A

On Software Component Co-Installability

Jérôme Vouillon, CNRS, PPS UMR 7126, Univ Paris Diderot, Sorbonne Paris Cité
Roberto Di Cosmo
, Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126 CNRS, INRIA

Modern software systems are built by composing components drawn from large *repositories*, whose size and complexity is increasing at a very fast pace. A fundamental challenge for the maintainability and the scalability of such software systems is the ability to quickly identify the components that can or cannot be installed together: this is the *co-installability* problem, which is related to boolean satisfiability and is known to be algorithmically hard. This paper develops a novel theoretical framework, based on formally certified semantic preserving graph-theoretic transformations, that allows us to associate to each concrete component repository a much smaller one with a simpler structure, that we call *strongly flat*, with equivalent co-installability properties. This flat repository can be displayed in a way that provides a concise view of the co-installability issues in the original repository, or used as a basis for various algorithms related to co-installability, like the efficient computation of strong conflicts between components. The proofs contained in this work have been machine checked using the Coq proof assistant.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal Methods*; G.2.2 [**Mathematics of Computing**]: Graph Theory—*Hypergraphs*

General Terms: Algorithms, Theory, Verification

Additional Key Words and Phrases: component, dependencies, conflicts, co-installability, package management, open source

ACM Reference Format:
ACM Trans. Softw. Eng. Methodol. V, N, Article A (January YYYY), 34 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

The mainstream adoption of free and open source software (FOSS) has widely popularised component-based software architectures, maintained in a distributed fashion and evolving at a very quick pace. Components are typically made available via a *repository*, and each of these components is equipped with metadata, such as *dependencies* and *conflicts*, used to specify concisely the contexts in which a component can or cannot be installed.

Author's addresses: Laboratoire PPS, Case 7014, 5, Rue Thomas Mann, F-75205 Paris, France.
Author's e-mail addresses: jerome.vouillon@pps.jussieu.fr, roberto@dicosmo.org
Work partially supported by the European Community's 7th Framework Programme (FP7/2007-2013), grant agreement n°214898, "Mancoosi" project, and performed at the IRILL center for Free Software Research and Innovation in Paris, France.
This is a revised and extended version of the article [Di Cosmo and Vouillon 2011] presented at ESEC/FSE 2011 in Szeged, Hungary.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

A typical example of the metadata attached to a component, taken from the Debian GNU/Linux distribution, is shown in Example 1.1: the logical language used for expressing dependencies and conflicts is quite powerful, as it allows conjunctions (symbol ‘,’), disjunctions (symbol ‘|’) and version constraints (the symbols ‘>=’, ‘<=’, ‘>>’ and ‘<<’ stand for the usual $\geq, \leq, >$ and $<$ operators).

Example 1.1. The inter-package relationships of `postfix`, an Internet mail transport agent in the Debian GNU/Linux distribution (`http://www.debian.org`) currently reads:

```
 1  Package: postfix
 2  Version: 2.5.5-1.1
 3  Depends: libc6 (>= 2.7), libdb4.6, ssl-cert,
 4    libsasl2-2, libssl0.9.8 (>= 0.9.8f-5),
 5    debconf (>= 0.5) | debconf-2.0,
 6    netbase, adduser (>= 3.48), dpkg (>= 1.8),
 7    lsb-base (>= 3.0-6)
 8  Conflicts: libnss-db (<< 2.2-3), smail,
 9    mail-transport-agent, postfix-tls
10  Provides: mail-transport-agent, postfix-tls
```

In most frameworks, determining whether a single component can be installed at all is already an NP-Complete problem, albeit the concrete instances arising in real-world systems, like GNU/Linux distributions, Eclipse plugins or OSGI component repositories, turn out to be tractable [Mancinelli et al. 2006; Tucker et al. 2007; Le Berre and Parrain 2008]. For the maintenance of component repositories, though, more sophisticated analyses are required. This includes identifying for each component the other components in the repository that it absolutely needs [Abate et al. 2009], and those that it can never be installed with [Di Cosmo and Boender 2010].

More generally, a fundamental challenge for component based software maintainability is the study of the problem of *co-installability* of components, that involves identifying and visualising the relevant sets of components that can or cannot be installed together.

Indeed, from a maintenance point of view, one needs to identify which components cannot be installed together, in order to check whether these incompatibilities are justified or due to erroneous dependencies. The dependency graph is also too rich for end users, which are interested in having a given set of functionalities provided by some components (for instance, they want a word processor and a Web browser) but do not care about the additional components also installed. This is illustrated by the fact that some package managers keep track of these additional components and automatically remove them when they are no longer needed.

The sheer size of current mainstream repositories, with tens of thousands of components and hundreds of thousands of relations, makes it completely unfeasible to study such properties of a repository directly: visualising such large graphs is both technically challenging, and of little interest, as one would need to follow recursively a large number of dependency and conflict relations to understand how components relate to one other.

In this paper, we develop a novel theoretical framework, based on formally certified semantic preserving graph-theoretic transformations, that allow us to associate to each concrete component repository a much smaller repository with a simpler structure, but with equivalent co-installability properties. This repository is small enough that its graphical representation gives a concise view of the co-installability issues in the original repository, that can be detected simply by visual inspection. It

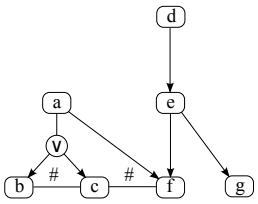


Fig. 1. Graphical depiction of a repository

can also be used as a basis for various algorithms related to co-installability, like the efficient computation of the strong conflicts defined in [Di Cosmo and Boender 2010].

We identified several bugs in the Debian distribution using the present work. For instance, up to version 0.1.33, the `harden-servers` package, meant to prevent the installation of unsafe packages, was in conflict with packages `proftpd` and `sendmail` but did not prevent the installation of the corresponding binaries which were in fact in packages `proftpd-base` and `sendmail-bin` (the first two packages were actually transitional packages used to ease the upgrade from a previous release).

This article is organised as follows: Section 2 recalls the basic notions about packages and dependencies, and overviews the repository transformation developed in the paper, which achieves impressive results on real-world GNU/Linux distributions (Section 3). The technical development follows: repositories are equipped with a partial order in Section 4, put into a flattened form in Sections 5, 6 and 7, simplified by removing irrelevant dependencies and conflicts in Section 8 and reduced by identifying equivalent packages in Section 9. Finally, we show how to draw a simplified graph of the kernel of a repository in Section 11, that also explains how to compute efficiently strong conflicts. We then present the `coinst` tool in Section 12, and discuss the results obtained on the Ubuntu 10.10 distribution in Section 13. Finally, we discuss related and future works, and conclude in Section 16.

2. OVERALL APPROACH

While the concrete details may vary from one technology to the other, the core metadata associated to component based systems always allows to express a few fundamental properties: a component, called *package* in the following, may *depend* on a combination of components, expressed as a conjunction of disjunctions of components, and a component may *conflict* with a combination of components, expressed as a conjunction of components.

Extra properties like *provides* (e.g. `postfix-tls` in Example 1.1), or *versioned constraints* (e.g. `libc6 (>= 2.7)` in Example 1.1) can be easily preprocessed out [Mancinelli et al. 2006], so that one can focus on a core dependency system that contains a binary symmetric conflict relation, and a dependency function $D(\pi) = \{\{\pi_1^1, ..., \pi_{n_1}^1\}, ..., \{\pi_1^k, ..., \pi_{n_k}^k\}\}$ that is satisfied when for each i at least one of the π_j^i is installed.

An example repository in this core dependency system is depicted in Figure 1, that also introduces the graphical notation used in the paper: package a has two dependencies, and can be installed only if, first, either package b or package c is installed, and second, package f is installed; package c conflicts with b and f: neither packages b and c, nor packages c and f, can be installed simultaneously; package d depends on package e, which in turn depends on both f and g.

2.1. Repositories

We follow the notations of earlier works [Mancinelli et al. 2006; Treinen and Zacchiroli 2008; Di Cosmo et al. 2006; Di Cosmo et al. 2006], that we recall here. A *repository* is a tuple $R = (P, D, C)$ where P is a finite set of *packages*, $D : P \to \mathcal{P}(\mathcal{P}(P))$ is the *dependency function* (we write $\mathcal{P}(X)$ for the set of subsets of X), and C, a symmetric irreflexive relation over P, is the *conflict relation*. An *installation* I of a repository (P, D, C) is a subset of P. An installation I is *healthy* when the following holds:

— *abundance*: every package has what it needs. Formally, for every package $\pi \in I$, and for every dependency $d \in D(\pi)$, we have $d \cap I \neq \emptyset$.
— *peace*: no two packages conflict, that is, $C \cap (I \times I) = \emptyset$.

We call *dependency* a set of packages d included in $D(\pi)$ for some package $\pi \in P$. Given a repository (P, D, C), the number of its dependencies is counted as $\sum_{\pi \in P} card(D(\pi))$.

Notice that the empty set $\emptyset$ is allowed by the definition: it is usually called a *broken* dependency, as it makes any package containing it uninstallable, because the abundance condition would be unsatisfiable for such package. Such broken dependencies are not uncommon in real component repositories, where a package may mention among its dependencies packages that are no longer present in the repository. We will also use such empty dependencies to mark uninstallable packages as broken, in Section 8.4.

One can give a logical interpretation of the dependency function and the conflict relation. The logical variables are the packages $\pi \in P$. A set of packages $d \in \mathcal{P}(P)$ is interpreted as a disjunction:

$$[\![d]\!] = \bigvee_{\pi \in d} \pi.$$

A set of set of packages $D(\pi) \in \mathcal{P}(\mathcal{P}(P))$ is interpreted as a conjunction:

$$[\![D(\pi)]\!] = \bigwedge_{d \in D(\pi)} [\![d]\!].$$

A dependency function D is then interpreted as the set of formulas of the shape:

$$\pi \implies [\![D(\pi)]\!]$$

where π ranges over P, which can also be written:

$$\pi \implies \bigwedge_{1 \leq i \leq n} \bigvee_{1 \leq j \leq m_i} \pi_{i,j}$$

with $D(\pi) = \{d_i \,|\, 1 \leq i \leq n\}$ and $d_i = \{\pi_{i,j} \,|\, 1 \leq j \leq m_i\}$. A conflict relation C is interpreted as the set of formulas $\neg\pi \vee \neg\pi'$ for $(\pi, \pi') \in C$. A healthy installation is an assignment which simultaneously satisfies all these formulas.

Remark 2.1 (*Circular dependencies*). It is normally recommended to avoid circular dependencies in package repositories, as it is not possible to define a deployment order in the presence of cycles. Nevertheless, circular dependencies pose no problem for the theoretical treatment: packages that depend on each other are simply logically equivalent, and will be merged into a single equivalence class by our algorithm.

2.2. Co-Installability

A package π is *installable* in a repository if it is included in a healthy installation I of this repository. A set of packages Π are *co-installable* in a repository if they are all included in some healthy installation I of the repository.

Checking co-installability has been shown equivalent to SAT by taking advantage of the logical interpretation of repositories [Mancinelli et al. 2006]. However, our experience is that this problem is easy in practice: SAT-solver based tools are currently used routinely to identify non-installable components on repositories that contains dozens of thousands of packages, and hundreds of thousands of dependencies and conflicts.

Remark 2.2. Let (P, D, C) and (P', D', C') be two repositories. If any healthy installation I of repository (P, D, C) is a healthy installation of repository (P', D', C'), then any co-installable set of packages Π in repository (P, D, C) is a co-installable set of packages in repository (P', D', C').

Proof: Immediate □

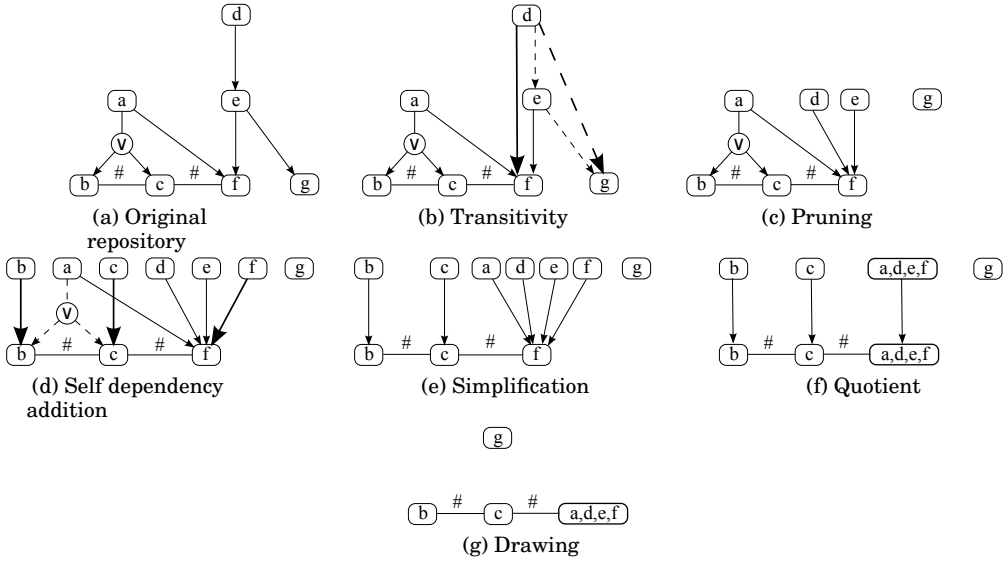


Fig. 2. Transformations of a repository (added dependencies are in bold, dotted ones are removed in the next phase)

2.3. Extracting a Co-Installability Kernel

Identifying all the sets of components that cannot be installed together is way more complex: even if we limit ourselves to the simplest case of sets of non co-installable components of size 2 (also known as *strong conflicts*), testing all possible pairs of packages is not a viable option, as the package number is in the tens of thousands. Even using all the optimisations described in [Di Cosmo and Boender 2010], the computation takes almost a week on a modern workstation.

In the present work, we lay down the essential theoretical basis and algorithmic insight for tackling this *non co-installability* problem: the fundamental idea is to extract from the component repository a *kernel* repository which is equivalent to the original one, as far as co-installability is concerned, but which turns out in practice to be orders of magnitude smaller, and easily manageable.

Looking at the example repository of Figure 1, for example, one can check that c cannot be installed together with b and and that c cannot be installed together with any of a,d,e and f, while g can always be installed; also, a,d,e and f are really equivalent as far as installability is concerned: if one of them can be installed in a given configuration,

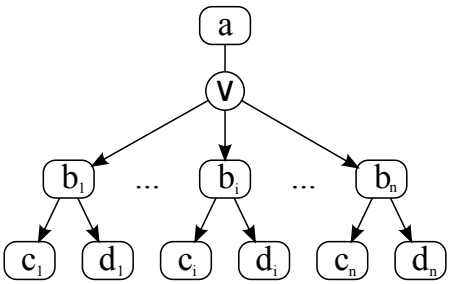


Fig. 3. Repository that blows-up when flattened

then all the others can, and viceversa. Instead of performing this check on the original repository, which requires following dependency chains and performing complex case analysis, we remark that not all the information present in the repository is relevant, and take advantage of this fact to simplify and reduce it.

The key steps of this process are now summarised. The effect of each step on the example repository of Figure 1 is shown in Figure 2.

Flattening. The recursive nature of dependencies is convenient for package developers, as it allows them to describe the dependencies among the different packages very concisely, in a modular way. To study the properties of a repository, though, it is way more convenient to use only a special *flattened* form $\widehat{D}$ of dependency functions that *directly* describes all dependencies of each package: if

$$\widehat{D}(\pi) = \{\{\pi_1^1, ..., \pi_1^{k_1}\}, ...\{\pi_n^1, ..., \pi_n^{k_n}\}\},$$

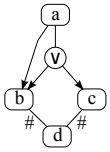
then the packages π_i^j are all the packages relevant for installing package π, and only them.

Any dependency function can be converted in this form by a sort of transitive closure that expands the dependencies of each intermediate package, and then converts the result again into a conjunction of disjunctions using distributivity: on our running example, this amounts to adding a dependency from d to f, and one from d to g (Figure 2b).

