On Software Component Co-Installability

Jérôme Vouillon
CNRS, PPS UMR 7126
Univ Paris Diderot, Sorbonne Paris Cité
F-75205 Paris, France
jerome.vouillon@pps.jussieu.fr

Roberto Di Cosmo
Univ Paris Diderot, Sorbonne Paris Cité
PPS, UMR 7126 CNRS, and Inria
F-75205 Paris, France
roberto@dicosmo.org

ABSTRACT

Modern software systems are built by composing components drawn from large *repositories*, whose size and complexity is increasing quickly. 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 to associate to each concrete component repository a much smaller one with a simpler structure, but with equivalent co-installability properties. This smaller repository can be represented graphically, giving 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 in Coq.

Keywords

component, dependencies, conflicts, co-installability, package management, open source

General Terms

Algorithms, Theory, Verification

Categories and Subject Descriptors

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

Partially supported by the European Community's 7th Framework Programme (FP7/2007-2013), grant agreement n°214898, "Mancoosi" project.
Work performed at the IRILL center for Free Software Research and Innovation (`http://www.irill.org`).

ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.

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.

A typical example of the metadata attached to a component, taken from the Debian GNU/Linux distribution, is shown in Example 1: the logical language used for expressing dependencies and conflicts is quite powerful, as it allows conjunctions (symbol '`,`'), disjunctions (symbol '`|`') and version constraints.

EXAMPLE 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 [1, 2, 3]. 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 [4], and those that it can never be installed with [5].

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

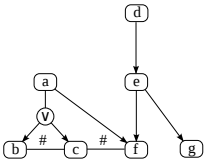


Figure 1: Graphical depiction of a repository

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 more 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 allows to associate to each concrete component repository a much smaller repository with a simpler structure, but with equivalent co-installability properties. This smaller repository can be represented graphically, giving 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 the strong conflicts defined in [5].

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

The paper 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 quotiented in Section 9. Finally, we show how to draw a simplified graph of the kernel of a repository in Section 11. We discuss related works, and conclude in Section 13.

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), or *versioned constraints* (e.g. `libc6 (>= 2.7)` in Example 1) can be easily preprocessed out [1], 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 [1, 6, 7, 8], 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 $\pi \in P$.

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.

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 [1]. 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.

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 [5], 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.

The key steps of our approach 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 in 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 [9]): the dependency function of the repository in Figure 3 has size $3n$, and when expanded, gives rise to a repository whose dependency function has size 2^n.

This is a strong limiting result, but we are only interested in studying co-installability 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 co-installability 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.

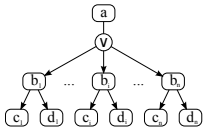


Figure 3: Repository that blows-up when flattened

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.

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 the pruning defined above);

(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 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.

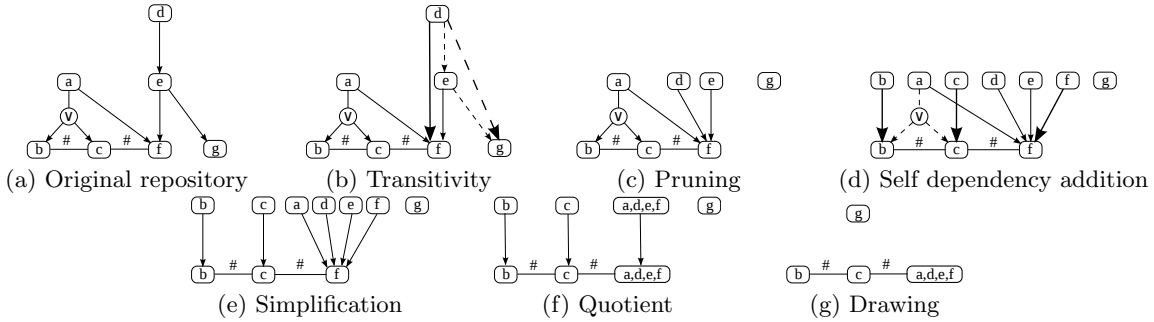


Figure 2: Transformations of a repository (added dependencies are in bold, dotted ones are removed in the next phase)

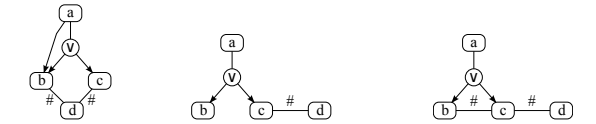


Figure 4: Some irrelevant dependencies

Figure 5: A redundant conflict

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 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 [10]. 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 on Table 1.

