Learning from the Future of Component Repositories[☆]

Pietro Abate[a], Roberto Di Cosmo[a], Ralf Treinen[a], Stefano Zacchiroli[a]

[a] *Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126, CNRS, F-75205 Paris, France*
{abate,treinen,zack}@pps.univ-paris-diderot.fr, roberto@dicosmo.org

Abstract

An important aspect of the quality assurance of large component repositories is to ensure the logical coherence of component metadata, and to this end one needs to identify incoherences as early as possible. Some relevant classes of problems can be formulated in term of properties of the *future repositories* into which the *current* repository may evolve. However, checking such properties on all possible future repositories requires a way to construct a finite representation of the infinite set of all potential futures. A class of properties for which this can be done is presented in this work.

We illustrate the practical usefulness of the approach with two quality assurance applications: (i) establishing the amount of "forced upgrades" induced by introducing new versions of existing components in a repository, and (ii) identifying outdated components that are currently not installable and need to be upgraded in order to become installable again. For both applications we provide experience reports obtained on the Debian free software distribution.

1. Introduction

As a consequence of the fact that software systems must undergo continuing evolution [2], any software has its own *evolution history*, made of changes, revisions, and releases. By mining those histories—which are often conveniently stored in software repositories—one may find interesting facts and properties of software systems [3]. The advent of component-based software systems [4] has not diminished the relevance of this approach. We now have *component repositories*, where new releases of individual components get pushed to.

Free and Open Source Software (FOSS) *distributions* are particularly interesting data sources for mining component repositories [5]. This is so partly because their components—called *packages* in this context—are freely available to study; and partly because some of them, such as the Debian distribution,[1] are among the largest coordinated software collections in history [6].

[☆] This paper is an extended version of [1], appeared in proceedings of CBSE 2012, the 15th International ACM SIGSOFT Symposium on Component Based Software Engineering.
Work performed at IRILL http://www.irill.org, center for Free Software Research and Innovation in Paris, France.

[1] http://www.debian.org

```
Package: libacl1-dev
Source: acl
Version: 2.2.51-5
Architecture: amd64
Provides: acl-dev
Depends: libc6-dev | libc-dev, libacl1 (= 2.2.51-5),
  libattr1-dev (>= 1:2.4.46)
Conflicts: acl (<< 2.0.0), acl-dev,
  kerberos4kth-dev (<< 1.2.2-4)
```

Figure 1: Example of Debian meta-data (excerpt)

Software packages share important features with *software component models* [7], but exhibit also some important differences. On one side, packages are, like components, reusable software units which can be combined freely by a system administrator; they are also independent units that follow their own development time-line and versioning scheme. On the other side, packages, unlike what happens in many software component models, cannot be composed together to build a larger component. In fact, packages are intended to be installed in the shared space of an operating system, and they have to share the resources provided by the system. This has important consequences on the inter-package relationships expressed using fairly expressive package metadata.

Figure 1 shows an example of the metadata of a package in the popular Debian distribution. (We will focus on Debian for the purpose of this paper, however our findings apply equally to all other popular package models [8].) As the example shows, inter-package relationships can get pretty complex. In general, packages have both *dependencies*, expressing what must be satisfied in order to allow for installation of the package, and *conflicts* that state which other packages must not be installed at the same time. While conflicts are simply given by a list of offending packages, dependencies may be expressed using logical conjunction (written '`,`') and disjunctions ('`|`'). Furthermore, packages mentioned in inter-package relations may be qualified by constraints on the version of the package. There are also some further types of metadata, like virtual packages [9], but we may ignore them for the purpose of this paper, as we ignore many types of metadata that are not expressing mandatory inter-package relationships.

An important feature of component architectures is that individual components may be upgraded independently of other components. In the case of package repositories, such upgrades may happen on two different levels: (a) system administrators upgrading the package installation on their machines, and (b) the maintenance team of a package distribution accepting upgrades of existing packages, or new packages, into the package repository. It is the latter that interests us for the present work since it leads to interesting quality assurance issues. Due to the frenetic pace of change in package repositories these can be properly dealt with only by using automated tool support. For instance, the development archive of the Debian distribution (the so called "unstable" package repository), with its more than 38.000 packages as of November 2012,

```
Package: foo
Version: 1.0
Depends: bar (<= 3.0) | bar (>= 5.0)

Package: bar
Version: 1.0

Package: baz
Version: 1.0
Depends: foo (>= 1.0)
```

Figure 2: Package `bar` challenges package `foo`

receives each single day about 150 upgrades.

Previous work [10] has focused on analyzing the metadata contained in a snapshot of a package repository. For instance, chasing not installable packages is a common quality assurance activity for distributions [11]. Efficient tools to attend it exist, despite the NP-completeness of the underlying algorithmic problem. In this paper we argue that not only the past and present of component repositories are worth studying. The *future* of component repositories, largely unexplored up to now, is equally interesting since it allows us to establish important facts and properties, especially in the area of component quality assurance.

We have investigated two practically relevant scenarios where the analysis of the possible future evolutions of a repository (or *futures* for short) provides precious information.

Challenging upgrades. When a repository is fine according to some quality measure (e.g. all packages contained therein are installable), but a specific class of its futures is problematic (e.g. they are affected by metadata incoherences that *will* make some packages not installable), we might want to prevent or delay such evolutions.

We use future analysis to develop algorithm and tools that identify the most "challenging" upgrades. In a given repository, we say that a version v of a package p *challenges* another package q when in all futures where p is upgraded to version v (while all other packages are kept unchanged) the package q becomes not installable, no matter how p's metadata might have changed. The number of packages that are challenged by an upgrade to version v of p tells us how disruptive for the repository the transition to the new v version of p is.

For example consider the repository in Figure 2. All packages are currently installable, but it is easy to check that upgrading package `bar` to any future version greater than 3.0 and smaller than 5.0 will render the current packages `foo` and `baz` not installable. We say that all upgrades of `bar` to versions in the range $3.0 < \cdot < 5.0$ *challenge* `foo` and `baz` as in any future evolution of this repository containing `bar` with any version between 3.0 and 5.0, packages `foo` and `baz` will be not installable.

This notion can therefore help the quality assurance team in charge of guaranteeing the installability of the components in the repository. By knowing in advance that a given new version of a given component will challenge, i.e. make not installable, many

```
Package: bar
Version: 2.3

Package: baz
Version: 2.5
Conflicts: bar (> 2.4)

Package: foo
Version: 1
Depends: (baz (=2.5) | bar (=2.3)), (baz (<2.3) | bar (>2.6))
```

Figure 3: Package `foo` is outdated

components, they can either delay the arrival of the new version to the point when all other challenged components have been changed to avoid non-installability issues, or else coordinate component upgrades so that the time window in which those issues are present is minimized.

Outdated packages. If a quality problem is observed in the current repository (e.g. a package is not installable due to an unsatisfiable dependency), and if we can show that the problem will be present in all futures of some kind, then we know that we have to *move out* of that class of futures in order to solve the problem. We use future analysis to develop tools that identify all the "outdated" packages. A package p contained in a repository is said to be *outdated* when it is not installable and *remains so* in all futures of the repository where p is unchanged. This means that the only way to make p installable is to upload a fixed version of the package. This information is useful for quality assurance since it pinpoints packages where action is required.

This definition of outdated packages grew out of a concrete problem with a previous quality analysis of package repositories. Following the earlier work of some of the authors of this paper [10], we implemented a daily automated check for not installable packages in the Debian repository [2]. This analysis displays on a typical day hundreds of packages that are not installable. One of the reasons for this amount of data is that a package may be not installable because one of its dependencies is not installable, that is the not installability might not be the "fault" of the package found to be not installable. While it seems impossible to define a formal criterion that decides in any single case who is to blame for a problem, we tried to find a criterion where the fault is definitely with the very package that is found to be not installable. This lead us to the definition of outdated packages — packages that are not installable, and that can only be fixed by the package itself (and not by one of its dependencies).

Consider, for instance, the repository in Figure 3. `foo` is not installable since its second dependency clause cannot be satisfied, as we have neither `baz` in a version smaller than 2.3, nor `bar` in a version greater than 2.6. Is it possible to make `foo` in version 1 installable by upgrading one or both of `baz` and `bar`? If we advance both of

[2] http://edos.debian.net/edos-debcheck/unstable.php

these packages then foo will not be installable since its first dependency clause will not be satisfied. Upgrading baz alone will not work, because baz can only be upgraded to versions greater than the current version 2.5, hence baz (<2.3) can never be satisfied. Upgrading bar alone will not work either since when we upgrade it to a version greater than 2.6 then we will get a conflict with baz in its current version. Hence, by investigating all possible cases we find that foo in version 1 is indeed outdated. The algorithms and tools we are going to describe are capable of doing this analysis automatically.

