Broken Sets in Software Repository Evolution

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

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

***Abstract*—Modern software systems are built by composing components drawn from large *repositories*, whose size and complexity increase at a fast pace. Software systems built with components from a release of a repository should be seamlessly upgradeable using components from the next release. Unfortunately, users are often confronted with sets of components that were installed together, but cannot be upgraded together to the latest version from the new repository. Identifying these *broken sets* can be of great help for a quality assurance team, that could examine and fix these issues well before they reach the end user.**

Building on previous work on component co-installability, we show that it is possible to find these *broken sets* for any two releases of a component repository, computing extremely efficiently a concise representation of these upgrade issues, together with informative graphical explanations.
A tool implementing the algorithm presented in this paper is available as free software, and is able to process the evolution between two major releases of the Debian GNU/Linux distribution in just a few seconds. These results make it possible to integrate seamlessly this analysis in a repository development process.

I. Introduction

Component-based software architectures, maintained in a distributed fashion and evolving at a very quick pace are nowadays commonplace, in particular in the world of free and open source software (FOSS). Components are usually made available via a *repository*, and are equipped with metadata, such as *dependencies* and *conflicts*, that specify concisely the contexts in which a component can or cannot be installed.

A typical example taken from the Debian GNU/Linux distribution, is shown in Figure I, where we can see that the logical language used for expressing dependencies and conflicts is quite powerful, as it allows conjunctions (symbol '`,`'), disjunctions (symbol '`|`') and version constraints.

Maintaining and evolving component repositories is an important task and requires an extensive quality assurance process: besides traditional issues concerning the bugs in the code inside each component, the quality of a large component repository rests also on how well components can be combined with each other, a property known as co-installability [24].

Similarly to what happens with the source code of a single component, which is passed through regression tests to ensure that one did not re-introduce issues that were fixed before [18], we are naturally led to check whether there is any set of components which were co-installable in the old repository, but become non co-installable in the new release.

```
Package: tesseract-ocr
Source: tesseract (2.04-2.1)
Version: 2.04-2.1+b1
Depends: libc6 (>= 2.2.5), libgcc1 (>= 1:4.1.1),
 libjpeg8 (>= 8c), libstdc++6 (>= 4.1.1),
 libtiff4, zlib1g (>= 1:1.1.4),
 tesseract-ocr-eng | tesseract-ocr-language

Package: tesseract-ocr-eng
Source: tesseract-eng
Version: 3.02-2
Conflicts: tesseract-ocr (<< 3.02-2)
```

Fig. 1. Inter-package relationships of `tesseract-ocr`, an optical character recognition engine, and `tesseract-ocr-eng`, the english language pack, as found on 20 February 2012, in the testing suite of the Debian GNU/Linux distribution.

Sets of components with this property, which we call *broken sets*, are particularly damaging in the evolution of a repository: their existence means that there may be perfectly functional deployments based on the old repository that will be disrupted as soon as one tries to upgrade their components to the version in the new repository. The configuration in Example I is a real world example of such a broken set: the `tesseract` optical character recognition program, which is split in several related packages[1], was perfectly functional before February 20^{th} 2012, but on that day, the introduction of the updated english language pack `tesseract-ocr-eng` made the installation of this program temporarily impossible for the English language.
Broken sets need to be identified early in the evolution process, and fixed well before the release of the new repository, but finding broken sets in repositories whose size is in the tens of thousands of components is a daunting task: there may be exponentially many broken sets, and listing them all would be both computationally unfeasible and of no use for a quality assurance team, which would be flooded under the error reports.

In this paper, we describe a highly efficient algorithm that solves the problem by finding a very small subset of the broken sets, which subsume all the others, and is close to *minimal*, in a sense that will be made precise later on.

Work performed at the IRILL centre for Free Software Research and Innovation (http://www.irill.org).

[1] Essentially, `tesseract-ocr` for the core engine, and `tesseract-ocr-`*lang* for all supported language *lang*

We have developed a tool that implements this algorithm and finds such minimal problematic configurations on real component repositories in a few seconds; it also provides very concise *explanations* that allow to identify the origin of the problem. We found several such issues in the evolution of the Debian distribution using the tool.

The paper is organised as follows: Section II briefly recalls the basic notions about packages and dependencies, and the formally certified semantic preserving repository transformations developed in [24] which our work builds upon, and then presents the key ideas of our approach. Section III presents some of the results obtained by our tool on real-world GNU/Linux distributions. The technical development follows: Section IV provides a precise formalisation of all the transformations and simplifications that can be performed to reduce dramatically the search space for explanations; Section V describes in detail an algorithm that uses these results and builds a set of broken sets that cover all upgrade issues, and we show in Section VI how we provide minimal explanations that contain all the relevant information for understanding the cause of these broken sets. We discuss related work in Section VII and Section VIII concludes.

II. Overall Approach

While the concrete details may vary from one technology to the other, the core metadata found in 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. Extra properties like *versioned constraints* (e.g. `libc6 (>= 2.2.5)` in Figure I) can be preprocessed out [14], so we focus on a core dependency system containing a binary symmetric conflict relation, and a dependency function $D(\pi) = \{\{\pi^1_1, ..., \pi^1_{n_1}\}, ..., \{\pi^k_1, ..., \pi^k_{n_k}\}\}$ that is satisfied when for each i at least one of the π^i_j is installed. We follow the notations of earlier works [14], [24], that we briefly recall.

A. Repositories

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*. One can represent a repository with a concise graphical notation, like in Figure 2a, where `c` is in conflict with `b` and `f`, while `a` requires `f` and either `b` or `c`, and `e` requires `f` and `g`, and `d` requires `e`. We call *dependency* a set of packages d included in $D(\pi)$ for some package $\pi \in P$. 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, $C \cap (I \times I) = \emptyset$.

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 R if they are all included in some healthy installation I of the repository.

B. Repository Transformations

To identify co-installability issues in a single repository, [24] introduced a set of repository transformations that allow to extract from a large repository a much smaller *co-installability kernel* which is equivalent as far as co-installability is concerned. The gain allowed by these transformations is impressive: starting from a Ubuntu repository containing 7277 packages, 31069 dependencies and 82 conflicts, one goes down to a kernel containing only 100 elements, with 29 dependencies and 60 conflicts (similar gains are obtained on Debian or Mandriva repositories). A description of these transformations can be found in section 2 of [24], where they are also shown correct (the proofs are certified in Coq [20]): for completeness we recall here the main steps involved.