We can notice that 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.

Table 1: Repository sizes

	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

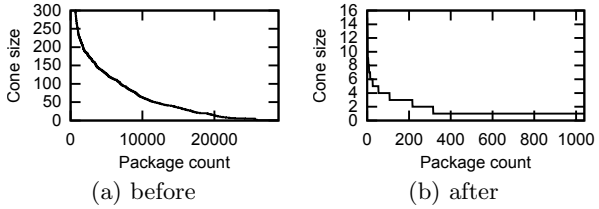


Figure 6: Distribution of cone size (Debian testing)

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 6): 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 letter size paper (see Figure 14 in Section 12) and can be easily inspected visually for errors. The corresponding graph

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

Figure 7: An example of dependency functions related by the preorder

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* [5], which is the simplest case of co-installability. On the same data as [5], the computation takes a few seconds, instead of the several days reported there. This time is in fact included in the running times of Table 1 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 7a is subsumed by the dependency function D' of the repository shown in Figure 7b.

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 induces an equivalence relation over dependency functions, and 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:

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

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

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

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

Given two repositories related under this preorder, going from the right hand side one to the left hand side one consists in removing conflicts or relaxing dependencies, thus making it easier to install packages.

THEOREM 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).*

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 monotonous, 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 $\widehat{D}(\pi)$.

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 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 following result (illustrated by Figure 2e):

	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	$\emptyset$	$\emptyset$

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

A flattened repository satisfies two properties (Theorem 3 below): a reflexivity property (if a package has a conflict, then it depends on itself), and a transitivity property (dependencies are stable under composition). Co-installability in such repositories, that we call *strongly flat*, can be shown equivalent to a more convenient property, that we call *weak co-installability* (Theorem 4); this is instrumental to 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).

We define precisely these two properties and study repositories that satisfies them.

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 3. *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.*

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 (that we refer to as *features* in the following) 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:

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.

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

THEOREM 4. *Any set of packages Π weakly co-installable in a strongly flat repository are also co-installable.*

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

LEMMA 5. *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:*

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

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

THEOREM 6. *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 $(P, \widehat{D}, C)$.*

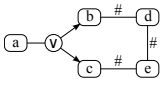


Figure 8: Illustration of monotony requirement

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 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 11), and for which co-installability and weak co-installability still coincide (Theorem 10), 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:

> **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 8: 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. *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.

THEOREM 8. *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 $\pi' \in d$.*

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.

As we shall see, weak co-installability is left invariant 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, so we need a weaker notion 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, for every dependency $d \in D(\pi)$, *either* $d \in \nabla_C$ *or* there exists a dependency $d' \in D'(\pi)$ such that $d' \subseteq d$.

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 9. *Any strongly flat repository is flat.*

THEOREM 10. *Any set of packages* Π *weakly co-installable in a flat repository are co-installable in this repository.*

THEOREM 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.*

One can still reason on flat repositories, as far as co-installability is concerned, as if their dependency function was transitive: just choose installations (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.)

8. IRRELEVANT CONSTRAINTS

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

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 9, 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, not explicited, 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.

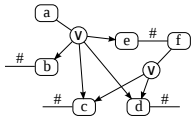


Figure 9: Dependency covered by the conflict requirements.

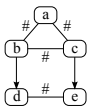


Figure 10: Redundant conflict belonging to a clique

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 leave co-installability invariant.

LEMMA 12. *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).

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 13. *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* $d \not\subseteq d'$.

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.

The graph reduction rule defined by this simplification has unsolvable critical pairs, so the result of this simplification could depend on the order of removal. 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 14. *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).

