ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, pp 199–208.

Managing the Complexity of Large Free and Open Source Package-Based Software Distributions*

Fabio Mancinelli, Jaap Boender, Roberto di Cosmo, Jérôme Vouillon
PPS - Université Paris VII, France
FirstName.LastName@pps.jussieu.fr†

Berke Durak, Xavier Leroy
INRIA Rocquencourt
France
FirstName.LastName@inria.fr‡

Ralf Treinen
LSV, ENS de Cachan, CNRS UMR 8643
& INRIA Futurs, France
treinen@lsv.ens-cachan.fr

Abstract

The widespread adoption of Free and Open Source Software (FOSS) in many strategic contexts of the information technology society has drawn the attention on the issues regarding how to handle the complexity of assembling and managing a huge number of (packaged) components in a consistent and effective way. FOSS distributions *(and in particular* GNU/Linux*-based ones) have always provided tools for managing the tasks of installing, removing and upgrading the (packaged) components they were made of. While these tools provide a (not always effective) way to handle these tasks on the client side, there is still a lack of tools that could help the distribution editors to maintain, on the server side, large and high-quality distributions. In this paper we present our research whose main goal is to fill this gap: we show our approach, the tools we have developed and their application with experimental results. Our contribution provides an effective and automatic way to support distribution editors in handling those issues that were, until now, mostly addressed using ad-hoc tools and manual techniques.*

1 Introduction

Component based software development [21] has been a very discussed topic in the Software Engineering domain for a long time. Assembling complex systems by integrating independently developed components has become a common practice nowadays. Components are either developed in-house or collected from a common marketplace where third-party organizations offer their products (e.g, commercial-off-the-shelf, COTS, components)

Maintaining a component based system and handling its evolution have always been a difficult task. The main difficulty resides in the fact that a complex system has many relationships, implicit or explicit, among components (i.e. *dependencies*). These relationships could be easily broken when performing standard life cycle management operations on components (i.e., component installation, removal and upgrading), leading to unusable and corrupted systems.

The widespread adoption of Free and Open Source Software (FOSS), in some way, has emphasized these problems. The traditional, centralized and closed way in which software components were developed has changed radically. Now they are developed by heterogeneous entities or individuals, not belonging to any company, who are sharing their work thanks to fast Internet connections and communication services. These components, that are licensed under all sorts of FOSS licenses [15], can be reused and redistributed without any formal agreements between companies and can be adapted to different contexts more easily, because of the fact that their source-code is available. The result is a more free and agile marketplace (i.e., the so-called *Bazaar* [16]) where there is a higher number of usable components that can be used to build systems in many original and often unforeseen ways.

The most manifest examples of complex systems built with FOSS components are *GNU/Linux*-based distributions. They provide complete UNIX-like operating systems by selecting and assembling a limited set of suitable components,

*This work was supported by the EDOS Specific Targeted Research Project of the 6th European Union Framework Programme.

†{Fabio.Mancinelli,Jaap.Boender, Roberto.Dicosmo, Jerome.Vouillon}@pps.jusseiu.fr

‡{Berke.Durak, Xavier.Leroy}@inria.fr

retrieved from the FOSS bazaar. Currently, there are more than 100 such distributions [3], each one targeted for a particular audience (e.g. end-users, server administrators, developers) or tailored in order to meet a given set of requirements (e.g., user-friendliness, smallest footprint, server-side optimizations)

GNU/Linux-based distributions are created and managed by a novel figure, whose role has no comparable counterpart in the traditional software development model: the *distribution editor*. She is an entity that takes care of collecting available FOSS components and builds from them a coherent and usable system. Typically, she monitors components at their original source (*upstream tracking*), recompiles and tests them against the distribution (*integration*), resolves and keeps track of all the needed relationships (*dependency management*)

Despite the heterogeneity of the large number of available distributions, many of them share the same basic foundations: they are *package-based*. They use a basic deployment unit for component distribution (i.e., the *package*) and a tool, the *package manager*, that is able to handle these packages and to correctly manage their life cycle (i.e., installation, removal and upgrade).

While *package managers* provide a (not always effective) way to handle these tasks on the user side, and could be used to help the distribution editors in tracking problems when assembling their products, there is still a lack of specific tools that could help them in maintaining large and high-quality distributions.

This is a big issue because a typical *GNU/Linux-based* distribution is made of several thousands of packages that have even more dependency relationships among them [12].

Until now, the maintenance and the evolution of the package repository underlying a distribution (i.e., the collection of all the packages that are available and can be "used" with a given distribution) have been done by ad-hoc techniques, relying on semi-automatic tools and, above all, bug reports filed by distribution users. Of course, this is clearly insufficient when dealing with huge package bases.

