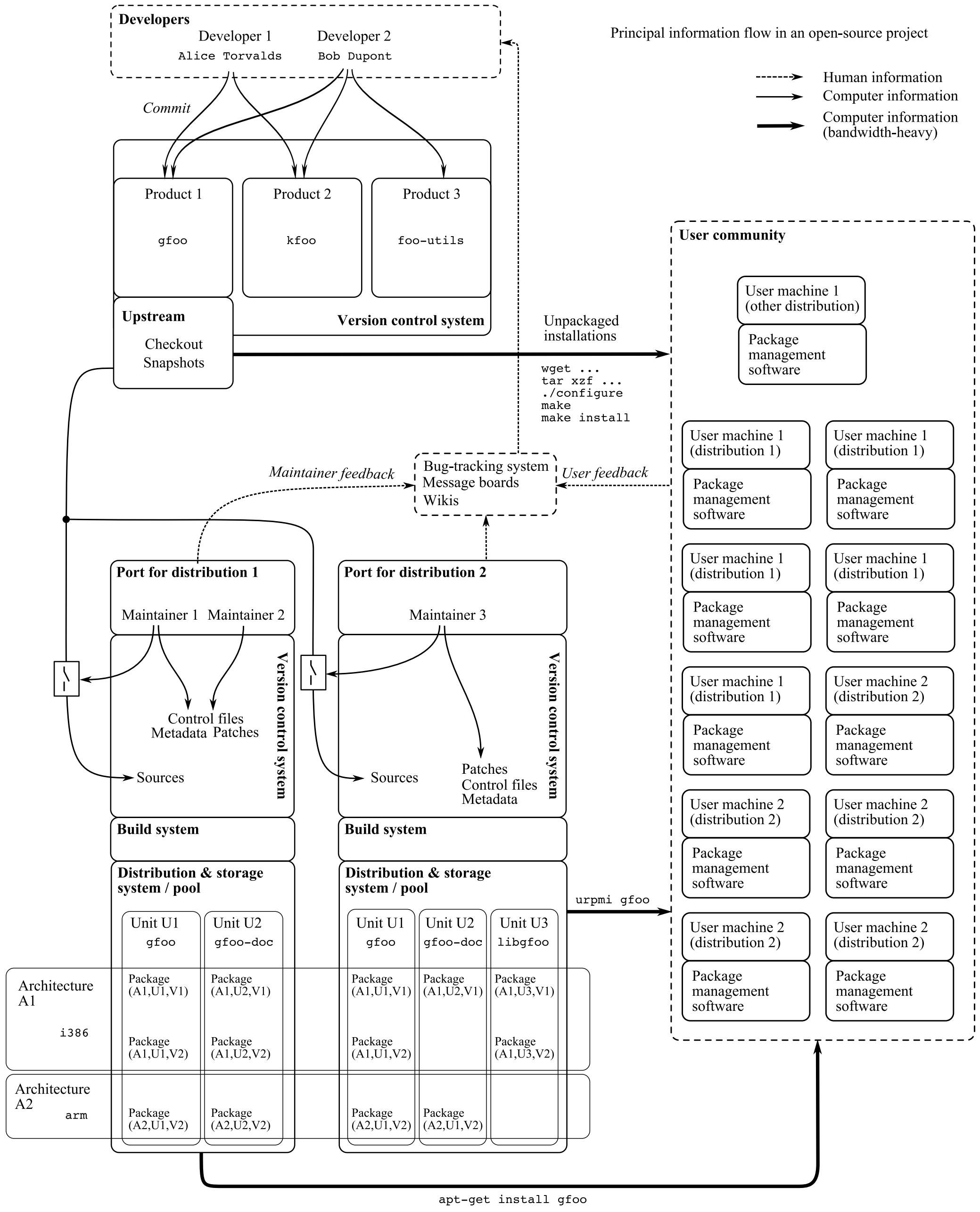
Consequently, most FOSS distribution today simply rely on the general notion of software *package*[2]: a bundle of files containing data, programs, and configuration information, with some metadata attached. Most of the metadata information deals with *dependencies*: the relationships with other packages that may be needed in order to run or install a given package, or that conflict with its presence on the system.