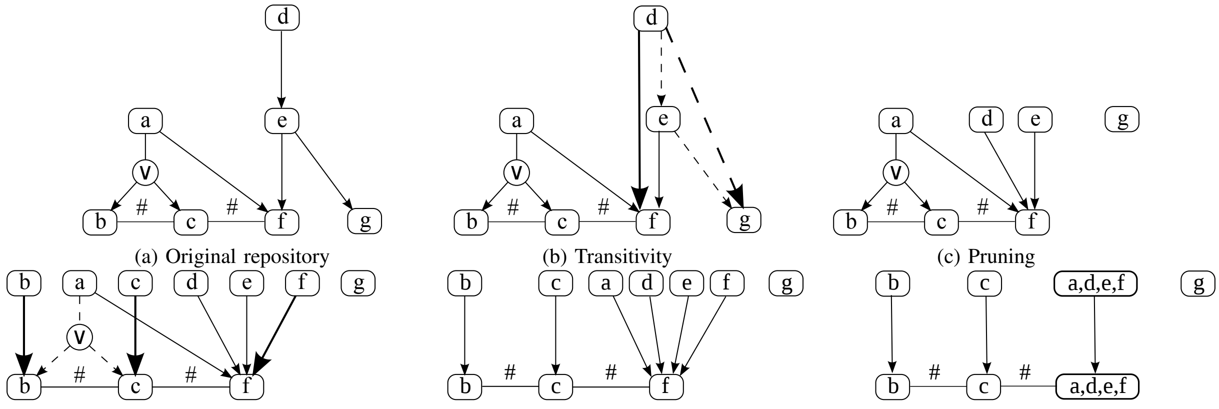
1) Flattening Dependencies: The first transformation replaces the dependency function D of a repository $R = (P, D, C)$ with a special *flattened* form $\widehat{D}$ with two key properties: first, by applying a sort of transitive closure to D, it *directly* describes all dependencies of each package, so if $\widehat{D}(\pi) = \{\{\pi^1_1, ..., \pi^{k_1}_1\}, ...\{\pi^1_n, ..., \pi^{k_n}_n\}\}$, then the packages π^j_i are all the packages relevant for installing package π, and only them. On the example repository of Figure 2a, this amounts to adding a dependency from d to f, and one from d to g (Figure 2b). Second, since we are only interested in co-installability, we can prune the expanded dependency function by removing any dependency containing a package with no conflicts. On the example repository of Figure 2a, 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. Finally, to render the repository more homogeneous $\widehat{D}$ adds a self dependency to each package with conflicts, leading to Figure 2d. 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.

These three operations (transitive closure, pruning, and addition of self conflicts) leave co-installability invariant (Theorem 6 in [24]). They are performed in a single pass, and can be 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:

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

The presentation of flattening through these inductive rules is particularly useful, as it is possible to see, by following the derivation tree, how the original dependencies are composed together into the final flattened form.

2) Removing Irrelevant Dependencies And Redundant Conflicts: As a second phase, [24] identifies several classes of dependencies that are irrelevant as far as co-installability is



(a) Original repository

(b) Transitivity

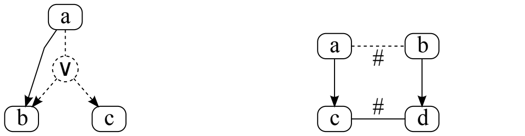
(c) Pruning

(d) Self dependency addition

(e) Simplification

(f) Quotient

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



(a) The dependency $\{b, c\}$ is implied by $\{b\}$

(b) Redundant conflict between a and b

Fig. 3. Implied dependency, redundant conflict

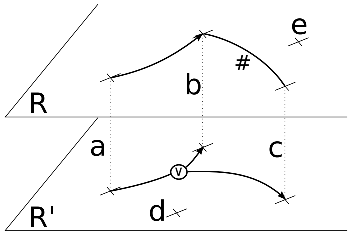


Fig. 4. Repository evolution.

concerned, and can thus be removed. In this work, we only use two such simplifications.

The first simplification is called *canonisation* in [24], as it corresponds to putting dependencies in a canonical minimal form, by removing any dependency d that is *implied* by another dependency d' of the same package (that is, $d' \subseteq d$). An example can be seen in Figure 3a, where the dependency of `a` on `b` or `c` is implied by the dependency of `a` on `b`.

The second simplification is the removal of *irrelevant dependencies*, with have no impact on co-installability. In particular, a dependency d is *clearly irrelevant* if it contains a package π that has conflicts only with packages belonging to d. In Figure 2d, the dotted disjunctive dependency is clearly irrelevant and can be removed, by Theorems 8 and 11 of [24], leading to the repository in Figure 2e.

It is also possible to remove conflicts which are implied by other conflicts, like the one between `a` and `b` in Figure 3b.

3) Quotienting Equivalent Packages: It is evident looking at Figure 2e, that packages a, d, e and f are, as far as co-installability is concerned, really *equivalent*, because they share the very same set of dependencies. This happens quite often in real repositories, so quotienting the final repository can greatly reduce its size, as can be seen in Figure 2f.

These transformations have been proven correct in [24] in a modular way, so we can now safely cherry pick those of them which are relevant for us.

C. Repository Evolution and Upgrades

This work is no longer concerned with a *single* repository R: we want to compare co-installable sets of packages in a *current* repository $R = (P, D, C)$ with those of a *previous state* $R' = (P', D', C')$ of the repository, that *evolved into* R.

One example of such evolution is shown in Figure 4, where one can see that in the process, we may find: *removed* packages, in $P' \setminus P$ (like d); *upgraded* packages, in $P \cap P'$ (like a, b, c); *new* packages, in $P \setminus P'$ (like e).

We assume that packages *keep the same name* across repository evolution: this allows to immediately relate packages between the two repositories with no extra notational burden, and corresponds to a best practice in GNU/Linux distributions, where repository maintainers keep a dummy package with the old package name and a dependence on the new package name when a change of name is needed; similarly, we assume that package upgrades are one-to-one operations.

A healthy installation I' of repository R' can be *successfully upgraded* if one can find an healthy installation I of repository R such that $I' \cap P \subseteq I$: this models closely the common practice of allowing to install extra packages during an upgrade, and accepting to uninstall old packages that have been removed in the new repository.

D. Finding Upgrade Issues: the Key Insights

In a perfect world, it would be possible to successfully upgrade any healthy installation, but in practice, this is not the case, as Example I shows. To improve this situation, we need to find a way of identifying all potential upgrade failures by looking only at the old and new repositories.