In this paper we present some of the results of our ongoing research that aims at providing a set of tools for the *distribution editor* that helps him in maintaining huge package bases and improving the quality of software distributions built on them, by detecting errors and inconsistencies in an effective, fast and automatic way.

The paper is structured as follows: in Section 2 we present and detail some package formats and their associated metadata that, for historical reasons, have become a *"de facto"* standard and are the basis of many *GNU/Linux-based* distributions. In particular we detail the information relevant to this paper explaining the relationships that can be specified among packages. In Section 3 we present the formalization of our model for package bases and the techniques we have used to reason on it. In Section 4 we describe our framework and its tools, and in Section 5 we show the results we have gathered from the analysis of some very popular *GNU/Linux-based* distributions. Finally in Section 6 we discuss related work and in Section 7 we draw conclusions and we sketch a road-map for the future work.

2 Package Metadata Overview

In this section we briefly introduce some package formats. In particular we focus on the ones that have been recognized as the *"de facto"* standard with respect to package-based GNU/Linux distributions: the DEB [23] and the RPM [6] formats. Though the actual (binary) formats of DEB and RPM packages are different, they have a lot of commonalities. In the following we describe the features that are relevant with respect to topic of this paper, i.e., dependency specification. Where not explicitly stated, these features are to be intended the same in both DEB and RPM package systems.

A *package* is a binary bundle that contains a component[1], all the data needed to its correct functioning and some metadata which describe its attributes and its requirements with respect to the environment in which it will be deployed.

Both DEB and RPM packages are actually compressed archives. However, while RPM is actually an ad-hoc format explicitly conceived for this purpose, DEB packages can be produced using standard tools (i.e., `ar` and `tar`), so they can be easily managed. Nevertheless, the most relevant difference between DEB and RPM package format concerns their metadata specification. While RPM packages encode it in a binary form as a part of its ad-hoc archive format, DEB packages use a textual representation for it. This fact makes its processing easier. Figure 1 shows an excerpt of the metadata taken from the `firefox`[2] DEB package.

Every package has a version that is used to give a temporal order to the packaged component release. Though the version structure of RPM and DEB packages is the same (i.e., `epoch` number, `version` id and `revision` id), the comparison algorithm is slightly different. Version information is central when specifying relationships between packages.

The kinds of relationships that can be expressed in RPM and DEB package metadata are almost the same, except some very specialized ones that are seldom used or processed (e.g., `obsoletes`, `replaces`, `suggests`, `enhances`)

[1] Packages not always contain "executable" components. Often documentation and non-executable artifacts are packaged as well. However, to our purposes, this distinction is not relevant because the package format is agnostic with respect to its actual content type

[2] `firefox_1.5.dfsg+1.5.0.1-2_i386.deb`

```
Package: firefox
Version: 1.5.dfsg+1.5.0.1-2 ...
Depends: fontconfig, psmisc,
 libatk1.0-0 (>= 1.9.0), libc6 (>= 2.3.5-1) ...
Suggests: xprint, firefox-gnome-support
 (= 1.5.dfsg+1.5.0.1-2), latex-xft-fonts
Conflicts: mozilla-firefox (<< 1.5.dfsg-1)
Replaces: mozilla-firefox
Provides: www-browser...
```

Figure 1. Excerpt of a DEB package metadata

In this paper we have taken into account only the three main relationships that are used with the same semantics in both RPM and DEB packages:

- `Depends` (DEB), `Requires` (RPM): used to establish a requirement on the packages that must be present in the system in order to make the current packaged component fully functional.
- `Conflicts` (DEB, RPM): used to establish a requirement on the packages that cannot be present at the same time in the system with the current one. A successful installation can be performed only if no conflicting packages are already present in the system.
- `Pre-Depends` (DEB), `PreReq` (RPM): similar to the `Depends` relationships but used to establish a requirement on the packages that must be already present in the system in order to successfully deploy the packaged component. The difference between `Pre-Depends` and `Depends` is that while `Depends` package might not be present in the system when deploying the packaged component (but only after, so they can be deployed *together* with the current component), `Pre-Depends` packages must be already installed even before attempting to deploy the current packaged component.

Relationships are specified by using a list of package names, optionally with version constraints. Each element in the list represents a requirement for the given relationship, and all these requirements must be met (i.e., they are a conjunction) in order to satisfy the dependency relationship. Actually, the DEB package format allows an element of the list to be a disjunction of requirements. This is done by using the `|` operator. In this case, in order to meet the (disjunctive) requirement it is sufficient that at least one of the constituting requirements is met.