We now give a general description of a typical FOSS process. In figure 1 we have an imaginary project, called `foo`, handled by two developers, Alice Torvalds and Bob Dupont, who use a common CVS or Subversion repository and associated facilities such as mailing lists at a typical FOSS development site such as Sourceforge. Open source software is indeed developed as *projects*, which may group one or more developers. Projects can be characterized by a common goal and the use of a common infrastructure, such as a common version control repository, bug tracking system, or mailing lists. For instance, the Firefox browser, the Linux kernel, the KDE and Gnome desktop environments or the GNU C compiler are amongst the largest FOSS projects and have their own infrastructures. Of course, even small bits of software like `sysstat` consitute projects, even if they are developed by only one author without the use of a version control system. A given project may lead to one or more *products*. For instance, the KDE project leads to many products, from the `konqueror` browser to the desktop environment itself. Each FOSS product may then be included in a distribution. In our example, the project `foo` delivers the products `gfoo`, `kfoo` and `foo-utils`. A *port* is the inclusion of a product into a distribution by one or more *maintainers* of that distribution. The maintainers must:

- Import and regularly track the source code for the project into the distribution's own version control or storage system (this is depicted in figure 1 by a switch controlling the flow of information from the upstream to the version control system of the distribution).
- Ensure that the dependencies of the product are already included in the distribution.
- Write or include patches to adapt the program to the distribution.
- Write installation, upgrading, configuration and removal scripts.
- Write metadata and control files.

[1]In the world of Windows-based personal computing, for example, the company controlling Windows can actually impose to the ISV the usage of its API and other rules.

[2]Not to be mistaken for the software organizational unit present in many modern programming languages.



```
wget ...
tar xzf ...
./configure
make
make install
```

```
urpmi gfoo
```

```
apt-get install gfoo
```

Figure 1: Major flow of information in a FOSS project.

- Communicate with the upstream developers by forwarding them bug reports, patches or feature requests.

We see that the job of maintainers is substantial for which attempts to automate some of those tasks, such as automated dependency extraction tools [Tuu03, TT01] or getting source code updates from developers [Ekl05] are no substitute. In our example, we have a Debian-based distribution 1, with two maintainers for `foo`, and an RPM-based distribution 2 with one maintainer. A given product will be divided into one or more *units*, which will be compiled for the different *architectures* supported by the distribution (a given unit may not be available on all architectures) and bundled as *packages*. The metadata and control files specify how the product is divided into units, how each unit is to be compiled and packaged and on which architectures, as well as the dependency information, the textual description of the units, their importance, and classification tags. These packages are then automatically downloaded (as well as their dependencies) by the package management software (for instance, `apt` or `urpmi`) of the users of that distribution. Some users may prefer to download directly the sources from the developers, in which case they will typically execute a sequence of commands such as `./configure && make && make install` to compile and install that software. However, they then lose the many benefits of a package management system, such as tracking of the files installed by the package, automated installation of the dependencies, local modifications and installation scripts.

We now turn to the problem of ensuring the quality of a distribution. This problem is the focus of the European FP6 project EDOS (Environment for the development and Distribution of Open Source software). This problem can therefore be divided into three main tasks:

Upstream tracking makes sure that the package in the distribution closely follows the evolution of the software development, almost always carried over by some team outside the control of the distributor.

Testing and integration makes sure that the program performs as expected in combination with other packages in the distribution. If not, bug reports need propagating to the upstream developer.

Dependency management makes sure that, in a distribution, packages can be installed and user installations can be upgraded when new versions of packages are produced, while respecting the constraints imposed by the dependency metadata.

In this paper, we focus on the last task: dependency management. This task is surprisingly complex [Tuu03, vdS04], owing to the large number of packages present in a typical distribution and to the complexity and richness of their interdependencies. It is at the very heart of the research activity conducted in workpackage 2 of the EDOS project.

More specifically, our focus is on the issues related to dependency management for large sets of software packages, with a particular attention to what must be done to maintain consistency of a software distribution *on the repository side*, as opposed to maintaining a set of packages installed *on a client machine*.

This choice is justified by the following observation: maintaining consistency of a distribution of software packages is *fundamental* to ensure quality and scalability of current and future distributions; yet, it is also an *invisible* task, since the smooth working it ensures on the end user side tends to be considered as normal and obvious as the smooth working of packet routing on the Internet. In other words, we are tackling an essential *infrastructure* problem that has long been ignored: while there are a wealth of client-side tools to maintain a user installation (`apt`, `urpmi`, `smart` and many others [Sil04, Man05, Nie05]), there is surprisingly little literature and publically available tools that address server-side requirements. We found very little significant prior work in this area, despite it being critical to the success of FOSS in the long term.

The paper is organised as follows. Section 2 contains a formal description of the main characteristics of a software package found in the mainstream FOSS distributions, as far as dependency are concerned. In Section 3 we identify and formally define three desirable properties of a distribution with respect to dependency management. Section 4 discusses the feasibility of checking these properties. A few empirical measurements are given in section 5, followed by conclusions in section 6.

