Using Strong Conflicts to Detect Quality Issues in Component-based Complex Systems [*]

Roberto Di Cosmo
PPS, Université Paris Diderot, Paris, France and
Inria Paris-Rocquencourt, France
roberto@dicosmo.org

Jaap Boender
PPS, Université Paris Diderot, Paris, France
Jaap.Boender@pps.jussieu.fr

ABSTRACT

The mainstream adoption of free and open source software (FOSS) has widely popularised notions like *software packages* or *plugins*, maintained in a distributed fashion and evolving at a very quick pace. Each of these components is equipped with metadata, such as *dependencies*, which define the other components it needs to function properly, and the incompatible components it cannot work with. In this paper, we introduce the notion of *strong conflicts*, defined from the component dependencies, that can be effectively computed. It gives important insights on the quality issues faced when adding or upgrading components in a given component repository, which is one of the facets of the predictable assembly problem.Our work contains concrete examples drawn from the world of GNU/Linux distributions, that validate the proposed approach. It also shows that the measures defined can be easily applied to the Eclipse world, or to any other coarse-grained software component model.

Categories and Subject Descriptors
D.2.8 Software Engineering: Metrics
D.2.9 Software Engineering: Management – Software Quality Assurance

General Terms
Measurement, Reliability

Keywords
Components, large component repositories, open source software, quality assurance, strong conflicts.

1. INTRODUCTION

The management of large software systems has always been a stimulating challenge in Software Engineering. Many seminal advances by founding fathers of Comp. Sci. were prompted by this challenge (see the book "Software Pioneers", edited by M. Broy and E. Denert [4], for an overview), but in recent years, the explosion of Internet connectivity and the mainstream adoption of free and open source software (FOSS) have deeply changed the scenarii faced by today's software engineers.

FOSS in particular has popularised notions like *software packages* or *plugins*, which evolve very quickly and are maintained in a distributed fashion by many different actors, often only loosely connected. The best known among these actors are *distribution editors.*

This new scenario has given a radically new meaning to quality assurance by raising the issue of how to track the overall quality of a huge collection of diverse software components developed in a loosely coupled framework, like FOSS *distributions.*

These *packages* or *plugins* are actually coarse-grained software components that can be assembled together to build complex systems, like a working GNU/Linux distribution or a sophisticated configuration of an Eclipse-based development platform.

Each of these components is equipped with metadata, such as *dependencies* that contain information on the other components it needs to function properly, and *conflicts* that contain information on the components it cannot work with.

Ensuring the quality of a particular configuration of a system based on these FOSS components is a technical and engineering challenge, owing to the size and complexity of the component base (tens of thousands of software packages in a GNU/Linux distribution), and the speed of its evolution.

The quality assurance teams working for FOSS distributions (in particular those that contain large numbers of packages, such as Debian or Mandriva), face this challenge daily, and they have to manually track and fix compatibility problems arising from the dependencies among the different packages. The EDOS research project [1] addressed some of these issues, more specifically by developing the theory and tools necessary to easily identify those components that are never installable in a repository (often called *broken packages*), and which we will briefly recall in Sec. 2.

In this paper, we make a significant step further, by proposing the notion of *strong conflicts*, formally defined from the dependencies among components. Strong conflicts can be efficiently computed and give important insights on the quality issues faced when adding or upgrading components in a given installation.

A component p is in *strong conflict* with another component q if it is never possible to install p and q together, no matter what other combination of components, drawn

[*]Partially supported by the European Community's 7th Framework Programme (FP7/2007-2013), grant agreement n°214898, "Mancoosi" project, and the CIRILL center.

[1]See `www.edos-project.org`.

from a given universe often called a *repository*, is made. As we will explain in more detail later on in the article, using strong conflicts gives much more precise results than simply looking at the dependencies and conflicts declared in the metadata. It is possible for a package with only a few explicit dependencies and conflicts to have hundreds of strong conflicts.

When a component p is in strong conflict with a large number of other components, installing it can significantly preclude further evolution of a software platform, so strong conflicts are major obstacles to the modularisation of assemblies built out of packages. In an ideal world, one would like to have very few of them in a repository, but in practice, as we will demonstrate, they seriously plague current GNU/Linux distributions. This shows the need to identify all strong conflicts in a repository, and explain the origin of these strong conflicts to the maintainers of the repository.

This is the goal of the current work, which provides a theoretical study of the complexity of the problem, an optimised algorithm that is efficient in practice, and a concise way of presenting the results to the quality assurance team.

The problem we deal with can be seen as a specific instance of the predictable assembly problem [5, 14], which is one of the numerous challenges arising when integrating software entities developed by third-party to form a coherent system [3, 5]. In [5] the predictable assembly problem is described as follows: "Given a set of components C, predict property P of an assembly A of these components." Although a package, as a static entity, only represents the notion of software component (w.r.t. the definition given in [14]) in a limited way, the predictable assembly problem can be recast in our context by considering the user installation as A, the set of packages that form the distribution as C, and the absence of conflicts as P.

While the work presented here contains concrete examples drawn from the world of GNU/Linux distributions, the measures defined can be easily applied to the Eclipse world, or to any other coarse-grained software component model.

The paper is organised as follows. Sec. 2 recalls the formal description developed during the EDOS research project [7] of the main characteristics of a software package found in the mainstream FOSS distributions, as well as a few decidability and complexity results from that project.

In Sec. 3 we identify and formally define a significant measure, computed from the explicit dependency metadata, useful for quality assurance in the maintenance of a FOSS distribution. Sec. 4 discusses the feasibility of computing this measure and introduces an optimised algorithm which can be efficient in practice. In Sec. 5–7 we introduce a way of presenting the information in a particularly appropriate form for quality assurance teams, and provide significant experimental evidence of the relevance of strong conflicts, by analysing both the Debian and the Mandriva GNU/Linux distributions. Finally, in Sec. 8 and 9, we present a discussion, related works and our conclusions.