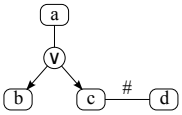
This transformation has some similarity with the conversion of logical formulae to conjunctive normal form, and is likewise subject to exponential blow-up (see for instance [Buning and Lettmann 2002]): for any n, a repository shaped like the one shown in Figure 3 contains $3n$ dependencies, but its expansion contains 2^n dependencies.

This is a strong limiting result, but we are only interested in studying coinstallability of packages, so we need not fully maintain the logical equivalence of repositories. In particular, we can prune the expanded dependency function by removing any dependency containing a package with no conflicts without changing the coinstallability property. In practice, this suffices to avoid the exponential blow-up. On our running example, this pruning phase removes the dependencies from d to e, from d to g and from e to g, leading to the repository of Figure 2c.

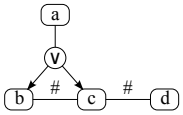
We take a further action to render the repository more homogeneous: we add a self dependency to each package with conflicts. This sort of reflexive closure will be very useful when quotienting the repository, in a later phase. (There is no point in adding self dependencies to other packages as they would be removed by pruning.) We then find it convenient to draw the repository using a two-level structure: on the top, we have all packages; on the bottom, we have packages with conflicts; dependencies connects the top layer to the bottom layer; conflicts are between packages on the bottom layer. On our running example, this leads to the repository of Figure 2d.



(a) Implied dependency



(b) Dependency on package with no conflict



(c) Dependency on package with only internal conflicts

Fig. 4. Some irrelevant disjunctive dependencies

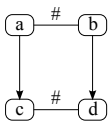


Fig. 5. A redundant conflict

The three phases of expansion, pruning and addition of self dependencies can be performed in a single pass, and we thereafter use the term *flattening* to denote all of them.

Elimination of irrelevant dependencies and conflicts. As a second phase, we identify several classes of dependencies and conflicts that are irrelevant as far as co-installability is concerned, and remove them. In Figure 4, we can see some interesting examples (more are given in Section 8), in all of which the disjunctive dependency connecting package a to packages b and c can be dropped:

(a) if some branches of a disjunction are forced by a stronger dependency, all other branches can be dropped;
(b) a package with no conflict can be added to any installation, so dependencies on such a package are always satisfiable and all disjunctive dependencies containing it can be simplified out (this is precisely the pruning defined above during flattening, which may need to be performed again on the flattened repository when a conflict is removed);
(c) if a package has a disjunctive dependency containing a package (here, b) that conflicts only with other packages in this dependency (here, c), this dependency is always satisfiable and can be dropped: either a package conflicting with package b is installed, or package b can be installed; in both cases, the dependency can be satisfied.

In Figure 5, the conflict between packages a and b is implied by the conflict between packages c and d and can be dropped.

Proving the soundness of such simplifications is far from trivial: in general, one has to rely on a peculiar structure of the repository to justify that a dependency can be removed, but removing a dependency may modify this very structure. Therefore, a suitable invariant has to be found, that allows us to remove most, if not all, irrelevant dependencies.

On our example repository, we already removed dependencies corresponding to Figure 4b during flattening; the disjunctive dependency from package a to packages b and c, corresponding to Figure 4c, can be removed, yielding the repository of Figure 2e.

Quotienting equivalent packages. It is now quite evident looking at Figure 2e, that packages a, d, e and f are, as far as co-installability is concerned, really equivalent: they share the very same set of dependencies (notice that this fact is easily detectable

	Debian		Ubuntu		Mandriva	
	before	after	before	after	before	after
Packages	28919	1038	7277	100	7601	84
Dependencies	124246	619	31069	29	38599	8
Conflicts	1146	985	82	60	78	62
Median cone size	38	1	38	1	59	1
Avg. cone size	66	1.7	84	1.3	153	1.1
Max. cone size	1134	15	842	4	1016	5
Running time (s)		10.6		1.19		11.6

Fig. 6. Repository sizes

on the graph thanks to the self dependency of package f introduced during the flattening phase).

As many packages in a repository share the same behaviour with respect to co-installability, it is useful to quotient the final repository, identifying these packages; this step contributes greatly in reducing the size of the repository, as can be seen on our running example in Figure 2f.

After removing self dependencies, one gets the final repository of Figure 2g, where it is now quite easy to see which package can be installed with which other package, and which package cannot.

3. EXPERIMENTAL RESULTS

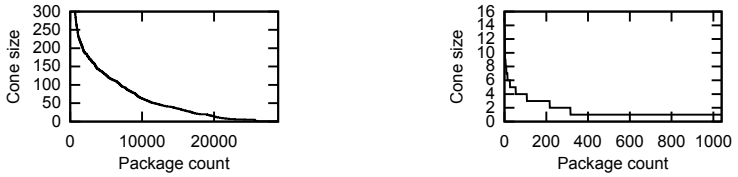
The transformations described in this paper have been proven correct, and all the proofs have been certified in Coq [The Coq Development Team 2008]. An OCaml program implementing these transformations has been run on several mainstream GNU/Linux distributions: Debian testing (full suite, amd64, snapshot taken August 22, 2010), Ubuntu 10.10 (main suite, x386) and Mandriva 2010.1 (main suite, x386). Running time were measured on a machine using a Intel Core 2 Duo Processor E6600 at 2.4GHz. The relevant statistics of the results are given in the table of Figure 6.

The number of packages is greatly reduced: many packages share the same behaviour as far as co-installability is concerned, and the quotienting phase identifies them. In particular, many packages can *always* be installed, which is good news for the GNU/Linux distributions; they are thus mapped to a single equivalence class. The number of dependencies is reduced even more: many dependencies are not relevant to co-installability, and are removed by our transformations. The simplification shown in Figure 4c turns out to be essential; for example, in Debian, thousands of packages depend on `debconf` which depends on either `debconf-i18n` or `debconf-english`, these two last packages being mutually exclusive: removing the disjunctive dependency hugely reduces the size of the final repository.

As for conflicts, we notice that distributions contain only few of them, which explains that flattening is practical; most of them cannot be removed.

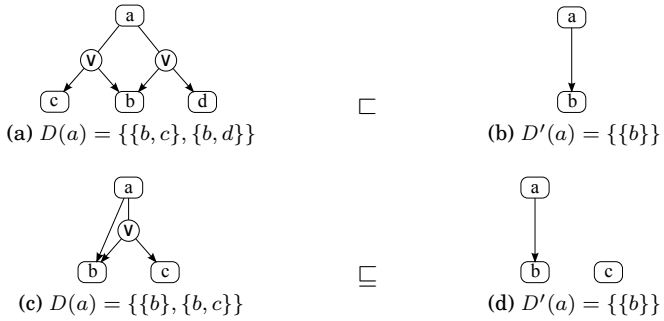
Finally, to each package p in a repository one can associate its *cone*, the set of packages that are reachable from p by following the dependency relations; the cone size of a package is typically quite large (Figure 7): two third of the packages have a cone of more than a hundred packages. On the other hand, after simplification, two third of the package equivalence classes have a cone of size one, meaning that they do not depend on any other package than themselves.

After simplification, the Ubuntu distribution fits on a three A5 size sheets (see Figures 16, 17 and 18 in Section 13) and can be easily inspected visually for er-



(a) before (b) after

Fig. 7. Distribution of cone size (Debian testing)



(a) $D(a) = \{\{b, c\}, \{b, d\}\}$ $\sqsubset$ (b) $D'(a) = \{\{b\}\}$

(c) $D(a) = \{\{b\}, \{b, c\}\}$ $\sqsubseteq$ (d) $D'(a) = \{\{b\}\}$

Fig. 8. Examples of dependency functions related by the preorder

rors. The corresponding graph for the Debian distribution is much larger, but it is our experience that it can still be displayed in a usable way with a suitable graph viewer: the visualisation of the Debian co-installability kernel can be tested online at `http://ocsigen.org/js_of_ocaml/graph/`.

As a consequence of the small size of the kernel extracted from a repository, many analyses can be performed very quickly. For example, it is easy to compute on the co-installability kernel of a repository the pairs of packages that can never be installed together, known as *strong conflicts* [Di Cosmo and Boender 2010], which is the simplest case of co-installability. On the same data as [Di Cosmo and Boender 2010], the computation takes a few seconds, instead of the several days reported there. This time is in fact included in the running times reported in Figure 6 as we use this information when drawing the simplified repository to emphasise packages that prevent the installation of many other packages.

4. ORDERING REPOSITORIES

A large part of our work consists of finding constraints that can be removed while leaving co-installability invariant. We make the idea of removing constraints precise by defining a preorder on repositories.

We first define a preorder on dependency functions: $D \sqsubseteq D'$ iff for every package π, for every dependency $d \in D(\pi)$, there exists a dependency $d' \in D'(\pi)$ such that $d' \subseteq d$. As an example, the dependency function D of the repository shown in Figure 8a is strictly subsumed by the dependency function D' of the repository shown in Figure 8b.

To see that this is really a preorder, consider the dependency function D of the repository shown in Figure 8c: it is subsumed by the dependency function D' of the repository shown in Figure 8d, but it also subsumes it, and yet the two functions are different.

This preorder corresponds exactly to the converse of logical implication: $D \sqsubseteq D'$ if and only if the formulas $[\![D'(\pi)]\!] \implies [\![D(\pi)]\!]$ can be derived for all packages $\pi \in P$. The preorder $\sqsubseteq$ is coarser than point-wise inclusion:

Remark 4.1. If, for all $\pi \in P$, $D(\pi) \subseteq D'(\pi)$, then $D \sqsubseteq D'$.

The preorder induces an equivalence relation over dependency functions : D is equivalent to D' if and only if $D \sqsubseteq D'$ and $D' \sqsubseteq D$. A canonical representative of an equivalence class can be defined by taking the element which is point-wise the smallest. Given a dependency function D, its canonical representative can be explicitly defined as:

$$D'(\pi) = \{d \in D(\pi) \,|\, \forall d' \in D(\pi), d' \subseteq d \implies d' = d\}.$$

In the example above, the dependency function D' of Figure 8d is the canonical representative of the dependency function D of the repository shown in Figure 8c.

This provides a first way to simplify the dependency function (illustrated by Figure 4a), and our implementation aggressively puts all dependencies in canonical form.

We can now define a preorder on repositories: $(P, D, C) \sqsubseteq (P', D', C')$ if and only if:

$$P = P', \qquad D \sqsubseteq D', \qquad C \subseteq C'.$$

We write $\sim$ for the equivalence relation associated to the relation $\sqsubseteq$. Given two repositories related under this preorder, going from the right hand side one to the left hand side one consists in possibly removing conflicts or relaxing dependencies, thus making it easier to install packages.

THEOREM 4.2. *When $(P, D, C) \sqsubseteq (P', D', C')$, any healthy installation I of repository (P', D', C') is also a healthy installation of repository (P, D, C).*

PROOF. We suppose that I is a healthy installation of (P', D', C') and prove that I is a healthy installation of (P, D, C).

We first prove abundance. Let $\pi \in I$ and $d \in D(\pi)$. As $D \sqsubseteq D'$, there exists $d' \in D'(\pi)$ such that $d' \subseteq d$. As I is a healthy installation of (P', D', C'), by abundance, $d' \cap I \neq \emptyset$. Hence, $d \cap I \neq \emptyset$ as wanted.

We now prove peace. As I is a healthy installation of (P', D', C'), we have $C' \cap (I \times I) = \emptyset$. On the other hand, $C \subseteq C'$. Hence, $C \cap (I \times I) = \emptyset$ as wanted. □

5. FLATTENING DEPENDENCIES

The *flattened* form of a dependency function, whose intuition has been given in the introduction, is formally defined as follows; given a repository (P, D, C), the *flattened* dependency function $\widehat{D}$ is the smallest function (with respect to point-wise inclusion) such that:

$$\textsc{Refl}\ \frac{(\pi, \pi') \in C}{\{\pi\} \in \widehat{D}(\pi)} \qquad \textsc{Trans}\ \frac{\{\pi_1, \ldots, \pi_n\} \in D(\pi) \quad d_1 \in \widehat{D}(\pi_1) \quad \ldots \quad d_n \in \widehat{D}(\pi_n)}{\bigcup_{1 \leq i \leq n} d_i \in \widehat{D}(\pi)}$$

As the above rules are monotonic, such a function $\widehat{D}$ exists.

The rule TRANS expands the intermediate dependencies of a package π and converts the result back into a conjunction of disjunctions; this rule silently drops circular dependencies: adding a dependency $\{\pi\} \in D(\pi)$ has no effect on $\widehat{D}(\pi)$.

	D	$\widehat{D}$
a	$\{\{b,c\},\{f\}\}$	$\{\{b,c\},\{f\}\}$
b	$\emptyset$	$\{\{b\}\}$
c	$\emptyset$	$\{\{c\}\}$
d	$\{\{e\}\}$	$\{\{f\}\}$
e	$\{\{f\},\{g\}\}$	$\{\{f\}\}$
f	$\emptyset$	$\{\{f\}\}$
g	$\emptyset$	$\emptyset$

Table I.

The rule REFL is designed to capture precisely the two properties we have outlined informally in the introduction: on one hand, we want to keep in $\widehat{D}(\pi)$ *only* dependencies on packages with at least a conflict (we prove in the next section that they are enough for keeping co-installability invariant); on the other hand, we want $\widehat{D}(\pi)$ to contain *explicitly* all the packages that have a conflict and are needed to install package π (so, if π has a conflict, it will also be an explicit dependency of itself).

The application of the transformation on the repository of Figure 1 gives the result shown in Table I, which is illustrated by Figure 2e:

Packages d, e and f now have the same dependencies, which reflects the intuition that they behave the same way as far as co-installability is concerned.

6. STRONGLY FLAT REPOSITORIES

After being flattened as described in the previous section, a repository satisfies two properties (Theorem 6.1 below): a reflexivity property (if a package has a conflict, then it depends on itself), and a transitivity property (dependencies are stable under composition).

In this section, we define precisely these two properties and study repositories that satisfies them, that we call *strongly flat*.

For strongly flat repositories, co-installability can be shown equivalent to a more convenient property, that we call *weak co-installability* (Theorem 6.6). Using this fact, we prove the key result of this section: a set of packages are co-installable in a repository if and only if they are co-installable in the corresponding flattened repository (Theorem 6.8).

We define the *composition* $D\,;D'$ of two dependency functions over a set of packages P as the smallest function (with respect to point-wise inclusion) such that for every package $\pi \in P$, for every set $\{\pi_1,\ldots,\pi_n\} \in D(\pi)$, for every sets $d_i \in D'(\pi_i)$, we have $\bigcup_{1\leq i\leq n} d_i \in (D\,;D')(\pi)$.

To any conflict relation C, we associate a dependency function Δ_C defined as follows:

$$\Delta_C(\pi) = \begin{cases} \{\{\pi\}\} & \text{if } (\pi,\pi') \in C \text{ for some } \pi' \in P \\ \emptyset & \text{otherwise} \end{cases}$$

A repository (P, D, C) is *strongly flat* when the following conditions hold:

— *reflexivity*: $\Delta_C \sqsubseteq D$ (every package with conflict depends on itself);
— *transitivity*: $D\,;D \sqsubseteq D$ (dependencies are closed under composition).

The flattening transformation produces strongly flat repositories.

THEOREM 6.1. *Let (P, D, C) be a repository. Let $\widehat{D}$ be the corresponding flattened dependency function. The repository $(P, \widehat{D}, C)$ is strongly flat.*

PROOF. Reflexivity, that is $\Delta_C \sqsubseteq \widehat{D}$, is a direct consequence of rule REFL.

We now show transitivity, that is $\widehat{D} \,;\, \widehat{D} \sqsubseteq \widehat{D}$. Let $\pi \in P$ and $d \in (\widehat{D} \,;\, \widehat{D})(\pi)$. By definition of composition, there exists $d' = \{\pi_1, \ldots, \pi_n\} \in \widehat{D}(\pi)$ and sets $d_i \in \widehat{D}(\pi_i)$ such that $d = \bigcup_{1 \leq i \leq n} d_i$. We need to show that there exists $d'' \in \widehat{D}(\pi)$ such that $d'' \subseteq d = \bigcup_{1 \leq i \leq n} d_i$. The proof is by induction on a derivation of $d' \in \widehat{D}(\pi)$. We have two cases.