The general problem in answering all sorts of questions about the future of a component repository is that there are infinitely many possible futures. The typical constraints on repository evolutions are indeed pretty liberal: *(i)* both additions and removals of packages are allowed; *(ii)* package upgrades must increase version numbers; and *(iii*) packages can be organized into clusters that must be *synchronized*, i.e. all packages in the cluster shall carry the same version. Common use cases of the latter requirement are packages that stem from a single source (and hence follow the same versioning scheme), or synchronization requirements on component versions that are known to work well together.

Paper structure. The main contribution of this paper is a formal framework to reason about future properties of a component repository, by only looking at a finite number of repositories. We start in Section 2 by giving an informal overview of the proposed approach. We formalize the notion of repositories in Section 3. In Section 4 we use the formalization to capture the set of properties about futures that can be established by considering a finite set of repositories. The set of futures is infinite, but we show in Section 5 that a general class of *straight* properties can be checked by exploring a finite and manageable set of futures. Section 6 discusses two quality assurance applications of the proposed formal framework: finding challenging upgrades and outdated packages. Experiences with the two applications obtained on the Opam and Debian distributions are presented in Section 7. Related work is discussed in Section 8.

2. Overview of our Approach

Checking properties of a repository that is defined in terms of all its possible futures may seem an impossible task, as there are infinitely many future evolutions to consider. Fortunately, for an interesting class of properties, it is possible to reduce the search space to a manageable finite subset of all the futures, enabling the design of efficient analysis algorithms.

In this section we illustrate the key ideas of our approach for the problem of checking whether a package in a specific version is *outated* with respect to the current repository.

A central notion in the analysis of metadata is the one of an *installation*: an installation is a subset of a repository such that all package interrelationships are satisfied inside that set. In particular, a package is *installable* if it is contained in some installation. This notion will be made precise in Section 3. Another central notion is the one of a *future* of some initial repository, which is a repository with the only restriction that any package in the initial repository must not be downgraded to a lower version in the future.

We call a package p *outdated* if it is not installable in any future where we have "frozen" p to its initial version. Section 4 will make this more formal, and will also define a general class of properties to which our approach applies. One of the defining characteristics of the properties in this class will be that they are universal quantifications over all futures of a repository. The property of package p in version v being outdated is indeed of this form since it can be rephrased as: for any future, if it contains p in version v than this p is not installable in this future.

Our approach consists of two main steps:

1. First we show that we may restrict the analysis to a finite, albeit still very large, set of futures that is representative of all possible futures.
2. Then we show that this large finite set of representative futures can be folded into only one repository, which then can be efficiently checked by an automatic tool.

Optimistic futures. The first insight is that when we upgrade packages in a repository, that is replace them by newer versions, then we may assume for the purpose of our analysis that these new versions behave as nicely as possible, that is they do not depend on any other packages and do not conflict with any packages. We call such a future *optimistic*. The restriction to optimistic futures is justified by the fact that if a package is installable in some future then it is also installable in some optimistic future (since dropping dependencies and conflicts just make installation easier); hence the package is not installable in any future iff it is not installable in any optimistic future. This argument will be made formal in Section 5.1.

Conservative futures. Next, we cope with the fact that when moving to a future of a repository we may arbitrarily remove packages or introduce new packages.

Removals of packages are not relevant for us since removals can never make a package installable: if a package is installable in some future from which we have removed some packages then it is also installable in the same future into which we put back the removed packages - the chosen installation just will not contain the packages that were removed in the first place.

Addition of new packages, however, may make a package installable. This may be the case when a package expresses a dependency on a package that is missing from the repository. In that case, adding the missing package may solve the issue. We will show that this is indeed the only case where adding new packages may solve an installability issue. As a consequence, it is sufficient to look at futures where new packages are only introduced when they are mentioned in the dependencies of some package already present in the current repository.

This leads us to the following notion: a future F of R is *conservative* iff F contains all packages of R, possibly in a newer version, and if F contains only packages whose names occur in R, either as names of existing packages or in dependencies. The restriction to conservative futures will be formally justified in Section 5.2.

Observational equivalence. So far we have seen that we may restrict our analysis to futures that are both optimistic and conservative. Unfortunately there are still infinitely many such futures since each single package may assume infinitely many different versions in the future. However, if two different future version numbers of a package q behave exactly the same with respect to all version constraints in dependencies and conflicts of other packages (these exist only in the versions of packages in R since we have already restricted ourselves to optimistic futures), then these two versions of q are equivalent in what concerns relations to other packages. We call two such versions of q *observationally equivalent*. Since R contains only finitely many packages R, and hence only finitely many constraints of versions of q, there may exist only finitely many different equivalence classes of versions of q. We will explain in Section 5.3 how a set of representatives of these equivalence classes can be efficiently computed.

Folding many futures into one. By considering optimistic and conservative futures, and only one representative per equivalence class of future versions of packages, we get a finite set of possible futures. If we are interested in exploring only futures that modify a single package, or a small set of packages that must evolve in lockstep, as is the case for computing challenging packages, this reduction is enough to design an efficient algorithm.

Computing outdated packages still requires to explore all the futures in the set, and this poses a significant challenge: we have reduced the infinite number of futures to a finite set, but the size of this set is astronomical. If we assume that different packages may advance independently, and even if we have only c possible (current or future) versions per package to consider, then we get c^n many future repositories where n is the number of packages, which in case of Debian is around 38.000.

To overcome this difficulty, we can again exploit the fact that our property is expressed as a quantification over all installations in all possible futures: we build one big repository U that contains all representatives of future versions of packages in optimistic and conservative futures of R, and remark that the set of installations allowed by U is exactly the same as the set of installations allowed by any future of R, because installations may never contain two different versions of a same package. Hence, p in version n is outdated if p in version n is not installable in the repository U, which contains "only" $c*n$ many packages, so that the analysis can be performed quite efficiently (less than a minute on a standard desktop computer—see Section 7).

Synchronized upgrades. There is one last point to consider: in reality, packages do not evolve in isolation in package repositories since large applications are usually split into several "synchronized" packages. For instance, LibreOffice is released as a single software by its developers, but it gets split into about a hundred different packages (one for each of its main applications, one for each localized version, etc.) by distribution editors. All those packages carry the same version and the upload of new LibreOffice releases to the repository will advance the versions of all involved packages at once. Let us call a *cluster* a set of packages stemming from the same source.

The problem now, when investigating installations w.r.t. the repository U, is to weed out installations containing packages that are in the same cluster but do not have synchronized versions. This can be achieved by having each package p of future version

n in a cluster conflict with versions of other packages in the same cluster that are not synchronized with n. In this way we can again fold back a constraint on the structure of future repositories into the packages that occur in the repository U.

We will discuss some of the issues related to clustering in Section 6.

3. Packages and repositories

Several analysis proposed in previous work use a simplified model of packages and repositories originally introduced in [10], where version constraints are expanded out.

In our work, we need to take version constraints explicitly into account, so we introduce here a model of packages and repositories that is more refined. We will build on it in Section 4 to capture the notion of a future of a repository.

3.1. Packages

Let N be a set of names and V be an ordered set of version numbers. We assume that the order $\leq$ on the set V:

- is *total*, i.e. for all versions v_1, v_2 one has either $v_1 < v_2$ or $v_1 = v_2$ or $v_1 > v_2$;
- is *dense*, i.e. for any two versions $v_1 < v_3$ there exists v_2 such that $v_1 < v_2 < v_3$;
- and has *no upper endpoint*, i.e. for every v_1 there exists v_2 with $v_2 > v_1$.