2. BASIC DEFINITIONS

Every package management system [6, 2] takes into account a set of interrelationships among packages. The most common relationships, present in all systems, are dependencies and conflicts: a package p *depends* on q if in order to install package p, it is necessary that package q is installed as well. A package p *conflicts* with package q if it cannot coexist with package q. In the rest of this section, we recall some basic definitions from the formalisation of package interrelationships developed by the EDOS project. Packages are archive files containing metadata and installation scripts, identified by a unit (package name) and a version (an arbitrary string equipped with distribution-specific orderings, but for our purposes, it can just be modelled as a positive integer).

DEFINITION 1 (PACKAGE, UNIT). *A* package *is a pair (u, v) where u is a unit and v is a version. Units are arbitrary strings, and versions are non-negative integers.*

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

For instance, if our Debian repository contains the versions `2.15-6`, `2.16.1cvs20051117-1` and `2.16.1cvs-20051206-1` of the unit `binutils`, we may encode these versions respectively as 0,1 and 2, giving the packages $(\texttt{binutils}, 0)$, $(\texttt{binutils}, 1)$, and $(\texttt{binutils}, 2)$.

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

- *The relation C is symmetric, i.e., $(p_1, p_2) \in C$ if and only if $(p_2, p_1) \in C$ for all $p_1, p_2 \in P$.*
- *Two packages with the same unit but different versions conflict*[3]*, that is, if $p_1 = (u, v_1)$ and $p_2 = (u, v_2)$ with $v_1 \neq v_2$, then $(p_1, p_2) \in C$.*

In a repository $R = (P, D, C)$, the dependencies of each package p are given by $D(p) = \{d_1, \dots, d_k\}$ which is a set of sets of packages, interpreted as follows. If p is to be installed, then all its k dependencies must be satisfied. For d_i to be satisfied, at least one of the packages of d_i must be available. In particular, if one of the d_i is the empty set, it will never be satisfied, and the package p is not installable. If a package p has no dependencies, then $D(p) = \emptyset$.

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

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

where $a = (u_a, 0)$, $b = (u_b, 0)$, $c = (u_c, 0)$ and so on. The repository R is represented in Fig. 1. For the package a to be installed, the following packages must be installed: b, either c or d, either d or e, and either d or f. Packages c and e, e and i, and g and h cannot be installed at the same time.

[2]We write $\mathscr{P}(X)$ for the set of subsets of X.
[3]This requirement is present in some package management systems, notably Debian's, but not all. For instance, RPM-based distributions allow simultaneous installation of several versions of the same unit, at least in principle.

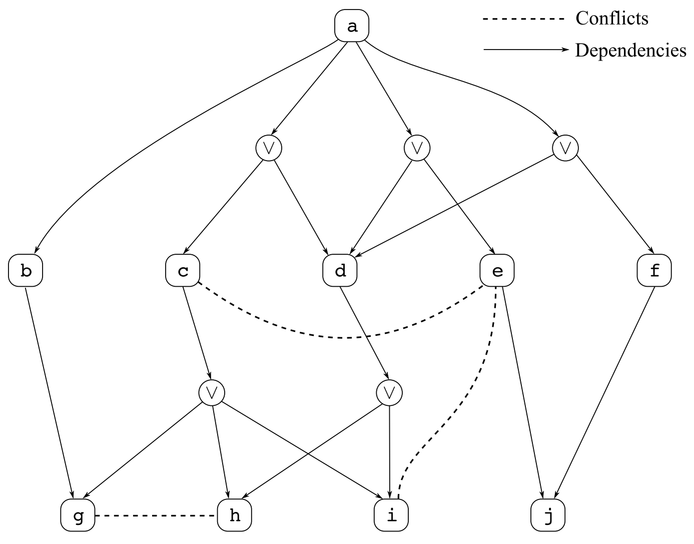


Figure 1: The repository of Ex. 1.

So, the dependencies we are dealing with have the general form of a conjunction of disjunctions:

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

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

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

Concerning the relation C, two packages $p_1 = (u_1, v_1), p_2 = (u_2, v_2) \in P$ conflict when $(p_1, p_2) \in C$. Since conflicts are a function of presence and not of installation order, the relation C is symmetric.

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

- ***Abundance:*** *Every package has what it needs*[4]. *Formally, for every $p \in I$, and for every dependency $d \in D(p)$ we have $I \cap d \neq \varnothing$.*
- ***Peace:*** *No two packages conflict. Formally, $(I \times I) \cap C = \varnothing$.*

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

As we have shown in detail in [7, 10], installability is equivalent to SAT, hence it is decidable and NP-complete. It is straightforward to use the same encoding to get the equivalent result for co-installability.

[4]This notion is similar to the notion of *version consistency* from [9].

Theorem 1. *Decidability of installability Given a repository R, deciding whether a given package p of R is installable is an NP-complete problem.*

In practice, though, all the real-world instances encountered up to now turn out to be tractable[5].

Definition 5 (Dependency closure (cone)). *The* dependency closure $\Delta(\Pi)$*, also called* cone*, of a set of packages Π of a repository R is the smallest set of packages included in R that contains Π and is closed under the* immediate dependency *function $\overline{D} : \mathscr{P}(P) \rightarrow \mathscr{P}(P)$ defined as*

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

In simpler words, $\Delta(\Pi)$ contains Π, as well as all the packages that are reachable from Π following the dependency relation. Since repositories are finite, this set is computable and can actually be calculated in linear time.