— Case REFL. There exists $\pi' \in P$ such that $(\pi, \pi') \in C$, and $d' = \{\pi\}$. We have, $d = d_1 \in \widehat{D}(\pi_1) = \widehat{D}(\pi)$, hence we can take $d'' = d$.
— Case TRANS. There exists a set $\{\pi'_1, \ldots \pi'_m\} \in \widehat{D}(\pi)$ and sets $d'_j \in \widehat{D}(\pi'_j)$ such that $d' = \bigcup_{1 \leq j \leq m} d'_j$. Besides, by induction hypothesis, for all j, if one can find packages π^k_j and dependencies d^k_j such that $d'_j = \{\pi^1_j, \ldots, \pi^{p_j}_j\}$, and $d^k_j \in \widehat{D}(\pi^k_j)$, then there exists $d''_j \in \widehat{D}(\pi'_j)$ such that $d''_j \subseteq \bigcup_{1 \leq k \leq p_j} d^k_j$. We have $d'_j \subseteq d'$. Hence we can choose for each indices j and k an index i such that $\pi^k_j = \pi_i$. Then, we take $d^k_j = d_i$. We therefore have dependencies $d''_j \in \widehat{D}(\pi'_j)$ such that $d''_j \subseteq \bigcup_{1 \leq k \leq p_j} d^k_j$. We take $d'' = \bigcup_{1 \leq j \leq m} d''_j$. By rule TRANS, $d'' \in \widehat{D}(\pi)$. Besides, $d'' \subseteq \bigcup_{1 \leq j \leq m} \bigcup_{1 \leq k \leq p_j} d^k_j$. But, as $d' = \bigcup_{1 \leq j \leq m} d_i$, the packages π^k_j ranges over all the distinct packages π_i. Hence, the dependencies d^k_j ranges over all dependencies d_i, and we have $d'' \subseteq \bigcup_{1 \leq i \leq n} d_i$ as wanted. □

□

Intuitively, strongly flat repositories have a two level structure. Looking for instance at Figure 2e, we find all packages on the top layer, and typically only packages with conflicts at the bottom layer: thanks to transitivity, everything a package π may need to be installed is fully described by $D(\pi)$, without recursive traversal of dependencies; and thanks to reflexivity, conflicts need only be considered on the image of D.

We can take advantage of this to define a more convenient way of capturing co-installability by defining it specifically on the two level structure of strongly flat repositories: while in our application we will have packages on the top and the bottom layer, the definition and the following mathematical development make perfect sense even if the objects on the two levels are different. To make this fact explicit, that we refer to the objects at the lower level, that only carry conflicts, as *features* in the following.

A *configuration* is a pair (I, F) of a set I of packages and a set F of features; we say that it is *healthy* when the following conditions hold:

— *abundance*: every package has what it needs. Formally, for every package $\pi \in I$, and for every dependency $d \in D(\pi)$, we have $d \cap F \neq \emptyset$.
— *peace*: no two features conflict, that is, $C \cap (F \times F) = \emptyset$.

This is subtly different from the homonymous definitions regarding installations. Conflicts are only checked in F, and abundance only checked for packages in I w.r.t. F; the sets F and I might as well be disjoint, here.

Remark 6.2. The two following properties hold:

— an installation I is healthy iff the configuration (I, I) is healthy;
— if (Π, F) is healthy and $\Pi' \subseteq \Pi$, then (Π', F) is healthy.

A set of packages Π are *weakly co-installable* if there exists a set of features $F \subseteq P$ such that the configuration (Π, F) is healthy. In general, this is a weaker notion.

LEMMA 6.3. *If a set of packages Π are co-installable, then they are also weakly co-installable.*

PROOF. Suppose the set of packages Π are co-installable. By definition, there exists a healthy installation F such that $\Pi \subseteq F$. We terminate the proof by using in turn both properties in Remark 6.2. The configuration (F, F) is healthy. Hence, by anti-monotony of the definition of healthiness, (Π, F) is healthy. □

In strongly flat repositories, though, the two notions are equivalent.

We present here two technical lemmas needed in the proof of Theorem 6.8.

LEMMA 6.4. *Let (P, D, C) be a repository such that $\Delta_C \sqsubseteq D$. Let (I, F) be a healthy configuration. For every package $\pi \in I$, if $(\pi, \pi') \in C$ for some $\pi' \in P$, then $\pi \in F$.*

PROOF. We consider a package $\pi \in I$ such that $(\pi, \pi') \in C$ for some package $\pi' \in P$. As $\Delta_C \sqsubseteq D$, there exists $d \in D(\pi)$ such that $d \subseteq \Delta_C(\pi) = \{\pi\}$. By abundance, $d \cap F \neq \emptyset$. Hence, $\pi \in F$. □

LEMMA 6.5. *Let (P, D, C) be a repository. Let D' be a dependency function such that $\Delta_C \sqsubseteq D'$ and $D \,;\, D' \sqsubseteq D'$. If a set of packages Π are weakly co-installable in (P, D', C), then they are co-installable in (P, D, C).*

PROOF. We consider a set of packages Π which are weakly co-installable in (P, D', C). This means that there exists a set of packages F such that the configuration (Π, F) is healthy in (P, D', C). We then consider a maximal set of packages I such that $\Pi \subseteq I$ and the configuration (I, F) is healthy in (P, D', C). We can conclude if we prove that I is healthy in (P, D, C).

We first prove abundance. We take $\pi \in I$ and $d = \{\pi_1, \ldots, \pi_n\} \in D(\pi)$ and need to prove that $d \cap I \neq \emptyset$. We want to show that we can find $\pi'' \in d$ such that $(I \cup \{\pi''\}, F)$, is healthy. Indeed, then, by maximality of I, we have $\pi'' \in I$. Hence, $d \cap I \neq \emptyset$ as wanted. The package π'' should be such that, for all $d'' \in D(\pi'')$, we have $d'' \cap F \neq \emptyset$. The proof is by contradiction. Suppose that for all i, there exists $d_i \in D'(\pi_i)$ such that $d_i \cap F = \emptyset$. As $D \,;\, D' \sqsubseteq D'$, there exists $d' \in D'(\pi)$ such that $d' \subseteq \bigcup_{1 \leq i \leq n} d_i$. We have $d' \cap F \subseteq \bigcup_{1 \leq i \leq n} d_i = \emptyset$. This contradicts the fact that $\pi \in I$ and (I, F) is healthy. Hence the existence of a suitable package π'' which allow us to finish the proof of abundance.

We now prove peace by contradiction. Suppose that $C \cap (I \times I) \neq \emptyset$, that is, there exists $\pi \in I$ and $\pi' \in I$ such that $(\pi, \pi') \in C$. As (I, F) is healthy, we have $\pi \in F$ and $\pi' \in F$ by Lemma 6.4, and then $(\pi, \pi') \notin C$ by peace. We reach a contradiction, hence $C \cap (I \times I) = \emptyset$. □

THEOREM 6.6. *Any set of packages Π weakly co-installable in a strongly flat repository are also co-installable.*

PROOF. This is an immediate consequence of Lemma 6.5, taking $D = D'$. □

It is interesting to remark that the result of the flattening operation can be mathematically characterised as follows.

LEMMA 6.7. *The flattened dependency function $\widehat{D}$ associated to a repository (P, D, C) is the least dependency function D' (for preorder $\sqsubseteq$, and up to equivalence) such that:*

— $\Delta_C \sqsubseteq D'$
— $D \,;\, D' \sqsubseteq D'$.

PROOF. Clearly, the flattened dependency function satisfies these equations. Let D' be another dependency function satisfying the equations. We want to prove that, for all $\pi \in P$, $\widehat{D}(\pi) \sqsubseteq D'(\pi)$. By unfolding the definition of the preorder, this can be rephrased as: for all $\pi \in P$, for all $d \in \widehat{D}(\pi)$, there exists $d' \in D'(\pi)$ such that $d' \subseteq d$. The proof is by induction on a derivation of $d \in \widehat{D}(\pi)$. We have two cases.

— Case REFL. There exists $\pi' \in P$ such that $(\pi, \pi') \in C$, and $d = \{\pi\}$. From the first assumption, there exists $d' \in D'(\pi)$, such that $d' \subseteq \{\pi\}$. Hence, $d' \subseteq d$ as wanted.
— Case TRANS. There exists a set $\{\pi_1, ..., \pi_n\} \in D(\pi)$ and sets $d_i \in \widehat{D}(\pi_i)$ such that $d = \bigcup_{1 \leq i \leq n} d_i$. Besides, by induction hypothesis, for all i, there exists $d'_i \in D'(\pi_i)$ such that $d'_i \subseteq d_i$. From the second assumption, there exists $d' \in D'(\pi)$, such that $d' \subseteq \bigcup_{1 \leq i \leq n} d'_i$. We have, $d' \subseteq \bigcup_{1 \leq i \leq n} d'_i \subseteq \bigcup_{1 \leq i \leq n} d_i = d$ as wanted. □

□

The essential result of this section is that co-installability is left invariant by flattening.

THEOREM 6.8. *Let (P, D, C) be a repository. Let $\widehat{D}$ be the corresponding flattened dependency function. Let Π be a set of packages. The following propositions are equivalent:*

(1) Π is co-installable in (P, D, C);
(2) Π is weakly co-installable in $(P, \widehat{D}, C)$;
(3) Π is co-installable in $(P, \widehat{D}, C)$.

PROOF. (3) implies (2) by Lemma 6.3. (2) implies (3) by Lemmas 6.6 and 6.1. (2) implies (1) by Lemmas 6.5 and 6.7. We now assume (1) and prove (3).

By definition, there exists a healthy installation I of repository (P, D, C) such that $\Pi \subseteq I$. We can just show that I is also a healthy installation of repository $(P, \widehat{D}, C)$. Peace is immediate. We prove abundance. We consider $\pi \in P$ and $d \in \widehat{D}(\pi)$. We need to show that if $\pi \in I$, then $d \cap I \neq \emptyset$. The proof is by induction on a derivation of $d \in \widehat{D}(\pi)$. We have two cases.

— Case REFL. We have $d = \{\pi\}$. Hence, if $\pi \in I$ then $d \cap I = \{\pi\} \neq \emptyset$.
— Case TRANS. We have $\{\pi_1, \ldots, \pi_n\} \in D(\pi)$ and $d = \bigcup_{1 \leq i \leq n} d_i$ where $d_i \in \widehat{D}(\pi_i)$ and, if $\pi_i \in I$, then $d_i \cap I \neq \emptyset$ (induction hypothesis). Let us assume $\pi \in I$. By abundance, as $\{\pi_1, \ldots, \pi_n\} \in D(\pi)$, there must exists some i such that $\pi_i \in I$. Then, $d_i \cap I \neq \emptyset$, and thus $d \cap I \neq \emptyset$. □

□

7. FLAT REPOSITORIES

In this section, we focus on a particular class $\nabla_C \subseteq \mathcal{P}(P)$ of dependencies that can be safely removed (Theorem 7.11). Removing these dependencies may destroy the *strongly flat* structure of a repository, but we introduce the weaker notion of *flat* repository which is preserved (Theorem 7.11), and for which co-installability and weak co-installability still coincide (Theorem 7.8), thus enabling further simplifications introduced in Section 8.

We want to capture in the class ∇_C a set of dependencies that have the following two key properties:

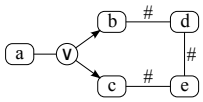


Fig. 9. Illustration of monotony requirement

always satisfiable any healthy configuration in the repository can be extended to satisfy these dependencies, so they are irrelevant for co-installability and we can remove them; formally, this means that if $d \in \nabla_C$, then for all $F \in \mathcal{P}(P)$ maximal with respect to set inclusion such that $C \cap (F \times F) = \emptyset$, we have $d \cap F \neq \emptyset$;
monotony a dependency which is in ∇_C must still be always satisfiable even if we remove some conflicts from the repository; formally, if $C' \subseteq C$, then $\nabla_C \subseteq \nabla_{C'}$.

The *monotony* property is necessary, because in the next section we introduce further simplifications that remove redundant conflicts. We want to be sure that removing a conflict later on does not invalidate the decision taken here of removing a dependency, as illustrated in Figure 9: the disjunctive dependency on packages b and c can always be satisfied because the conflict between d and e prevents the simultaneous installation of d and e; but this dependency is not in ∇_C since it is no longer satisfiable if the conflict between d and e is removed. This leads to the following:

Definition 7.1. Given a repository (P, D, C), the set ∇_C is the largest set such that $d \in \nabla_C$ if and only if, for all $C' \subseteq C$, for all $F \in \mathcal{P}(P)$ maximal with respect to set inclusion such that $C' \cap (F \times F) = \emptyset$, we have $d \cap F \neq \emptyset$.

We can give a more explicit characterisation of the elements of ∇_C: these are *exactly* the dependencies that contain at least a package having only *internal* conflicts, like in Figure 4c. This is quite a remarkable fact, as the property introduced in definition 7.1 is a global property of the repository, and yet it turns out to be possible to characterise it purely locally, as a condition on individual dependencies.

THEOREM 7.2. *Let (P, D, C) be a repository. The set ∇_C is the set of dependencies d such that there exists a package $\pi \in d$ such that, for all pairs $(\pi, \pi') \in C$, we have $\pi' \in d$.*

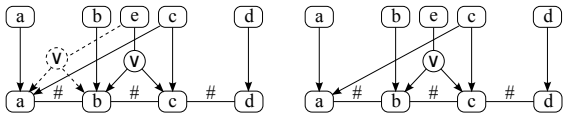
PROOF. We first assume that there exists no package $\pi \in d$ such that, for all pairs $(\pi, \pi') \in C$, we have $\pi' \in d$. We show that then $d \notin \nabla_C$. Let C' be the set of conflicts in C that cross d:

$$C' = \{(\pi, \pi') \in C \mid \#(\{\pi, \pi'\} \cap d) = 1\}.$$

Clearly, this is a conflict relation (it is symmetric and irreflexive). We consider the set of features $F = P \setminus d$. We have $C' \cap (F \times F) = \emptyset$. If we can now prove that the set F is a maximal set satisfying this property, we can conclude. Indeed, then, as $d \cap F = \emptyset$, the set d is not in ∇_C. We consider a strictly larger set of features F' and show that it does not satisfies the property. The set F' must contain at least a package π in dependency d. But then, by assumption, there exists $\pi' \in P \setminus d = F$ such that $(\pi, \pi') \in C$. By definition of C', we have $(\pi, \pi') \in C'$. Hence, $C' \cap (F' \times F') \neq \emptyset$, as wanted.

We now assume that there exists a package $\pi \in d$ such that, for all pairs $(\pi, \pi') \in C$, we have $\pi' \in d$. We show that then $d \in \nabla_C$. Let C' be a conflict relation such that $C' \subseteq C$. Let F be a maximal set of features such that $C' \cap (F \times F) = \emptyset$. We need to show that $d \cap F \neq \emptyset$. If we assume that there exists no $\pi' \in F$ such that $(\pi, \pi') \in C'$, then, by maximality of F, we must have $\pi \in F$. Hence, $d \cap F \neq \emptyset$ as wanted. Otherwise, there exists $\pi' \in F$ such that $(\pi, \pi') \in C'$. But then, $\pi' \in d$. Hence, $d \cap F \neq \emptyset$ as well. □

Notice that, if a dependency d contains a package π with no conflict, then it is in ∇_C; so ∇_C also contains the redundant dependencies shown in Figure 4b.



(a) Before removal (b) After removal

Fig. 10. Transitivity of dependencies is lost

Remark 7.3. Let (P, D, C) be a repository. The set ∇_C is upward-closed: if $d \subseteq d'$ and $d \in \nabla_C$, then $d' \in \nabla_C$.