Of course, if an installation I' cannot be upgraded, it must contain a set of packages that were co-installable in the old

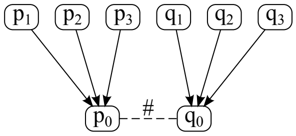


Fig. 5. Many minimal broken sets, one minimal explanation

repository $R' = (P', D', C')$, and are no longer co-installable in the new repository $R = (P, D, C)$: we call such sets *broken sets*. and $I' \cap P$ is a broken set.

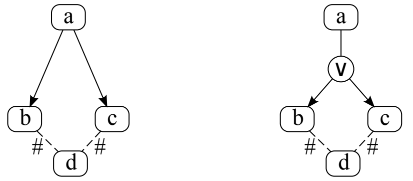
1) Finding broken sets is not easy: Listing all broken sets would require to enumerate all co-installable sets of the old repository R' to see whether their image in the new repository becomes non co-installable. Unfortunately, this might lead to testing all the 2^n subsets of a repository of size n: with n in the range of thousands to tens of thousands[2], this approach is unfeasible.

A dual approach would be to reuse the highly efficient `coinst` tool from [24] to find all minimal non co-installable subsets of the new repository R, and check whether they were installable in the old repository R'. Unfortunately, this would also be unsatisfactory: on one side, the number of minimal non co-installable subsets in R can be huge, and one would need to test them all, even if the changes made from R' to R have no impact on them; on the other side, the efficiency of `coinst` relies on computing equivalence classes of co-installable packages, but these classes may be completely disrupted by an evolution of a repository.

2) Our Solution: Fortunately, it is possible to avoid listing all broken sets, by looking for a special collection of broken sets that has the property that any installation that cannot be upgraded contains at least one element of this collection; we call any such collection a *cover*.

To efficiently build such a cover, we needed to combine several key ideas. We illustrate them here on small examples of repository evolutions, that are represented graphically by indicating the newly added packages, dependencies or conflicts by dashed lines or borders.

Using *explanations* to choose relevant broken sets. We remark that broken sets are closed by superset: if Π cannot be upgraded, then all $\Pi' \supseteq \Pi$ fail as well. So, we might want to only consider broken sets of minimal size. But the fact that a broken set has minimal size does not mean that it must be included in a cover, as we can see in the repository shown in Figure 5. Here we have three packages depending on $\mathtt{p}_0$ and three packages depending on $\mathtt{q}_0$: adding the conflict between $\mathtt{p}_0$ and $\mathtt{q}_0$ in the new repository prevents the upgrade of any installation containing simultaneously one of the $\mathtt{p}_i$ and one of the $\mathtt{q}_i$ packages. In this case, all the sets $\{\mathtt{p}_i, \mathtt{q}_j\}$ built by taking one package on the left and one on the right are



(a) conjunction (b) disjunction

Fig. 6. Multiple explanations

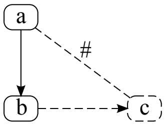


Fig. 7. Broken sets: minimal size vs. minimal explanation

broken sets of minimal size, but one of these broken sets, namely $\{\mathtt{p}_0, \mathtt{q}_0\}$, is much better than the others: indeed, any installation containing one of the $\{\mathtt{p}_i, \mathtt{q}_j\}$ necessarily contains also $\{\mathtt{p}_0, \mathtt{q}_0\}$, which is sufficient, alone, to provide a cover.

To capture the reason why $\{\mathtt{p}_0, \mathtt{q}_0\}$ is better than the other sets, we call *explanation* of a broken set Π in a repository $R = (P, D, C)$ any subrepository (as formally defined in Section IV-A) of R in which Π is still a broken set, and then we give our preference to the broken sets with minimal explanations. In Figure 5, $\{\mathtt{p}_0, \mathtt{q}_0\}$ is the broken set with the minimal explanation: all other pairs have an explanation that contains it.

There can be several minimal explanations associated with a broken set: for instance, in the repository R of Figure 6a, the broken set $\{\mathtt{a}, \mathtt{d}\}$ has one minimal explanation that is R restricted to packages $\{\mathtt{a}, \mathtt{b}, \mathtt{d}\}$, and another minimal explanation that is R restricted to packages $\{\mathtt{a}, \mathtt{c}, \mathtt{d}\}$. In this case, it is easy to see that any installation containing $\{\mathtt{a}, \mathtt{d}\}$ must also contain either $\{\mathtt{b}, \mathtt{d}\}$ or $\{\mathtt{c}, \mathtt{d}\}$. This means that we have the choice, when building a cover, to replace $\{\mathtt{a}, \mathtt{d}\}$ by either one of the two other broken sets, and we prefer $\{\mathtt{b}, \mathtt{d}\}$ or $\{\mathtt{c}, \mathtt{d}\}$, because they have smaller explanations than $\{\mathtt{a}, \mathtt{d}\}$. The repository R of Figure 6b illustrates a dual situation: here the broken set $\{\mathtt{a}, \mathtt{d}\}$ has a unique minimal explanation, the full repository. But we prefer to add to a cover the two broken sets $\{\mathtt{b}, \mathtt{d}\}$ and $\{\mathtt{c}, \mathtt{d}\}$, instead of $\{\mathtt{a}, \mathtt{d}\}$, because they both have smaller explanations than the broken set $\{\mathtt{a}, \mathtt{d}\}$.

Unfortunately, it is not always possible to minimise simultaneously the cardinality of a broken set and the size of its explanation, as can be seen in the repository of Figure 7; here we have the choice between taking as best broken set either $\{\mathtt{a}\}$, which has as explanation all the repository, or the set $\{\mathtt{a}, \mathtt{b}\}$, which is bigger, but has a smaller explanation consisting of the repository without the dependency from `a` to `b`. Our algorithm combines the two minimisation techniques by first enumerating all possible minimal explanations of a given shape, and then taking minimal broken sets for each of them.

Focusing on explanations containing only new conflicts

[2] The current Debian testing distribution is approaching 30.000 packages, and the basic collection of Eclipse plugins contains over 5.000 elements.

or new dependencies. The second insight is that if it is impossible to upgrade an installation, the cause must lie with the *changes* made to the repository, so we should start our search from the *differences* between the old and the new repository. Indeed, a fundamental result proven in this work is that when searching for broken sets to add to a cover, we can restrict our attention to explanations that have as root cause only *new conflicts* and *new dependencies*. This allows to reduce dramatically the search space, by only enumerating explanations involving new conflicts and new dependencies.