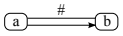


Figure 11: Dependence on conflicting package

Removing redundant conflicts involves a trade off. On one side, it may allow to remove some additional dependencies; on the other, it can also break some interesting structures. In Figure 10, 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 11: 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 15. *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).*

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: $[\Pi] = \{[\pi] \,|\, \pi \in \Pi\}$. 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: $P' = P/\equiv = \{[\pi] \,|\, \pi \in P\}$;
- the dependency function D' is such that $D'([\pi]) = \{[d] \,|\, d \in D(\pi)\}$ for all $\pi \in P$;
- the conflict relation C' is defined by

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

If the original repository does not contain dependencies of the form of Figure 11 nor redundant conflicts, then the quotient repository is indeed a repository (the key point to check is irreflexivity of the conflict relation).

LEMMA 16. *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.*

Quotienting preserves flatness and keeps co-installability invariant.

THEOREM 17. *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.*

THEOREM 18. *If a repository is flat, then the corresponding quotiented repository is flat as well.*

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 [11, 12], in constract 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 19. *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).*

LEMMA 20. *Let (P, D, C) be a repository. Let D' be the dependency function defined by $D'(\pi) = \{d \in D \,|\, \pi \notin d\}$. Any healthy installation I of repository (P, D', C) is also a healthy installation of repository (P, D, C).*

11. PUTTING ALL TOGETHER

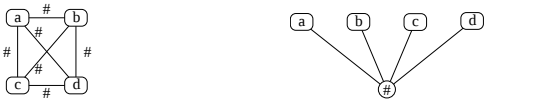
We now have all the ingredients at hand to perform on any repository (P, D, C) the transformations that allow to produce the final repository, which is suitable both for drawing a simplified graph, or performing efficiently various analysis related to co-installability.

11.1 Extracting a Co-Installability Kernel

The complete algorithm is shown in Figure 12. 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 11 to the empty dependency $\emptyset$ (Section 8.4), and we remove redundant conflicts

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

Figure 12: Simplifying the repository.



(a) Explicit representation (b) Compact representation

Figure 13: Conflict clique

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

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 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 [13] 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 13.

We compute strong conflicts [5] (non-coinstallable pairs of packages) and use this information to emphasise packages that prevent the installation of many other packages.

12. VISUALIZING UBUNTU

In Figure 14 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 couple of examples. The isolated node in the middle left stands for 7049 packages that are *always* installable. The five conflict cliques are justified, as they all correspond to incompatible implementations of libraries which are compiled with different backends. One can 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 few issues with this repository. Our tool is most useful for detecting issues proactively when the distribution is in flux (during alpha and beta stages).

13. RELATED AND FUTURE WORK

Many relevant research issues stem from the concrete and widespread applications of software component repositories: identifying components that can never be installable has been shown NP-complete, but tractable in practice [1], and there is currently very active research on computing installations that optimise some given objective functions, organised around the Mancoosi International Solver Competition (`http://www.mancoosi.org/misc-2010/`); determining what other components a package will always need [4] and what pairs of packages are incompatible [5] have been shown to be relevant for quality assurance in package repositories; finally, since feature diagrams, used in software product lines, can be encoded as component repositories [14], all the problems related to configuration management can be equivalently stated in terms of repositories.

Component repositories have been in widespread use for quite a long time, but Little research work has been devoted to study the *formal* properties of component repositories: this situation is similar to what happened with the ubiquitous tool `make`, whose optimisations were only formalised and proven in [15].

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 [1, 2] and the Eclipse platform [3], but these connections, and other recent developments such as [4, 5] do not exploit the *special structure* of the dependencies and conflicts found in a repository.

The underlying structure of software component repositories exposed here can also be seen as a generalisation of some

known mathematical structures : prime event structure [16] correspond to repositories without loops and disjunctions; directed hypergraphs [11] correspond to repositories without conflict arcs, and Dual Horn theories correspond to repositories with conflicts [17].

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 [10], and a long version of this paper containing all the proofs is available online as `http://hal.archives-ouvertes.fr/hal-00605491/`.

The results presented here 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.

Artifact evaluation: the tool `coinst` implemeting the analysis described in this article has been evaluated by the ESEC/FSE Artifact Evaluation Committee, and it has be found to *exceed expectations*; more information on this tool can be found online at `http://coinst.irill.org`.

14. REFERENCES

[1] F. Mancinelli, J. Boender, R. Di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen, "Managing the complexity of large free and open source package-based software distributions." in *Automated Software Engineering (ASE)*. IEEE Computer Society, 2006, pp. 199–208.