PROOF. This is immediate thanks to Theorem 7.2. □

As we shall see, weak co-installability is preserved by the transformation that removes the elements of set ∇_C from a dependency function D of a strongly flat repository.
But, in general, the strongly flat property is lost: when we remove one dependency from a strongly flat repository, it may be the case that the transitivity of the flattened dependency function does no longer hold. Consider for example the repository shown in Figure 10: the dependency $\{a, b\}$ for package e is in ∇_C, but it can also be obtained by composing the depency $\{c, b\}$ with the dependency $\{a\} \in Dc$ and $\{b\} \in Db$. So, if we remove $\{a, b\}$, we break transitivity.
So we need a weaker notion of repository that is preserved by this simplification. We start by defining a coarser preorder on dependency functions that ignores dependencies in ∇_C :

> $D \prec_C D'$ if and only if for every package π in the domain of D, for every dependency $d \in D(\pi)$, *either* $d \in \nabla_C$ *or* there exists a dependency $d' \in D'(\pi)$ such that $d' \subseteq d$.

This preorder can be extended to repositories by setting $(P, D, C) \prec (P', D', C')$ iff $P = P'$, $D \prec_C D'$, and $C \subseteq C'$. We write $\approx$ for the associated equivalence relation.

Remark 7.4. The following implications hold:

— if $C' \subseteq C$ and $D \prec_C D'$, then $D \prec_{C'} D'$;
— if $D \sqsubseteq D'$, then $D \prec_C D'$;
— if $(P, D, C) \sqsubseteq (P', D', C')$, then $(P, D, C) \prec (P', D', C')$.

A repository (P, D, C) is *flat* when it satisfies the following properties:

— *reflexivity*: $\Delta_C \prec_C D$;
— *transitivity*: $D \,;\, D \prec_C D$.

Flat repositories have a series of good properties: they include strongly flat repositories, co-installability and weak co-installability still coincide, removing ∇_C preserves flatness and keeps co-installability invariant.

LEMMA 7.5. *Any strongly flat repository is flat.*

PROOF. This is a direct consequence of Remark 7.4. □

Remark 7.6. Let (P, D, C) be a flat repository. Let $\pi \in P$ such that $(\pi, \pi') \in C$ for some $\pi' \in P$. Then, there exists $d \in D(\pi)$ such that $d \subseteq \{\pi\}$.

PROOF. Let $\pi \in P$ such that $(\pi, \pi') \in C$ for some $\pi' \in P$. By reflexivity ($\Delta_{C'} \prec_C D$), we have either $\{\pi\} \in \nabla_C$ or there exists $d \in D(\pi)$ such that $d \subseteq \{\pi\}$. By Theorem 7.2, $\{\pi\} \notin \nabla_C$. Hence the result. □

LEMMA 7.7. *If $(P, D, C) \prec (P', D', C')$, $(P', D', C') \sqsubseteq (P, D, C)$, and the repository (P, D, C) is flat, then the repository (P', D', C') is flat.*

PROOF. We assume that $(P, D, C) \prec (P', D', C')$, $(P', D', C') \sqsubseteq (P, D, C)$, and (P, D, C) is flat. We show that (P', D', C') is flat.

We first show reflexivity, that is, $\Delta_{C'} \prec_{C'} D'$. We have $\Delta_C \prec_C D \prec_C D'$ by hypothesis. Hence, $\Delta_C \prec_{C'} D \prec_{C'} D'$ by Remark 7.4. On the other hand, we have $C' \subseteq C$. Hence, for all $\pi \in P$, we have $\Delta_{C'}(\pi) \subseteq \Delta_C(\pi)$. Thus, $\Delta_{C'} \sqsubseteq \Delta_C$ by Remark 4.1, and finally $\Delta_{C'} \prec_{C'} \Delta_C$ by Remark 7.4. We conclude by transitivity of $\prec_{C'}$.

We now show transitivity, that is, $D' ; D' \prec_{C'} D'$. Let $\pi^0 \in P$ and $d^0 \in (D' ; D')(\pi^0)$. We show that either $d^0 \in \nabla_{C'}$ or there exists a dependency $d \in D'(\pi^0)$ such that $d \subseteq d^0$. By definition of composition, there exists a dependency $d' = \{\pi_1, \ldots, \pi_n\} \in D'(\pi^0)$ and dependencies $d'_i \in D'(\pi_i)$ such that $d^0 = \bigcup_{1 \leq i \leq n} d'_i$. As $D' \sqsubseteq D$, there exists $d \in D(\pi^0)$ such that $d \subseteq d'$. Similarly, for all i, there exists $d_i \in D(\pi_i)$ such that $d_i \subseteq d'_i$. By transitivity in (P, D, C), either $\bigcup_{1 \leq i \leq n} d_i \in \nabla_C$ or there exists $d^1 \in D(\pi^0)$ such that $d^1 \subseteq \bigcup_{1 \leq i \leq n} d_i$. In the first case, we have $\bigcup_{1 \leq i \leq n} d_i \subseteq \bigcup_{1 \leq i \leq n} d'_i = d^0$. Then, as ∇_C is upward-closed (Remark 7.3) and $C' \subseteq C$, we have $d^0 \in \nabla_C \subseteq \nabla_{C'}$ as wanted. In the second case, we have $d^1 \subseteq \bigcup_{1 \leq i \leq n} d_i \subseteq \bigcup_{1 \leq i \leq n} d'_i = d^0$. As $D \prec_C D'$, either $d^1 \in \nabla_{C'}$ or there exists $d^2 \in D'(\pi^0)$ such that $d^2 \subseteq d^1$. If $d^1 \in \nabla_{C'}$, then $d^0 \in \nabla_{C'}$ as well, as $\nabla_{C'}$ is upward-closed. This is what we wanted to prove. Otherwise, we have $d^2 \in D'(\pi^0)$ such that $d^2 \subseteq d^1 \subseteq d^0$, which is again what is needed to conclude. □

THEOREM 7.8. *Any set of packages Π weakly co-installable in a flat repository are co-installable in this repository.*

PROOF. The proof follows the proof of Theorem 6.6, with minor modifications.

Let (P, D, C) be a flat repository. Let Π be a set of packages weakly co-installable in (P, D, C). There exists a set of packages F such that the configuration (Π, F) is healthy in (P, D, C). We assume that F is a maximal set of packages satisfying this property. We then consider a maximal set of packages I such that $\Pi \subseteq I$ and the configuration (I, F) is healthy in (P, D, C). We can conclude if we prove that I is healthy in (P, D, C).

We first prove abundance. We take $\pi \in I$ and $d = \{\pi_1, \ldots, \pi_n\} \in D(\pi)$ and need to prove that $d \cap I \neq \emptyset$. We want to show that we can find $\pi'' \in d$ such that $(I \cup \{\pi''\}, F)$, is healthy. Indeed, then, by maximality of I, we have $\pi'' \in I$. Hence, $d \cap I \neq \emptyset$ as wanted. The package π'' should be such that, for all $d'' \in D(\pi'')$, we have $d'' \cap F \neq \emptyset$. The proof is by contradiction. Suppose that for all i, there exists $d_i \in D(\pi_i)$ such that $d_i \cap F = \emptyset$. As $D ; D \prec_C D$, either $\bigcup_{1 \leq i \leq n} d_i \in \nabla_C$ or there exists $d' \in D(\pi)$ such that $d' \subseteq \bigcup_{1 \leq i \leq n} d_i$. In the first case, by maximality of F and definition of ∇_C, $(\bigcup_{1 \leq i \leq n} d_i) \cap F \neq \emptyset$. This contradicts the assumption that $d_i \cap F = \emptyset$ for all i. In the second case, we have $d' \cap F \subseteq \bigcup_{1 \leq i \leq n} d_i = \emptyset$. This contradicts the fact that $\pi \in I$ and (I, F) is healthy. Hence the existence of a suitable package π'' which allow us to finish the proof of abundance.

We now prove peace by contradiction. Suppose that $C \cap (I \times I) \neq \emptyset$, that is, there exists $\pi \in I$ and $\pi' \in I$ such that $(\pi, \pi') \in C$. As (I, F) is healthy, we have $\pi \in F$ and $\pi' \in F$ by Lemma 6.4, and then $(\pi, \pi') \notin C$ by peace. We reach a contradiction, hence $C \cap (I \times I) = \emptyset$. □

LEMMA 7.9. *If $(P, D, C) \prec (P', D', C')$, then any set of packages Π weakly co-installable in repository (P', D', C') is weakly co-installable in repository (P, D, C).*

PROOF. We assume that $(P, D, C) \prec (P', D', C')$ and that the set of packages Π is weakly co-installable in (P', D', C'). There exists a set of packages F such that (Π, F) is a healthy configuration in (P', D', C'). In particular, we have $C' \cap (F \times F) = \emptyset$. As $C \subseteq C'$, we also have $C \cap (F \times F) = \emptyset$. There exists thus a set of packages F' maximal such that $C \cap (F' \times F') = \emptyset$ and $F \subseteq F'$. We can conclude by showing that (Π, F') is a healthy configuration in (P, D, C). We have peace by definition of F'. We show abundance. Let $\pi \in \Pi$ and $d \in D(\pi)$. We need to show that $d \cap F' \neq \emptyset$. As $D \prec_C D'$, either $d \in \nabla_C$ or there exists $d' \in D'(\pi)$ such that $d' \subseteq d$. In the first case, by definition of ∇_C and maximality of F', we have $d \cap F' \neq \emptyset$ as wanted. In the second case, as (Π, F) is a healthy configuration, we have $d' \cap F \neq \emptyset$, and therefore $d \cap F' \neq \emptyset$ as wanted. □

LEMMA 7.10. *Let (P, D, C) be a repository. We define a simplified dependency function D' by: $D'(\pi) = D(\pi) \setminus \nabla_C$ for all $\pi \in P$. Then:*

— $(P, D', C) \sqsubseteq (P, D, C)$;
— $(P, D', C) \approx (P, D, C)$.

PROOF. For all $\pi \in P$, we have $D'(\pi) \subseteq D(\pi)$. Hence, by Remark 4.1, $D' \sqsubseteq D$, and therefore, $(P, D', C) \sqsubseteq (P, D, C)$. Then, by Remark 7.4, $(P, D', C) \prec (P, D, C)$.

We now prove $(P, D, C) \prec (P, D', C)$. It is sufficient to show that $D \prec_C D'$. Let $\pi \in P$ and $d \in D(\pi)$. We need to prove that either $d \in \nabla_C$ or there exists $d' \in D'(\pi) = D(\pi) \setminus \nabla_C$ such that $d' \subseteq d$. This is clearly satisfied, taking $d' = d$ in the second case. □

THEOREM 7.11. *Let (P, D, C) be a flat repository and D' be the dependency function such that $D'(\pi) = D(\pi) \setminus \nabla_C$ for all $\pi \in P$. The repository (P, D', C) is flat, and co-installability is left invariant by this transformation.*

PROOF. Flatness is a consequence of Lemmas 7.7 and 7.10. Invariance is a consequence of Theorem 7.8, and Lemmas 7.9 and 7.10. □

One can still reason on flat repositories, as far as co-installability is concerned, as if their dependency function was transitive: just choose configurations (I, F) where F is maximal, and then any dependency obtained by composition is satisfied, even when it is not explicitly in the dependency function. (For strongly flat repositories, this holds for arbitrary sets F.)

The following technical lemma is a crucial tool for reasoning on flat repositories. It states that a stronger form of transitivity where dependencies are not fully composed holds for flat repositories.

LEMMA 7.12. *Let (P, D, C) be a flat repository, $\pi \in P$, $d \in D(\pi)$. Let $\Pi = \{\pi_1, \ldots, \pi_n\}$ be a subset of d. We assume that, for all i, there exists $d_i \in D(\pi_i)$. Then, there exists $d' \in D(\pi) \cup \nabla_C$ such that $d' \subseteq (d \setminus \Pi) \cup \bigcup_{1 \leq i \leq n} d_i$.*

PROOF. Suppose that there exists $\pi' \in d \setminus \Pi$ such that $\{\pi'\} \in \nabla_C$ (that is, by Theorem 7.2, package π' conflicts with no package). Then we can take $d' = \{\pi'\}$ and conclude. Thus, we can assume that, for all $\pi' \in d \setminus \Pi$, there exists $\pi'' \in P$ such that $(\pi', \pi'') \in C$.

We show that for all $\pi' \in d$ we can find $d' \in D(\pi')$ such that $d' \subseteq (d \setminus \Pi) \cup \bigcup_{1 \leq i \leq n} d_i$. We can then use transitivity ($D \,;\, D \prec_C D$) to conclude. Indeed, by transitivity, there exists $d'' \in D \,;\, D$ such that $d'' \subseteq (d \setminus \Pi) \cup \bigcup_{1 \leq i \leq n} d_i$ and either $d'' \in \nabla_C$ or there exists $d''' \in D(\pi)$ such that $d''' \subseteq d''$. We can then conclude by taking either d'' or d'''.

Let $\pi' \in d$. If $\pi' = \pi_i$ for some i, we can take $d' = d_i$. Otherwise, $\pi' \in d \setminus \Pi$. We have assumed that there exists π'' such that $(\pi', \pi'') \in C$. By Remark 7.6, there exists $d' \in D(\pi')$ such that $d' \subseteq \{\pi'\}$. This dependency is suitable as $\pi' \in d \setminus \Pi$. □

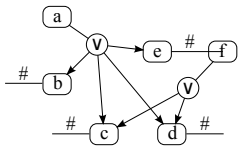


Fig. 11. Dependency covered by the conflict requirements.

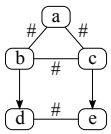


Fig. 12. Redundant conflict belonging to a clique

LEMMA 7.13. *Let (P, D, C) be a flat repository. Let C' be a conflict relation such that $C' \subseteq C$. The repository (P, D, C') is flat.*

PROOF. We first prove reflexivity: $\Delta_{C'} \prec_C D$. By assumption, $\Delta_C \prec_C D$. Hence, by transitivity, it is sufficient to prove that $\Delta_{C'} \prec_C \Delta_C$. Clearly, as $C' \subseteq C$, we have $\Delta_{C'}(\pi) \subseteq \Delta_C(\pi)$ for all π. We conclude by Remark 4.1 and 7.4.

We now prove transitivity. By assumption, we have $D \,;\, D \prec_C D$. Hence, by Remark 7.4, as $C' \subseteq C$, we have $D \,;\, D \prec_{C'} D$ as wanted. □

8. IRRELEVANT CONSTRAINTS

We review now several classes of dependencies and conflicts that are redundant and can be simplified out.

In the following, we will write $D \setminus \{\pi \mapsto d\}$ for the dependency function D where the dependency d of package π has been removed. Formally,

$$\begin{array}{lcll} (D \setminus \{\pi_0 \mapsto d_0\})(\pi_0) & = & D(\pi_0) \setminus \{d_0\} & \\ (D \setminus \{\pi_0 \mapsto d_0\})(\pi) & = & D(\pi) & \text{when } \pi \neq \pi_0. \end{array}$$

8.1. Clearly Irrelevant Dependencies

The results of the previous section let us remove the dependencies in ∇_C from a flat repository while leaving weak co-installability invariant and keeping the repository flat.

8.2. Conflict Covered Dependencies

An example of another very interesting class of irrelevant dependencies is shown in Figure 11, where the dependency for package a can always be satisfied despite the conflict between packages e and f (we assume the other packages in this dependency also have conflicts, even if not shown in the picture, so the dependency cannot be obviously removed): indeed, for this conflict to be relevant for the dependency, f needs to be installed; but if f is installed, at least one of packages c and d is installed as well, and thus the dependency is satisfied without needing to install e. This generalizes to the case where package e is in conflict with several packages with the same property as package f.

More formally, we say that a dependency d is *conflict covered at* π if it contains a package π such that for all $(\pi, \pi') \in C$, there exists a dependency $d' \in D(\pi')$ such that $d' \subseteq d \setminus \{\pi\}$. Removing one such dependency leaves co-installability invariant.