Building explanations by combining flattened dependencies. The final ingredient of our approach is the reuse of several transformations from [24] that leave co-installability invariant, which we recalled in subsection II-B. After these transformations, the new repository is put in a simplified, flattened form, that allows to enumerate explanations by just combining together the flattened dependencies, joining them when they share conflicts.

Combining these three key ideas, we could devise an algorithm that is complete and extremely efficient in practice, finding covers that contain very small broken sets, and providing minimal explanations for each broken set.

3) Visualising upgrade issues: To visualise in a compelling way the upgrade issues, we first draw a graph of the broken sets in the cover, as can be seen in Figure 9: this allows a repository maintainer to quickly identify the new incompatibilities, and select the ones he wants to focus on.

Then, for each broken set in the cover, we draw a graph, called *full explanation*, that conveys concisely all the information necessary to understand why it was co-installable in the original repository, and is no longer co-installable in the new one. In this graph, dependencies and conflicts are drawn with different styles: solid lines indicate an object present in both the old and new repositories, dashed lines indicate an object present only in the new repository, and dotted lines indicate an object present only in the old one.

Two examples of full explanations are given in Figure 10, which can be read as follows. In Figure 10a, we see that `tdbsodbc` is no longer co-installable with `libiodbc2` because the new version of package `tdbsodbc` is in conflict with both versions, old and new, of `libiodbc2`. In Figure 10b, we see that `speechd-el` is no longer co-installable with `emacs23-nox` because all versions of `emacs23` conflict with all versions of `emacs23-nox`, and both versions of `speechd-el` depend on any version of `emacs23`, or on the old version of `eieio`.

III. Experimental Results

An OCaml program implementing the algorithm described in this paper is available, together with a few sample outputs, from http://coinst.irill.org/upgrades.

We have run it on several instances of repository evolutions of the Debian GNU/Linux distributions that correspond to the two main use cases of the tool. *Analysing major upgrades* requires comparing one version of a distribution to the next; we tested two such cases by comparing two successive releases of Debian (oldstable and stable), and the last Debian stable release (February 2011) versus the 13 February 2012 snapshot of the testing release (i386 architecture). *Identifying recently introduced issues* requires comparing versions of a distribution during the development phase; we tested several instances of this scenario by comparing various pairs of testing snapshots monthly from October 2011 to February 2012.

A summary of the result is shown in the table below; the running time has been measured on a machine using an Intel Core i7-870 at 2.93GHz.

Distributions	Running time	Broken sets
oldstable – stable	7.4s	202
stable – testing	14.9s	346
2012-01-20 – 2012-01-20	6.2s	64

In figure 9, one can see the full set of upgrade issues between the stable version of Debian and the snapshot of the testing version, which fits on a single A4 page. On this graph, an edge connecting two packages stands for a broken pair of packages. A larger broken set is represented by a circle connected to the packages it contains. One can see that, with only two exceptions, the broken sets that compose the cover contain just two packages.

Looking at some of the transitions, we could find several real upgrade issues in the evolution of the Debian repositories; many of them are due to incomplete migrations of related packages, like in the case of the `tesseract` example shown in the introduction, but there are more damaging ones, like the transition from `gnat-4.4` to `gnat-4.6` where some packages changed to depend from the old version to the new one starting in December 2011, while most packages are still depending on the old one in February 2012.

IV. Formal Development

We now turn to the formal development of our approach, and present the fundamental theoretical results of our work, which allow to find a cover containing a limited number of small broken sets, and to provide concise and readable explanations for each of them.

A. Finding Broken Sets

As highlighted in the previous section, in order to find a cover of broken sets for a repository R, we focus on *explanations*, which are particular *subrepositories* of R' consisting of dependencies and conflicts that may prevent the installation of a set of packages that were previously co-installable.

Formally, a repository $R = (P_1, D_1, C_1)$ is a *subrepository* of another repository (P_2, D_2, C_2), written $R \subseteq R'$, when

$$P_1 \subseteq P_2, \qquad \forall \pi \in P_1, D_1(\pi) \subseteq D_2(\pi), \qquad C_1 \subseteq C_2.$$

Our algorithm enumerates such explanations, striving to satisfy two conflicting objectives: on one side, we want to find enough explanations to get a cover, but on the other side we want to keep the number of explanations small enough to keep the problem tractable and the report readable.

1) Reduced Repositories: We start by putting the two initial repositories in a *reduced* form by applying, in this order, the following co-installability invariant transformations from [24]:

1) perform flattening (see Theorem 6 of [24]);
2) remove redundant conflicts (see Lemma 14 of [24]);
3) remove remaining irrelevant dependencies (see Theorem 8 and 9, and Lemma 13 of [24]);
4) canonise dependencies (see Theorem 2 of [24]).

We write $(P, \widetilde{D}, \widetilde{C})$ for the reduced repository computed from a repository (P, D, C). Compared to [24], we do not remove an irrelevant dependency d of a package π if we can find $d' \in \widetilde{D}(\pi_0)$ and $d_0 \in D(\pi_0)$ such that $\pi \in d_0$ and $d \subseteq d'$, so as to validate the crucial lemma 4 below. All clearly irrelevant dependencies and in practice almost all other irrelevant dependencies can still be removed.

2) Weak Co-Installability: An important feature of reduced repositories is that one does not need to consider dependencies recursively to check co-installability. Instead, we can adopt a two-level view of repositories: given a set of packages at the upper level, they are (weakly) co-installable if one can find a set of packages at the lower level (called *features*) that satisfies their dependencies and that do not conflict with one another.

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 the features 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, $C \cap (F \times F) = \emptyset$.

The following key theorem is a direct consequence of our previous work [24], as its proof is just a composition of the theorems corresponding to each of the transformations performed to obtain a reduced repository:

Theorem 1: A set of packages is co-installable in an initial repository if and only if it is weakly co-installable in the corresponding reduced repository.

3) Minimal Explanations: Given a set of packages Π not weakly co-installable in a reduced repository $(P, \widetilde{D}, \widetilde{C})$, we call *explanation* of Π any subrepository of $(P, \widetilde{D}, \widetilde{C})$ in which Π is not weakly co-installable. Explanations correspond to the so-called *unsatisfiable cores* of SAT problems [7]. There may be several explanations for a set of packages Π. Considering only the minimal ones, with respect to the subrepository relation, is sufficient. They have a very specific shape, illustrated in Figure 8, as shown by the following theorem.