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

Definition 6 (Generated subrepository). *Let $R = (P, D, C)$ be a repository and $\Pi \subseteq P$ be a set of packages. The* subrepository generated by Π *is the repository $R|_\Pi = (P', D', C')$ whose set of packages is the dependency closure of Π and whose dependency and conflict relations are those of R restricted to that set of packages. More formally we have $P' = \Delta(\Pi)$, $D' : P' \rightarrow \mathscr{P}(\mathscr{P}(P')), p \mapsto \{d \cap P' \mid d \in D(p)\}$ and $C' = C \cap (P' \times P')$.*

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

Proposition 1 (Subrepository completeness). *A package p is installable w.r.t. R if and only if it is installable w.r.t. $R|_p$. (Similar for co-installability.)*

Proof. Let I be a healthy (i.e., abundant and peaceful) installation of p in R.

Since I is abundant and $p \in I$, for every $d \in D(p)$, $d \cap I \neq \varnothing$. However, if $d \in D(p)$, any element of d is also an element of $\Delta(\{p\})$, and thus, an element of $d \cap I$ will also be in $I \cap \Delta(\{p\})$. Therefore, $I \cap \Delta(\{p\})$ is abundant.

I is peaceful, therefore $(I \times I) \cap C = \varnothing$. Since $I \cap \Delta(\{p\}) \subseteq I$, $(I \cap \Delta(\{p\}) \times I \cap \Delta(\{p\}) \cap C = \varnothing$. Therefore, $I \cap \Delta(\{p\})$ is peaceful. □

3. STRONG CONFLICTS VS. EXPLICIT CONFLICTS

GNU/Linux distributions have been around for more than 15 years now, and they all provide tools that help the quality assurance teams to spot problems in the component collection they maintain, and some of this information is freely available. The most typical interface are bug tracking systems like the one found

[5]The EDOS tools are used daily on tens of thousands of instances, see for example `http://edos.debian.net`.

at `http://wiki.debian.org/qa.debian.org/pts`, and component browsers like those found at `http://www.debian.org/distrib/packages` or `http://doc4.mandriva.org`.

These tools allow one to follow bugs filed against given packages, or to browse the *explicit* dependency and conflict relationships, but they fall very short of what is needed to be able to identify the reasons for package incompatibilities, since these may be caused by a complex interplay of dependencies and conflicts (see Sec. 7 for concrete examples).

Recent research work in the EDOS project has made a first step towards a more effective method to identify problems in packages, via the development of tools able to efficiently find packages that are not installable at all (so-called *broken packages*). These tools are now used daily: for example, Jerôme Vouillon's `edos-debcheck` is now incorporated into `edos.debian.net`.

We propose a next step, allowing to identify components that are not broken, but that prevent, if selected, the installation of a large set of other components. In an ideal world, there would be no such components in a repository, but in practice, we have found thousands of them. Hence, it is important to find them efficiently, classify them and explain the reasons of their existence, so that the faulty components can be fixed.

With the current tools, it is almost impossible for the quality assurance teams to identify such packages. Therefore, a stronger, *semantic*, notion of conflicts is needed, which we introduce here as *strong conflicts*; informally, two packages p and q strongly conflict w.r.t. a repository R if it is not possible to install them together in R.

Here is the formal definition of strong conflicts:

Definition 7 (Strong Conflicts). *Given a repository R, we say that a package p in R* strongly conflicts *with a package q in R if there exists a healthy installation of R containing p, a healthy installation of R containing q and no healthy installation of R containing p and q.*

Note that the formal definition requires p and q to be installable separately in R, as otherwise they would trivially conflict with every other package in the distribution. It is easy to see that strong conflicts are decidable: since R is finite, the set of all installations in R is finite too, and one can check the property on each of these installations (the question of doing so efficiently is addressed later in the paper).

Once we dispose of all the strong conflicts in a given repository R, it is possible to know, for every package p, the set of the packages that strongly conflict with p. We call this set the *exclusion set* of p in R.

Definition 8 (Exclusion set of a component). *Given a repository R and a package p in R, the* exclusion set *of p in R is the set $Excl(p, R) = \{q \in R | q \text{ strongly conflicts with } p\}$.*

Problematic packages are now easily identifiable as those having large exclusion sets: in table 1, we give a significant example by presenting the 20 packages with the highest number of strong conflicts for the Debian Lenny distribution (containing over 23.000 packages and more than 400.000 explicit dependency and conflict relationships), that we have fully analysed using tools built on the results of this article.

To show the importance of the new notion, in this table, for each package, we show the size of its exclusion set (number of strong conflicts), alongside with the other, simpler measures that may appear natural to use when classifying it: the number of explicit conflicts it is involved in, the size of the dependency closure, and the height of the dependency closure.

The figures speak by themselves: none of the simplistic measures allows to identify the problematic packages identified using strong conflicts: the most astounding example is the case of `ppmtofb`, which is negligible under all of these simplistic measures, while it prevents, when installed, the installation of *over two thousand* other packages.

Once we have found the problematic packages, the other important issue is finding a meaningful way to *explain* where their large exclusion sets come from. It turns out that such explanations will be easier to find once we have designed an optimised algorithm for computing all strong conflicts in a repository, which is the subject of the next section.

4. EFFICIENT COMPUTATION OF STRONG CONFLICTS

The naive method to check whether a particular pair of packages (p, q) strongly conflicts in a repository R would consist in taking all pairs of packages in R and check them for co-installability. However, this is not fast enough: if R has n elements, the naive method requires us to examine all 2^n subsets of R, and check that p and q never occur together in those that are abundant and peaceful.

An improvement on this naive method is to encode the co-installability of p and q in a SAT instance, which can be done efficiently (see [10]).

In practice, though, one really wants to find *all* pairs (p, q) that strongly conflict in a repository R; the naive way of doing this is the following:

```
procedure strongconflicts(R)
strongconflicts ← $\emptyset$
forall $p, q \in R$
  if $p$ and $q$ are not co-installable in $R$
  then strongconflicts ← $\{p, q\} \cup$ strongconflicts
return strongconflicts
```

This implies checking n^2 SAT instances. With current repositories, which contain over 20.000 packages, this is totally unfeasible, so we need to look for an optimised approach. Since our analysis of the structure of the repositories shows that conflicts do not often occur, in the rest of this section, we will focus on designing an algorithm that works well in this particular case.