[2] C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner, "Opium: Optimal package install/uninstall manager," *International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2007, pp. 178–188.

[3] D. L. Berre and A. Parrain, "On sat technologies for dependency management and beyond," in *Software Product Lines Conference (SPLC)*, S. Thiel and K. Pohl, Eds. Lero Int. Science Centre, University of Limerick, Ireland, 2008, pp. 197–200.

[4] P. Abate, J. Boender, R. Di Cosmo, and S. Zacchiroli, "Strong dependencies between software components," in *Empirical Software Engineering and Measurement (ESEM)*. IEEE Press, Oct. 2009, pp. 89–99. [Online]. Available: http://ieeexplore.ieee.org:80/search/wrapper.jsp?arnumber=5316017

[5] R. Di Cosmo and J. Boender, "Using strong conflicts to detect quality issues in component-based complex systems," in *ISEC '10: Proceedings of the 3rd India software engineering conference*. New York, NY, USA: ACM, 2010, pp. 163–172.

[6] R. Treinen and S. Zacchiroli, "Solving package dependencies: from EDOS to Mancoosi," in *DebConf8 (9th annual conference of the Debian project)*, Argentine, 08 2008, pp. 18–43. http://hal.archives-ouvertes.fr/hal-00340581/en/

[7] R. Di Cosmo, F. Mancinelli, J. Boender, J. Vouillon, B. Durak, X. Leroy, D. Pinheiro, P. Trezentos, M. Morgado, T. Milo, T. Zur, R. Suarez, M. Lijour, and R. Treinen, "Report on formal mangement of software dependencies,", Tech. Rep., Apr. 2006, EDOS project Deliverable 2.2, available as http://www.edos-project.org/xwiki/bin/download/Main/Deliverables/edos-wp2d2.pdf.

[8] R. Di Cosmo, B. Durak, X. Leroy, F. Mancinelli, and J. Vouillon, "Maintaining large software distributions: new challenges from the FOSS era," in *Proceedings of the FRCSS 2006 workshop*, 2006, eASST Newsletter.

[9] H. K. Buning and T. Lettmann, "Propositional logic: Deduction and algorithms," *Studia Logica*, vol. 71, pp. 247–258, 2002.

[10] The Coq Development Team, *The Coq Proof Assistant Reference Manual – Version V8.2*, 2008. http://coq.inria.fr

[11] G. Ausiello, A. D'Atri, and D. Saccà, "Graph algorithms for functional dependency manipulation," *Journal of the ACM*, vol. 30, no. 4, pp. 752–766, 1983.

[12] G. Ausiello, A. D'Atri, and D. Saccá, "Minimal representation of directed hypergraphs," *SIAM Journal of Computing*, vol. 15, no. 2, pp. 418–431, 1986.

[13] J. Ellson, E. Gansner, L. Koutsofios, S. North, and G. Woodhull, "Graphviz - open source graph drawing tools," in *Graph Drawing*, ser. Lecture Notes in Computer Science, P. Mutzel, M. Jünger, and S. Leipert, Eds. Springer Berlin / Heidelberg, 2002, vol. 2265, pp. 594–597. [Online]. Available: http://dx.doi.org/10.1007/3-540-45848-4_57

[14] R. Di Cosmo and S. Zacchiroli, "Feature diagrams as package dependencies," in *Software Product Lines Conference (SPLC)*, ser. Lecture Notes in Computer Science, J. Bosch and J. Lee, Eds., vol. 6287. Springer, 2010, pp. 476–480.

[15] C. A. Gunter, "Abstracting dependencies between software configuration items," *ACM Trans. Softw. Eng. Methodol.*, vol. 9, pp. 94–131, January 2000. [Online]. Available: http://doi.acm.org/10.1145/332740.332743

[16] G. Winskel, "Event structures," in *Petri Nets: Applications and Relationships to Other Models of Concurrency*, ser. Lecture Notes in Computer Science, W. Brauer, W. Reisig, and G. Rozenberg, Eds. Springer Berlin / Heidelberg, 1987, vol. 255, pp. 325–392. [Online]. Available: http://dx.doi.org/10.1007/3-540-17906-2_31

[17] W. F. Dowling and J. H. Gallier, "Linear-time algorithms for testing the satisfiability of propositional horn formulae," *Journal of Logic Programming*, vol. 1, no. 3, pp. 267–284, 1984.