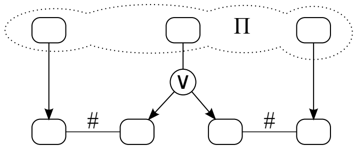


Fig. 8. Typical shape of a minimal explanation

Theorem 2: A minimal explanation (P_Π, D_Π, C_Π) of a broken set Π has the following properties:

- the set of dependencies and conflicts is connected
- only packages in Π have dependencies: if $\pi \notin \Pi$, then $D_\Pi(\pi) = \emptyset$
- all packages in a dependency have a conflict: $\forall \pi \in P, \forall d \in D_\Pi(\pi), \forall \pi' \in d, \exists \pi'' \in P, (\pi', \pi'') \in C_\Pi$;
- packages in conflict belong to a dependency: $\forall (\pi', \pi'') \in C_\Pi, \exists d \in D_\Pi(\pi), \pi' \in d$;
- if the set Π is minimal and $\pi \in \Pi$, then $D_\Pi(\pi) \neq \emptyset$.

B. Focusing on Interesting Explanations

We can take advantage of the similarities between the old and new repositories to only consider some of the minimal explanations above, built using only a subset of the dependencies and conflicts of the new reduced repository.

First of all, the packages in a broken set, by definition, are present in both repositories, so new packages have no dependencies in a minimal explanation. In the following subsections we show that we can restrict our attention to minimal explanations containing only a very restricted form of dependencies; this characterisation is essential to prune the search space when we enumerate minimal explanations. For the sake of clarity, we first explain some of the pruning we could perform if we were working on the initial repositories, before turning to the case of reduced repositories.

1) Interesting Explanations in Non Reduced Repositories: We consider an old repository (P', D', C') and a new repository (P, D, C). We call *new dependency* a dependency $d \in D(\pi)$ of some package $\pi \in P \cap P'$, such that there is no dependency $d' \in D'(\pi)$ such that $d' \subseteq d$. An *explanation* for a non co-installable set of packages is a subrepository in which these packages are still not co-installable. The following lemma shows that if we were to enumerate minimal explanations in the initial repositories, it would be sufficient to consider explanations in which the root dependencies are all new. The proof of the lemma relies on iteratively removing root dependencies that are not new in the explanations. We write $R \setminus \{\pi \mapsto d\}$ for the repository R where the dependency d of package π has been removed.

Theorem 3: There exists a cover $\mathcal{C}$ containing only broken sets Π having an explanation (P_Π, D_Π, C_Π) such that all dependencies in $D_\Pi(\pi)$ for $\pi \in \Pi$ are new dependencies.

Proof: We show that by starting with any cover $\mathcal{C}$ and iteratively applying a suitable transformation, we eventually reach a cover that satisfies the above property.

Consider a cover $\mathcal{C}$ containing a broken set Π, with explanation $R_\Pi = (P_\Pi, D_\Pi, C_\Pi)$. Suppose that there exists a package $\pi \in \Pi$ and a dependency $d \in D_\Pi(\pi)$ which is not new. Then, we claim that the set

$$\mathcal{C}_1 = \mathcal{C} \setminus \{\Pi\} \cup \{\Pi \cup \{\pi_1\}\text{broken} \mid \pi_1 \in d \cap P'\}$$

is also a cover, and that the subrepository $R_\Pi \setminus \{\pi \mapsto d\}$ is an explanation for all broken sets $\Pi \cup \{\pi_1\}$. Let I' be an healthy installation of R' that cannot be successfully upgraded. We need to prove that it includes a broken set in $\mathcal{C}_1$. As $\mathcal{C}$ is a cover, there exists a broken set Π_1 in $\mathcal{C}$ that is contained in I'. If $\Pi_1 \neq \Pi$, then $\Pi_1 \in \mathcal{C}_1$ and we are done. We now consider the case where $\Pi \subseteq I'$. As d is not new, there exists $d' \in D'(\pi)$ such that $d' \subseteq d$. As I' is an healthy installation of R' that contains package π, by abundance, there exists a package $\pi' \in d' \cap I'$. Note that $\pi' \in d \cap P'$. Thus the set $\Pi \cup \{\pi'\}$, included in I' was co-installable, and therefore is included in $\mathcal{C}_1$, as wanted, provided it is not co-installable in R. Suppose this were not the case. Then, it would also be co-installable in the subrepository $R_\Pi \setminus \{\pi \mapsto d\}$. But then, it is easy to see that Π would be co-installable in R_Π, which contradicts our hypotheses. Hence, the set is not co-installable, which complete this part of the proof.

To prove termination, we keep track of an explanation for each of the broken sets in the cover, that is, we consider sets formed of pairs (Π, R_Π) composed of a broken set and the corresponding explanation such that the collection of broken sets Π is a cover. Then, the operation above consists in removing pairs from the set, adding new pairs with strictly smaller explanations. Clearly, this process terminates. At this point, the operation cannot be applied anymore, which means that we have a cover that proves the lemma. ■

2) Taking Advantage of Reduced Repositories: We now turn to reduced repositories, and establish a similar result, but with the important difference that we do not need all the new dependencies: we show that it is enough to consider those dependencies which are both new w.r.t. the reduced repositories, and whose derivation tree has at the root a dependency which is new in the non reduced repository. We call such dependencies *fully new*.

To the new repository (P, D, C), we associate the flattened repository $(P, \widehat{D}, C)$. Each dependency d in $\widehat{D}(\pi)$ can be seen as a composition of dependencies: in the derivation below, d is the composition of dependency d_0 from the initial repository and dependencies d_i from the flattened repository.

$$\dfrac{d_0 = \{\pi_1, \ldots, \pi_n\} \in D(\pi) \quad \dfrac{\cdots}{d_1 \in \widehat{D}(\pi_1)} \quad \ldots \quad \dfrac{\cdots}{d_n \in \widehat{D}(\pi_n)}}{d = \bigcup_{1 \leq i \leq n} d_i \in \widehat{D}(\pi)}$$

A crucial lemma shows that a similar property holds for dependencies in the reduced repository $(P, \widetilde{D}, \widetilde{C})$. We call *lean derivation tree* a derivation tree of $d \in \widehat{D}(\pi)$ such that, for any conclusion $d' \in \widehat{D}(\pi')$ of a subderivation of this tree, we have $d' \in \widetilde{D}(\pi')$ as well. With such a tree, we can decompose a dependency in $\widetilde{D}$ while remaining in $\widetilde{D}$.

Lemma 4: All dependencies in $\widetilde{D}(\pi)$ for all $\pi \in P$ have a lean derivation tree.

Proof: We show that, given a dependency $d \in \widetilde{D}(\pi)$ and any derivation tree of $d \in \widehat{D}(\pi)$, we can find a lean derivation tree. The proof is by induction on the cardinality of the dependency d, then by induction on the size of the initial derivation tree.