2 Basic definitions

Every package management system [DG98, Bai97] takes into account the interrelationships among packages (to different extents). We will call these relationships *requirements*. Several kinds of requirements can be considered. The most common one is a *dependency* requirement: in order to install package P_1, it is necessary that package P_2 is installed as well. Less often, we find *conflict* requirements: package P_1 cannot coexist with package P_2.

Some package management systems specialize these basic types of requirements by allowing to specify the *timeframe* during which the requirement must be satisfied. For example, it is customary to be able to express *pre-dependencies*, a kind of dependency stating that a package P_1 needs package P_2 to be present on the system *before* P_1 can be installed [DG98].

In the following, we assume the distribution and the architecture are fixed. We will identify packages, which are archive files containing metadata and installation scripts, with pairs of a unit and a version.

Definition 1 (Package, unit). *A* package *is a pair (u, v) where u is a unit and v is a version. Units are arbitrary strings, and we assume that versions are non-negative integers.*

While the ordering over version strings as used in common OSS distributions is not discrete (i.e., for any two version strings v_1 and v_2 such that $v_1 < v_2$, there exists v_3 such that $v_1 < v_3 < v_2$), taking integers as version numbers is justified for two reasons. First, any given repository will have a finite number of packages. Second, only packages with the same unit will be compared.

For instance, if our Debian repository contains the versions `2.15-6`, `2.16.1cvs20051117-1` and `2.16.1cvs20051206-1` of the unit `binutils`, we may encode these versions respectively as 0,1 and 2, giving the packages $(\texttt{binutils}, 0)$, $(\texttt{binutils}, 1)$, and $(\texttt{binutils}, 2)$.

Definition 2 (Repository). *A* repository *is a tuple $R = (P, D, C)$ where P is a set of packages, $D : P \to \mathscr{P}(\mathscr{P}(P))$ is the dependency function*[3]*, and $C \subseteq P \times P$ is the conflict relation. The repository must satisfy the following conditions:*

- *The relation C is symmetric, i.e., $(\pi_1, \pi_2) \in C$ if and only if $(\pi_2, \pi_1) \in C$ for all $\pi_1, \pi_2 \in P$.*

[3]We write $\mathscr{P}(X)$ for the set of subsets of X.

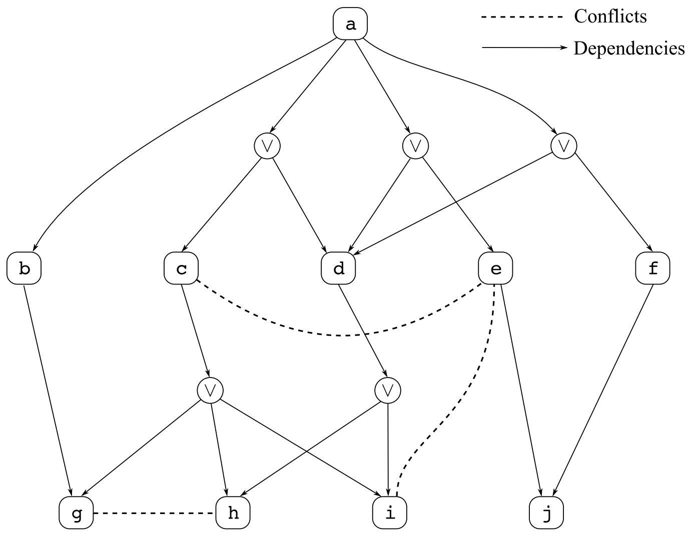


Figure 2: The repository of example 1.

- *Two packages with the same unit but different versions conflict*[4]*, that is, if* $\pi_1 = (u, v_1)$ *and* $\pi_2 = (u, v_2)$ *with* $v_1 \neq v_2$*, then* $(\pi_1, \pi_2) \in C$*.*

In a repository $R = (P, D, C)$, the dependencies of each package p are given by $D(p) = \{d_1, \ldots, d_k\}$ which is a set of sets of packages, interpreted as follows. If p is to be installed, then all its k dependencies must be satisfied. For d_i to be satisfied, at least one of the packages of d_i must be available. In particular, if one of the d_i is the empty set, it will never be satisfied, and the package p is not installable.

Example 1. Let $R = (P, D, C)$ be the repository given by

$$\begin{array}{l} P = \{a, b, c, d, e, f, g, h, i, j\} \\ D(a) = \big\{\{b\}, \{c, d\}, \{d, e\}, \{d, f\}\big\} \\ D(b) = \big\{\{g\}\big\} \quad D(c) = \big\{\{g, h, i\}\big\} \quad D(d) = \big\{\{h, i\}\big\} \\ D(e) = D(f) = \big\{\{j\}\big\} \\ C = \{(c, e), (e, c), (e, i), (i, e), (g, h), (h, g)\} \end{array}$$

where $a = (u_a, 0)$, $b = (u_b, 0)$, $c = (u_c, 0)$ and so on. The repository R is represented in figure 2. For the package a to be installed, the following packages must be installed: b, either c or d, either d or e, and either d or f. Packages c and e, e and i, and g and h cannot be installed at the same time.