Figure 1 shows that the package `firefox` version `1.5.dfsg+1.5.0.1-2` has, among the others, a `Depends` requirement on the package `fontconfig` and on a package `debianutils` with a version $>=$ `1.16`. This means that once the packaged component `firefox` is deployed, in order to correctly function, it will need the `debianutils` component with a version greater than or equal to `1.16` to be deployed as well. When no version constraint is specified, then any packaged component version will do.

The allowed version constraints can be specified by using only the standard comparison operators with their standard semantics: $<$, $<=$, $=$, $>=$ and $=$.

```
Depends: libc6 (>= 2.0),
 xlibs (>= 4.0) | xlib6g (>= 3.3.3.1), ...
```

Figure 2. DEB disjunctive dependency requirement

Figure 2, on the other hand, shows an example of a disjunctive dependency requirement. In order to correctly function, the current packaged component needs `libc6` with a version $>=$ `2.0` and *either* `xlibs` with a version $>=$ `4.0` *or* `xlib6g` with a version $>=$ `3.3.3.1`.

Another essential information is given by the `Provides` metadata. When specified it allows to declare an identifier that can be used to reference the package. Such an identifier is often called either *virtual package* or *feature*. For instance, in Figure 1 the `firefox` package `Provides` the identifier `web-browser`. It is important to point out that this identifier can be used when specifying dependency relationships. So, for example, another package could declare a dependency requirement by specifying `web-browser`. In this case this requirement should be considered as a disjunctive requirement whose elements are *all* the packages that provide `web-browser`.

In RPM metadata, these *features* are the only way of declaring a relationship between packages; it is not possible to declare a relationship directly between packages, only by way of features. Also, it is worth noting that while RPM does not allow disjunctive dependencies per se, they can be emulated by declaring a dependency on a feature that is provided by two or more packages.

DEB also allows *virtual packages*, but their usage is strictly limited to a few cases (there is an explicitly maintained list of virtual packages) and using direct relations between packages is the standard practice.

3 Formalization

In this section we formalize the definitions we have described in Section 2 and we introduce the notion of *package installation* that identify the central property we want to guarantee for each package present in a package repository.

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.*

Now we have two choices: either we try to represent formally the version constraints used in the different package formats, and then our model will need to take into account relationships between versions like the $>=, =, >>, <<, <=$ found in the Figure 1; or we keep our model simple and eliminate all version constraints but one, equality (==), by replacing all the other version constraints with the explicitly listing of all packages versions that satisfy the version constraints (we call such replacement an *expansion*). This second choice has the additional advantage that the model becomes independent on the specific version comparison operators, present and future, of any given package format, but of course such expansions are relative to a given repository, so any change to the repository will make it necessary to perform the expansion phase again.

As an example, consider a repository containing versions `0.9`, `1.0` and `1.1` of mozilla-firefox. Then the conflict relationship in Figure 1 would expand as shown in Figure 3.

```
Package: firefox
Version: 1.5.dfsg+1.5.0.1-2
Conflicts: mozilla-firefox (==0.9),
 mozilla-firefox (== 1.0),
 mozilla-firefox (== 1.1)...
```

Figure 3. Dependencies expansion

Once this expansion has been performed, we are left only with direct dependency or conflict relationships among actual packages, i.e. $(unit, version)$ pairs, and we can then straightfowardly encode them it into a graph.

In the following we will hence assume that the distribution is fixed, and that the version comparison operators have been expanded. Notice that, due to this assumption, we can also get rid of all the details concerning the particular ordering over version strings used in common FOSS packages: we use the specific version comparison algorithm only to expand the dependencies, and then we can use any set of unique identifiers for representing each version[3].

For instance, if the repository of our Debian distribution 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 $(\texttt{binutils}, 2)$.

[3]Notice, for example, that the version numbering schemas used in DEB and RPM packages are 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 $v_1 < v_3 < v_2$), and representing such property explicitly would be cumbersome.

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*[4]*, 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* $\pi_1, \pi_2 \in P$*.*
- *Two packages with the same unit but different versions conflict*[5]*, 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 dependency requirements 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.

The attentive reader might have noticed that we do not have a separate dependency function for modeling `Pre-Depends` relationships. In fact, we consider them as if they were normal `Depends` relationships. This is reasonable because we put ourselves on the *distribution editor*-side where we have a package repository (and not an installed system). At this level, the two kinds of relationships can be considered equivalent.