The base case, where the tree is just composed of rule Refl, is immediate. Otherwise, the tree starts by an application of rule Trans to some dependency $d_0 \in D(\pi)$ and subtrees with conclusions $d_i \in \widehat{D}(\pi_i)$ for $\pi_i \in d_0$. We are going to show that for each $\pi_i \in d_0$, we can find a lean derivation tree with conclusion $d'_i \in \widehat{D}(\pi_i)$ for some dependency $d'_i \subseteq d_i$. Then, by applying rule Trans on d_0 and these trees, we get a derivation tree for a dependency $\bigcup_i d'_i \in \widehat{D}(\pi_i)$. Due to canonisation, as $\bigcup_i d'_i \subseteq \bigcup_i d_i = d$, we must have $\bigcup_i d'_i = d$, and thus we have a lean derivation for d.

We now prove the existence of suitable dependencies d'_i. A dependency in $\widehat{D}(\pi_i)$ but not in $\widetilde{D}(\pi_i)$ is either irrelevant or implied by another dependency in $\widetilde{D}(\pi_i)$. Since $d_i \subseteq d$ and the dependency d is still in $\widetilde{D}(\pi)$, the dependency d_i has not been removed due to irrelevancy either. Therefore, by definition of canonisation, we can find $d'_i \in \widetilde{D}(\pi_i)$ such that $d'_i \subseteq d_i$. If $d_i = d'_i$, we find a lean derivation tree by applying the induction hypothesis of the inner induction. Otherwise, the cardinality of d'_i is strictly smaller than the cardinality of d. Hence, we can apply the induction hypothesis of the outer induction to get the lean derivation tree. ■

Theorem 5: There exists a cover $\mathcal{C}$ containing only broken sets Π with an explanation (P_Π, D_Π, C_Π) such that all dependencies in $D_\Pi(\pi)$ for $\pi \in \Pi$ are either self-dependencies or fully new dependencies.

Proof: Starting from the cover composed of all possible broken sets, we can associate to each of these sets an explanation, and to each dependency in this explanation, we can associate a lean derivation tree by Lemma 4.

Adapting the proof of Lemma 3, if one of these derivation trees start by a rule Trans involving a dependency which is not new in the non flattened repository, we can build a new cover by removing the corresponding broken set and adding other broken sets with the explanation where the tree has been replaced by all immediate subtrees above the rule Trans. We are replacing subtrees by strictly smaller subtrees. Hence, the process eventually terminates, yielding a cover for which no associated derivation trees start with a dependency which is not new in the non flattened repository.

After this first transformation, we mimic again the argument of the proof of Lemma 3 to remove all flattened dependencies d that are not new; after each removal, it may be necessary to add some self-dependencies, which are important for weak co-installability. ■

3) Pruning Conflicts and Self Dependencies: We can further restrict the search space when enumerating minimal explanations as follows.

If a minimal explanation contains a conflict and self-dependencies for the two packages in conflict, then it is reduced to this conflict and these dependencies. Besides, this conflict must be a *new conflict*, that is, a pair of packages π and π', both in $P \cap P'$ such that $(\pi, \pi') \in C \setminus C'$.

Otherwise, the minimal explanation is composed of fully new dependencies, of conflicts that connect a new dependency to either another fully new dependency or a package in $P \cap P'$, and of the self dependencies for such packages.

So we can first inspect all new conflicts, to capture the first kind of explanations, and then drop all conflicts and self dependencies that do not correspond to the configuration of the second kind before continuing to enumerate explanations.

C. Simplifying covers

A cover containing only broken sets with minimal explanations may be further simplified, using a notion of *implication* that generalises to sets of packages the *strong dependencies* introduced in [1].

We say that a set of packages S_1 *implies* a set of packages S_2, written $S_1 \Rightarrow S_2$, if S_1 is installable and every healthy installation containing S_1 also contains S_2.

If in a cover $\mathcal{C} = \{\Pi_1, ...\Pi_n\}$ there is some broken set Π_i that implies another broken set Π_j in the old repository, then $\mathcal{C} \setminus \{\Pi_i\}$ is still a cover, and we can safely drop Π_i.

Computing implications may be expensive, though, so we only implement an approximation of this simplification step in our algorithm, that turns out to provide a good compromise in practice, and is based on the following notions.

We say that a broken set Π_1 *implies pointwise* another broken set Π_2 if for every package $\pi_2 \in \Pi_2$ there exists a package $\pi_1 \in \Pi_1$ such that $\{\pi_1\}$ implies $\{\pi_2\}$ in the old repository. We approximate the implication between packages by following only conjunctive dependencies: this can be done with the following rules.

$$\textsc{Conj-Refl}\quad \pi \Rightarrow \pi \qquad\qquad \textsc{Conj-Trans}\quad \frac{\{\pi_1, \ldots, \pi_n\} \in D(\pi) \qquad \pi_1 \Rightarrow \pi' \quad \ldots \quad \pi_n \Rightarrow \pi'}{\pi \Rightarrow \pi'}$$

V. Putting it all together: the Algorithm

The algorithm takes as input the old repository (P', D', C') and the new repository (P, D, C). It starts by computing the corresponding reduced repositories $(P', \widetilde{D'}, \widetilde{C'})$ and $(P, \widetilde{D}, \widetilde{C})$, as described in Section IV-A1), and initialises a SAT solver with their encodings, as described in [14]: since our solver allows to check co-installability without rebuilding the encodings, such checks are blazingly fast and can be performed at little cost in the rest of the process.

We then progressively build a cover in two phases, corresponding to the observations of Section IV-B3.

Broken sets corresponding to new conflicts. We first find the pairs of packages $(\pi_1, \pi_2) \in \widetilde{C} \setminus \widetilde{C'}$ which are broken sets. For each of these pairs, we add to the cover a broken set of minimal cardinality contained in it.

Other Broken Sets. We first compute a subrepository (P, D_1, C_1) of $(P, \widetilde{D}, \widetilde{C})$ that contains all relevant minimal explanations, by removing from $(P, \widetilde{D}, \widetilde{C})$ the conflicts and dependencies that may be dropped according to Section IV-B3. For this, we start by computing the fully new dependencies (Section IV-B2), and the subset $C_1 \subseteq \widetilde{C}$ of conflicts that involve at least a package in a new dependency, and whose other package is either in $P \cap P'$ or belongs to a new dependency as well. We then remove fully new dependencies that are irrelevant with respect to C_1 and finally add self-dependencies on packages in $P \cap P'$ with a conflict. This gives us the repository (P, D_1, C_1).