The precise definition of package names is not important for the purpose of this work. All packaging systems allow to decide which of any two given versions is more recent, which translates in our setting to V being totally ordered. All packaging systems that the authors know of allow a package maintainer to produce versions that are lexicographic combinations of elements, like 27.14.1.2. In the case of Debian or Red Hat GNU/Linux, for instance, a package maintainer might package a software that was released by its author under version 1.2, as 1.2-1 and possibly later as 1.2-2, 1.2-3 and so on, being sure that these package versions will be smaller than anything that starts on 1.3. The abstract property behind this is the density property required above. The absence of upper endpoints is of course necessary to ensure that a package maintainer will never run out of versions for new packages when he needs them.
A version *constraint* is a unary predicate on versions: for any $v \in$ V we may write constraints $= v$, $\neq v$, $< v$, $> v$, $\leq v$, $\geq v$, or no constraint at all. The set of constraints is denoted by $\textsc{Con}$; this set also contains a special element $\top$ indicating the absence of a version constraint. Package constraints may be used in order to refine package dependencies like dependencies and conflicts.

We define for any constraint $c \in \textsc{Con}$ its *denotation* $[[c]]$:

$$\begin{array}{rcl}
[[= v]] & = & \{v\} \\
[[\neq v]] & = & \{w \in \mathrm{V} \mid w \neq v\} \\
[[< v]] & = & \{w \in \mathrm{V} \mid w < v\} \\
[[\geq v]] & = & \{w \in \mathrm{V} \mid w \geq v\} \\
[[> v]] & = & \{w \in \mathrm{V} \mid w > v\} \\
[[\leq v]] & = & \{w \in \mathrm{V} \mid w \leq v\} \\
[[\top]] & = & \mathrm{V}
\end{array}$$

The denotation $[[c]]$ of a constraint c is simply the set of versions that satisfy the constraint c. In particular, any version satisfies the empty constraint $\top$.

Definition 1. A *package* p is a tuple (n, v, D, C) consisting of:

- a package name $n \in \mathrm{N}$,
- a *version* $v \in \mathrm{V}$,
- a set of dependencies $D \in \mathscr{P}(\mathscr{P}(\mathrm{N} \times \textsc{Con}))$,
- a set of conflicts $C \in \mathscr{P}(\mathrm{N} \times \textsc{Con})$.

where we use $\mathscr{P}(X)$ for the set of all subsets of X. We also use the infix notation $(.)$ to access individual components (e.g. $p.n$ stands for the name of package p). This notion extends to sets of packages, e.g. $R.n$ is the set of names of packages in the repository R.

Example 1. Figure 4 shows a set of packages in a textual representation inspired by Debian GNU/Linux. The first package in this figure, denoted by the first three lines in the left column, would be written in the abstract syntax of Definition 1 as

$$(\mathtt{a}, 1, \{\{(\mathtt{b}, (\geq 2)), (\mathtt{d}, \top)\}\}, \{\})$$

If we call this first package p and the complete set R then we have that $p.n = \mathtt{a}$, and $R.v = \{1, 2, 3, 5\}$

The dependencies of a package indicate which packages must be installed together with it, the conflicts which packages must not. Dependencies are represented as conjunctive normal form (CNF) propositional logic formulae over constraints: the outer set is interpreted as a conjunction, the elements of which are interpreted as disjunctions. Conflicts are represented as flat lists of constraints.

Example 2. A dependency

$$\{\{(\mathtt{a}, (> 2)), (\mathtt{b}, (< 5))\}, \{(\mathtt{c}, (\neq 4)), (\mathtt{d}, (= 42))\}\}$$

is to be read as “at least one of `a` in a version greater than 2 or `b` in a version smaller than 5, and also at least one of `c` in a version different from 4 or `d` in version 42.” A conflict

$$\{(\mathtt{e}, (\leq 7)), (\mathtt{d}, \top)\}$$

means “you must not install `e` in a version smaller or equal 7, and you must not install `d` in any version whatsoever.” However, installing `e` in some version greater than 7 is permitted.

```
Package: a                 Package: b             Package: d
Version: 1                 Version: 2             Version: 3
Depends: b (>= 2) | d      Conflicts: d

Package: a                 Package: c             Package: d
Version: 2                 Version: 3             Version: 5
Depends: c (> 1)           Depends: d (> 3)
                           Conflicts: d (= 5)
```

Figure 4: Sample (abstract) repository R, containing 6 packages. Note that packages `a` and `d` appear in multiple versions.

3.2. Repositories and installations

A repository is a set of packages that are uniquely identified by name and version:

Definition 2. A *repository* is a set R of packages such that for all $p,q \in R$: if $p.n = q.n$ and $p.v = q.v$ then $p = q$. A repository R also comes with an equivalence relation $\sim_R$ on the set $R.n$ of package names, called its *synchronization relation* that tells which packages shall carry the same version in the repository.

By abuse of notation we will write $p \sim_R q$ when $p.n \sim_R q.n$. Common examples of the synchronization relation are equality (i.e. no synchronization at all) and the relation that has as equivalence classes the names of packages stemming from the same source. For the sake of conciseness, we will also refer to packages using couples $(name, version)$ (instead of full tuples) when doing so does not lead to ambiguities. Note that it is always safe to do so in the context of a single repository because name and version together uniquely identify packages.

Example 3. Figure 4 shows a set of packages in a textual representation inspired by Debian GNU/Linux. Note that there are two packages with name `a`; the set is indeed a repository since these two packages are distinguished by different versions.

An installation is a consistent set of packages, that is a set of packages satisfying abundance (every package in the installation has its dependencies satisfied) and peace (no two packages in the installation are in conflict). Formally:

Definition 3. Let R be a repository. An R-*installation* I is a subset $I \subseteq R$ such that for every $p \in I$ the following properties hold:

abundance For each element $d \in p.D$ there exists $(n,c) \in d$ and a package $q \in I$ such that $q.n = n$ and $q.v \in [[c]]$.

peace For each $(n,c) \in p.C$ and package $q \in I$, if $q.n = n$ then $q.v \notin [[c]]$.

We write $Inst(R)$ for the set of all R-installations. We say that a package p is *installable* in a repository R if there exists an R-installation I such that $p \in I$.

Example 4 (R-installations). We illustrate this definition at hand of the repository R of Figure 4:

- $\{(a,1),(c,3)\}$ is not an R-installation due to the lack of abundance (the dependencies of $(a,1)$ is not satisfied, nor is the dependency of $(c,3)$);
- $\{(a,2),(c,3),(d,5)\}$ is not an R-installation either due to the lack of peace (there is a conflict between $(c,3)$ and $(d,5)$;
- $\{(a,1),(b,2)\}$ and $\{(a,1),(d,5)\}$ are both R-installations;
- the package $(a,1)$ is installable in R, because $(a,1) \in \{(a,1),(d,5)\}$;
- the package $(a,2)$ is not installable in R, because: (i) any R-installation containing $(a,2)$ must also contain $(c,3)$, (ii) any R-installation containing $(c,3)$ must also contain $(d,5)$, and (iii) this destroys the peace as $(c,3)$ and $(d,5)$ are in conflict. □

Some packaging systems (such as Debian's) have an additional implicit *flatness* condition on installations: one cannot install at the same time two different versions of the same package. In Example 4 above, the repository R in itself is not an R-installation because, among other reasons, flatness is violated by the presence of `a` and `d` in multiple versions. Although not explicitly supported, flatness can be easily encoded in our model by adding conflicts between packages of same name and different version.

Checking package installability is an NP-Complete problem, although modern SAT-Solvers can easily solve real instances of it [10].

Definition 4. Let P be a set of packages. We define $dep(P) := \bigcup_{p\in P}\bigcup_{d\in p.D} d$, and $depnames(P) := \{n|(n,c) \in dep(P)\}$.

That is, $dep(P)$ is the set of all pairs of names and constraints occurring in dependencies in P, and $depnames(P)$ its projection to names. We abbreviate $dep(\{p\})$ to $dep(p)$.

Example 5. Continuing with the example repository R of Figure 4 we have that

$$\begin{aligned} dep(R) &= \{(b,(\geq 2)),(d,\top),(c,(>1)),(d,(>3))\} \\ depnames(R) &= \{b,c,d\} \end{aligned}$$

A repository is synchronized if packages that have to be synchronized (according to $\sim_R$) share the same version number. Notice that a synchronized repository may contain only one version for any package name since the synchronization relation $\sim_R$, being an equivalence relation, satisfies that $p \sim_R p$ for every package $p \in R$.

Definition 5. A *cluster* of a repository R is an equivalence class w.r.t. the synchronization relation $\sim_R$. A repository R is *synchronized* if all packages belonging to the same cluster have the same version. We write $cluster(p)$ to denote the packages $\{r \in R|r \sim_R p\}$ which are synchronized with p.