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 4. 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.*

As described in Section 2, dependencies are usually *conjunctive*, that is they are of the form

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

where a is the target and b_1, b_2, ... are its requirements. However, more complex dependencies can be specified,

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

[5]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(if, for example, they do not install files in the same location).

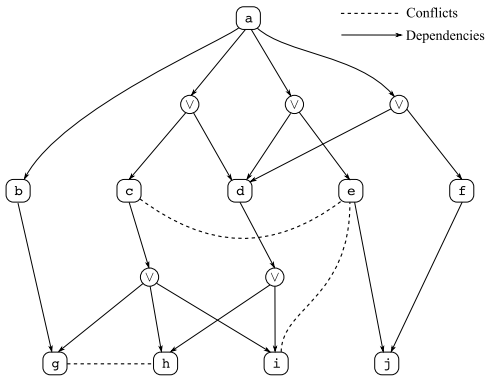

Figure 4. The repository of Example 1.

with name *disjunctive* requirements. So the general form for a dependency specification is a conjunction of disjunctions:

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

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 I 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 c, b 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 (Dependency closure) *The* dependency closure $\Delta(\Pi)$ *of a set of packages 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*

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

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:

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 6 (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 5 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.

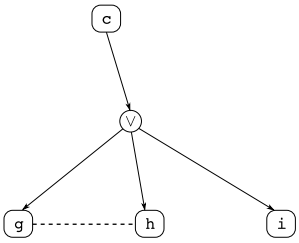

Figure 5. The subrepository generated by package c. The dependency closure is $\{c, g, h, i\}$.

Proposition 1 (Completeness of subrepositories) *A package π is installable w.r.t. R if and only if it is installable w.r.t. $R|_\pi$.*

The desirable property that we want to ensure over a repository R is the following:

Definition 7 (Trimmed repository) *A repository R is* trimmed *if every package $\pi \in R$ is installable with respect to R itself.*

The intuition is that if a repository is not *trimmed*, then it contains packages that cannot be installed in any configuration: these packages behave as if they were not part of the repository. This means that either they should not be actually there or there is some a problem in their metadata specification that should be corrected.

In the following sections we show how we have formalized the Installability problem using two approaches: Constraint Programming and SAT.

3.1 Encoding the Installability problem as a SAT problem

The formalization of installability provided above leads quite naturally to an encoding as a boolean satisfiability problem. Each package becomes directly a boolean variable whose value indicates whether that particular version of a unit is installed or not. The constraint relationships become immediately boolean formulae over the package variables, using just the logical connectives $\rightarrow, \wedge, \vee$. A package is installable if and only if the corresponding boolean formula is satisfiable, as can be checked using any SAT solver.

3.2 Encoding the Installability problem as a CP problem

We can also formulate the installability problem for a given package in a Debian repository R as a CP problem over finite domains, but in this case we must start from the repository *before* expanding the version relationships.

To simplify the problem by getting rid of the inessential details related to version comparison algorithms in DEB or RPM formats, we first preprocess the repository and replace version strings by integer as follows: for each unit u, collect all of its mentioned version strings v, and order them accordingly to the appropriate, format specific, comparison algorithm; then replace each occurrence of $u\ op\ v$ by $u\ op n_v$, where n_v is the position of v in the increasingly ordered sequence of versions of u. In other terms, we simply *project over an initial segment of the integers starting at 1* the order structure of the versions of each package. This does not change the nature of the constaint problem, but reduces it to a problem over the Integer domain, for which solvers are more easily available.

We then build a constraint satisfaction problem over a finite domain by constructing a set of constraints R_c out of R as follows:

Variables and domains: For each unit u in the repository R, we introduce a finite domain variable, with domain equal to the set of available versions of the unit present in the repository, plus one special value 0 denoting the fact that no version of the unit is installed. We add the constraint $X_u \in \{0, v_1, \ldots, v_k\}$ to R_c.