To find the explanations we are interested in, we enumerate all dependency functions D_2 pointwise included in D_1 such that all dependencies in D_2 are connected through conflicts in C_1, using a backtracking algorithm.

In order not to consider all permutations of dependencies, we keep track of two sets of dependencies when building explanations: the positive set of dependencies that are part of the explanation; a negative set of dependencies that are not going to be part of any further explanation. When backtracking, we add the dependency to the negative set, as all explanations it may be involved into have already been considered. The process starts with the two sets being empty. We add a first dependency to the positive set, then repeatedly add dependencies connected to other dependencies in the positive sets. All possible combinations are considered through backtracking.

At each step, the set of all packages $\{\pi \,|\, D_2(\pi) \neq \emptyset\}$ included in the positive set is a *candidate broken set*. Each time a minimum broken set is built, it is added to the cover.

Final simplification. Finally, we identify all broken sets Π that can be shown to imply pointwise another broken set following only conjunctive dependencies, and remove them to obtain the final cover (Section IV-C).

VI. Explaining Broken Sets

As said in Section II-D3, we associate to each broken set Π a graph called full explanation, which is computed as follows. We run a SAT solver on the unsatisfiable problem obtained by encoding the new repository and asserting the packages in Π, and then extract an unsatisfiable core [7].

Since in our encoding each Boolean clause of the SAT problem corresponds exactly to one dependency or conflict, the unsatisfiable core can be immediately turned into an explanation. Then, we annotate appropriately each dependency and conflict in this explanation: a dependency can be present in both the old and the new repository, or just in one of them; the possible targets of a dependency might satisfy it in the old repository, the new one, or both; similarly, a conflict might be between all possible versions of a pair of packages, between one version of a package and any version of the other, or between two precise versions of the packages. As explained in Section II-D3, different line styles distinguish these cases.

VII. 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; *dynamic* aspects of the components are considered to ensure certain properties of their composition [13], [21], to automatically detect behavioural incompatibilities from the component source code [15], or to deploy and upgrade such systems [4], [9]. Research on *static* inter-module dependencies is also largely performed at the level of the component source code, with dependencies automatically extracted from huge sets of source code, and then used to predict failures [8], [16], [17], [26], or to guide automated testing [25]; some work, like [19], manually analyse the architectural dependencies to improve the modularisation of the software architecture, on problems

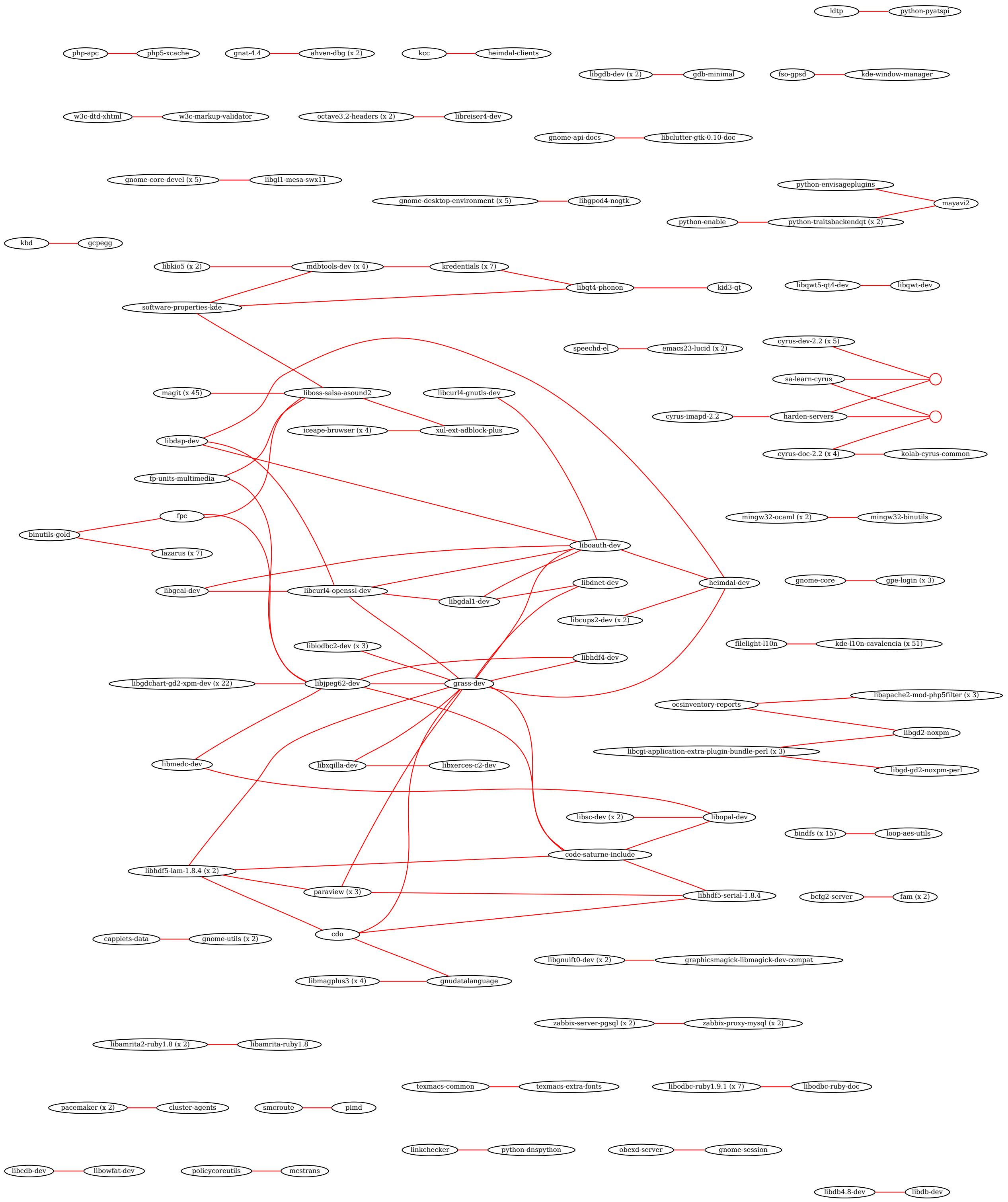
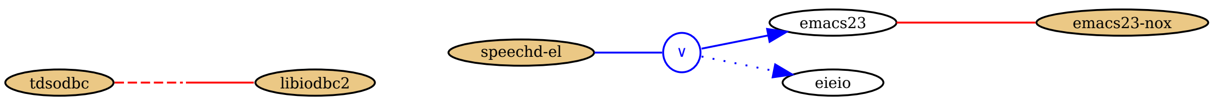


Fig. 9. Problematic upgrades in Debian, i386, Stable vs. Testing (13 February 2012)



(a)

(b)

Fig. 10. Actual broken sets and their full explanations

small enough (some 20 components) to avoid the need for automation.