Example 6. The repository of Figure 4 is not synchronized for *any* choice of $\sim_R$ since any synchronized repository must be flat, as we have observed above, and R isn't since it contains two different packages with name a, and also two different packages with name d. In case that $a \sim_R b$, the repository becomes synchronized when we drop $(a,1)$ and $(d,5)$.

```
Package: a               Package: c              Package: d
Version: 2               Version: 3              Version: 6
Depends: c (> 1)         Depends: d (> 3)
                          Conflicts: d (= 5)
```

Figure 5: A future of the repository of Figure 4.

4. Repositories and Their Futures

4.1. Futures

We now formalize the notion of possible future states of a component repository, or *futures* for short.

Real world component repositories evolve via modifications of the set of packages they contain such as package additions, removals, and upgrades. All component models impose some restrictions on how repositories may evolve; we consider three very common restrictions:

1. Name and version uniquely identify packages. It is not possible to have in a future two packages with the same name and version, but otherwise different metadata.
2. If a package with the same name of an existing package is introduced, then its version must be greater than the one of the existing package. In this case we say that the existing package is being *upgraded*.
3. Futures must be synchronized, that is, all packages belonging to a same cluster must evolve together.

Note that the only way to change package metadata is via an upgrade to a newer version. The above restrictions on repository evolution are captured by the following definition:

Definition 6. A repository F is a *future* of a repository R, written $F \in \mathit{futures}(R)$, if the following properties hold:

uniqueness $R \cup F$ is a repository; this ensures that if F contains a package p with same version and name as a package q already present in R, then $p = q$;

monotonicity For all $p \in R$ and $q \in F$: if $p.n = q.n$ then $p.v \leq q.v$;

synchronization F is synchronized (w.r.t. the synchronization relation $\sim_R$).

Note that $R \in \mathit{futures}(R)$, and that uniqueness implies that name and version together uniquely identify packages. We require the future to be synchronized w.r.t. the original relation $\sim_R$, that is, packages are not allowed to migrate from one cluster to another.

Example 7. The repository F shown in Figure 5 is a future of the repository R of Figure 4 in case the synchronization relation $\sim_R$ is simply equality of names. Uniqueness is satisfied since the two packages (a,2) and (c,3) are the same in R and in F, and the

```
Package:   white                Package: green
Version:   2.1                  Version: 5.2
Source:    nocolor              Source:  color
Depends:   black
Conflicts: red (> 5.2)

Package:   black                Package: red
Version:   2.1                  Version: 5.2
Source:    nocolor              Source:  color
Depends:   green (> 5.2), red
```

Figure 6: {`white`,`black`} and {`green`,`red`} are clusters

package (`d`,6) does not exist in R. Monotonicity is satisfied since the only package in F that is not also in R is (`d`,6), and 6 is greater than the two versions of packages with name `d` in R. Synchronization is trivial in our case. Note that F contains no package of name `c`, contrary to the repository R, which is perfectly allowed.

4.2. Outdated and Challenging Packages

We now formally define the notions of outdated and challenging packages that are the motivating examples of our work.

Definition 7. Let R be a repository. A package $p \in R$ is *outdated* in R if p is not installable in any future F of R.

In other words p is outdated in R if it is not installable (since $R \in \mathit{futures}(R)$) and if it has to be upgraded to make it ever installable again. This means that p cannot be made installable by “fixing” some of its dependencies — the only way of solving the issue is upgrading p itself to a newer version.

Example 8. We have seen in Section 1 an example of a package being outdated, namely package (`foo`,1) in the repository of Figure 3. In the example of the repository in Figure 4, the package (`a`,2) is not installable as we have seen in Example 4, it is however *not* outdated as it is installable in the future shown in Figure 5.

Example 9. Figure 6 shows an example involving non-trivial clustering. In this textual representation, clusters are formed by packages with the same value of the `Source` field, that is `white`∼`black` and `green`∼`red`. First, package `white` is currently not installable since any installation of it requires `black`, which in its current version requires a version of `green` that does not yet exist. If we ignored clustering then upgrading `black` to a newer version without any dependencies would solve the issue, that is `white` in version 2.1 would not be outdated. In the presence of clustering as shown, however, we find that that `black` and `white` form one cluster, as do `red` and `green`. Due to the Synchronization condition it becomes impossible to install at the same time different versions of `black` and `white`, or different versions of `red` and `green`. Hence, any installation of white in version 2.1 also requires black in version 2.1, which in turn requires green in a future version, and also red *a fortiori* in a future version, which conflicts with white. Hence, when taking clustering into account, we find that white in version 2.1 is outdated.

Definition 8. Let R be a repository, $p,q \in R$, and q installable in R. The pair $(p.n,v)$, where $v > p.v$, *challenges* q if q is not installable in any future F which is obtained by upgrading to version v all packages in $cluster(p)$ in R.

Intuitively $(p.n,v)$ challenges q, when upgrading p and its cluster to a new version v without touching any other package makes q not installable. The interest of this property is that it permits to pinpoint critical future upgrades that challenge many packages and that might therefore need special attention before being pushed to the repository.

4.3. Abstract properties of repositories

The properties *outdated* and *challenging* have in common the fact that they are defined in terms of installability w.r.t. some future of R. More generally, the two properties are *instances* of a more general class of *straight* properties of a repository. We will obtain the definition of straight properties in three steps:

1. We first characterize a class of *R-focused* properties of package sets (e.g., installations) that can be established by only looking at the packages whose names already exist in the original repository R (Definition 10).
2. We characterize a class of *R-safe* properties of futures of R that can be described by saying that all installations allowed in the future have some R-focused property (Definition 11).
3. Finally, we arrive at the characterization of *straight* properties of repositories that hold when all its futures satisfy some R-safe property (Definition 12).

Definition 9. Let R,P be two sets of packages. The *R-focus* of P is

$$\pi_R(P) := \{(p.n, p.v)) \mid p \in P, p.n \in R.n\}$$

Informally, $\pi_R(P)$ consists of the pairs (n,v) of package name and version such that a package p with that name and version appears in P, restricted to package names that are defined in R. In particular, $\pi_R(R)$ is the set of all the pairs (n,v) of package name and version that can be found in packages of R.

Example 10. Continuing Example 7, we see that $\pi_R(F) = \{(a,2),(c,3),(d,6)\}$, that is the focus contains indeed all the packages of F, since all names of packages in F also occur in R.

Definition 10. A property ϕ of package sets is called *R-focused* if for all package sets P_1 and P_2 (not necessarily subsets of R)

$$\pi_R(P_1) = \pi_R(P_2) \quad \text{implies} \quad \phi(P_1) = \phi(P_2)$$

A property ϕ of package sets is *downward-closed* if $P_1 \subseteq P_2$ and $\phi(P_2)$ implies $\phi(P_1)$.

Example 11. We take again the repository R from Figure 4, and consider different properties of a package set P:

- "P contains (a,2)": this property is R-focused (since it can be expressed as $(a,2) \in \pi_R(P)$) but is not downward closed as R itself satisfies this property, but the empty set does not.
- "P does not contain (a,2)" is R-focused for the same reason as in the first example, and is downward closed: if $(a,2) \notin \pi_R(P_2)$ and $P_1 \subseteq P_2$ then $(a,2) \notin \pi_R(P_1)$.
- "P does not contain a package with name `xyz`": note that `xyz` is not the name of a package in R, and as a consequence it can be easily seen that this property is not R-focused. It, is, however, downward-closed.
- "P has an even number of elements" is neither R-focused, nor downward-closed.

It is easy to see that the set of R-focused properties is closed under boolean combination.

A property of futures of R is R-*safe* if it can be expressed in terms of an R-focused property:

Definition 11. Let R be a repository. A property ψ_R of futures of R is called R-*safe* if there is an R-focused property ϕ s.t. for all futures F of R:

$$\psi_R(F) \Leftrightarrow \text{ for all } F\text{-installations } I\text{: } \phi(I)$$

Example 12. The property "p is not installable in F", where $p \in R$, is R-safe , since it is equivalent to the fact that any F-installation does not contain p, which is an R-focused property as we have seen (since $p \in R$).

An interesting property of R-safe predicates is that they are closed under conjunctions.

Finally, a property of a repository R is called *straight* if it is the closure of an R-safe property of futures of R.