We start with some key properties of repositories with respect to abundance and peace.

Observation 1 (Abundance is closed under union). *Given a repository R and two installations I and I' of R, if I and I' are abundant, then so is $I \cup I'$.*

Lemma 1 (Peace vs. union). *Given a repository R and two installations I and I' of R, if I and I' are peaceful, and $I \cup I'$ is not, then there exists a conflict (c_1, c_2) with $c_1 \in I$ and $c_2 \in I'$.*

Strong Conflicts	Package	Explicit Conflicts	Explicit Dependencies	Closure Size	Closure Height
2368	ppmtofb	2	3	6	4
127	libgd2-noxpm	4	6	8	4
127	libgd2-noxpm-dev	2	5	15	5
107	heimdal-dev	2	8	121	10
71	dtc-postfix-courier	2	22	348	8
71	dtc-toaster	0	11	429	9
70	citadel-mta	1	6	123	9
69	citadel-suite	0	5	133	9
66	xmail	4	6	105	8
63	apache2-mpm-event	2	5	122	10
63	apache2-mpm-worker	2	5	122	10
62	harden	0	4	214	9
62	harden-servers	36	2	103	8
57	gpe	0	31	263	10
56	heimdal-servers	10	16	139	9
55	heimdal-servers-x	2	15	142	9
53	libapache2-mod-php5filter	2	16	129	9
52	dtc-cyrus	2	17	345	8
50	kdepimlibs5-dev	1	6	225	9
46	kdebase-runtime-data-common	2	0	1	1

Table 1: The 20 packages with the highest number of strong conflicts in Debian `stable` (`main`) 5.0, February 2009

PROOF. $I \cup I'$ is not peaceful, so there is a conflict (c_1, c_2) with c_1 and c_2 elements of $I \cup I'$. However, it cannot be the case that both c_1 and c_2 are elements of I, since I is peaceful. The same goes for I', so it must be that $c_1 \in I$ and $c_2 \in I'$ (or $c_1 \in I'$ and $c_2 \in I$, but since the conflict relation is symmetric, this is equivalent). $\square$

LEMMA 2 (REACHABILITY OF PACKAGES IN A CLOSURE). *Given a repository R, if p' belongs to $\Delta(\{p\})$, then there is a path from p to p' in the dependency graph.*

PROOF. $\Delta(\{p\})$ is closed under the repeated application of the immediate dependency function, so if $p' \in \Delta(\{p\})$, there must be a list $p, p_1, \ldots, p_n, p'$ so that $p_1 = \overline{D}(p)$, $p_2 = \overline{D}(p_1)$, $\ldots$, and $p' = \overline{D}(p_n)$. Since $\overline{D}$ is the union of all immediate dependencies of a package, the list $p, p_1, \ldots, p_n, p'$ is also a path in the dependency graph. $\square$

Now we can prove the main result of this section

THEOREM 2 (ORIGIN OF STRONG CONFLICTS). *If p is installable in R and q is installable in R, but p and q are not co-installable in R, then there exists an explicit conflict (c_1, c_2), a dependency path from p to c_1 and a dependency path from q to c_2.*

PROOF. Since p and q are separately installable in R, then by Proposition 1 there exist $I_p \subseteq cone(p, R)$ and $I_q \subseteq cone(q, R)$ which are both abundant and peaceful.

Since p and q are not co-installable, we have that $I_p \cup I_q$ cannot be abundant and peaceful (otherwise, it would be an installation containing both p and q). Since $I_p \cup I_q$ is abundant, by Observation 1, we have that $I_p \cup I_q$ is not peaceful, and then Lemma 1 gives us a conflict (c_1, c_2) in R with c_1 in I_p and c_2 in I_q, so we conclude by Lemma 2. $\square$

This theorem demonstrates that the strong conflicts are a strict subset of all the pairs of packages (p, q) found following the predecessors of explicit conflicts.

This allows to rewrite the procedure `strongconflicts` as follows:

```
procedure strongconflicts(R)

candidates ← ∅
strongconflicts ← ∅
explicit ← {$(c,c')$| $c$ and $c'$ are an explicit conflict in $R$}

(* compute all pairs $(p,q)$ of predecessors of ex-
plicit conflicts in $R$ *)
forall $(c,c') \in$ explicit
 forall $p$ predecessor of $c$ in $R$
  forall $q \neq p$ predecessor of $c'$ in $R$
   candidates ← {$p,q$} ∪ candidates

(* check strong conflicts only among the selected candi-
dates *)
forall $p,q \in$ candidates
  if $p$ and $p$ are not co-installable in $R$
  then strongconflicts ← {$p,q$} ∪ strongconflicts
```

The notion of predecessor used in the algorithm is reflexive: a package p is a predecessor of q if there is a, possibly empty, dependency path from p to q.

If the number of explicit conflicts is small, and the set of their predecessors too, the search space can be hugely reduced: this is the case in our actual experiments with real-world data.

We also note the following:

PROPOSITION 2. *Given a repository R, if c, c' is an explicit conflict, there is a path from p to c following only conjunctive dependencies, and there is a path from q to c' following only conjunctive dependencies, then (p, q) are a strong conflict in R.*

PROOF. Any healthy installation containing p will also contain c; likewise for q and c'. Since c and c' conflict, there are no healthy installations that contain p and q. $\square$

Since the co-installability check is extremely expensive (NP-complete in the size of R), while conjunctive predecessors can be precomputed in linear time and then checked in constant time, one can further optimise the above algorithm, moving directly from `candidates` to `strongconflicts` all pairs p, q which are obtained following only conjunctive dependencies in the predecessor relation.