In computer science, dependencies are usually *conjunctive*, that is they are of the form

$$a \rightarrow b_1 \wedge b_2 \wedge \cdots \wedge b_s$$

[4]This requirement is present in some package management systems, notably Debian's, but not all. For instance, RPM-based distributions allow simultaneous installation of several versions of the same unit, at least in principle.

where a is the target and b_1, b_2, ... are its prerequisites. This is the case in `make` files, where all the dependencies of a target must be built before building the target. Such dependency information can be represented by a directed graph, and dependencies can be solved by the well-known topological sort algorithm. Our dependencies are of a more complex kind, which we name *disjunctive* dependencies. Their general form is a conjunction of disjunctions:

$$a \longrightarrow (b_1^1 \vee \cdots \vee b_1^{r_1}) \wedge \cdots \wedge (b_s^1 \vee \cdots \vee b_s^{r_s}). \tag{1}$$

For a to be installed, each term of the right-hand side of the implication 1 must be satisfied. In turn, the term $b_i^1 \vee \cdots \vee b_i^{r_i}$ when $1 \leq i \leq s$ is satisfied when at least one of the b_i^j with $1 \leq j \leq r_i$ is satisfied. If a is a package in our repository, we therefore have

$$D(a) = \{\{b_1^1, \ldots, b_1^{r_1}\}, \cdots, \{b_s^1, \ldots, b_s^{r_s}\}\}.$$

In particular, if one of the terms is empty (if $\varnothing \in D(a)$), then a cannot be satisfied. This side-effect is useful for modeling repositories containing packages mentioning another package b that is not in that repository. Such a situation may occur because of an error in the metadata, because the package b has been removed, or b is in another repository, maybe for licensing reasons.

Concerning the relation C, two packages $\pi_1 = (u_1, v_1), \pi_2 = (u_2, v_2) \in P$ conflict when $(\pi_1, \pi_2) \in C$. Since conflicts are a function of presence and not of installation order, the relation C is symmetric.

Definition 3 (Installation). *An* installation *of a repository $R = (P, D, C)$ is a subset of P, giving the set of packages installed on a system. An installation is* healthy *when the following conditions hold:*

- ***Abundance:*** *Every package has what it needs. Formally, for every $\pi \in I$, and for every dependency $d \in D(\pi)$ we have $I \cap d \neq \varnothing$.*
- ***Peace:*** *No two packages conflict. Formally, $(I \times I) \cap C = \varnothing$.*

Definition 4 (Installability and co-installability). *A package π of a repository R is* installable *if there exists a healthy installation I such that $\pi \in I$. Similarly, a set of packages Π of R is* co-installable *if there exists a healthy installation I such that $\Pi \subseteq I$.*

Note that because of conflicts, every member of a set $X \subseteq P$ may be installable without the set X being co-installable.

Example 2. Assume a depends on b, c depends on d, and c and d conflict. Then, the set $\{a, b\}$ is not co-installable, despite each of a and b being installable and not conflicting directly.

Definition 5 (Maximal co-installability). *A set X of co-installable packages of a repository R is* maximal *if there is no other co-installable subset X' of R that strictly contains X. We write $\operatorname{maxco}(R)$ for the family of all maximal co-installable subsets of R.*

Definition 6 (Dependency closure). *The* dependency closure *$\Delta(\Pi)$ of a set of package Π of a repository R is the smallest set of packages included in R that contains Π and is closed under the* immediate dependency *function $\overline{D} : \mathscr{P}(P) \to \mathscr{P}(P)$ defined as*

$$\overline{D}(\Pi) = \bigcup_{\substack{\pi \in \Pi \\ d \in D(\pi)}} d.$$

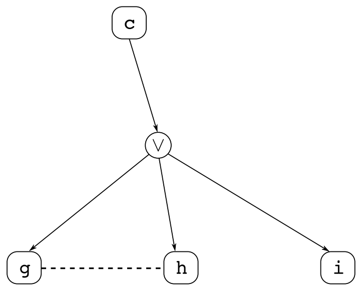


Figure 3: The subrepository generated by package c. The dependency closure is $\{c, g, h, i\}$.