Definition 12. Let R be a repository. A property Ψ of R is called *straight* if there is an R-safe property ψ_R of futures of R such that

$$\Psi(R) \Leftrightarrow \forall F \in \mathit{futures}(R).\psi_R(F)$$

Example 13. A property stating that a fixed package q is outdated in a repository R is a straight property of repositories R since

$$\Psi_{q\ outdated}(R) = \forall F \in \mathit{futures}(R).\forall I \in \mathit{Inst}(F).(q.n, q.v) \notin \pi_R(I)$$

However, it is not so obvious that this holds also for *challenged* packages: indeed, (p,v) challenges q in R can be naturally written as follows

$$\begin{array}{ll} & \forall F \in \mathit{futures}(R). \\ & \quad \big((p.n,v) \in \pi_R(F) \wedge (\forall n' \neq p.n.(n',v') \in \pi_R(F) \rightarrow (n',v') \in \pi_R(R))\big) \\ \rightarrow & \qquad \forall I \in \mathit{Inst}(F).(q.n,q.v) \notin \pi_R(I) \end{array}$$

and this formula has the shape $\forall F.\phi_1(F) \rightarrow \forall I.\phi_2(I)$, which is *not* the shape of a straight property.

For downward closed properties, though, the following result holds:

Lemma 1. *Let ϕ_1, ϕ_2 be properties of package sets, and ϕ_1 be downward-closed. Then the following two statements are equivalent:*

1. $\forall F \in \mathit{futures}(R) : \big(\phi_1(F) \rightarrow \forall I \in \mathit{Inst}(F) : \phi_2(I)\big)$
2. $\forall F \in$ *futures(R)*, $\forall I \in \mathit{Inst}(F) : (\phi_1(I) \rightarrow \phi_2(I))$

Proof. Observe that futures are closed under subset, that is if $I \subseteq F \in \mathit{futures}(R)$ then $I \in \mathit{futures}(R)$.

First we show that (1) implies (2). Let $F \in \mathit{futures}(R), I \in \mathit{Inst}(F)$, and let $\phi_1(I)$ be true. As we have seen, $I \in \mathit{futures}(R)$, so we may apply (1) by choosing I as the value of F, and obtain that $\phi_2(I)$ is true.

To show that (2) implies (1) let $F \in \mathit{futures}(R)$, $\phi_1(F)$ be true, and $I \in \mathit{Inst}(F)$. Since ϕ_1 is downward closed we also have that $\phi_1(I)$ is true, and we conclude by (2) that $\phi_2(I)$ holds. □

In other words, we can under certain circumstances fold restrictions on futures into restrictions on installations, and in particular we can reformulate (p, v) challenges q in R as $\forall F \in \mathit{futures}(R).\forall I \in \mathit{Inst}(F).\phi_{challeng.}(I)$ where

$$\begin{aligned}\phi_{challeng.}(I) = \ & \big((p.n, v) \in \pi_R(I) \\ & \wedge(\forall n' \neq p.n.(n', v') \in \pi_R(I) \rightarrow (n', v') \in \pi_R(R))\big) \\ & \rightarrow (q.n, q.v) \notin \pi_R(I)\end{aligned}$$

and hence *challenged* is a straight property, too.

As a consequence, both the property stating that a package p is outdated in R and the property stating that a pair (p, v) challenges a package q in R are straight predicates, and we will write them $\Psi_{outdated}$, resp. $\Psi_{challeng.}$.

The interest of straight properties will become apparent in the next section, where we show that to check straight properties of a repository R it is enough to consider futures containing the same package names as R: the set of features in these futures does not deviate from the one in R, only their version can, hence the name *straight*.

5. Controlling the future

In this section we show that we can compute a *finite* set of futures of a given repository, such that all straight properties can be verified by only looking at this set. We proceed by iteratively narrowing down the set of futures to consider, following the approach described in Section 2.

5.1. Optimistic futures

As a first step we show that it is sufficient to consider futures where all new versions of packages have no dependencies and no conflicts.

We may make this assumption since the property we are interested in talks about all possible installations that are allowed by all possible future repositories. For instance, a package p in version n is outdated if for any installation set I that is allowed in any possible future of R, the set I does not contain package p in version n. Our assumption is hence justified by the fact that any installation allowed in any future F is also an installation allowed in some optimistic future, namely the one obtained from F by dropping all dependencies and conflicts from all new versions of packages in F.

Definition 13. Let p be a package. We define a *blank package* $\omega(p)$ that has the same package name and version as p, but no dependencies and conflicts:

$$\omega(p) = (p.n, p.v, \emptyset, \emptyset)$$

Note that a blank package is always installable by definition, since in any installation a blank package is always abundant and peaceful. Blank packages are an extreme case the future where all constraints associated to a package disappear.

We can now build an *optimistic* vision of the future.

Definition 14. Let F be a future of R, and $S \subseteq F$. The *blanking* of S is:

$$\omega_R(S) = (S \cap R) \cup \{\omega(p) \mid p \in S \setminus R\}$$

Blanking a set S corresponds to blanking out all packages in S that are not in R. Obviously, when $F \in \mathit{futures}(R)$ then also $\omega_R(F) \in \mathit{futures}(R)$.

Definition 15. A repository F is an *optimistic future* of a repository R iff F is a future of R, $F = \omega_R(F)$, and if for every package $p \in F$ with $p.n \notin R.n$ the cluster of p is a singleton set.

In an optimistic future of R all packages that are not in R are hassle-free: they have no dependencies, no conflicts, and they do not impose any synchronization constraint. A simple kind of optimistic future can be obtained by replacing a package by a blank package with a newer version:

Definition 16. Given a repository R and a set $P = \{p_1, ..., p_n\} \subseteq R$, we write $R[P \mapsto v]$ for the repository obtained from R by replacing each package $p_i \in P$ with a blank package $(p_i.n, v, \emptyset, \emptyset)$. When $v > p_i.v$ for each i, this is called a *blank upgrade*.

Lemma 2. *Let R be a repository, $F \in \mathit{futures}(R)$, and I an F-installation. Then, $\omega_R(I)$ is an $\omega_R(F)$-installation, and the packages in I and $\omega_R(I) \cap R$ have the same name and version.*

PROOF. The blanking operation does not change version names, so the packages in I and $\omega_R(I) \cap R$ have the same name and version. Furthermore $\omega_R(I)$ is an $\omega_R(F)$-installation since for every package $p \in \omega_R(I)$ there exists a package $q \in I$ with $p.D \subseteq q.D$ and $p.C \subseteq q.C$: Since peace and abundance are satisfied for q in I they are also satisfied for p in $\omega_R(I)$. □

5.2. Conservative futures

As a second step of our reduction we address the fact that when moving from a repository to one of its futures we may drop packages, that is a future may not contain a package with a certain that existed in the original repository, or we may add new packages, that is a future may contain a package with a name that did not exist in the original repository.

One sees easily that we may ignore package removals from R since every installation I w.r.t. a future F of R in which we removed a set of packages D is also an installation w.r.t. $F \cup D$: If $I \subseteq F$ then, obviously, $I \subseteq F \cup D$. In other words, by looking only at future repositories where we do not have removed packages we cover already all possibles installations w.r.t. all possible futures. This is one part of the argument behind Lemma 3

Restricting the introduction of new packages, however, is less immediate as introduction of new packages into R may indeed make a package p installable that was not installable before—for instance, if p depends on some package q that was missing from R. The second part of the argument behind Lemma **??** below is that the only new packages that are essential for checking straight properties are packages whose name was mentioned in the dependencies of package p. Hence, it is sufficient to look only at futures where we possibly introduce new packages that are mentioned in the dependencies of p.

Definition 17. $F \in \mathit{futures}(R)$ is a *conservative* future of R if $F.n = R.n \cup \mathit{depnames}(R)$.

Lemma 3. *Let R be a repository, $F \in \mathit{futures}(R)$, and $I \in \mathit{Inst}(F)$. There exists an optimistic and conservative future G of R and $J \in \mathit{Inst}(G)$ such that $\phi(I) = \phi(J)$ for any R-focused property ϕ.*

PROOF. It is enough to build a future G of R and an installation J of G that has the same R-focus as I.
By Lemma 2, $\omega_R(I)$ is an $\omega_R(F)$-installation with $\pi_R(I) = \pi_R(\omega_R(I))$, and $\omega_R(F)$ is optimistic. We can then construct a future G of R by blanking all packages of F, and adding back all packages of R that may be missing in F; formally

$$\begin{array}{rcl} G & = & \{p \in \omega_R(F) \mid p.n \in R.n \cup \mathit{depnames}(R)\} \\ & \cup & \{p \in R \mid p.n \notin F.n\} \end{array}$$

By construction, G is an optimistic and conservative future of R. Let $J = \{p \in \omega_R(I) \mid p.n \in R.n \cup \mathit{depnames}(R)\}$ Obviously, $\pi_R(I) = \pi_R(J)$.