For example, if we take a look at the repository from Fig. 1, we see that it is impossible to install package `a` without also installing package `b` (there is a conjunctive dependency). Hence, `b` is a strong dependency of `a`. However, for the other dependencies of `a`, (all disjunctive), matters are more complicated and we will have to check for co-installability using a SAT solver.

5. EMPIRICAL MEASUREMENTS

In parallel with our formal complexity and algorithmic investigations, we also performed empirical measurements on the Debian and Mandriva distributions: this allowed us both to assess the relevance of the optimisations outlined in the previous sections, and to identify the most relevant form for presenting the measurements to allow their immediate exploitation in terms of quality assurance.

5.1 Computation of Strong Conflicts for Debian and Mandriva Distributions

We have developed a tool[6] that implements all the optimisations mentioned in the previous section and allows to compute the strong conflicts of a given repository.

We used this tool to analyse two kinds of repositories which are representative of different development models and packaging policies: the `stable` repository of the Debian distribution (version 5.0, February 2009) and the `main` release of the Mandriva 2009.1 distribution[7].

The table in Fig. 2 summarises the experimental data we gathered. The “Packages” row shows the number of packages and the time spent parsing the distribution metadata; the “Explicit conflicts” row shows the number of explicit conflicts mentioned in the metadata (and the time spent gathering this data).

The two “Pairs to check” rows show the reduction in search space obtained by using the optimised algorithm. Using the naive method, all pairs need to be checked, which, for a repository of size n, will result in checking $n(n-1)$ pairs. The optimised method uses the result of Proposition 2 and only checks the pairs that are necessary, reducing the search space by 63,3 percent for Debian and 57,2 percent for Mandriva.

5.2 Assessment

The optimised algorithm provides a significant improvement over the naive one, and it allows to compute the strong conflict measures weekly on a modern commodity Unix workstation.[8].

The algorithm being fully parallelisable, it would be straightforward to reduce the computation time to a few hours using a small cluster, or a few minutes using a medium-sized cluster. However, we noticed on all cases that we have analysed (not only the two we have selected for presentation in this paper), that the vast majority of strong conflicts found in GNU/Linux distributions can be found following only conjunctive dependencies. These do not require a call to a SAT-solver, and can be computed extremely quickly (less than a second). In the figures above, for example, 4878 strong conflicts in Mandriva can be found in less than a second following conjunctive dependencies, and the huge amount of time spent to cover the rest of the search space only brings in 39 extra strong conflicts.

In practice, computing the conjunctive strong conflicts might provide a useful approximation when a quick estimate of the final result is required: all packages having large exclusion sets due to conjunctive strong conflicts will have a large exclusion set anyway, and they will be brought to the attention of a quality assurance team right away.

6. PRESENTATION OF THE STRONG CONFLICT DATA

Since the Mandriva and the Debian repository have several thousands of strong conflicts, the question of how to present the information for quality assurance purposes is essential.

We have already shown in table 1 earlier in this article how a simple textual representation of the analysis results, with a list of the packages in decreasing order of their exclusion set, allows to immediately find the more problematic packages.

At an aggregate level, one can also present to the quality assurance team a simple graph showing the density distribution of the strong conflicts, like the one in figure 2b. This may provide a visual way of assessing the overall distribution of the exclusion sets present in the repository: a point on this graph at (x, y) means that there are y packages with x strong conflicts (with the shape of the dot indicating their cause); dots in the far right of the picture are the ones coming from the more problematic packages: for example in figure 2a we see that there are three packages with 881 strong conflicts.

So, finding highly problematic packages is easy, but we need now a way to allow the quality assurance team to fix them, and for this it is necessary to precisely identify *the reasons* of the abnormal level of strong conflicts for a given package: the data presented in the table is very useful to see the packages one should focus on (for example, `ppmtofb` is clearly a disaster), but says nothing about the reason of the disaster. When discussing with quality assurance people from the Debian project, it became clear that the raw data presented like this is really inadequate.

Our first step to provide useful reports has then been to compute, for every *strong conflict* between a package p and a package q, one of the *explicit conflicts* that are necessary for the strong conflict to occur, according to Theorem 2, and then to group all strong conflicts using these “roots”.

In the case of `ppmtofb`, for example, this allows to produce a report like the following one.

[6]Available from the subversion repository at `http://tinyurl.com/strongconflicts`
[7]The data can be downloaded from `snapshot.debian.net` for Debian and `http://tinyurl.com/sc-mdv-2009-1` for Mandriva
[8]Intel Xeon 3 GHz processor, 3 Gb of memory

	Debian 5.0		Mandriva 2009.1	
	Number	Time	Number	Time
Packages (parsing)	22311	15.88 s	5849	12.03 s
Explicit conflicts	1003	0.45 s	121	0.06 s
Pairs to check (naive)	497758410	-	34204952	-
Pairs to check (optimised)	183050886	0.42 s	14640594	0.06 s
Strong conflicts (conjunctive)	6384	0.06 s	4878	0.38 s
Strong conflicts (other)	531	471365.77 s	39	40660.91 s
Strong conflicts (overall)	7918	471384.64 s	4917	40668.93 s

Table 2: Strong conflicts in Debian and Mandriva

```
Package p          Explicit conflict    Package q
ppmtofb-0.32-0.1   python-2.5.2-3       atomix-2.14.0-1
                   ppmtofb-0.32-0.1
ppmtofb-0.32-0.1   python-2.5.2-3       live-magic-1.5
                   ppmtofb-0.32-0.1
                          ⋮
                    (2368 lines)
                          ⋮
```

We went one step further, though: since the explicit conflict, even if it is not unique in general, plays clearly the role of an *explanation*, we used it as a key to organise the error report to the quality assurance team. For each package, we list the packages it strongly conflicts with, *clustered* according to the explicit conflict that leads to discovering them, in the following format:

```
nsc package-p :
 nsce1  c1 <-> c1' :
  * package-q-1-1
      dependency path from package-p to c1
      dependency path from package-q-1-1 to c1'
  ⋮
  * package-q-1-nsce1
      dependency path from package-p to c1
      dependency path from package-q-1-nsce1 to c1'
  ...
 nsce2  c2 <-> c2' :
  * package-q-2-1
      dependency path from package-p to c2
      dependency path from package-q-2-1 to c2'
  ⋮
  * package-q-2-nsce2
      dependency path from package-p to c2
      dependency path from package-q-2-nsce2 to c2'
  ...
```

where nsc is the number of strong conflicts that package p has; then, for each explicit conflict c_i, c_i', we indicate the number nscei of strong conflicts that come from it for each package q_i^j we provide the dependency path linking p to c_i and q_i^j to c_i' that exists according to Theorem 2.