Constraints We add constraints to R_c that encode the dependency information associated to each package $\pi = (u, v) \in R$ as follows. If R contains a dependency for $\pi = (u, v)$ of the form

$$Depends : (u_1^1\ op_1^1\ v_1^1 \vee \cdots \vee u_1^{r_1}\ op_1^{r_1}\ v_1^{r_1}) \wedge \cdots \wedge (u_s^1\ op_s^1\ v_s^1 \vee \cdots \vee u_s^{r_s}\ op_s^{r_s}\ v_s^{r_s}).$$

we introduce the constraint

$$(X_u = v) \Rightarrow (X_{u_1^1}\ op_1^1\ v_1^1 \vee \cdots \vee X_{u_1^{r_1}}\ op_1^{r_1}\ v_1^{r_1}) \wedge \cdots \wedge (X_{u_s^1}\ op_s^1\ v_s^1 \vee \cdots \vee X_{u_s^{r_s}}\ op_s^{r_s}\ v_s^{r_s})$$

If R contains a conflict for $\pi = (u, v)$ of the form

$$Conflicts : (u_1^1\ op_1^1\ v_1^1 \vee \cdots \vee u_1^{r_1}\ op_1^{r_1}\ v_1^{r_1}) \wedge \cdots \wedge (u_s^1\ op_s^1\ v_s^1 \vee \cdots \vee u_s^{r_s}\ op_s^{r_s}\ v_s^{r_s}).$$

we introduce the constraint

$$(X_u = v) \Rightarrow (X_{u_1^1}\ compl(op_1^1)\ v_1^1 \wedge \cdots \wedge X_{u_1^{r_1}}\ compl(op_1^{r_1})\ v_1^{r_1}) \vee \cdots \wedge (X_{u_s^1}\ compl(op_s^1)\ v_s^1 \wedge \cdots \wedge X_{u_s^{r_s}}\ compl(op_s^{r_s})\ v_s^{r_s})$$

where $compl(op)$ is the operation opposite to op (e.g., $compl(>>)$ is $<=$, etc.)

Notice that, in the encoding above, if we encounter a package name with no version constraint (so we find just `u` instead of `u >> 3`, for example), we simply produce $X_u > 0$ as the encoding. It is now possible to prove the following:

Proposition 2 *A package $\pi = (u, v)$ is installable in the repository R if and only if the constraint $X_u == v$ is compatible with R_c.*

Hence, to check installability of a package u, v in a repository R, we can pass the constraint set R_c to any CP solver and ask whether $X_u = v$ is satisfiable. We can also simply ask whether there exist a version of a unit that is installable, by asking whether $X_u > 0$ is satisfiable.

4 The framework

We have developed a framework that can be used by a *distribution editor* to asses the quality of its distribution and to track problems concerning the underlying package repository (i.e., broken packages in non trimmed repositories). Actually we have two distinct sets of tools (Figure 6): a set of independent elements (i.e.,the *toolchain*) that executes the analysis by incrementally processing the data collected from a repository; and A very specialized tool that does the analysis in one step.

The rationale for having these two sets of tools is that we want to be able to use the data for other kind of analyses as well. Having a modularized toolchain enables us to reuse part of it even for other purposes. On the other hand, having a specialized, small and efficient tool that performs this particular kind of analysis is good for those *distribution editors* that do not need other kind of features.

Moreover, having plenty of independent tools that execute the same kind of analysis allowed us to validate the whole approach by comparing the results obtained from these different sources.

In the following we describe each element of our framework.

4.1 Ceve

Ceve, the first element of the toolchain, is a generalized package parser. It can read several package formats (most importantly RPM and DEB packages and distributions), but also the XML rpm-metadata format [24]), and output their metadata in several different formats, of which the most important is the EGraph format described in the next section. Ceve is written in OCaml, and it uses the CDuce [22] language for XML input and output.

The function of Ceve in the toolchain is that of parsing package metadata from all sorts of formats into one common format (the EGraph format). Since there are many differences between the various package formats, Ceve cannot simply read metadata and output them; the data has to be manipulated. The most notable example of this is RPM dependencies; RPM cannot declare dependencies on other packages directly, but only by way of features, as explained in section 2. Ceve can resolve these indirect dependencies, so that if package A declares a dependency on feature F, and feature F is provided by packages B and C, in the output package A will need either package B or package C. Also, packages that install different files in the same location conflict with each other, but this is not explicitly declared; Ceve can explicitly add these conflicts to the output.

4.2 The EGraph file format

The *EGraph* file format is a package-format agnostic representation of the information that is encoded in the packages stored in a package repository. It is based on the XML [9] language and in particular it complies with the *GraphML* [8] specification. This intermediate format has been conceived in order to have a common and uniform representation of the metadata that can be used as input by all the tools of our framework, without having to develop a filter for each package format.

The *GraphML* format is well suited for this purpose because the dependency relationships among packages are easily represented by using graphs and, moreover, it can be extended in order to accommodate other significant metadata.

The *EGraph* format is an effort for proposing a new metadata format that can complement and extend the existing metadata specified at the package level in order to perform more complicated and effective checking on package repositories [10].