LEMMA 8.1. *Let (P, D, C) be a flat repository, d be a* conflict covered *dependency, and D' be the dependency function obtained by removing d from D. Any set of packages weakly co-installable in (P, D', C) is weakly co-installable in (P, D, C).*

PROOF. Let $d_0 \in D(\pi_0)$ be one of these dependencies. There exists $\pi_1 \in d_0$ such that for all $(\pi_1, \pi) \in C$, there exists a dependency $d \in D(\pi)$ such that $d \subseteq d_0 \setminus \{\pi_1\}$. Let $D' = D \setminus \{\pi_0 \mapsto d_0\}$ be the dependency function with this dependency removed.

Let Π be a weakly co-installable set of packages in the repository (P, D', C). There exists a set of features F such that (Π, F) is a healthy configuration of (P, D', C). We consider a maximal such a set F (with respect to inclusion). Clearly, the set F is then maximal such that $C \cap (F \times F) = \emptyset$. If $d_0 \cap F \neq \emptyset$, then (Π, F) is a healthy configuration of (P, D, C) as well, as wanted. We now consider the case where $d_0 \cap F = \emptyset$. Let $F' = (F \setminus \{\pi \mid (\pi_1, \pi) \in C\}) \cup \{\pi_1\}$. We show that (Π, F') is a healthy configuration of (P, D, C). Peace is clear. We need to show abundance.

Let $\pi \in I$ and $d \in D(\pi)$. We want to show that $d \cap F' \neq \emptyset$. If $\pi_1 \in d$, this is the case. We can thus assume that $\pi_1 \notin d$. In particular, this means that $d \neq d_0$, and therefore, by abundance, $d \cap F \neq \emptyset$. The remainder of the proof is by contradiction. We assume that $d \cap F' = \emptyset$. We consider the packages π^i such that $d \cap F = \{\pi^1, \dots \pi^n\}$. We have $\pi^i \in F \setminus F'$. Thus, for all i, we have $(\pi_1, \pi^i) \in C$, and therefore, by assumption, there exists $d^i \in D(\pi^i)$ such that $d^i \subseteq d_0 \setminus \{\pi_1\}$. By Lemma 7.12, there exists a dependency d' such that $d' \in D(\pi) \cup \nabla_C$ and $d' \subseteq (d \setminus (d \cap F)) \cup \bigcup_{1 \leq i \leq n} d^i$. From this last inclusion, we get $d' \subseteq (d \setminus F) \cup (d_0 \setminus \{\pi_1\})$. From this, on the one hand, as $\pi_1 \notin d$, we can see that $\pi_1 \notin d'$, and therefore $d' \neq d$. Hence, $d' \in D'(\pi) \cup \nabla_C$. By abundance and maximality of F, we get $d' \cap F \neq \emptyset$. On the other hand, $d' \cap F \subseteq ((d \setminus F) \cup (d_0 \setminus \{\pi_1\})) \cap F = (d_0 \setminus \{\pi_1\}) \cap F = ((d_0 \cap F) \setminus \{\pi_1\}) = \emptyset$. We reach a contraction, which completes the proof. □

Unfortunately, removing such dependencies may destroy the flatness of the repository, so we remove them one after another, in a greedy way, and only after checking that flatness is preserved by using the following result.

LEMMA 8.2. *Let (P, D, C) be a flat repository. Let $\pi \in P$ and $d \in D$. Let D'be the dependency function D where the dependency d of package π has been removed. If the following two conditions hold, then (P, D', C) is flat.*

— $d \not\subseteq \{\pi\}$;
— for all $d' \in (D' ; D')(\pi)$, we have $d \not\subseteq d'$.

PROOF. We first prove reflexivity, that is, $\Delta_C \prec_C D'$. Let $\pi' \in P$ and $d' \in \Delta_C(\pi')$. We show that either $d' \in \nabla_C$, or there exists $d'' \in D'(\pi')$ such that $d'' \subseteq d'$. By definition, we must have $d' = \{\pi'\}$. By reflexivity in repository (P, D, C), we have either $d' \in \nabla_C$, or there exists $d''' \in D(\pi')$ such that $d''' \subseteq d'$. In the first case, we can conclude immediately. If $d''' \in D'(\pi')$, we can conclude as well by taking $d'' = d'''$. Otherwise, we must have $\pi' = \pi$ and $d''' = d$. But, then, $d = d''' \subseteq d' = \{\pi'\} = \{\pi\}$. This is not possible due to the first assumption.

We now prove transitivity, that is, $D' ; D' \prec_C D'$. Let $\pi' \in P$ and $d' \in (D' ; D')(\pi')$. We show that either $d' \in \nabla_C$, or there exists $d'' \in D'(\pi')$ such that $d'' \subseteq d'$. By definition, there exists a dependency $\{\pi_1, \dots, \pi_n\} \in D'(\pi')$ and dependencies $d_i \in D'(\pi_i)$ such that $d' = \bigcup_{1 \leq i \leq n} d_i$. By transitivity in repository (P, D, C), either $d' \in \nabla_C$ or there exists $d''' \in D(\pi')$ such that $d''' \subseteq d'$. In the first case, we can conclude immediately. If $d''' \in D'(\pi')$, we can conclude as well by taking $d'' = d'''$. Otherwise, we must have $\pi' = \pi$ and $d''' = d$. But, then, $d = d''' \subseteq d' \in D' ; D'$. This is not possible due to the second assumption. □

In practice, it can be simpler to get all possible dependencies d' above by taking all dependencies $d'' \in D(\pi) \setminus \{d, \{\pi\}\}$ and composing them with dependencies in D.

In theory, the result of this simplification may depend on the order in which one applies the conflict covered reductions to remove dependencies: indeed, as a graph rewriting system [Baader and Nipkow 1998], this transformation has unsolvable

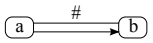


Fig. 13. Dependence on conflicting package

critical pairs.

In practice, though, we remove all instances present in the initial repository.

8.3. Redundant Conflicts

We consider some of the conflicts that can be removed from a repository while leaving co-installability invariant. A conflicting pair $(\pi_1, \pi_1') \in C$ is *redundant* if there exists a dependency $d \in D(\pi_1)$ such that for all $\pi_2 \in d$, there exists a package π_2' such that:

— $(\pi_2, \pi_2') \in C$;
— $\{\pi_1, \pi_1'\} \neq \{\pi_2, \pi_2'\}$;
— there exists $d' \in D(\pi_2)$ such that $d' \subseteq \{\pi_2'\}$.

Redundant conflicts can be removed, but only one at a time: for example, if two conflicts are considered redundant thanks to the existence of one another, then removing both of them simultaneously is incorrect.

LEMMA 8.3. *Let (P, D, C) be a repository. Let (π_1, π_2) be a redundant conflict in this repository. Any healthy installation of repository $(P, D, C \setminus \{(\pi_1, \pi_2), (\pi_2, \pi_1)\})$ is healthy in repository (P, D, C).*

PROOF. Let Π be a healthy set of packages of repository $(P, D, C \setminus \{(\pi_1, \pi_2), (\pi_2, \pi_1)\})$. We show that it is healthy in repository (P, D, C). Abundance is clear. We prove peace, that is, $C \cap (\Pi \times \Pi) = \emptyset$. We have by hypothesis $(C \setminus \{(\pi_1, \pi_2), (\pi_2, \pi_1)\})) \cap (\Pi \times \Pi) = \emptyset$. Thus, we can conclude if packages π_1 and π_2 are not both in Π. The proof is by contradiction: we assume the packages are in Π and reach a contradiction. By abundance in the initial repository, for any dependency $d \in D(\pi_1)$, there exists a package $\pi_1' \in d \cap \Pi$. Then, by definition of redundant conflicts, there exists a dependency $d_1 \in D(\pi_1)$, a package π_1' in $d_1 \cap \Pi$, a dependency $d_2 \in D(\pi_2)$ and a package $\pi_2' \in \Pi$ such that $(\pi_1', \pi_2') \in C \setminus \{(\pi_1, \pi_2), (\pi_2, \pi_1)\})$ and $d_2 \subseteq \{\pi_2'\}$. By abundance, $d_2 \cap \Pi \neq \emptyset$, hence $\pi_2' \in \Pi$. This contradicts peace in the initial repository. Hence the result. □

Removing redundant conflicts involves a trade off. On one side, it may allow removing some additional dependencies; on the other, it can also break some interesting structures. In Figure 12, the conflict between b and c is redundant, but removing it breaks the clique a, b, c, which is useful when drawing a simplified graph.

8.4. Dependence on Conflicting Packages

A special configuration may surface in the repository during simplification when the initial repository contains broken packages, as depicted in Figure 13: clearly, package a cannot be installed, and leaving such a configuration in the repository would pollute the graphical representation. In this case, we mark explicitly package a as broken by replacing its dependencies by the empty dependency $\emptyset$. All conflicts involving a can then be also removed, as they are redundant (Section 8.3). The transformation preserves healthiness.

LEMMA 8.4. *Let π be a package not installable in some repository (P, D, C). Let D' be the dependency function that coincide with D for all packages but π and such that $D'(\pi) = \{\emptyset\}$. Any healthy installation of repository (P, D, C) is also healthy in repository (P, D', C).*

PROOF. Let I be an healthy installation of repository (P, D, C). We show that it is an healthy installation of repository (P, D', C). Clearly, peace is satisfied and abundance is satisfied for all packages distinct from π. We now show abundance for package π. As $D'(\pi) = \emptyset$, this reduces to showing that $\pi \notin I$. This is indeed the case as π is not installable in the initial repository. □

The flatness of the repository may be destroyed, as some of the removed dependencies may be involved in transitivity. Thus, after applying such simplification, flattening should be performed again.

9. QUOTIENTING THE SET OF PACKAGES

In real-world repositories, many packages share the same behaviour as far as co-installability is concerned: for example, a lot of packages can always be installed, and some groups of packages only conflicts with a single other package.

In this section, we define an equivalence relation between packages, and show that the quotient w.r.t. this relation preserves all the good properties of a repository. We define two packages as equivalent in a repository (P, D, C) if they have the same dependencies:

$$\pi \equiv \pi' \text{ if and only if } D(\pi) = D(\pi').$$

We write $[\pi]$ for the equivalence class of package π, and extend this definition to set of packages: $[\Pi] = \{[\pi] \mid \pi \in \Pi\}$. The *quotient repository* (P', D', C') of a repository (P, D, C) is naturally defined as follows:

— P' is the set of all equivalence classes: $P' = P/\equiv = \{[\pi] \mid \pi \in P\}$;
— the dependency function D' is such that $D'([\pi]) = \{[d] \mid d \in D(\pi)\}$ for all $\pi \in P$;
— the conflict relation C' is defined by

$$C' = \{([\pi], [\pi']) \mid (\pi, \pi') \in C\}.$$

If the original repository does not contain dependencies between conflicting packages, like those between a and b shown in Figure 13, nor redundant conflicts, then the quotient repository is indeed a repository (the key point to check is irreflexivity of the conflict relation).

LEMMA 9.1. *Let (P, D, C) be a flat repository such that, for all $\pi \in P$ and for all $d \in D(\pi)$, if $d \subseteq \{\pi'\}$ for some $\pi' \in P$, then $(\pi, \pi') \notin C$. Its quotient is indeed a repository.*

PROOF. Let (P', D', C') be the tentative quotiented repository associated to repository (P, D, C), as defined above. The result is clear, except for irreflexivity of C', which we show below. We assume that $(\pi', \pi') \in C'$ for some package $\pi' \in P'$ and reach a contradiction. By definition of C', there exists $\pi_1 \in P$ and $\pi_2 \in P$ such that $\pi' = [\pi_1] = [\pi_2]$ and $(\pi_1, \pi_2) \in C$. By Remark 7.6, there exists a dependency $d \in D(\pi_2)$ such that $d \subseteq \{\pi_2\}$. By definition of quotienting, $D(\pi_1) = D(\pi_2)$, and thus $d \in D(\pi_1)$. But then, by hypothesis, we should have $(\pi_1, \pi_2) \notin C$. Hence a contradiction, as wanted. □

Remark 9.2. Let (P, D, C) be a repository. Let (I, F) be a healthy configuration of repository (P, D, C). We assume that the set F is a maximal set (with respect to inclusion) satisfying this property. Then, for all packages π in $P \setminus F$, there exists a package π' such that $(\pi, \pi') \in C$.

PROOF. We have $\{\pi\} \notin \nabla_C$. Indeed, otherwise, we would have, by definition, $\{\pi\} \cap F \neq \emptyset$, and thus $\pi \in F$. Then, by Theorem 7.2, there exists π' such that $(\pi, \pi') \in C$. □

LEMMA 9.3. *Let (P, D, C) be a flat repository. Let (I, F) be a healthy configuration of repository (P, D, C). We assume that the set F is a maximal set (with respect to*

inclusion) satisfying this property. Let $\pi \in P \setminus F$. Let $\pi' \in P$ such that $D(\pi') = D(\pi)$ (in other words, $\pi' \in [\pi]$). Then, there exists $d \in D(\pi')$ such that $d \subseteq \{\pi\}$.

PROOF. By Remark 9.2, there exists π'' such that $(\pi, \pi'') \in C$. By Remark 7.6, there exists $d \in D(\pi)$ such that $d \subseteq \{\pi\}$. We have $D(\pi') = D(\pi')$, hence the result. □

LEMMA 9.4. *Let (P, D, C) be a repository. Let (P', D', C') be the corresponding quotient repository. Let (I, F) be a healthy configuration of repository (P, D, C). We assume that the set F is a maximal set (with respect to inclusion) satisfying this property. Then $([I], \overline{[\overline{F}]})$ is a healthy configuration of repository (P', D', C'). (We write $\overline{F}$ for the complement $P \setminus F$ of set F.)*

PROOF. Let $F' = \overline{[\overline{F}]} = \{[\pi] \in P \,|\, \forall \pi' \in [\pi], \pi' \in F\}$.

We first prove abundance. Let $\pi'_0 \in [I]$ and $d'_0 \in D'(\pi'_0)$. There exists $\pi_0 \in I$ and $d_0 \in D(\pi_0)$ such that $\pi'_0 = [\pi_0]$ and $d'_0 = [d_0]$. By abundance in repository (P, D, C), we have $d_0 \cap F \neq \emptyset$. Suppose there exists $\pi \in d_0 \cap F$ such that $[\pi] \in F'$. Then, as $[\pi] \in d'_0$, we have $d'_0 \cap F' \neq \emptyset$ as wanted. Otherwise, for all package $\pi \in d_0 \cap F$, there exists a package $\pi' \in [\pi] \setminus F$. By Lemma 9.3, there also exists a dependency $d \in D(\pi)$ such that $d \subseteq \{\pi'\}$, and therefore $d \cap F = \emptyset$. We can apply Lemma 7.12 to compose d_0 with all these dependencies. Hence, there exists $d' \in D(\pi_0) \cup \nabla_C$ such that $d' \cap F = \emptyset$. But by abundance and maximality of F, we should have $d' \cap F \neq \emptyset$. We reach a contradiction. Hence, this second case is actually not possible.

We now show peace. The proof is by contradiction. Suppose that peace does not hold. Then, there exists $\pi'_1 \in F'$ and $\pi'_2 \in F'$ such that $(\pi'_1, \pi'_2) \in C'$. By definition of C', there exists a pair $(\pi_1, \pi_2) \in C$ such that $\pi'_1 = [\pi_1]$ and $\pi'_2 = [\pi_2]$. By definition of F', $\pi_1 \in F$ and $\pi_2 \in F$. Then, by peace in repository (π, D, C), $(\pi_1, \pi_2) \notin C$. We reach a contradiction, hence the result. □

LEMMA 9.5. *Let (P, D, C) be a repository. Let (P', D', C') be the corresponding quotient repository. Let (I, F) be a healthy configuration of repository (P', D', C'). We pose $I' = \{\pi \,|\, [\pi] \in I\}$ and $F' = \{\pi \,|\, [\pi] \in F\}$. Then (I', F') is a healthy configuration of repository (P, D, C).*

PROOF. We first prove abundance. Let $\pi \in I'$ and $d \in D(\pi)$. We need to show that $d \cap F' \neq \emptyset$. We have $[\pi] \in I$. By definition of D', we have $[d] \in D'([\pi])$. By abundance, we thus have $[d] \cap F \neq \emptyset$, that is, there exists a package $\pi \in d$ such that $[\pi] \in F$. We thus have $\pi \in F'$, hence the result.