It remains to show that J is an installation.

Since $J \subseteq \omega_R(I)$ and since $\omega_R(I)$ is in peace, J is in peace, too. Let $p \in J$. If $p \in R$ then let $d \in p.D$, $(n,c) \in d$, and $q \in \omega_R(I)$ such that $q.n = n$ and $q.v \in [[c]]$. Hence, we have that $q.n \in \mathit{depnames}(R)$, and consequently that $q \in J$. This means that the dependencies of p are also satisfied in J. If $p \notin R$ then p is a blank package, and hence has no dependencies. □

Proving straight properties. We can now prove the fundamental result that straight properties can be verified by looking only at optimistic and conservative futures:

Theorem 4. *Let R be a repository, and ψ a* straight *property of repositories. Then ψ holds for all futures of R iff it holds for all optimistic and conservative futures of R.*

Proof. If ψ holds for all futures of R then it also holds in particular for all optimistic and conservative futures. The inverse direction follows immediately from Lemma 3. □

Note that the result would not hold without the restriction to R-focused properties. For instance, if $\phi(I)$ is the property "I does not contain package `foo`," where `foo` $\notin R.n$, then ϕ holds for all installations in conservative futures, but not in all futures. Also, the property "each package not originally in R has no conflicts" holds in installations of optimistic futures, but not in all futures.

5.3. Observational equivalence

We have shown that we can restrict our search, without loss of generality, to conservative futures of a repository R: conservative and optimistic futures alone *represent*, for our purpose, all possible futures. But there is still one important obstacle: the number of possible *future versions* to examine is potentially infinite. The key observation to solve this problem is that abundance and peace only depend on the satisfiability of version constraints—a boolean judgement—and not on the particular versions that make such valuations hold.

Definition 18. Let $n \in \mathrm{N}$ and $v, w \in \mathrm{V}$. The pairs (n, v) and (n, w) are R-observationally equivalent, noted $(n, v) \simeq (n, w)$, when for all packages $p \in R$ and all $(n, c) \in dep(p) \cup p.C$ we have that $v \in [[c]]$ iff $w \in [[c]]$.

Example 14. Going back to the repository R of Figure 4 we find that the pairs (`d`,6) and (`d`,7) are observationally equivalent since both 6 and 7 satisfy the constraint > 3, and both do not satisfy the constraint $= 5$. However, the pairs (`c`,1) and (`c`,2) are not observationally equivalent since 1 does not satisfy the constraint > 1, while 2 does.

This definition can be used directly to check equivalence of two versions v and w of a package p: just collect all constraints mentioning p in R, which form a finite set, and evaluate them on v and w; the two versions are equivalent iff the valuations agree. Since an equivalence class of versions of p is uniquely determined by the value of the finite set of constraints mentioning p, it is easy to prove the following.

Lemma 5. *For any repository R, and any package name n, there is a* finite *set of equivalence classes of versions of n in R.*

In our algorithms, we use representatives of these finite equivalence classes, that we call *discriminants of p*.

Definition 19. For any package p such that $p.n \in depnames(R) \cup R.n$ we call *discriminants of p* any set of representatives of the equivalence classes of the versions of *p that may appear in a future of R* with respect to $\simeq$.

Algorithm 1 Computing discriminants

```
function DISCRIMINANTS($S$ : package set, $R$ : repository)
    $C \leftarrow \{$all constraints in R mentioning a package name in S$\}$
    $V \leftarrow \{$all versions appearing in C$\}$
    $D \leftarrow \{\}$
    $E \leftarrow \{\}$
    for all $v \in interpolate(V)$ (in descending order) do
        $valuation \leftarrow map(fun\ c \rightarrow v \in [[c]], C)$
        if $valuation \notin E$ then
            $D \leftarrow D \cup \{v\}$
            $E \leftarrow E \cup \{valuation\}$
        end if
    end for
    return $D$
end function
```

Algorithm 1 describes the method used in our tools, and we show how it works on a simple example. Notice though that there are many ways of picking one representative out of each equivalence class, so our algorithm is not the only possible one.

Example 15 (discriminants). We start from the set of versions V of a package p with name n mentioned in the repository R. For instance, suppose we have version 3.1 of a package with name n, and there are dependencies on n with constraints $= 5.0$ and ≥ 9.1 in the repository. Then $V = \{3.1, 5.0, 9.1\}$.

As a first step, we add to V one intermediate version between any two successive versions in V, plus one version bigger than all and one version smaller than all: since the version space is dense, we can always do this, and the choice of the intermediate version is arbitrary. On our example, we can get a sequence $V' = \{2, 3.1, 4, 5.0, 7, 9.1, 10\}$.

Then we need to trim our sequence by removing versions that cannot appear in a future: if the package p is not present in R, there is nothing to do, but if p is present with a given version, as is our case, we need to remove all versions smaller or equal than the current one, 3.1, as a future can only contain upgrades. On our example, this leads to $V'' = \{4, 5.0, 7, 9.1, 10\}$. All these steps leading from V to V'' are performed by the function $interpolate(V)$ used in Algorithm 1.

Finally, we evaluate the constraints $= 5.0$ and ≥ 9.1 on each version in $interpolate(V)$, and discover that $(n, 4) \simeq (n, 7)$ as both 4 and 7 falsify $= 5.0$ and ≥ 9.1; we also find $(n, 9.1) \simeq (n, 10)$ as 9.1 and 10 both falsify $= 5.0$ and satisfy ≥ 9.1. This test is done by mapping over all constraints $c \in C$ a function that checks whether v satisfies c (that is $v \in [[c]]$). Hence, we end up with the discriminants $\{4, 5.0, 10\}$ (see also Table 1).

When performing upgrades of clusters of packages, to upgrade them in a synchronized way, we need to take discriminants not of a package, but of a set of packages S. The notion of discriminant extends naturally to sets of packages, and Algorithm 1 computes them by checking all possible valuations of the constraints mentioning a package in the set S and picking a representative version for each of them.

Version equivalence can be extended to repositories.

	4	5.0	7	9.1	10
$=5.0$	f	t	f	f	f
≥ 9.1	f	f	f	t	t

Table 1: Evolution of constraints on version numbers in Example 15. The versions 4 and 7 are observationally equivalent as their columns contain exactly the same truth values, as are 9.1 and 10.

Definition 20. Let F, G be optimistic and conservative futures of R. Then F and G are R-observational equivalent, noted $F \simeq G$, when for all $(n,v) \in F$ and $(n,w) \in G$ one has that $(n,v) \simeq (n,w)$.

The interest of this equivalence is that the properties we are interested in cannot distinguish equivalent repositories, so it is enough to check them only on a representative of each equivalence class.

Theorem 6. *Let $F,G \in futures(R)$, and ψ an R-safe predicate. If $F \simeq G$ then $\psi(F) = \psi(G)$.*

The upshot of these theoretical considerations for the quality assurance applications we are interested in is:

- **Challenging upgrades**. To find the packages challenged by future versions of a package p, it is enough to examine the optimistic and conservative futures of the repository obtained by replacing p with a blank package for p with one of the versions in the discriminants of p.
- **Outdated packages**. To check for outdated packages in a repository R, it is enough to examine for all packages $p \in R$ all optimistic and conservative futures of R, obtained by replacing some number of packages different from p by their futures. Furthermore, it is enough to consider for these futures of packages p' only versions that are discriminants of p' in R.

6. Applications

The theoretical results we have obtained lead to efficient algorithms to compute challenging upgrades and outdated packages, that we detail in this section.

6.1. Challenging upgrades

We can collect information about all the upgrades of a repository R in a single data structure, called a *prediction map*. Intuitively that is a function that associates to each cluster C of packages, and each *relevant* future version v of the packages it contains, the set of packages that will *surely* become not installable by upgrading the cluster C to version v, no matter the actual dependency and conflicts.