After some further interaction with the Debian team, we have found that this is the most appropriate way to present the data. It allows to easily identify the explicit conflicts at the origin of the strong conflict, and then to zoom on each package in the exclusion set to see the chain of dependencies leading to the strong conflict.

7. ANALYSIS OF REAL-WORLD EXAMPLES

We validated our approach on real world examples coming from both the Debian and Mandriva distributions. Our findings confirm that it is quite easy to spot and fix the errors, when the root reason is a single explicit conflict. The public availability of all the quality assurance information has also allowed to verify that some of the serious problems identified via strong conflicts were either unknown, or, when identified, not handled at all because no objective measure was available to assess their impact on the overall quality of the component collection.

7.1 Dependencies on Obsolete Components

An excerpt of the 2368 lines of output produced by our tool for the most conflictual package ppmtofb in Debian is shown below:

```
2368 ppmtofb-0.32-0.1 :
 2368 (python-2.5.2-3 <-> ppmtofb-0.32-0.1) :
  * atomix-2.14.0-1 (conjunctive)
    ...
  * live-magic-1.5 (conjunctive)
    ...
  ...
```

This shows us that *all* of the 2368 strong conflicts involving ppmtofb are actually due to *one single* explicit conflict (the one between python-2.5.2-3 and ppmtofb-0.32-0.1). Here is the relevant part of the metadata for ppmtofb:

```
Package: ppmtofb
Version: 0.32-0.1
Replaces: ppmtoagafb
Provides: ppmtoagafb
Depends: libc6 (>= 2.7-1), libnetpbm9, libpopt0 (>= 1.10)
Recommends: python, netpbm
Conflicts: ppmtoagafb, python (>> 2.4)
```

Looking at this report and the metadata above a member of the quality assurance team sees immediately that the cause is the conflict between ppmtofb and an old version of python: as time went by, when the python package evolved to new versions, this dependency produced an incompatibility with all other packages using python. Clearly, ppmtofb has not been updated, or tested, for a while, and the responsibility is with its maintainer.

This problem had been cursorily reported to the Debian bug tracking system in 2006[9]; since there was no way of recognizing the importance of this bug report, no action had been undertaken: in the newest stable version of Debian from February 2009, the problem is still present.

[9]http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=385209

After we added our findings to the bug report, its severity was upgraded to 'serious' (which means that the package is unsuitable for release), and it was even suggested that the package be removed completely from the distribution.

This shows clearly that packages that have a high number of strong conflicts are really problematic, and yet tend to go unnoticed using current measures.

7.2 Insufficient Precision in the Metadata

Another relevant example in Debian is the package `libgd2-noxpm-2.0.36~rc1~dfsg3`, with 127 strong conflicts, which again emanate from one specific conflict:

```
127 libgd2-noxpm-2.0.36~rc1~dfsg-3 :
 127 (libgd2-xpm-2.0.36~rc1~dfsg-3 <-> libgd2-noxpm-
2.0.36~rc1~dfsg-3)
  * moodle-book-1.6.3-1.1 (conjunctive)
   ...
  ...
```

The explicit conflict in itself seems justified: both packages provide the same library, one in a version with support for XPM, and the other in a version without XPM.

However, this is not a simple conflict; in fact, the functionality of `libgd2-xpm` is a superset of the functionality of `libgd2-noxpm`. Therefore, it should in all cases be possible to *replace* the `libgd2-noxpm` package, if installed, by the `libgd2-xpm` package, without affecting the functionality of the system.

A simple fix for this problem would be to add an entry `Replaces: libgd2-noxpm` in the package specification for `libgd2-xpm`. The `Replaces` field informs the packaging system that it should remove the `libgd2-noxpm` package when installing the `libgd2-xpm` package, so that the installation can proceed.

7.3 Overlooked File Conflicts

In Mandriva 2009.1 distribution we find 881 strong conflicts for the package `perl-ExtUtils-ParseXS-2.19-2mdv2009.1`:

```
881 perl-ExtUtils-ParseXS-2.19-2mdv2009.1 :
  881 (perl-2:5.10.0-25mdv2009.1 <->
     perl-ExtUtils-ParseXS-2.19-2mdv2009.1)
    * openoffice.org64-gnome-1:3.0.1-5mdv2009.1 (conjunctive)
      ...
    ...
```

When we look at the metadata, we see that the conflict between `perl` and `perl-ExtUtils-ParseXS` is caused by the fact that they both contain the file `/usr/share/man/man3/ExtUtils::ParseXS.3pm.lzma`. In fact, both packages install a version of the `ExtUtils::ParseXS` library, though in different directories: it is only the man page that is installed in the same place and thus causes the conflict.

A possible solution to the problem is to remove the `ExtUtils::ParseXS` library from the `perl` package and either add a dependency on `perl-ExtUtils-ParseXS` to `perl`, or change those packages that use the `ExtUtils::ParseXS` library from `perl` to depend on `perl-ExtUtils-ParseXS` instead.

There are two other packages in the distribution that suffer from a similar problem: `perl-Pod-Escapes` and `perl-DB_File`. Both of these packages have 881 strong conflicts as well.