We now prove peace, that is $C \cap (I' \times I') \neq \emptyset$, The proof is by contradiction. Let π_1 and π_2 be two packages in F' such that $(\pi_1, \pi_2) \in C$. We have $[\pi_1]$ and $[\pi_2]$ in F and, by definition of C', $([\pi_1], [\pi_2]) \in C'$. This contradicts peace in the quotient repository. Hence the result. □

Quotienting preserves flatness and keeps co-installability invariant.

THEOREM 9.6. *A set of packages Π is weakly co-installable in the flat repository (P, D, C) if and only if the set $[\Pi]$ is weakly co-installable in the associated quotient repository.*

PROOF. This is a direct consequence of Lemmas 9.4 and 9.5. □

THEOREM 9.7. *If a repository is flat, then the corresponding quotiented repository is flat as well.*

PROOF. Let (P, D, C) be a flat repository. We show that its quotient repository (P', D', C') is also flat.

We first prove reflexivity, that is, $\Delta_C \prec_C D'$. It is sufficient to prove that $\Delta_C \sqsubseteq D'$. Let $\pi_1, \pi_2 \in P$ such that $([\pi_1], [\pi_2]) \in C'$. We need to prove that there exists a dependency $d' \in D'([\pi_1])$ such that $d' \subseteq \{[\pi_1]\}$. By definition of C', there exists a pair of packages $(\pi_3, \pi_4) \in C$ such that $[\pi_3] = [\pi_1]$ and $[\pi_4] = [\pi_2]$. By reflexivity in the initial repository, either $\{\pi_3\} \in \nabla_C$ or there exists $d \in D(\pi_3)$ such that $d \subseteq \{\pi_3\}$. By Theorem 7.2, $\{\pi_3\} \notin \nabla_C$. Hence, there exists a dependency $[d]$ in $D'([\pi_3]) = D'([\pi_1])$ such that $[d] \subseteq \{[\pi_3]\} = \{[\pi_1]\}$, as wanted.

We now prove transitivity, that is, $D' \,;\, D' \prec_C D'$. Let π be a package. Let $d = \{\pi_1, ..., \pi_n\}$ a dependency such that $[d] \in D'([\pi])$ (that is, $d \in D(\pi)$).

Let d_i be dependencies such that $[d_i] \in D'([\pi_i])$ (that is, $d_i \in D(\pi_i)$) for each i. We need to prove that either $\bigcup_i [d_i] \in \nabla'_C$ (where ∇'_C is our class of always satisfiable dependencies for the quotiented repository) or there exists $d' \in D'([\pi])$ such that $d' \subseteq \bigcup_i [d_i]$. We assume that $\bigcup_i [d_i] \notin \nabla'_C$ and show we are in the second case. By Theorem 7.2, for all $[\pi] \in \bigcup_i [d_i]$, there exists a package $[\pi'] \in P' \setminus \bigcup_i [d_i]$ such that $([\pi], [\pi']) \in C'$. This means that we can define a function f which associates to each $[\pi] \in \bigcup_i [d_i]$ a package $f([\pi]) \in [\pi]$ such that there exists a package π'' such that $[\pi''] \in P' \setminus \bigcup_i [d_i]$ and $(f([\pi]), \pi'') \in C$. Note that $\pi'' \in P \setminus f(\bigcup_i [d_i])$. Hence, for any dependency d'', if $d'' \subseteq f(\bigcup_i [d_i])$, then $d'' \notin \nabla_C$. Now, by reflexivity and Theorem 7.2 both applied to $f(\pi'')$, for each $\pi'' \in d_i$, there exists a dependency $d'' \in D(f([\pi'']) = D(\pi'')$ such that $d'' \subseteq \{f([\pi''])\}$. Thus, for all i, by transitivity applied to d_i and the corresponding d'', there exists a dependency $d''_i \in D(\pi_i)$ such that $d''_i \subseteq f(\bigcup_j [d_j])$. By transitivity again, applied to dependency d and dependencies d''_i, there exists a dependency $d'' \in D(\pi)$ such that $d'' \subseteq f(\bigcup_i [d_i])$. We have $[d''] \in D'([\pi])$ and $[d''] \subseteq [f(\bigcup_i [d_i])] = \bigcup_i [d_i]$ as wanted. □

10. REFLEXIVE TRANSITIVE REDUCTION

It would not be suitable to graph directly a repository after flattening as it would be polluted by dependencies which are not informative: due to reflexivity, we have packages π where $\{\pi\} \in D(\pi)$, and some dependencies can be deduced from others by transitivity. Thus, we perform a kind of reflexive transitive reduction of the dependency function: given a repository (P, D, C), we find a minimal dependency function D' with the same flattening (that is, $\widehat{D} = \widehat{D'}$).

Because of disjunctive dependencies, the complexity of finding an optimal solution is high [Ausiello et al. 1983; Ausiello et al. 1986], in constrast with the case of reflexive transitive reduction for graphs. As this is mostly a cosmetic issue for us, we use a simple non-optimal algorithm. As a first step, we iteratively remove dependencies which are implied from other dependencies by transitivity, in a greedy way. The second step is to remove all self dependencies, that is, dependencies $d \in D(\pi)$ such that $\pi \in d$. Co-installability is left invariant by these operations.

LEMMA 10.1. *Let (P, D, C) be a repository. Let $\pi \in P$ be a package and $d \in D(\pi)$ be a dependency of this package. Let $D' = D \setminus \{\pi \mapsto d\}$ be the dependency function D where the dependency has been removed. If $d \in (D' \,;\, D')(\pi)$ then any healthy installation I of repository (P, D', C) is a healthy installation of repository (P, D, C).*

PROOF. Let I be a healthy installation of (P, D', C). We show that it is a healthy installation of (P, D, C). Peace is clear. We prove abundance.

Let $\pi' \in I$ and $d' \in D(\pi')$. We need to show that $d' \cap I \neq \emptyset$. If $\pi' \neq \pi$ or $d' \neq d$, then $d' \in D'(\pi')$ and the result holds by abundance in (P, D', C). We now show that $d \cap I \neq \emptyset$. As $d \in (D' \,;\, D')(\pi)$, there exists a dependency $d'' = \{\pi_1, \ldots, \pi_n\} \in D'(\pi)$ and, for all i, a dependency $d_i \in D'(\pi_i)$ such that $d = \bigcup_{1 \leq i \leq n} d_i$. By abundance, as $\pi \in I$, we

repeat
  $(P, D, C) \leftarrow$ *flatten*(P, D, C)
  $(P, D, C) \leftarrow$ *canonise*(P, D, C)
  $(P, D, C) \leftarrow (P, D, C) \setminus \nabla_C$
  $(P, D, C) \leftarrow$ *remove-clearly-broken*(P, D, C)
  $(P, D, C) \leftarrow$ *remove-redundant-conflicts*(P, D, C)
until the last two steps above have no effect
$(P, D, C) \leftarrow$ *remove-conflict-covered-deps*(P, D, C)
return *quotient*(P, D, C)

Fig. 14. Simplifying the repository.

have $d'' \cap I \neq \emptyset$, that is, there exists i such that $\pi_i \in I$. Then, by abundance again, we have $d_i \cap I \neq \emptyset$. Hence, $d \cap I \neq \emptyset$ as wanted. □

LEMMA 10.2. *Let (P, D, C) be a repository. Let D' be the dependency function defined by $D'(\pi) = \{d \in D \mid \pi \notin d\}$. Any healthy installation I of repository (P, D', C) is also a healthy installation of repository (P, D, C).*

PROOF. Let I be a healthy installation of (P, D', C). We show that it is a healthy installation of (P, D, C). Peace is clear. We prove abundance.

Let $\pi \in I$ and $d \in D(\pi)$. We need to show that $d \cap I \neq \emptyset$. If $\pi \in d$, this is the case. Otherwise, $d \in D'(\pi)$, and we can conclude by healthiness of I in (P, D', C). □

11. PUTTING IT ALL TOGETHER

We now have all the ingredients at hand to perform on any repository (P, D, C) the transformations that allow us to produce the final repository, which is suitable both for drawing a simplified graph, or performing efficiently various analysis related to co-installability.

11.1. Extracting a Co-Installability Kernel

The complete algorithm is shown in Figure 14. We first flatten the initial repository (Section 5), canonise the dependency function (Section 4), and remove the clearly irrelevant dependencies in ∇_C (Section 7). In our implementation, all these operations are performed simultaneously: this is significantly more efficient, as we have less dependencies to consider while flattening. Then, we set the dependencies of broken packages of the form of Figure 13 to the empty dependency $\emptyset$ (Section 8.4), and we remove redundant conflicts (Section 8.3). As changing the dependencies of broken packages may break flatness and removing may grow ∇_C, these fives steps are iterated until no change occurs. The process terminates as at each iteration either $D(\pi)$ is set to $\{\emptyset\}$ for a package π or a conflict is removed. In practice, only two iterations are performed: more iterations are only needed in unlikely configurations where dealing with a broken package exposes another package as broken. Finally, we remove the conflict covered dependencies that can be safely dropped (Section 8.2) and the repository is quotiented (Section 9).

By combining the results of the previous sections, we obtain the fundamental result on the simplification performed by the algorithm.

THEOREM 11.1. *The transformation performed by the simplification algorithm leaves co-installability invariant. This algorithm produces a flat repository.*

PROOF. First, the quotienting operation performed at the end indeed produces a repository, as the hypothesis of Lemma 9.1 are satisfied: if $\emptyset \in D(\pi)$, then package π has no conflicts thanks to redundant conflict removal; if $\{\pi'\} \in D(\pi)$, then $(\pi, \pi') \notin C$

thanks to the dependency strengthening of clearly broken packages. These two properties are preserved by dependency removal.

We now show that once the loop in Figure 14 is exited, the repository remains flat. Indeed, flattening produces a flat repository (Theorem 6.1 and Lemma 7.5), which remains flat by canonisation and removal of dependencies in ∇_C (Remark 7.4, Lemmas 7.7 and 7.10). Then, the removal of conflict cover dependencies is defined so as to preserve flatness, and quotienting preserves flatness (Lemma 9.7).

To show that co-installability is left invariant, we rely heavily on the equivalence between co-installability and weak co-installability in flat repositories (Lemma 6.3 and Theorem 7.8), which let us use weak co-installability preservation results to prove co-installability preservation. We also use the fact that healthiness preservation implies co-installability preservation (Remark 2.2). Then, co-installability is left invariant by flattening (Theorem 6.8), canonisation (Theorem 4.2), removal of dependencies in ∇_C (Theorem 7.11), dependency strengthening of clearly broken packages (Lemma 8.4 and Theorem 4.2), removal of redundant conflicts (Lemma 8.3 and Theorem 4.2), removal of conflict covered dependencies (Lemma 8.1) and Theorem 4.2), and quotienting (Theorem 9.6). □

As noticed above, on a repository with no broken package, it is not necessary to iterate the flattening phase, so the algorithm could run slighly faster; but finding all broken packages is slower than performing the whole simplification, as it requires to call a SAT solver repeatedly on large problems. On the other hand, repositories with good quality control should contain no broken packages, and a simpler version of the simplification algorithm could be used on them.

11.2. Computing Non Co-Installable Pairs

We are now in a position to compute efficiently all the pairs of packages that cannot be installed together, which are the *strong conflicts* of [Di Cosmo and Boender 2010]. First, we notice that, when checking whether a package is installable, one only need to consider features on which this package may depend.

LEMMA 11.2. *Let (P, D, C) be a repository. If the package $\pi \in P$ is weakly installable in this repository, there exists a set of features F such that the configuration $(\{\pi\}, F)$ is healthy and for all $\pi' \in F$ there exists a dependency $d \in D(\pi)$ such that $\pi' \in d$.*

PROOF. Let $\pi \in P$ be a weakly installable in repository (P, D, C). There exists a set of features F' such that $(\{\pi\}, F')$ is installable. We take $F = F' \cap \bigcup_{d \in D(\pi)} d$. Clearly, for all $\pi' \in F$ there exists a dependency $d \in D(\pi)$ such that $\pi' \in d$. We now prove that $(\{\pi\}, F)$ is healthy.

We first prove abundance. Let $d \in D(\pi)$. By abundance for configuration $(\{\pi\}, F')$, $d \cap F' \neq \emptyset$. By definition of F, we have $d \cap F = d \cap F'$. Hence $d \cap F \neq \emptyset$ as wanted.

We now prove peace. By peace for configuration $(\{\pi\}, F')$, $C \cap (F' \times F') = \emptyset$. As $F \subseteq F'$, we $C \cap (F \times F) = \emptyset$ as wanted. □

And then, we can prove that whenever two packages are not coinstallable in a flat repository, they directly share a conflict.

LEMMA 11.3. *If the packages π_1 and π_2 are both weakly installable in a repository (P, D, C) but the set $\{\pi_1, \pi_2\}$ is not weakly co-installable in this repository, there must exists $d_1 \in D(\pi_1)$, $d_2 \in D(\pi_2)$, $\pi'_1 \in d_1$, and $\pi'_2 \in d_2$ such that $(\pi'_1, \pi'_2) \in C$.*

PROOF. We prove the contrapositive. Let π_1 and π_2 be two packages in repository (P, D, C). We assume that there is no dependencies d_1 and d_2 as specified above and show that then the set $\{\pi_1, \pi_2\}$ is not weakly co-installable.

By Lemma 11.2, there exists sets of features F_1 and F_2 such that configurations $(\{\pi_1\}, F_1)$ and $(\{\pi_2\}, F_2)$ are healthy. Besides, $F_1 \subseteq \bigcup_{d \in D(\pi_1)} d$, and similarly for F_2. We want to show that the set $\{\pi_1, \pi_2\}$ is not weakly co-installable. We define $F = F_1 \cup F_2$ and show that the configuration $(\{\pi_1, \pi_2\}, F)$ is healthy.

We first prove abundance. Let $\pi \in \{\pi_1, \pi_2\}$ and $d \in D(\pi)$. We need to show that $d \cap F \neq \emptyset$. Suppose $\pi = \pi_1$. Then, by abundance, $d \cap F_1 \neq \emptyset$. Thus, $d \cap F \neq \emptyset$ as wanted. We get the same result when $\pi = \pi_2$.

We now prove peace, that is, $C \cap (F \times F) = \emptyset$. The proof is by contradiction. Let π and π' in F such that $(\pi, \pi') \in C$. If π and π' are both in F_1, we reach a contradiction as, by peace for configuration $(\{\pi_1\}, F_1)$, we have $(\pi, \pi') \notin C$. The same result holds if they are both in F_2. We now assume that $\pi \in F_1$ and $\pi' \in F_2$. By definition of F_1, there exists $d_1 \in D(\pi_1)$ such that $\pi \in D_1$. Similarly, there exists $d_2 \in D(\pi_2)$ such that $\pi' \in D_2$. Besides, $(\pi, \pi') \in C$. This case is not possible, as this contradicts our initial assumption that no such dependencies existed. The last case, where $\pi \in F_1$ and $\pi' \in F_2$, is not possible as well. □

To find all non co-installable pairs, we iterate over all such pairs (in the quotiented repository, which is way smaller that the initial repository) and check co-installability using a SAT solver. This way we can find all strong conflicts in a few seconds, versus 5 days in [Di Cosmo and Boender 2010]. One major reason for this huge performance improvement is that the transformations leading to the flat repository prune a large part of the search space by removing many dependencies that would otherwinse need to be explored. Another part of the performance improvement comes from quotienting, which makes it possible to test a large number of packages at once.

Besides the usage in quality assurance advocated in [Di Cosmo and Boender 2010], this information can be used to give a sense of how much a package may be problematic: we can count the number of packages it cannot be installed with, and use this information to emphasize the nodes with more strong conflicts.