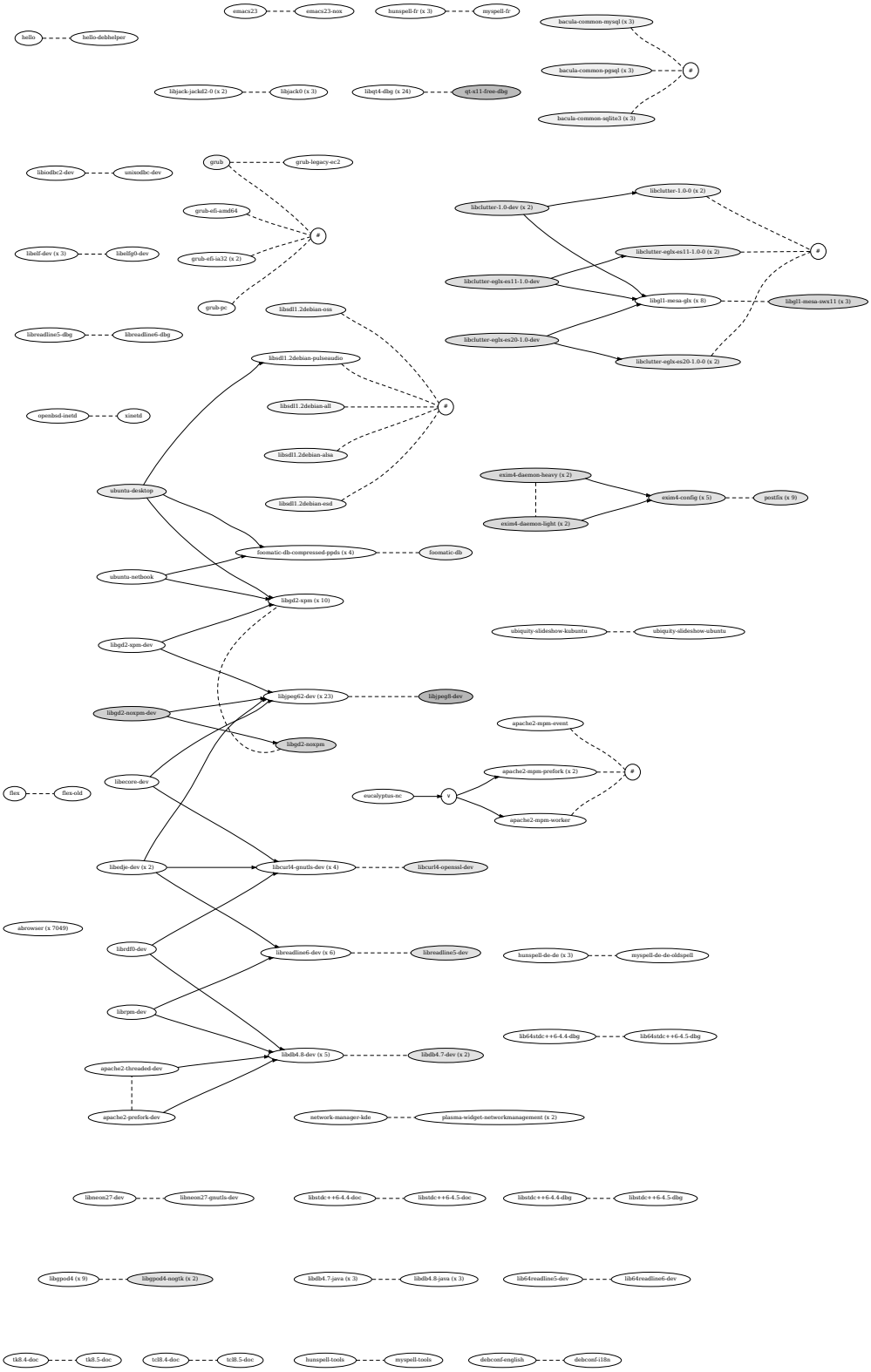


Figure 14: Output graph for Ubuntu 10.10 (main)