7.4 Assessment

The three examples highlighted above correspond to very serious quality issues in mainstream GNU/Linux distribution that had never been addressed before, and show that there was no metric previously available capable of clearly identifying them. Any user of `ppmtofb`, for example, would experience significant problems when trying to install any of the 2368 packages in its exclusion set (some of them very popular), but the data made available to him is not enough to reveal the extent of the issue, and the problem has little chances to be reported, and even less chances to be taken seriousy.

The availability of our new metric based on strong conflicts and exclusion sets has allowed to immediately pinpoint these issues (and several others) as relevant ones to the quality assurance team, that has taken steps to fix them: looking at the bug tracking system entry for `ppmtofb` (at `http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=385209`), for example, it is easy to see how the dependency problem was reported by one user back in August 2006, and totally ignored for three years, while as soon as we filed a report in February 2009 with the evidence coming from the exclusion sets, the issue was dealt with immediately.

A natural research line woudl be to push the analysis further and identify automatically the precise problem at the origin of a large exclusion set. As we have seen in the examples above, when a single explicit conflict is a the root of a large exclusion set, it is natural to identify it as the root cause, but understanding why this single explicit conflict is there requires actual knowledge of the functionality of the package (for `ppmtofb` it is a matter of updating the package to a more recent version of `python`, for `perl-ExtUtils-ParseXS` it is a forgotten file conflict): it is surely interesting to try and build a taxonomy of common error patterns, but the authors believe that a full automatisation will be difficult to achieve.

8. DISCUSSION AND RELATED WORKS

Predicting and ensuring the correct behaviour of components when composed into an assembly is a central issue in modern research, and has been extensively stuudied, but we believe that the maintenance of GNU/Linux distributions poses several novel challenges.

On one side, a significant part of the literature studies the dynamic aspects of composition: knowing the behaviour of the components, one looks for means to ensure certain properties of the behaviour of the system obtained by assembling them, like for example in [8, 15]. This is a crucial issue, but in the world of GNU/Linux distributions we are still very far from having any information available on the behaviour of each component after installation, so it is too early to tackle this facet of the problem.

On the other side, the research on static inter-module dependencies is essentially performed at the level of the source code, with a different focus: in [11, 12], dependencies are automatically extracted from huge sets of source code, and then used to predict failures, but not to identify issues in the architecture of the code, unlike what we do with strong conflicts here; in [16] and [17] dependencies are used as a guideline for testing component-based systems; finally, [13], which shares similar concerns with us, as the analysis of the architectural dependencies is used to improve the modularisation of the software architecture, the size of the problem is sufficiently small (some 20 components) to allow manual analysis and resolution of component relationships, which is totally unfeasible in our case. More recently, a notion of *strong dependency* has been introduced by the Abate, Zacchiroli and the authors in [1].

What makes strong conflicts novel and appealing is the fact that they allow us to identify architectural problems in *huge* collections of components that are impossible to detect in other ways, as we have clearly shown with the selected examples presented in Sec. 7.

Furthermore, using strong conflicts results in significantly higher precision than more direct (and less costly) methods such as using the conflicts or dependencies mentioned directly in the metadata; a package like `ppmtofb` can have only a few conflicts or dependencies, and still has an enormous exclusion set.

The number of conflicts in Debian distributions is decreasing slightly over the different releases, even though the number of packages increases steadily. Thus it seems that many declared conflicts are not caused by incompatibility between components, but by other factors, as seen in the examples presented above. With strong conflicts, distribution editors can check the explicit conflicts in their distribution systematically, and keep only the conflicts that are absolutely necessary.

We believe that all large bodies of software components, like OSGI, Eclipse plugins[10] or Firefox extensions, will eventually adopt a dependency model similar to the one existing in GNU/Linux distributions, and the notion of strong conflict, with the associated algorithms, will be of paramount importance in their maintenance.

9. CONCLUSIONS

In this paper, we have presented and motivated a fundamental property for large repositories of FOSS packages, *strong conflicts*.

Despite its theoretical algorithmic complexity, we have already performed large-scale tests on two major GNU/Linux distributions, Debian and Mandriva, indicating that strong conflicts can be mechanically checked in reasonable time on real-world component repositories, and we have shown how major issues in the quality of the repositories can be easily identified using strong conflicts.

These tools are currently being integrated in the quality assurance process of these distributions, and we believe that exactly the same concepts can be applied to many other coarse-grained loosely coupled component collections, like Eclipse plugins.

Acknowledgements.

The starting idea came from a discussion with Arnaud Laprevote, from the Mandriva group, during the meeting held in Lisbon in October 2008 of the Mancoosi research project[11], which is dedicated to improving the maintenance of installations for users of GNU/Linux distributions.

Interesting discussions on these issues have taken place during the Mancoosi weekly meeting at the University Paris Diderot since then: special thanks go to Pietro Abate, Yacine Boufkhad, Alfonso Pierantonio, Ralf Treinen, Jerôme Vouillon and Stefano Zacchiroli.

[10]This is already happening with the Eclipse P2 provisioning platform.

[11]See `www.mancoosi.org`.

Data Availability

The metadata for Debian and Mandriva used for the experimental validation, together with the full presentation of the strong conflicts found is available from `http://www.dicosmo.org/Mancoosi/measures-data.tar.bz2`.

10. REFERENCES

[1] P. Abate, J. Boender, R. Di Cosmo, and S. Zacchiroli. Strong dependencies between software components. In *Proceedings of ESEM 2009*, pages 89–99. IEEE Press, 15-16 Oct. 2009.

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

[3] B. Boehm and C. Abts. COTS integration: Plug and pray? *IEEE Computer*, 32(1):135–138, Jan. 1999.

[4] M. Broy and E. Denert. *Software Pioneers: Contributions to Software Engineering.* Springer-Verlag, 2002.

[5] I. Crnkovic, H. Schmidt, J. Stafford, and K. Wallnau. Anatomy of a research project in predictable assembly. In *Proceedings of CBSE5*, 2002. White paper.