Algorithm 2 Computing prediction maps

```
$PM \leftarrow []$
for all $C \in Clusters(R)$, at version $cv$ do
    for all $v \in Discriminants(C,R)$ with $v > cv$ do
        $F = R[C \mapsto v]$
        for all $q \in R$ do
            if $\neg checkinstall(F,q)$ then
                $PM[(C,cv)] \leftarrow PM[(C,cv)] \cup \{q\}$
            end if
        end for
    end for
end for
return $PM$
```

Definition 21. The *prediction map* of a repository R is a function PM that maps every cluster $C \subseteq R$ of *synchronized* packages, at version cv, and each version $v \in Discriminants(C,R)$ with $v > cv$, to the set of all packages $Q \subseteq R \setminus C$ that are not installable in the future $F = R[C \mapsto v]$.

The prediction map for a repository R can be computed using Algorithm 2, which calls the function $checkinstall(F,q)$ that uses a SAT solver to return true if the package q is installable in the repository F or false otherwise.

Notice that the discriminants are computed taking into account all the packages in the cluster C, and that the future F is the result of upgrading all packages in the cluster C to the same discriminant version v.

The algorithm needs to check for all not installable packages in all futures F. That can lead to a worst case scenario of $nc \cdot nv \cdot np$ SAT solver calls, where nc is the number of clusters in the repository, nv the maximum number of discriminants of a cluster, and np the number of packages in the repository. Notice, however, that the size of the discriminants of a cluster is usually smaller than the sum of the size of the discriminants of the packages it contains, as many discriminants may be shared.

We have implemented the above algorithm to experiment on real world repositories (see Section 7).

Optimizing for singleton clusters. In the tools that we have built to implement our approach, there are several important optimizations that make their use feasible.

The first one uses the notions of *strong dependency* and *impact set* defined in [12] that we recall here.

Definition 22 (Strong dependency). Given a repository R, we say that a package p in R *strongly depends* on a package q, written $p \Rightarrow q$, if p is installable in R and every installation of R containing p also contains q.

Intuitively, p strongly depends on q with respect to R if it is not possible to install p without also installing q. Strong dependencies have been used [12] to propose the

Algorithm 3 Computing the prediction map: case of singleton clusters.

```
$PM \leftarrow []$
**for all** $(p,v) \in U$ **do**
    **for all** $v_i \in Discriminants(p)$ **do**
        **for all** $q \in IS(p)$ **do**
            $U[p,v \mapsto dummy(p,v_i)]$
            **if** $\neg check(U',q)$ **then**
                $PM[(p,v_i)] \leftarrow PM[(p,v_i)] \cup \{q\}$
            **end if**
        **end for**
    **end for**
**end forreturn** $PM$
```

impact set of p as a first approximation of the packages that may be affected by an upgrade of a package p.

Definition 23 (Impact set of a component). Given a repository R and a package p in R, the *impact set* of p in R is the set $Is(p) = \{q \in R \mid q \Rightarrow p\}$.

The impact set of p with respect to R is the set of packages of R that it is not possible to install without also installing p, so changing p *may* have an impact on all the packages contained in its impact set.

Using the algorithms presented in [12], it is possible to pre-compute the impact sets of a large distribution like Debian in just a few minutes on a standard laptop. This is quite useful, because of the following

Proposition 1. *Let q be a package in an R-installation for a repository R. Suppose that for a version $v \in \mathrm{V}$, $(n\ , = v) \in q.D$. Let $p \in R$ such that $(p.n, p.v) = (n, v)$. If $p \in SD(q)$, then for any future F such that $p \notin F$, we have that $q \notin I$ for any F-installations.*

Proof. Assume that there exists an installation I for a future F of R such that $q \in I$ and $p \notin F$. Since $q \in I$ then the installation of q must be abundant. In particular since $p \in SD(q)$, then p must be in I. But this is impossible since $I \subseteq F$ and $p \notin F$. □

This means that if q is installable in R, but breaks in an updated repository $R[P \mapsto v]$, then q really needs p with version $p.v$ to be installed in R, so q *strongly depends* on p and belongs to the *impact set of p*. So, for clusters containing only one package, we can use the modified Algorithm 3.

In practice this optimization is extremely effective since impact sets are small, can be computed in a few minutes, and many clusters contain only one package.

Further optimizations. In case that two versions v and w of a package p are present in the universe U, we know that every transition from v to w of p breaks all the impact set of (p,v), without computing anything.

Proposition 2. *Let (p,v), (p,w) and (q,z) in U. If (q,z) implies (p,v) then (q,z) is not installable when (p,w) replaces (p,v).*

PROOF. Assume that (p,w) replaces (p,v) and (q,z) is installable. Then the installation of (q,z) is abundant. In particular, since (q,z) implies (p,v), the dependency on (p,v) must be satisfied. This is impossible since (p,v) is no longer in U. □

We also remark that

Proposition 3. *A blank package is always installable*

PROOF. By construction a blank package has no dependencies nor conflicts. □

In particular, if a package in the impact set of p is a blank package, it is always installable.

And finally, we can skip packages that are not mentioned in any version constraint.

Proposition 4. *Given a package (p,v_i) in U if the list of the constraints of p is empty, the prediction map of (p,v_j) is empty for any j.*

On top of all these optimizations, we remark that the prediction algorithm is highly parallelizable: each future can be checked independently of the others, and indeed our implementation allows to take advantage of a multicore machine to significantly speed up the overall computation.

6.2. Outdated packages

As discussed in Section 2, it is not feasible to find outdated packages by checking, for every package $p \in R$, whether p is not installable in each conservative and optimistic future of R where packages other than p have been upgraded to a blank package, even when only considering the discriminant versions.

The key idea to an efficient algorithm is then that we can fold all the relevant futures together in one large set U of packages. In case all clusters have size 1 this is trivial to achieve:

$$U = \{p\} \cup \{(q.n, v, \emptyset, \emptyset \mid q \in R, q \neq p, v \in \mathit{Discriminants}(q)\}$$

The set of all U-installations is exactly the union of all F-installations over all conservative and optimistic futures of R. Hence, package p is installable in some future of R iff it is installable in U.

In case of non-trivial clusters this is not sufficient since the big universe U allows for installations that are not synchronized. This can be avoided by adding additional constraints to blank packages that forbid to install a package together with a package from the same cluster but a different version. This is done by first constructing U as above, and then applying the following *synchronization*:

$$\begin{array}{rcl} \mathit{synch}(U) & = & \{(\mathit{cluster}(p), p.v, \emptyset, \{(\mathit{cluster}(p), \neq p.v)\} \mid p \in U\} \\ & \cup & \{(p.n, p.v, p.D \cup \{\mathit{cluster}(p), = p.v\}, p.C) \mid p \in U\} \end{array}$$

Algorithm 4 Computing outdated packages

```
$OutDatedPackages \leftarrow []$
$U \leftarrow R$
for all $p \in R$ do
    for all $v \in Discriminants(q) \setminus p.v$ do
        $U \leftarrow U[p \mapsto v]$
    end for
end for
for all $p \in R$ do
    if $\neg checkinstall(synch(U), p)$ then
        $OutDatedPackages \leftarrow OutDatedPackages \cup \{p\}$
    end if
end for
return $OutDatedPackages$
```

Dummy packages for clusters are always installable since they have no dependencies, however, at most one version of a cluster package can be installed at a time due to its conflict with other versions. Furthermore, all packages p in R are amended by adding to p an additional dependency on its cluster package, with a version that is equal to the version of p. Since installation of any package with version v entails installation of the cluster package with version v, and since it is not possible to install two cluster packages with the same name and different version, a $synch(U)$-installation cannot contain two packages that are in the same cluster but have different version. This leads to our final Algorithm 4.

6.3. Implementing Clustering

Whether two packages stem from the same source can be easily detected since the source package name is usually part of package metadata. However, even if packages from the same source do advance together, we do not necessarily know what exactly the version numbers of future versions of packages will be, as the version numbers of packages may diverge from the version of the source package (e.g. due to distribution specific changes).

We hence make two hypotheses here: packages do not change their adherence to a source package (in reality, this is possible but very rare), and furthermore we assume that if we can detect a similarity between the current version numbers of two packages stemming from the same source, then the same similarity will hold for future updates.

7. Experimental evaluation

We have implemented the algorithms presented in the previous section in a suite of tools that find challenging upgrades and outdated packages in component repositories. We have performed an extensive experimental evaluation of the tools on the Debian distribution, one of the largest known software repositories, and on the Opam repository, a component repository for the OCaml programming language. All our experiments have been performed on a commodity 2.4GHz quad-core desktop computer running Debian.