In simpler words, $\Delta(\Pi)$ contains Π, then all packages that appear as immediate dependencies of Π, then all packages that appear as immediate dependencies of immediate dependencies of Π, and so on. Since the domain of $\overline{D}$ is a complete lattice, and $\overline{D}$ is clearly a continuous function, we immediately get (by Tarski's theorem) that such a smallest set exists and can be actually computed as follows:

Proposition 1. *The dependency closure $\Delta(\Pi)$ of Π is:*

$$\Delta(\Pi) = \bigcup_{n \geq 0} \overline{D}^n(\Pi).$$

The notion of dependency closure is useful to extract the part of a repository that pertains to a package or to a set of packages.

Definition 7 (Generated subrepository). *Let $R = (P, D, C)$ be a repository and $\Pi \subseteq P$ be a set of packages. The* subrepository generated by Π *is the repository $R|_\Pi = (P', D', C')$ whose set of packages is the dependency closure of Π and whose dependency and conflict relations are those of R restricted to that set of packages. More formally we have $P' = \Delta(\Pi)$, $D' : P' \to \mathscr{P}(\mathscr{P}(P')), \pi \mapsto \{d \cap P' \mid d \in D(\pi)\}$ and $C' = C \cap (P' \times P')$.*

Figure 3 shows the subrepository generated by the package c of example 1. The dependency closure of c is the set of package nodes of that subrepository.

We then have the following property, which allows to consider only the relevant subrepositories when answering questions of installability.

Proposition 2 (Completeness of subrepositories). *A package π is installable w.r.t. R if and only if it is installable w.r.t. $R|_\pi$. (Similarly for co-installability.)*

3 Maintaining a package repository

The task of maintaining a package repository is difficult: the maintainance team must monitor the evolution of thousand of packages over time, and address the error reports coming from different sources (users, QA teams, developers, etc.). It is desirable to automate as much of this work as possible. Our medium-term goal is to build tools that help distribution maintainers track dependency-related problems in package repositories. We detail here some of the desirable properties of a repository. The first is *history-free*, in that it applies to a given state of a repository.

Being trimmed We say that a repository R is *trimmed* when every package of R is installable w.r.t. R. The intuition behind this terminology is that a non-trimmed repository contains packages that cannot be installed in any configuration. We call those packages *broken*. They behave as if they were not part of the repository. It is obviously desirable that at any point in time, a repository is trimmed, that is, contains no broken packages.

The next properties are *history-sensitive*, meaning that they take into account the evolution of the repository over time. Due to this dependency on time, the precise formulation of these properties is delicate. Just like history-free properties are relevant to users who install a distribution from scratch, history-sensitive properties are relevant to users who upgrade an existing installation.

Monotonicity Let R_t be the repository at time t and consider a coinstallable set of packages C_t. Some users can actually have packages C_t installed simultaneously on their system. These users have the possibility of installing additional packages from R_t, resulting in a coinstallable set of packages C'_t. These users can reasonably expect that they will be able to do so (extend C_t into C'_t) at any future time t', using the repository $R_{t'}$, which, being *newer*, is supposed to be *better* than the old R_t.
Of course, users are ready to accept that in $R_{t'}$ they will not get exactly ${C'}_t$, but possibly ${C'}_{t'}$, where some packages were updated to a greater version, and some others have been replaced as the result of splitting into smaller packages or grouping into larger ones. But, clearly, it is not acceptable to evolve R_t into $R_{t'}$ if R_t allows to install, say, `apache` together with `squid`, while $R_{t'}$ does not.
We say that a repository history line is *monotone* if the *freedom* of a user to install packages is a monotone function of time. Writing $F(x, R)$ for the set of possible package sets in R that are a possible replacement of package x according to the metadata, monotonicity can be formally expressed as

$$\mathrm{Mon}(R) = \forall t < t'.\, \forall P \in \mathrm{Con}(R_t).\, \exists Q \in \mathrm{Con}(R_{t'}).\, \forall x \in P.\, Q \cap F(x, R_{t'}) \neq \varnothing$$

Upgradeability Another reasonable expectation of the user is to be able to upgrade a previously installed package to the most recent version (or even any more recent version) of this package that was added to the repository since her latest installation. She is ready to accept that this upgrade will force the installation of some new packages, the upgrade of some other packages, and the replacement of some sets of package by other sets of packages, as the result of the reorganization of the structure of the packages.
She may even accept, in order to perform an important upgrade, to see some previously installed packages removed, as it happens when using all the meta package management tools available today.

We remark that these properties are *not* interdefinable. We give here a proof of this assertion by exhibiting example repositories showing this independence of the properties. For the first two cases, consider three repositories R_1, R_2, R_3 whose sets of packages are $P_1 = \{(a,1),(b,1),(c,1)\}$, $P_2 = \{(a,1),(b,1)\}$, $P_3 = \{(a,1),(a,2),(b,1)\}$ with no conflicts nor dependencies among the version 1 packages and a conflict among $(a,2)$ and $(b,1)$. Notice that at each moment t in time, R_t is trimmed.