4.3 EDOSLib

EDOSLib is a library which provides the foundation of some tools that are used in the framework. In particular *EDOSLib* implements: an object model for representing package repositories information and their structure; a set of functionalities to explore manage the package repository struture (e.g., extracting subrepositories and dependency closures); *EGraph* input/output functionalities.

4.3.1 ProblemGenerator

ProblemGenerator is an *EDOSLib*-based preprocessor that takes as input the *EGraph* format representing a package repository and gives an encoding of the installability problem in order to verify if the repository is trimmed or not. In particular *ProblemGenerator* performs three steps: i) Extract the package subrepository underlying the dependency closure for each package that has to be analyzed. ii) Map the standard package version numbers to integers, as mentioned in Section 3; iii) Expands all the reference to virtual packages by substituting them with actual package names and versions.

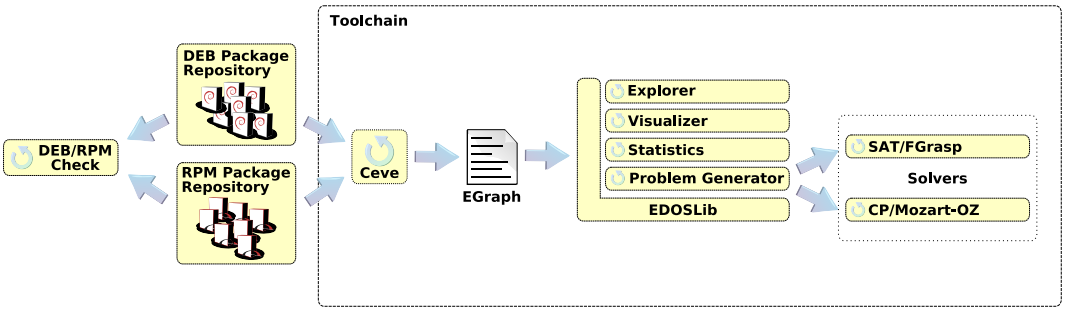

Figure 6. The framework

The output is in the format suitable to be processed by the solvers that will perform the actual verification.

4.3.2 Explorer, Statistics and Visualizer

These are three complementary tools that can be useful for exploring the package repository, giving a help in navigating through the intrinsic complexity of it. In particular *Explorer* offers a command line interface to the functionalities provided by *EDOSLib* (e.g., dependency closures extraction, package metadata exploration, etc.). Statistics extract some useful metric with respect to the complexity of the package repository (e.g., number of dependencies, average closure sizes, etc.). Finally, *Visualizer* is a graph visualization tool that allows the distribution editor to visually navigate the package repository through its graphical representation. Even though these tools are not really part of the analysis engine, they provide a useful aid when it comes to navigate the package repository in order to track down the problems discovered by the actual analysis.

4.4 SAT/FGrasp Solver

The *SAT/FGrasp* solver takes as input the encoding of an installability problem as produced by *ProblemGenerator* and translates it to a boolean formula as outlined in section 3.1. After conversion to conjunctive normal form, the formula is fed through the FGrasp SAT solver from T. U. Lisbon. This solver can return a minimal set of assignments that satisfy the formula, which the *SAT/FGrasp* tool translates back into minimal lists of packages that need to be installed to enable the installation of the initial package. The SAT solving is reasonably efficient, taking less than 1 second on the hardest installability problems.

4.5 CP/Mozart-Oz Solver

The *CP/Mozart-Oz Solver* translates the output of *ProblemGenerator* to a CP problem as described in section 3.2, then solves it using a solver written in the Mozart-Oz language [5] . The solver uses a custom branch-and-bound strategy programmed in Oz itself. While effective on small to medium-sized installability problems, the solver tends to exhibit exponential divergence on large problems.

4.6 DEB/RPMCheck

The `debcheck` and `rpmcheck` utilities are separate from the toolchain. They take as input a package repository and check whether one, several or all packages in the repository are installable with respect to that repository. Both utilities are based on the SAT encoding of section 3.1 and exploit a customized Davis-Putnam SAT solver [11]. Since all computations are performed in-memory and some of the encoding work is shared between all packages considered, the `debcheck` and `rpmcheck` are significantly faster than the *SAT/FGrasp* tool for checking the installability of all packages of a repository.

5 Experimental results

In this section we show some experimental results that we have gathered by analyzing with our tools two of the most famous *GNU/Linux*-based distributions: Debian GNU/Linux [2] and Mandriva Linux [4].

The reference package repositories are a snapshot of the complete Debian pool located at `http://ftp.debian.org/pool` and the packages distributed with the Mandriva 2006 Edition.