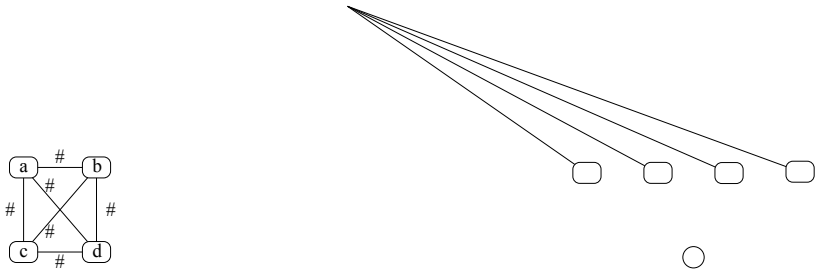
11.3. Drawing a Simplified Graph

Before drawing the final repository, we perform the transitive reflexive reduction of Section 10. The structure of the graph is then passed as input to the `dot` program of the Graphviz toolkit [Ellson et al. 2001] that performs the layout.

It is important to name nodes using meaningful representatives of each equivalence class: we give preference to packages π that are directly involved in conflicts, as they have more chances to be relevant for the repository maintainers; these are easy to find by checking if $\{\pi\} \in D(\pi)$.

There can be many packages all mutually in conflict. For instance, this is the case of all mailer agents in Debian. We identify maximal such cliques and draw them in a more concise way, as shown in Figure 15.

We compute strong conflicts [Di Cosmo and Boender 2010] (non-coinstallable pairs of packages) as described in Section 11.2 above, and we colour nodes according to the logarithm of the number of packages it is incompatible with. This way one can immediately identify in the picture the nodes that prevent the installation of many other packages.



(a) Explicit representation (b) Compact representation

Fig. 15. Conflict clique

12. THE `COINST` TOOL

All the transformations described above have been implemented by Jérôme Vouillon in the `coinst` tool, that can be used to extract and visualize the coinstallability kernel for any GNU/Linux distribution.

The `coinst` program is a command line tool that takes package information from standard input in either one of the most popular formats among GNU/Linux distributions, RPM and DEB; DEB format is the default, and RPM metadata can be specified using the `-rpm` option.

It writes on standard output:

— the list of equivalence classes, providing, for each, a selected representative package as well as all the packages it corresponds to;
— the list of packages that cannot be installed at all;
— the list of non-coinstallable pairs: for each package, the tool lists all packages it can never be installed with.

The coinstallability kernel is written as a dot graph which can be passed to the Graphviz tool suite [Ellson et al. 2001]; the file name can be specified with the `-o` option.

To unclutter the results, `coinst` omits by default packages with simple conflict configurations, but one can use the `-all` option to include all packages in the output.

A typical run of the tool will look like this:

Compute If you have all the metadata for Ubuntu Maverick in a directory `~/maverick-beta/`, you can run the tool by issueing the command

```
zcat ~/maverick-beta/* | coinst -all -o g.dot
```

Layout The graph can be layed out, using the Graphviz `dot` tool

```
dot g.dot -o graph.dot
```

Display The result can be visualised using your choice of tool; we strongly recommend to use the companion `coinst_viewer` tool, though:

```
coinst_viewer graph.dot
```

The `-ignore` option can be used to ignore a package, like in

```
zcat ~/testing-2010-08-22/* | coinst -ignore liboss-salsa-asound2 -o g.dot
```

To focus on a particular package, use the `-root` option.

```
zcat ~/testing-2010-08-22/* | coinst -root libhdf5-openmpi-1.8.4 -o g.dot
```

Artifact evaluation and availability: `coinst` has been evaluated by the ESEC/FSE 2011 Artifact Evaluation Committee, and it has been awarded the *Distinguished Artifact Award*. Full source code, build and installation instructions as well as some running examples of this tool can be found online at `http://coinst.irill.org`.

13. VISUALIZING UBUNTU

In the Figures 16, 17 and 18, we can see the result of applying the simplifications described in this paper to the main section of release 10.10 of the Ubuntu GNU/Linux distribution; solid arrows indicate dependencies, dotted lines indicate conflicts, and conflict cliques are represented with a node containing a # connected with dotted lines to all packages in the clique. Of the thousands of packages, and dozens of thousands of relationships, only a handful are left, and it is possible to *read* interesting information directly on the graph.

We give here just a few examples of what can be easily discovered at a glance without needing to sift through thousands of hyperlinked web pages.

In Figure 16 we see all the simple binary conflicts; for example, `qt-x11-free-dbg` is in conflict with 24 packages which are equivalent to `libqt4-dbg`. The isolated node in the bottom left, named `abrowser`, stands for 7049 packages that are *always* installable. The four conflict cliques in Figure 17 and the conflict clique in Figure 18 are justified, as they all correspond to incompatible implementations of libraries which are compiled with different backends. In Figure 18, we see that package `libjpeg8-dev` is in conflict with 28 other packages, which is likely to make it unconvenient to use. The core package `ubuntu-desktop` is in conflict with a number of packages. These packages should either be removed from the main section, or the dependencies should be revised. For instance, packages `foomatic-db` and `libgd2-noxpm` should probably be removed. On the other hand, package `libsdl1.2debian-all` provides a superset of the functionalities of `libsdl1.2debian-pulseaudio`. Thus, it should not be in conflict with `ubuntu-desktop`. Overall, there are still a few issues left in this repository, which are immediately apparent when looking at the output of our tool. Since this repository corresponds to a stable distribution that has undergone extensive testing and quality assurance, we believe that this is clear proof of the effectiveness of our approach.
Of course, our tool is even more useful for detecting issues proactively when the distribution is in flux (during alpha and beta stages).

14. RELATED WORKS

Ensuring the correct behaviour of components when composed into an assembly is a fundamental concern for modern software architectures, and has been extensively studied, but the maintenance of GNU/Linux distributions is quite a different subject, that poses new challenges, and has started to be studied formally only recently. In this section, we survey various related reserch ares, pointing out the differences with our work, which is, to the best of our knowledge, completely novel.

A large body of literature is dedicated to the *dynamic* aspects of component composition: knowing the behaviour of the components, one looks for means to determine the behaviour of the system obtained by assembling them [de Alfaro and Henzinger 2001], or to ensure certain properties of the composition [Inverardi et al. 2000; Tivoli and Inverardi 2008]; when confronted with large systems, one may look for ways of detecting automatically behavioural incompatibilities from the component source code [McCamant and Ernst 2003; 2004], or to deploy and upgrade such systems [Ajmani et al. 2006; Crameri et al. 2007].

These are important issues, but in the world of GNU/Linux distributions we are still very far from having any information available on the behaviour of each component after installation, so it is too early to tackle this facet of the problem.

On the other side, the mainstream research on *static* inter-module dependencies is essentially performed at the level of the source code, with a different focus: in [Nagappan and Ball 2007a; Neuhaus et al. 2007a], dependencies are automatically extracted from huge sets of source code, and then used to predict failures, but not to identify issues in the architecture of the code, unlike what our co-installability kernel allows to do; in [Yoon et al. 2007] and [Yoon et al. 2008] dependencies are used as a guideline for testing component-based systems; finally, [Pei-Breivold et al. 2008], which shares some concerns similar to ours, analyses the architectural dependencies to improve the modularisation of the software architecture, but the size of the problem is sufficiently small (some 20 components) to allow manual analysis and resolution of component relationships, which is totally unfeasible in our case.

In the area of quality assurance for large software projects, many authors have correlated component dependencies with past failure rates, in order to predict future failures [Nagappan and Ball 2007b; Neuhaus et al. 2007b; Zimmermann and Nagappan 2008]. The underlying hypothesis is that software "fault-proneness" of a component is correlated to changes in components that are tightly related to it. In particular, if a component A has many dependencies on a component B and the latter changes a lot between versions, one would expect that errors propagate through the "component network" reducing the reliability of A. A related interesting statistical model to predict failures over time is the *weighted time damp model* that correlates recent changes to software fault-proneness [Graves et al. 2000]. Social network methods [Hanneman and Riddle 2005] have also been used to validate and predict the list of *sensitive* components in the Windows platform [Zimmermann and Nagappan 2008].

Our work is part of a recent research area that focuses on the properties of component repositories that can be established automatically without looking at the source code of the components, and without testing them: it is only assumed that each component carries with itself a small amount of metadata describing what the component provides and what it requires to be deployed and run; this metadata may be inferred automatically using tools, or built manually, or a mixture of the two, but what is important to notice is that this is the raw material which is used by researchers in this area.

A first natural question in this setting is how difficult is to determine whether a component can be installed at all: in [Mancinelli et al. 2006], it has been shown that this problem is NP-complete, but tractable in practice, for packages in GNU/Linux distributions, and this result has been recently generalised to a wide class of component systems, including OSGI bundles and Eclipse plugins [Abate et al. 2012a]. Since feature diagrams, used in software product lines, can be encoded as component

repositories [Di Cosmo and Zacchiroli 2010], all the problems related to configuration management can be equivalently stated in terms of repositories.

An important issue for users of component repositories is computing installations that optimise some given objective functions: various experiments have been done since 2006 [Tucker et al. 2007; Argelich et al. 2010; Trezentos et al. 2010; Di Cosmo et al. 2011; Gebser et al. 2011], and are now organised around the Mancoosi International Solver Competition (`http://www.mancoosi.org/misc-2011/`); the results of this research line have been used to develop next generation package mangers [Abate et al. 2011], and several of these solvers are now available as plugins for the Debian package manager, `apt`, starting from version `0.9.5.1`.

To help repository maintainers in their quality assurance task, It has have been shown relevant the ability to identify what other components a package will always need [Abate et al. 2009], what pairs of packages are incompatible [Di Cosmo and Boender 2010], and what component upgrades are more likely to impact a repository [Abate et al. 2012b].

The present work is a significant step forward in this direction, as it provides for the first time a means to produce a very compact representation of all the sources of incompatibilities in a repository that stem from its component metadata.

It is also the first work, to our knowledge, that deals with the fine structure of repositories directly, and not throuh encodings: connections between component repositories and boolean satisfiability and constraint solving have been made only a few years ago in the framework of GNU/Linux distributions [Mancinelli et al. 2006; Tucker et al. 2007] and the Eclipse platform [Le Berre and Parrain 2008], but these connections, and other recent developments such as [Abate et al. 2009; Di Cosmo and Boender 2010; Abate et al. 2012b] do not exploit the *special structure* of the dependencies and conflicts found in a repository.

15. FUTURE WORK

The underlying structure of software component repositories exposed here can be seen as a generalisation of some well known mathematical structures : prime event structure [Winskel 1987] correspond to repositories without loops and disjunctions; directed hypergraphs [Ausiello et al. 1983] correspond to repositories without conflict arcs, and Dual Horn theories correspond to repositories without conflicts [Dowling and Gallier 1984].

The notions of *strongly flat* and *flat* repository introduced in this article, together with the simplification that preserve co-installability, can be used as building blocks for tackling new interesting problems for the maintenance of component repositories. A first natural question that we will investigate is how co-installability evolves along with repository evolution: it is quite interesting for quality assurance to identify those set of components that are co-installable in a repository, and are no longer co-installable in a repository that has been updated with newer versions of some components.

We will also continue to investigate the mathematical properties of the structure of component repositories.

16. CONCLUSIONS

In this paper, we have developed a theory and algorithms to extract from a repository a co-installability kernel, which can be seen as a minimal representation of the dependency and conflict relations: despite the apparent simplicity of the definition of the problem, and the intuitive appealing of the hypergraph transformations we have developed, the proofs of the crucial properties turned out to be surprisingly complex, so we decided to machine check them using Coq [The Coq Development Team 2008].

The results presented here allow to quickly identify non co-installable components in a repository, and pave the way to attacking significantly more complex problems concerning software component repositories.

More generally, we believe this work clearly shows the interest of the mathematical objects underlying software repositories, which turn out to be amenable to an elegant formal treatment and of high practical interest.

Acknowledgements. We thank the anonymous referees for their suggestions and contributions.

REFERENCES

ABATE, P., BOENDER, J., DI COSMO, R., AND ZACCHIROLI, S. 2009. Strong dependencies between software components. In *ESEM*. IEEE Press, 89–99.

ABATE, P., DI COSMO, R., TREINEN, R., AND ZACCHIROLI, S. 2011. Mpm: a modular package manager. In *Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering*. CBSE '11. ACM, New York, NY, USA, 179–188.

ABATE, P., DI COSMO, R., TREINEN, R., AND ZACCHIROLI, S. 2012a. Dependency solving: a separate concern in component evolution management. *J. Syst. Softw. 85,* 10, 2228 – 2240. Automated Software Evolution.

ABATE, P., DI COSMO, R., TREINEN, R., AND ZACCHIROLI, S. 2012b. Learning from the Future of Component Repositories. In *15th International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE-2012)*. ACM, Bertinoro, Italie.

AJMANI, S., LISKOV, B., AND SHRIRA, L. 2006. Modular software upgrades for distributed systems. In *ECOOP*, D. Thomas, Ed. Lecture Notes in Computer Science Series, vol. 4067. Springer, 452–476.

ARGELICH, J., LE BERRE, D., LYNCE, I., MARQUES-SILVA, J., AND RAPICAULT, P. 2010. Solving Linux upgradeability problems using boolean optimization. In *LoCoCo: Logics for Component Configuration*. EPTCS Series, vol. 29. 11–22.

AUSIELLO, G., D'ATRI, A., AND SACCÀ, D. 1983. Graph algorithms for functional dependency manipulation. *J. ACM 30,* 4, 752–766.

AUSIELLO, G., D'ATRI, A., AND SACCÁ, D. 1986. Minimal representation of directed hypergraphs. *SIAM J. Comput. 15,* 2, 418–431.

BAADER, F. AND NIPKOW, T. 1998. *Term rewriting and all that*. Cambridge University Press.

BUNING, H. K. AND LETTMANN, T. 2002. Propositional logic: Deduction and algorithms. *Studia Logica 71*, 247–258.

CRAMERI, O., KNEZEVIC, N., KOSTIC, D., BIANCHINI, R., AND ZWAENEPOEL, W. 2007. Staged deployment in mirage, an integrated software upgrade testing and distribution system. *SIGOPS Oper. Syst. Rev. 41,* 6, 221–236.

DE ALFARO, L. AND HENZINGER, T. A. 2001. Interface automata. In *ESEC / SIGSOFT FSE*.

DI COSMO, R. AND BOENDER, J. 2010. Using strong conflicts to detect quality issues in component-based complex systems. In *ISEC '10: Proceedings of the 3rd India software engineering conference*. ACM, New York, NY, USA, 163–172.

DI COSMO, R., DURAK, B., LEROY, X., MANCINELLI, F., AND VOUILLON, J. 2006. Maintaining large software distributions: new challenges from the FOSS era. In *Proceedings of the FRCSS 2006 workshop*. EASST Newsletter.

DI COSMO, R., LHOMME, O., AND MICHEL, C. 2011. Aligning component upgrades. In Proceedings Second Workshop on Logics for Component Configuration, C. Drescher, I. Lynce, and R. Treinen, Eds. *EPTCS 65*, 1–11.

DI COSMO, R., MANCINELLI, F., BOENDER, J., VOUILLON, J., DURAK, B., LEROY, X., PINHEIRO, D., TREZENTOS, P., MORGADO, M., MILO, T., ZUR, T., SUAREZ, R., LIJOUR, M., AND TREINEN, R. 2006. Report on formal mangement of software dependencies. Tech. rep., EDOS. Apr. EDOS project Deliverable 2.2, available as `http://www.edos-project.org/xwiki/bin/download/Main/Deliverables/edos-wp2d2.pdf`.

DI COSMO, R. AND VOUILLON, J. 2011. On software component co-installability. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13rd European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, T. Gyimóthy and A. Zeller, Eds. ACM, 256–266.

DI COSMO, R. AND ZACCHIROLI, S. 2010. Feature diagrams as package dependencies. In *SPLC*, J. Bosch and J. Lee, Eds. Lecture Notes in Computer Science Series, vol. 6287. Springer, 476–480.

DOWLING, W. F. AND GALLIER, J. H. 1984. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *J. Log. Program. 1,* 3, 267–284.

ELLSON, J., GANSNER, E., KOUTSOFIOS, L., NORTH, S., WOODHULL, G., DESCRIPTION, S., AND TECHNOLOGIES, L. 2001. Graphviz : open source graph drawing tools. In *Lecture Notes in Computer Science*. Springer-Verlag, 483–484.