1. A repository that stays trimmed over a period of time is not necessarily monotone, nor upgradeable. Since $(c,1)$ disappears between times 1 and 2, this step in the evolution does not preserve monotonicity. Since $(a,2)$ has a new conflict (namely with $(b,1)$) in R_3, the evolution from R_2 to R_3 does not preserve upgradeability.

2. A repository that stays trimmed over a period of time and evolves in a monotone fashion is not necessarily upgradeable. The evolution from R_2 to R_3 above is monotone, each of R_2 and R_3 is trimmed, but we fail upgradeability because there is no way of going from $\{(a,1),(b,1)\}$ to $\{(a,2),(b,1)\}$ because of the conflict.

3. A repository that stays trimmed over a period of time and is upgradeable is not necessarily monotone.

   Consider repositories R_1 and R_2 with $P_1 = \{(a,1),(b,1)\}$ and $P_2 = \{(a,2),(b,1)\}$. Assume $(a,1)$ and $(b,1)$ are isolated packages, while $(a,2)$ conflicts with $(b,1)$. Now, a user having installed all of R_1 and really willing to get $(a,2)$ can do it, but at the price of giving up $(b,1)$. This evolution of the repository is therefore upgradeable but not monotone.

4. A repository that evolves in a monotone and upgradeable fashion is not necessarily trimmed at any time: indeed, the monotonicity and upgradeability property only speak of *consistent* subsets of a repository, that cannot contain, by definition, any broken packages.

   Consider for example repositories R_1, R_2 with $P_1 = \{(a,1)\}$, $P_2 = \{(a,1),(b,1)\}$. Assume $(a,1)$ and $(b,1)$ are broken because they depend on a missing package $(c,1)$. Here, the evolution of R_1 to R_2 is trivially monotone and upgradeable, because there is *no* consistent subset of R_1 and R_2, and both R_1 and R_2 are not trimmed because they contain broken packages.

The examples above to prove that the three properties are actually independent may seem contrived, but are simplifications of real-world scenarii. For instance, example 3 can actually happen in the evolution of real repositories, when for some reason the new version of a set of interrelated packages is only partially migrated to the repository. Many packages are split into several packages to isolate architecture-independent files, as in the Debian packages `swi-prolog` and `swi-prolog-doc`. When performing this split, it is quite natural to add a conflict in `swi-prolog-doc` against old, non-splitted versions of `swi-prolog`. If the new version of `swi-prolog-doc` slips into a real repository before the new, splitted version of `swi-prolog`, we are exactly in situation number 3 above.

Package developers seem aware of some of these issues: they actually do their best to ensure monotonicity and upgradeability by trying to reduce as much as possible the usage of conflicts, and sometime resorting to naming conventions for the packages when a radical change in the package happens, like in the case of `xserver-common` vs. `xserver-common-v3` in Debian, as can be seen in the dependencies for `xserver-common`.

```
Package: xserver-common
Conflicts: xbase (<< 3.3.2.3a-2), xsun-utils, xbase-clients (<< 3.3.6-1),
  suidmanager (<< 0.50), configlet (<= 0.9.22),
  xserver-3dlabs (<< 3.3.6-35), xserver-8514 (<< 3.3.6-35),
  xserver-agx (<< 3.3.6-35), xserver-common-v3 (<< 3.3.6-35),
  xserver-fbdev (<< 3.3.6-35), xserver-i128 (<< 3.3.6-35),
  xserver-mach32 (<< 3.3.6-35), xserver-mach64 (<< 3.3.6-35),
  xserver-mach8 (<< 3.3.6-35), xserver-mono (<< 3.3.6-35),
  xserver-p9000 (<< 3.3.6-35), xserver-s3(<< 3.3.6-35),
  xserver-s3v (<< 3.3.6-35), xserver-svga (<< 3.3.6-35),
  xserver-tga (<< 3.3.6-35), xserver-vga16 (<< 3.3.6-35),
  xserver-w32 (<< 3.3.6-35), xserver-xsun (<< 3.3.6-35),
  xserver-xsun-mono (<< 3.3.6-35), xserver-xsun24 (<< 3.3.6-35),
  xserver-rage128, xserver-sis
```

4 Algorithmic considerations

Our research objective within the EDOS project is to formally define the desirable properties of repositories stated in section 3 (and possibly other properties that will appear useful), and to develop efficient algorithms to check these properties automatically.