Out of the 4211 packages in the main part of the Mandriva 2006 distribution, 42 are not installable. In 38 cases, this is due to a dependency that is not available; in the case of four packages, `mozilla-thunderbird-enigmail-{de,es,fr,it}`, the problem is a simultaneous dependency and conflict with the package `mozilla-thunderbird-enigmail`.

The Debian pool snapshot contains 34701 packages, described in a Debian Control File [23] of almost 30Mb of metadata defining, among other things, almost 200.000 package relationships (dependencies and conflicts). Inserting new information that can possibly break the consistency

of the pool and finding what is the source of the problem that lead to that is a difficult task if not supported by automatic tools for the analysis.

Figure 7 shows an excerpt of the output generated by our `debcheck` tool after the analysis of the previously described Debian pool snapshot. The result of this analysis is that for 123 packages there are no possible way to install them. In particular, 111 of them are not installable because of a missing dependency (i.e., there is no package available in the pool that could satisfy a dependency relationship). The other 12 are more interesting: they are not installable because the specified dependency relationships induce an unavoidable conflict.

For example, the package `cacti-cactid` version 0.8.6e-2 depends on `libsnmp5`, version 5.2.1.2 or greater; but the only appropriate version of `libsnmp5` in the pool, 5.2.1.2-2, conflicts with all versions of `cacti-cactid` up to and including 0.8.6e-2. In this case either there is some problem in the specification of the package relationships or there are some packages that have been removed from the pool (e.g., other versions of `libsnmp5`) leaving it in an inconsistent state.

Such kind of information gives the distribution editor a better understanding of what is going on when performing operations on the package metadata or on the repositories.

```
cacti-cactid (= 0.8.6e-2): FAILED
The following constraints cannot be satisfied:
  cacti-cactid (= 0.8.6e-2)
    conflicts with libsnmp5 (= 5.2.1.2-2)
  cacti-cactid (= 0.8.6e-2)
    depends on libsnmp5 (>= 5.2.1.2)
                    {libsnmp5 (= 5.2.1.2-2)}
```

Figure 7. Excerpt of the `debcheck` output on the Debian pool packages

The processing speed[6] is very encouraging and this makes the tools a valid and effective aid to the distribution editor. Checking the Mandriva package repository took 55 seconds with the SAT solver and 8 seconds with the standalone `rpmcheck` tool. Checking the Debian pool snapshot took 18 minutes and 13 seconds with the SAT solver and 43 seconds with the standalone `debcheck` tool. The timing difference is largely due to the fact the `rpmcheck` and `debcheck` tools parse the package metadata only once and produce the constraints associated to all the package satisfiability problems in a single run, while the toolchain generate one distinct subproblem, and calls the generic SAT solver, for each package.

[6]The machine used for the tests is a single-processor Intel Xeon 3.4 GHz machine running Mandriva Linux

6 Related work

Our work is related to what has been called *Software Release Management* [25], i.e., the process through which software is made available and obtained by the final users. While our goal was not to address the problem pertaining to physical distribution[7] our efforts have been directed in providing tools that can improve the quality of the Software Release Management.

There is a significant literature on the problem of managing dependencies between components: [20, 19] propose techniques that are actually implemented in tools like [18].

But most of these approaches are targeted at *designing* component systems, with a clear top-down process typical of a software architecture viewpoint, while the component systems we have to handle come from the huge preexisting infrastructures and legacy systems (e.g. well established GNU/Linux distributions) that have been conceived without having such a kind of architectural specifications in mind.

Our long term goal is to integrate these systems by abstracting and enriching their features while maintaining backward compatibility. For example by proposing extensions to the dependency specification that enables the checking of more sophisticated properties as in [27, 28]

On the algorithmic side, an interesting work is [7], that uses adjacency matrices to represent different kinds of dependency graphs. By performing operation on these matrices it is possible to verify some kinds of properties related to the evolution of the system (e.g., the number of components that are required if a given component has to be reused).

Notice though that in all the approaches just mentioned, the notion of conflict is not taken into account, as we do in ours. This makes all these approaches not suitable in a context, like the one we have analyzed, where conflicts are an essential (and complex) part of the dependency requirement specification.

A more similar work is [26] that uses feature diagrams to model dependencies and configurations, and BDD algorithms to perform automated checking of model properties.