GEBSER, M., KAMINSKI, R., AND SCHAUB, T. 2011. aspcud: A linux package configuration tool based on answer set programming. In *LoCoCo*, C. Drescher, I. Lynce, and R. Treinen, Eds. EPTCS Series, vol. 65. 12–25.

GRAVES, T. L., KARR, A. F., MARRON, J. S., AND SIY, H. 2000. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng. 26,* 7, 653–661.

HANNEMAN, R. A. AND RIDDLE, M. 2005. *Introduction to social network methods*. University of California, Riverside.

INVERARDI, P., WOLF, A. L., AND YANKELEVICH, D. 2000. Static checking of system behaviors using derived component assumptions. *ACM Trans. Softw. Eng. Methodol. 9,* 3, 239–272.

LE BERRE, D. AND PARRAIN, A. 2008. On sat technologies for dependency management and beyond. In *SPLC (2)*, S. Thiel and K. Pohl, Eds. Lero Int. Science Centre, University of Limerick, Ireland, 197–200.

MANCINELLI, F., BOENDER, J., DI COSMO, R., VOUILLON, J., DURAK, B., LEROY, X., AND TREINEN, R. 2006. Managing the complexity of large free and open source package-based software distributions. In *ASE*. IEEE Computer Society, 199–208.

MCCAMANT, S. AND ERNST, M. D. 2003. Predicting problems caused by component upgrades. In *ESEC / SIGSOFT FSE*. ACM, 287–296.

MCCAMANT, S. AND ERNST, M. D. 2004. Early identification of incompatibilities in multi-component upgrades. In *ECOOP*, M. Odersky, Ed. Lecture Notes in Computer Science Series, vol. 3086. Springer, 440–464.

NAGAPPAN, N. AND BALL, T. 2007a. Using software dependencies and churn metrics to predict field failures: An empirical case study. *ESEM 2007 0*, 364–373.

NAGAPPAN, N. AND BALL, T. 2007b. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *ESEM*. 364–373.

NEUHAUS, S., ZIMMERMANN, T., HOLLER, C., AND ZELLER, A. 2007a. Predicting vulnerable software components. In *Proceedings of CCS 2007*, P. Ning, S. D. C. di Vimercati, and P. F. Syverson, Eds. ACM, 529–540.

NEUHAUS, S., ZIMMERMANN, T., HOLLER, C., AND ZELLER, A. 2007b. Predicting vulnerable software components. In *ACM Conference on Computer and Communications Security*. 529–540.

PEI-BREIVOLD, H., CRNKOVIC, I., LAND, R., AND LARSSON, S. 2008. Using dependency model to support software architecture evolution. In *Proceedings of Evol'08*.

THE COQ DEVELOPMENT TEAM. 2008. *The Coq Proof Assistant Reference Manual – Version V8.2*.

TIVOLI, M. AND INVERARDI, P. 2008. Failure-free coordinators synthesis for component-based architectures. *Sci. Comput. Program. 71,* 3, 181–212.

TREINEN, R. AND ZACCHIROLI, S. 2008. Solving package dependencies: from EDOS to Mancoosi. In *DebConf8 (9th annual conference of the Debian project) DebConf8*. Argentine, 18–43.

TREZENTOS, P., LYNCE, I., AND OLIVEIRA, A. L. 2010. Apt-pbo: solving the software dependency problem using pseudo-boolean optimization. In *ASE*, C. Pecheur, J. Andrews, and E. D. Nitto, Eds. ACM, 427–436.

TUCKER, C., SHUFFELTON, D., JHALA, R., AND LERNER, S. 2007. Opium: Optimal package install/uninstall manager. In *ICSE*. IEEE Computer Society, 178–188.

WINSKEL, G. 1987. Event structures. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, W. Brauer, W. Reisig, and G. Rozenberg, Eds. Lecture Notes in Computer Science Series, vol. 255. Springer Berlin / Heidelberg, 325–392.

YOON, I.-C., SUSSMAN, A., MEMON, A., AND PORTER, A. 2007. Direct-dependency-based software compatibility testing. In *Proceedings of ASE '07*. ACM, New York, NY, USA, 409–412.

YOON, I.-C., SUSSMAN, A., MEMON, A., AND PORTER, A. 2008. Effective and scalable software compatibility testing. In *Proceedings of ISSTA '08*. ACM, New York, NY, USA, 63–74.

ZIMMERMANN, T. AND NAGAPPAN, N. 2008. Predicting defects using network analysis on dependency graphs. In *ICSE'08*. ACM, 531–540.

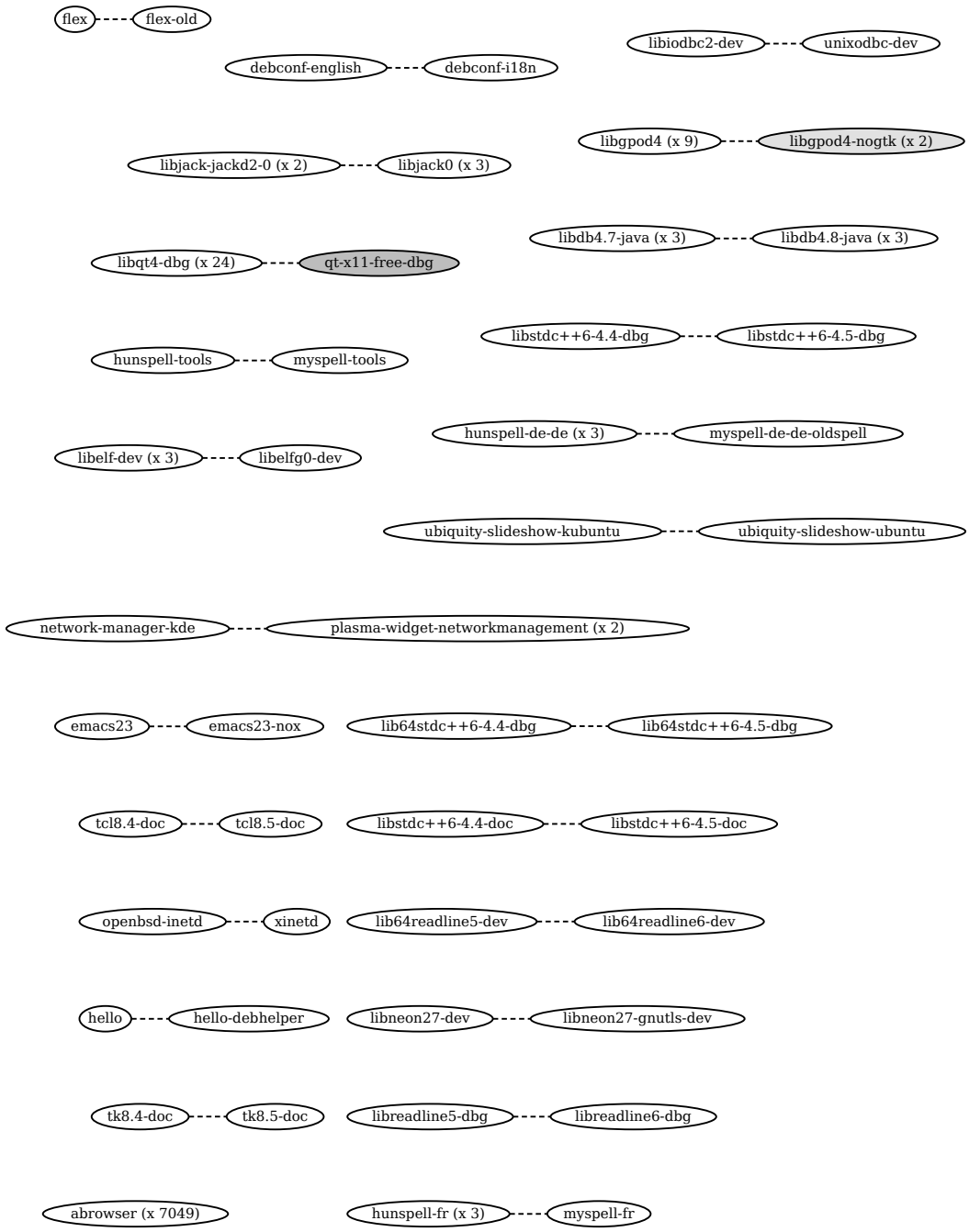


Fig. 16. Output graph for Ubuntu 10.10 (main), simple conflicts

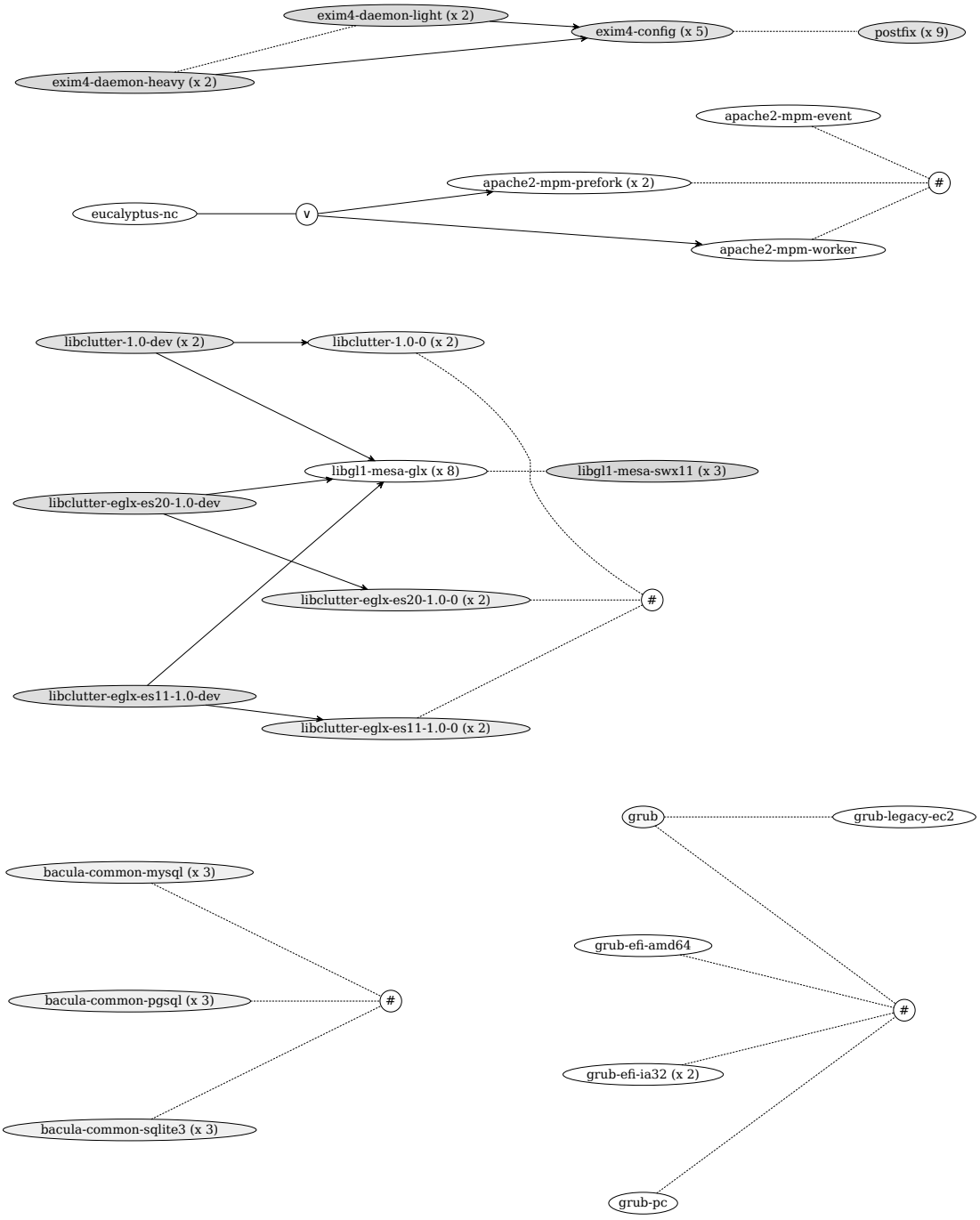


Fig. 17. Output graph for Ubuntu 10.10 (main), more complex cases

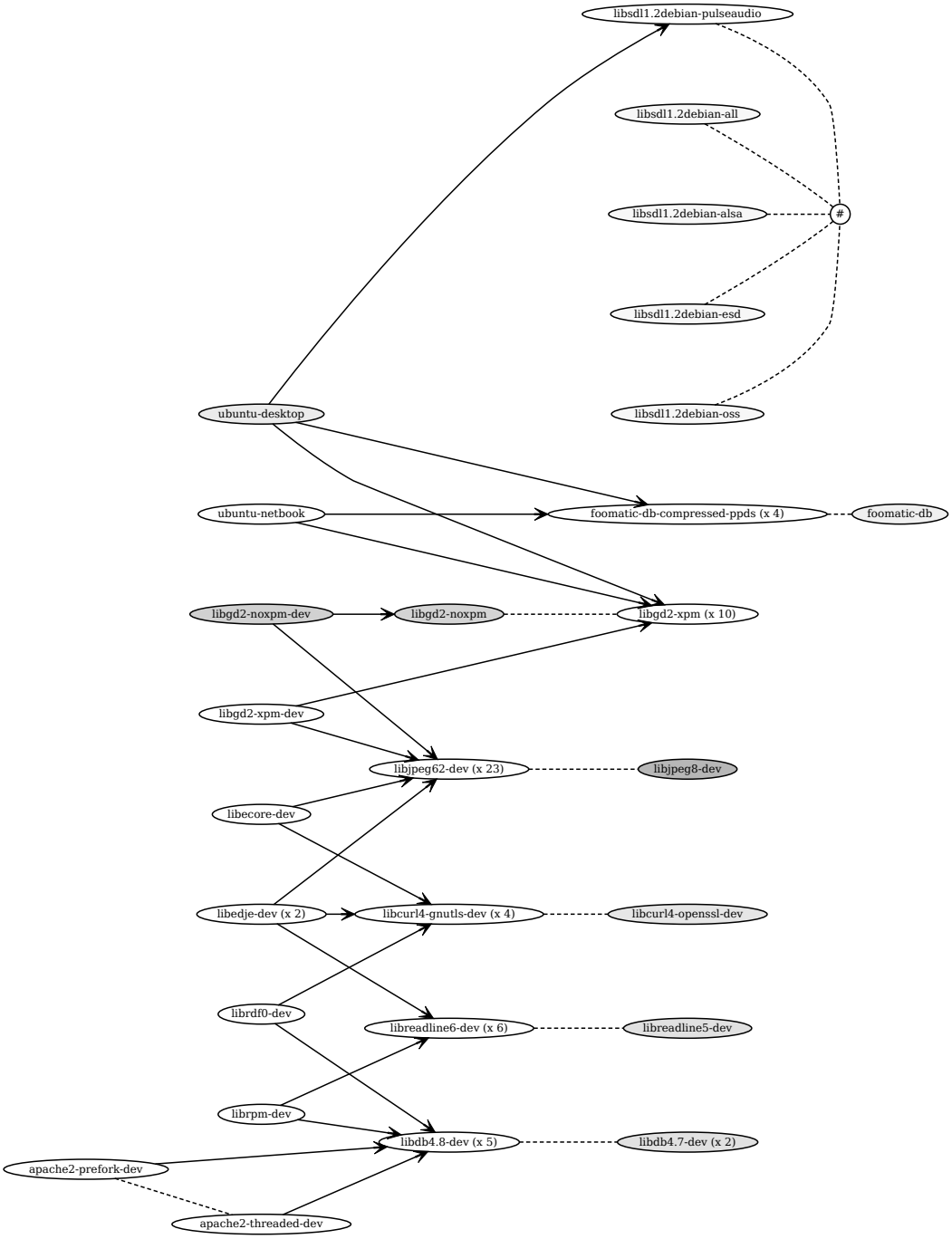


Fig. 18. Output graph for Ubuntu 10.10 (main), last and largest connected component