It is really not evident that any of these problems are actually tractable in practice: due to the rich language allowed to describe package dependencies in the mainstream FOSS distributions, even the simplest problems (checking installability of a single package) may involve verifications over a large number of other packages. During our first investigations of these problems, we have indeed already proven the following complexity result.

Theorem 1 (Package installability is an NP-complete problem). *Checking whether a single package P can be installed, given a repository R, is NP-complete.*

The full proof of this result will be published separately. It relies on a simple, polynomial-time reduction of the 3SAT problem to the installability problem. Given an instance of 3SAT, a repository is constructed having one package for the whole 3SAT formula, one package per clause of that formula, and three packages for each propositional atom occurring in that formula. Dependencies and conflicts between these packages are added in such a way that the package for the whole formula is installable if and only if the 3SAT formula is satisfiable.

Nevertheless, this strong limiting result does not mean that we will not be able to decide installability and the other problems in practice: the actual instances of these problems, as found in real repositories, could be quite simple in the average.

In particular, the converse of the reduction used for the NP-completeness proof leads to an effective way of deciding package installability. We developed an algorithm that encodes a repository R and its dependencies as a Boolean formula $C(R)$. (Details of the encoding will be published in a forthcoming paper.) Assignments of truth values to boolean variables that satisfy $C(R)$ are in one-to-one correspondence with sets of co-installable packages. Therefore, a package P is installable if and only if the Boolean formula $C(R) \wedge P$ is satisfiable, which we can check relatively efficiently using off-the-shelf SAT solving

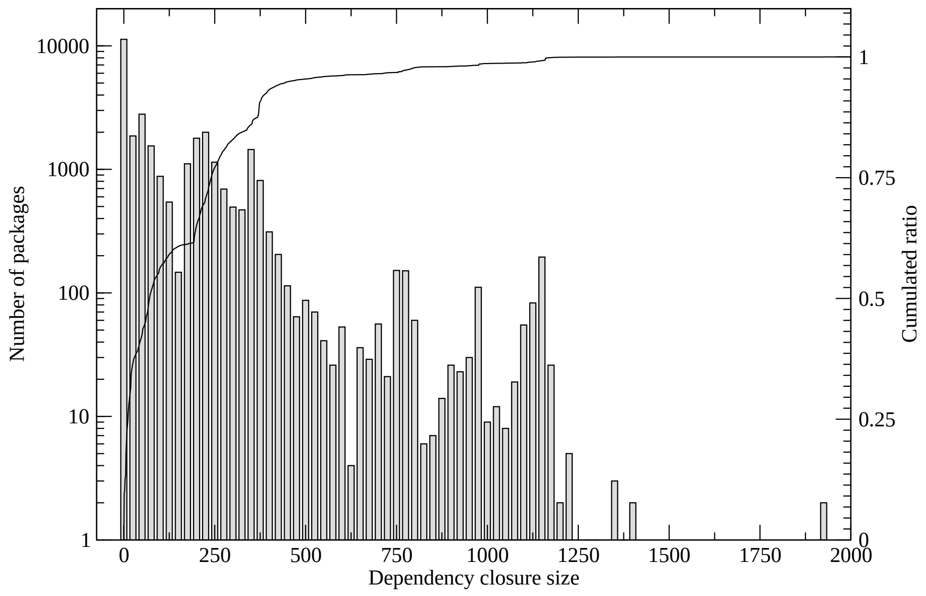


Figure 4: Number of packages as a function of the size of their dependency closures.

technology.

We implemented the conversion algorithm as well a SAT solver [ES04] and ran it over both the Debian pool (over 30,000 packages) and the Mandriva Cooker distribution (around 5,000 packages). The execution time is entirely acceptable, and the tool found a number of non-installable packages in both distributions.

We are now focusing our attention on the two time-dependent desirable properties for the repositories, which are, algorithmically speaking, much harder.

5 Empirical measurements

In parallel with our formal complexity and algorithmic investigations, we also performed some empirical measurements on the Debian and Mandriva distributions, to try and grasp the practical complexity of the problems.

Figure 4 gives a histogram showing the number of packages as a function of the size of the dependency closure, from the Debian stable, unstable and testing pools on 2005-12-13, which has 31149 packages. The average closure size is 158; 50% have a closure size of 71 or less, 90% of 1077 or less. These numbers show that naive combinatorial algorithms, exponential in the size of the dependency closure, are clearly out of the question.