Our work is part of a recent research area that focuses on the properties of large 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. In most frameworks, determining whether a single component can be installed at all is an NP-Complete problem [2], albeit the concrete instances arising in real-world systems, like GNU/Linux distributions, Eclipse plugins or OSGI component repositories, turn out to be tractable [14], [23], [6], [5], [22], [11], [12].

For the maintenance of component repositories, though, more sophisticated analyses are required. This includes identifying for each component the other components that it absolutely needs [1], those that it can never be installed with [10], and what component upgrades are more likely to impact a repository [3].

An important advance in this area was made in [24], that shows how to efficiently extract from any component repository a much smaller *co-installability kernel* that allows to identify directly the sets of components that cannot be installed together. This makes it possible for a quality assurance team to quickly find component incompatibilities *in a given repository*.

The present work is a significant step forward in this direction, as it provides for the first time a means to identify how coinstallability evolves from one version of the repository to another, and provides simple and effective explanations for quality assurance teams that want to understand why some components that were coinstallable become all of a sudden incompatible. While the examples in this paper all come from the world of GNU/Linux distributions, the algorithm presented can be applied directly to other frameworks, like Eclipse plugins or OSGI component repositories.

VIII. Conclusion

We have shown that it is possible to determine how *co-installable* sets of packages change across repository evolutions and developed an algorithm that computes efficiently a concise representation of the upgrade issues introduced by such changes. We have also shown how to present concisely all the information relevant for understanding the origin and the importance of an upgrade issue with an informative graphical explanation. A tool implementing the algorithm presented in this paper is available as free software, and is able to process the evolution between two major releases of the Debian GNU/Linux distribution, and compute all the explanations, in just a few seconds. These results make it possible to integrate seamlessly this analysis in a repository development process.

Acknowledgments: the authors would like to thank Mehdi Dogguy, Ralf Treinen and Stefano Zacchiroli for interesting discussions on the Debian repository evolution process.

Code and data availability: the tool, together with a few sample outputs, can be found at http://coinst.irill.org/upgrades

References

[1] P. Abate, J. Boender, R. Di Cosmo, and S. Zacchiroli. Strong dependencies between software components. In *ESEM*, pages 89–99. IEEE Press, Oct. 2009.

[2] P. Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli. Dependency solving: a separate concern in component evolution management. *Journal of System and Software Science*, 85(10):2228 – 2240, 2012.

[3] P. Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli. Learning from the Future of Component Repositories. In *CBSE-2012*, Bertinoro, Italie, June 2012. ACM.

[4] S. Ajmani, B. Liskov, and L. Shrira. Modular software upgrades for distributed systems. In *ECOOP*, pages 452–476, 2006.

[5] J. Argelich, D. Le Berre, I. Lynce, J. Marques-Silva, and P. Rapicault. Solving Linux upgradeability problems using boolean optimization. In *LoCoCo*, volume 29 of *EPTCS*, pages 11–22, 2010.

[6] D. L. Berre and A. Parrain. On sat technologies for dependency management and beyond. In *SPLC (2)*, pages 197–200, 2008.

[7] R. Bruni. Approximating minimal unsatisfiable subformulae by means of adaptive core search. *Discrete Appl. Math.*, 130(2):85–100, Aug. 2003.

[8] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE*, pages 342–351, 2005.

[9] O. Crameri, N. Knezevic, D. Kostic, R. Bianchini, and W. Zwaenepoel. Staged deployment in mirage, an integrated software upgrade testing and distribution system. *SIGOPS Oper. Syst. Rev.*, 41(6):221–236, Oct. 2007.

[10] R. Di Cosmo and J. Boender. Using strong conflicts to detect quality issues in component-based complex systems. In *ISEC*, pages 163–172, New York, NY, USA, 2010. ACM.

[11] R. Di Cosmo, O. Lhomme, and C. Michel. Aligning component upgrades. In C. Drescher, I. Lynce, and R. Treinen, editors, *LoCoCo*, volume 65 of *EPTCS*, pages 1–11, 2011.

[12] M. Gebser, R. Kaminski, and T. Schaub. aspcud: A linux package configuration tool based on answer set programming. In *LoCoCo*, volume 65 of *EPTCS*, pages 12–25, 2011.

[13] P. Inverardi, A. L. Wolf, and D. Yankelevich. Static checking of system behaviors using derived component assumptions. *ACM TOSEM*, 9(3):239–272, 2000.

[14] 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 *ASE*, pages 199–208, 2006.

[15] S. McCamant and M. D. Ernst. Early identification of incompatibilities in multi-component upgrades. In *ECOOP*, pages 440–464, 2004.

[16] N. Nagappan and T. Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *ESEM*, pages 364–373, 2007.

[17] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In *ACM CCS*, pages 529–540, 2007.

[18] A. Orso, M. J. Harrold, D. Rosenblum, G. Rothermel, H. Do, and M. L. Soffa. Using component metacontent to support the regression testing of component-based software. In *ICSM* pages 716–, Washington, DC, USA, 2001. IEEE Computer Society.

[19] H. Pei-Breivold, I. Crnkovic, R. Land, and S. Larsson. Using dependency model to support software architecture evolution. In *Evol*, September 2008.

[20] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.2*, 2008.

[21] M. Tivoli and P. Inverardi. Failure-free coordinators synthesis for component-based architectures. *Sci. Comput. Program.*, 71(3):181–212, 2008.

[22] P. Trezentos, I. Lynce, and A. L. Oliveira. Apt-pbo: solving the software dependency problem using pseudo-boolean optimization. In *ASE*, pages 427–436, 2010.

[23] C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. Opium: Optimal package install/uninstall manager. In *ICSE*, pages 178–188, 2007.

[24] J. Vouillon and R. Di Cosmo. On software component co-installability. In *SIGSOFT FSE*, pages 256–266, 2011.

[25] I.-C. Yoon, A. Sussman, A. Memon, and A. Porter. Direct-dependency-based software compatibility testing. In *Proceedings of ASE '07*, pages 409–412, New York, NY, USA, 2007. ACM.

[26] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *ICSE'08*, pages 531–540. ACM, 2008.