Finally, there is a wealth of largely used tools in the Free Software Community, known as meta-package managers [17, 14, 13] that are clearly the most similar in spirit to what we have developed here, as they are able to explore the dependency requirements of a given package and perform all the steps needed to correctly install it by automatically finding the missing dependent packages, downloading and, finally, installing them, taking into acocunt dependencies and conflicts. The essential difference between our tools and those meta-package manager is that meta-package manager try to *optimize* the installation of packages on a

[7]Notice though that this is a problem addressed by other workpackages of the EDOS Project.

user machine, which is a task radically different, and in principle much more difficult than veryfying that a repository does not contain broken packages. As a consequence, many of these tools contain specific heuristics that make them incomplete (some packages are actually reported broken while they are installable), while those tools, like Smart, that strive to be complete are unacceptably inefficient when used to verify installability. And none of them has a formal basis.

7 Conclusions

In this paper we have presented our approach and the framework we have developed for managing the complexity and improving the quality of large package based FOSS systems. The approach has been validated by implementing two different set of tools that report the same results. By applying our automatic analysis on two of the most popular Linux-based distributions (i.e., Debian and Mandriva) we have been able to spot several problems in their package repositories that, otherwise, could have been ignored for longtime. Having this kind of (previously unavailable) tools is a great help for distribution editors that can use them in order to track problems in their package bases and, hence, improve the quality of their distributions.

Our tools are successfully used at Caixa Magica (an RPM-based distribution editor [1]) and Mandriva in parallel with their standard distribution life cycle phases. Future work concerns the integration of these tools into their production chain, and in other distributions as well. Moreover, some other work will concern the improvement of the current framework, for example, by refactoring it in a more integrated and user friendly tool and by providing more complex and active analysis.

References

[1] Caixa Magica Linux. `http://www.caixamagica.pt`.

[2] Debian GNU/Linux. `http://www.debian.org`.

[3] Distrowatch. `http://www.distrowatch.com`.

[4] Mandriva Linux. `http://www.mandriva.com`.

[5] The mozart programming system. `http://www.mozart-oz.org`.

[6] E. C. Bailey. Maximum rpm. `http://www.rpm.org/max-rpm`.

[7] L. Bixin. Managing dependencies in component-based systems based on matrix model. In *Proceedings of NETObjectDays'03*, 2003.

[8] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. Marshall. Graphml specification, 2002. `http://graphml.graphdrawing.org/specification.html`.

[9] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. M. F. Yergeau. Extensible markup language (xml) 1.0 (third edition), 2004. `http://www.w3.org/TR/REC-xml`.

[10] EDOS Project Workpackage 2 Team. Deliverable 1, 2005. `http://www.edos-project.org/xwiki/bin/Main/Deliverables`.

[11] N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. 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.

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

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

[14] G. Niemeyer. Smart package manager. `http://labix.org/smart`, 2005.

[15] Opensource.org. Opensource licenses. `http://www.opensource.org/licenses`.

[16] E. S. Raymond. The cathedral and the bazaar. `http://www.catb.org/~esr/writings/cathedral-bazaar`.

[17] G. N. Silva. Apt-howto. `http://www.debian.org/doc/manuals/apt-howto`, 2004.

[18] J. Stafford, D. Richardson, and A. Wolf. Aladdin: A tool for architecture-level dependence analysis of software systems. Technical Report CU-CS-858-98, Department of Computer Science, University of Colorado, 1997.

[19] J. Stafford, D. Richardson, and A. Wolf. Chaining: A software architecture dependence analysis technique. Technical Report CU-CS-845-97, Department of Computer Science, University of Colorado, 1997.

[20] J. A. Stafford and A. L. Wolf. Architecture-level dependence analysis in support of software maintenance. In *ISAW '98: Proceedings of the third international workshop on Software architecture*, pages 129–132. ACM Press, 1998.

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

[22] The CDuce Team. Cduce. `http://www.cduce.org`.

[23] The Debian Project. Debian policy manual. `http://www.debian.org/doc/debian-policy/index.html`.

[24] The RPM-Metadata Project. Xml package metadata. `http://linux.duke.edu/projects/metadata`.

[25] A. van der Hoek, R. S. Hall, D. Heimbigner, and A. L. Wolf. Software release management. In *ESEC '97/FSE-5: Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 159–175. Springer-Verlag New York, Inc., 1997.

[26] T. van der Storm. Variability and component composition. In *ICSR*, pages 157–166, 2004.

[27] M. Vieira and D. Richardson. Analyzing dependencies in large component-based systems. In *ASE'02: Proceedings of the International Conference of Automated Software Engineering*, pages 241–244, 2002.

[28] M. Vieira and D. Richardson. The role of dependencies in component-based systems evolution. In *IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution*, pages 62–65. ACM Press, 2002.