Figure 5 estimates the complexity of solving the Boolean formulae generated by our encoding of the installability problem. The “temperature” T of a formula in 3SAT conjunctive normal form is defined as $T = m/n$ where m is the number of clauses and n the number of variables. There is strong theoretical

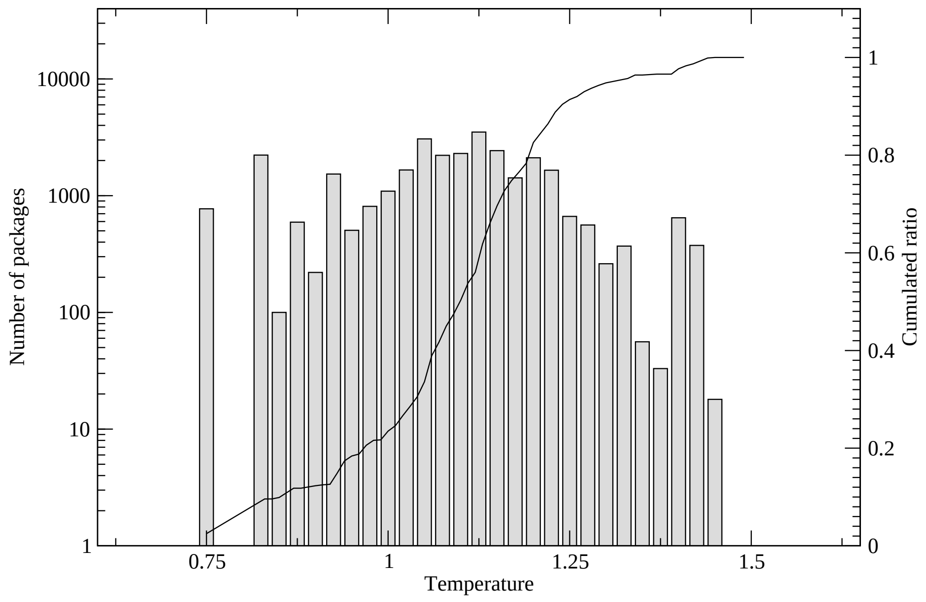


Figure 5: Number of packages as a function of the “temperature” of the SAT problems corresponding to their installability problems.

and practical evidence that hard SAT problems have a temperature close to 4.2, while SAT problems with temperatures well below or above that limit are easier to solve. The temperatures for the SAT problems corresponding to installability of the Debian packages range from 0.75 to 1.49, well below the threshold value of 4.2. This result confirms that we are dealing with relatively easy satisfiability problems, maybe owing to the small-world nature of the dependency graphs [LW05].

6 Conclusions

We have presented and motivated in this paper three fundamental properties for large repositories of FOSS packages that are quite different from the usual properties of component collections, due to the large spectrum of languages, technologies, frameworks and interfaces spanned by a contemporary FOSS distribution.

Despite their algorithmic complexity, we have already performed large-scale tests indicating that the first of these properties can be mechanically checked in reasonable time. We continue similar investigations on the other properties.

We claim that providing efficient tools to check these properties is an essential step in order to ensure that the FOSS development model stays sustainable, and we suggest that researchers should look into the specificities brought by FOSS in the software engineering world.

References

[Bai97] Edward C. Bailey. Maximum RPM, taking the Red Hat package manager to the limit. http://rikers.org/rpmbook/,http://www.rpm.org, 1997.

[BD02] Manfred Broy and Ernst Denert. *Software Pioneers: Contributions to Software Engineering*. Springer-Verlag, 2002.

[DG98] Debian Group. Debian policy manual. http://www.debian.org/doc/debian-policy/, 1996–1998.

[Ekl05] David Eklund. The lib update/autoupdate suite. http://luau.sourceforge.net/, 2003–2005.

[ES04] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2004.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[LW05] Nathan LaBelle and Eugene Wallingford. Inter-package dependency networks in open-source software. *Submitted to Journal of Theoretical Computer Science*, 2005.

[Man05] Mandriva. URPMI. http://www.urpmi.org/, 2005.

[Nie05] Gustavo Niemeyer. Smart package manager. http://labix.org/smart/, 2005.

[Sil04] Gustavo Noronha Silva. Apt-howto. http://www.debian.org/doc/manuals/apt-howto/, 2004.

[Szy97] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley Professional, 1997.

[TT01] L. Taylor and L. Tuura. Ignominy: a tool for software dependency and metric analysis with examples from large HEP packages. In *Proceedings of CHEP'01*, 2001.

[Tuu03] L. A. Tuura. Ignominy: tool for analysing software dependencies and for reducing complexity in large software systems. In *Proceedings of the VIII International Workshop on Advanced Computing and Analysis Techniques in Physics Research*, volume 502, pages 684–686, 2003.

[vdS04] Tijs van der Storm. Variability and component composition. In *Proceedings of the Eighth International Conference on Software Reuse (ICSR-8)*, 2004.