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

[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. Technical report, EDOS, Apr. 2006. D2.2, available as `http://tinyurl.com/cd9mzo`.

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

[9] M. Larsson, A. Wall, C. Norström, and I. Crnkovic. Using prediction-enabled technologies for embedded product line architectures. In *Proceedings of CBSE5*, 2002.

[10] F. Mancinelli, J. Boender, R. di Cosmo, J. Vouillon, Berke, X. Leroy, and R. Treinen. Managing the complexity of large free and open source package-based software distributions. *ASE '06*, 0:199–208, 2006.

[11] N. Nagappan and T. Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. *ESEM 2007*, 0:364–373, 2007.

[12] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In P. Ning, S. D. C. di Vimercati, and P. F. Syverson, editors, *Proceedings of CCS 2007*, pages 529–540. ACM, 2007.

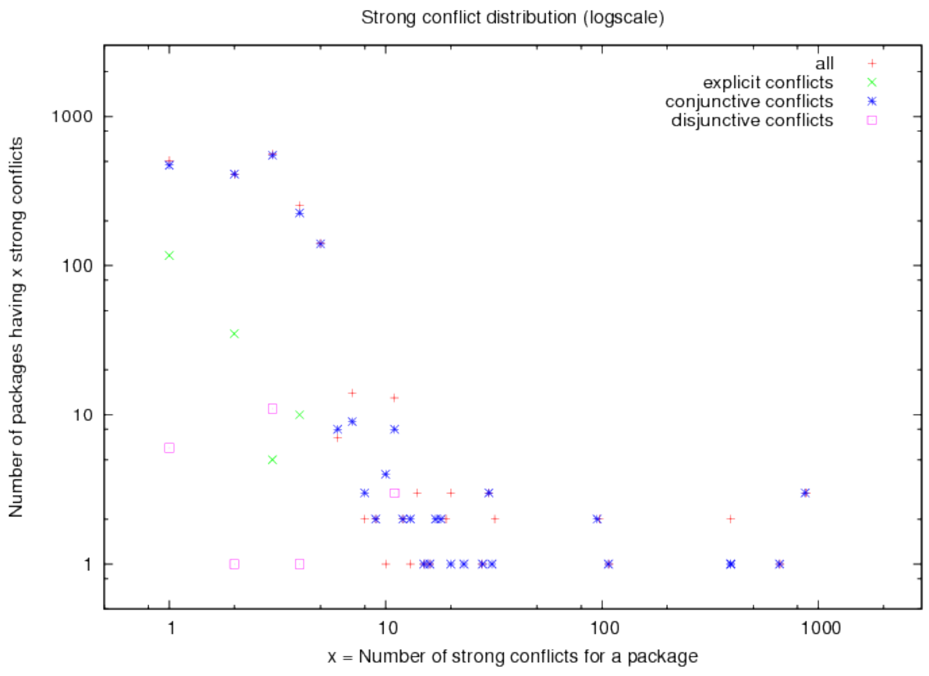
[13] H. Pei-Breivold, I. Crnkovic, R. Land, and S. Larsson. Using dependency model to support software architecture evolution. In *Proceedings of Evol'08*, September 2008.

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

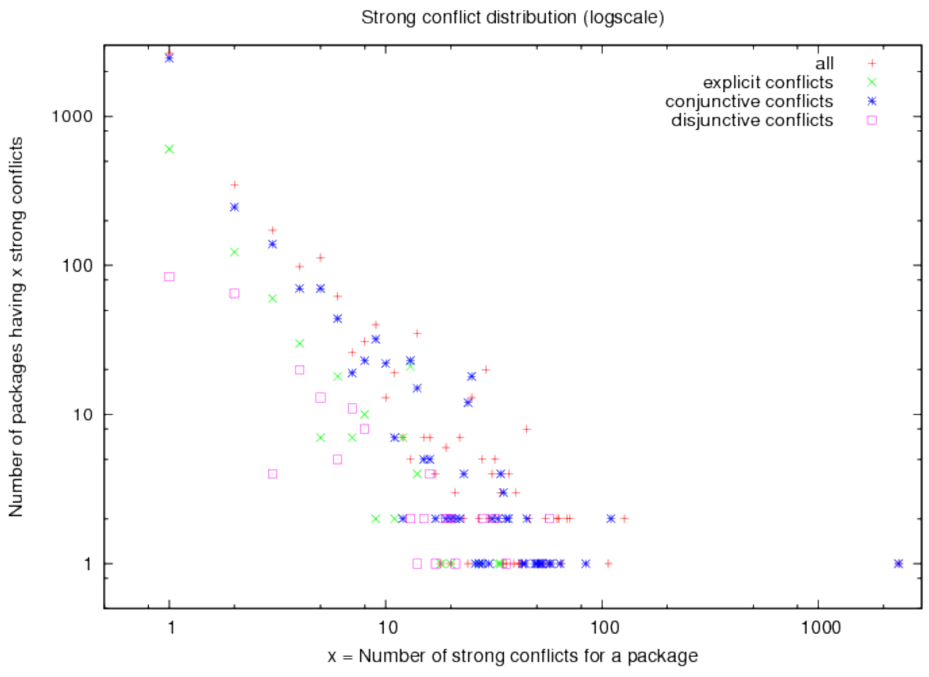
[15] M. Tivoli and P. Inverardi. Failure-free coordinators synthesis for component-based architectures. *Sci. Comput. Program.*, 71(3):181–212, 2008.

[16] 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.

[17] I.-C. Yoon, A. Sussman, A. Memon, and A. Porter. Effective and scalable software compatibility testing. In *Proceedings of ISSTA '08*, pages 63–74, New York, NY, USA, 2008. ACM.



(a) Mandriva `2009.1`



(b) Debian 5.0

Figure 2: Distribution of strong conflicts in package repositories