7.1. Debian

In Debian, packages built from the same source are upgraded together due to the fact that the process of uploading new (versions of) packages to the repository is based on uploading (or recompiling) source packages. Clusters stemming from source packages can be quite large: the cluster corresponding to `gcc-4.3` contains 44 different packages including `libgcc1` and `gcc-4.3-base`.

Finding challenging upgrades on the entire development version of the Debian distribution takes about 20 minutes, with a parallel version of Algorithm 2 that uses the Parmap library [13] to analyze different chunks of the repository independently. Finding outdated packages on the same repository takes about 30 seconds, so we did not feel the need to parallelize the algorithm in that case.

Since we aimed at seeing our tools incorporated in the routine quality assurance processes of the Debian distribution, we had to make them robust by taking into account practical details such as the fact that packages built from the same source may adopt slightly different versioning schemes. In particular, we had to implement heuristics to detect version synchronization in spite of versioning scheme differences. As opposed to the rest of this work, these heuristics are quite specific to Debian, therefore we do not discuss them here any further, but their implementation can be found in our tools.

7.1.1. Challenging upgrades

We run our algorithm to detect challenging upgrades on the 5.0 “Lenny” Debian release that contains 22.314 packages. Table 2 shows the top 25 challenging cluster upgrades sorted by the number of packages that would become not installable due to the upgrade.

Discussion. These results reveal some interesting facts. Not all cluster upgrades break the same number of components: for example, upgrading the cluster `python-defaults` to a version strictly greater than `2.6` and strictly smaller than `3` will break 1.075 packages, moving to version `3` or later breaks 1.079 packages. There are cases where problems will arise only after a while: for `python-defaults` upgrades are dangerous only starting with version `2.6`, and `ghc6` will be problematic only when upgrading to versions higher than `6.8.2dfsg1-999`. These critical version changes roughly correspond to changes in the ABI of the Python interpreter.

It may seem surprising that complex cluster structures as `gnome` or `kde` do not appear in the table. In practice these systems are often assembled from highly interdependent components that evolve together and that are connected by a tight dependency structure: upgrading all packages of these clusters together will not break the in-cluster dependency structure and therefore will not cause problems.

The results of our analysis can be checked independently by changing in a Debian `Packages` package archive index the version of all packages of a cluster to a version in the target version column, and then running the `dose-distcheck` [14] tool on the resulting repository to find the not installable packages.

Table 2: Top 25 challenging upgrades in Debian (clustered)

Source	**Version**	**Target version**	**Breaks**
python-defaults	2.5.2-3	$\geq$ 3	1079
python-defaults	2.5.2-3	2.6 $\leq . <$ 3	1075
e2fsprogs	1.41.3-1	any	139
ghc6	6.8.2dfsg1-1	$\geq$ 6.8.2+	136
libio-compress-base-perl	2.012-1	$\geq$ 2.012.	80
libcompress-raw-zlib-perl	2.012-1	$\geq$ 2.012.	80
libio-compress-zlib-perl	2.012-1	$\geq$ 2.012.	79
icedove	2.0.0.19-1	$>$ 2.1-0	78
iceweasel	3.0.6-1	$>$ 3.1	70
haskell-mtl	1.1.0.0-2	$\geq$ 1.1.0.0+	48
sip4-qt3	4.7.6-1	$>$ 4.8	47
ghc6	6.8.2dfsg1-1	6.8.2dfsg1+ $\leq . <$ 6.8.2+	36
haskell-parsec	2.1.0.0-2	$\geq$ 2.1.0.0+	29
haskell-parsec	2.1.0.0-2	2.1.0.0-2 $< . <$ 2.1.0.0+	26
sip4-qt3	4.7.6-1	4.8	25
haskell-network	2.1.0.0-2	$\geq$ 2.1.0.0+	22
ghc6	6.8.2dfsg1-1	$<$ 6.8.2dfsg1-1	22
pidgin	2.4.3-4	$\geq$ 3.0	20
ghc6	6.8.2dfsg1-1	6.8.2dfsg1-999 $\leq . <$ 6.8.2dfsg1+	20
pcre3	7.6-2.1	$\geq$ 7.8	17
haskell-regex-base	0.93.1-3	$\geq$ 0.93.1+	16
haskell-regex-posix	0.93.1-1	$\geq$ 0.93.1+	14
haskell-time	1.1.2.0-2	$\geq$ 1.1.2.0+	13
haskell-quickcheck	1.1.0.0-2	$\geq$ 1.1.0.0+	13
haskell-hsql	1.7-2	$\geq$ 1.7+	13

7.1.2. Outdated packages

Finding outdated packages requires both automated tools and critical review of their results from distribution editors. We started by narrowing down a first list by applying to the Debian archive our implementation of the outdated packages algorithm. On October 6, 2011, the archive contained 34.444 packages in the unstable/main branch for the i386 architecture. The `dose-distcheck` tool reported 386 of these as not installable in the current repository, 110 of which our tool found to be outdated.

Discussion. It turns out that a Python transition was going on at the time, which means that many packages were being upgraded from a dependency on Python version 2.6 to Python version 2.7[3]. Since transitions like this are closely monitored by the Debian release team we choose to exclude these packages from our analysis. To do this, we added to the repository a dummy `python` package with version `2.6` to make packages depending on that version of Python installable. Our tool reported only 42 outdated packages, 22 of which stem from the source package `kdebindings`. Contacting the

[3] `http://release.debian.org/transitions/html/python2.7.html`, retrieved November 2012

`kdebindings` package maintainer we discovered that the package was being reorganized, and that many transient breakages were to be expected. We hence removed (by hand) all reports about concerned packages.

Of the remaining 20 outdated packages, 17 were outdated due to a versioned dependency on some package that had been upgraded, like this:

```
Package: nitpic
Version: 0.1-12
Depends: binutils (< 2.21.53.20110923)

Package: binutils
Version: 2.21.90.20111004-1
```

This means that the `nitpic` package has to be migrated to the newer version of `binutils`. We filed bugs against these packages, resp. asked for recompilation in case the offending dependency was filled in during the package compilation process.

The remaining 3 reports of outdated packages were due to the same mistake in the `cyrus-imapd-2.4` source package. This source package generates, among others, some transitional packages that only exist to facilitate the transition between different packages. Transitional packages have to use the right combination of inter-package relationships, which are quite easy to get wrong. This is what happened in this case:

```
Package: cyrus-common-2.2
Version: 2.4.12-1
Depends: cyrus-common-2.4

Package: cyrus-common-2.4
Version: 2.4.12-1
Conflicts: cyrus-common-2.2
```

Once reported by us, this bug was then promptly acknowledged and fixed[4] by changing the conflict to `cyrus-common-2.2 (< 2.4)`. Bugs filed as consequence of our analysis are available for public inspection[5] and seem to have provided useful feedback to the respective maintainers.

7.1.3. A release cycle perspective

We have argued that the theoretical framework to analyze repository future can be used to verify quality assurance properties and proposed outdated packages as one such property. In particular, outdated packages are a special subset of the not installable packages present in a repository, because they require manual intervention to become

[4] `http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=644503`, retrieved November 2012

[5] `http://bugs.debian.org/cgi-bin/pkgreport.cgi?tag=edos-outdated;users=treinen@debian.org`, retrieved November 2012

installable again. Given that computing outdated packages with our tools is quite inexpensive (∼30 seconds on a repository with more than 30.000 packages), we can track the number of outdated packages of a specific repository over time. We have done so for the complete duration of the development cycle of Debian "Wheezy," which started in February 2011 and continues to date (November 2012), aiming at releasing in the first quarter of 2013. The working hypothesis is that, *if* the number of outdated packages is a sensible quality metric for a package repository, then we should observe some kind of correlation with the various phases and relevant events of the Wheezy development cycle. We have proceeded as follows.

We have taken biweekly snapshots of the repositories used to develop Wheezy, i.e. the Debian "unstable" repository where new uploads by developers arrive, and the Wheezy repository itself (also known as "testing") where packages migrate after a qualification period of usually 10 days, as long as no serious bugs have been found in them via user testing. Retrieving repository snapshots can be done retrospectively, by downloading package metadata from `http://snapshot.debian.org`, using its service API.[6] For each sample we have noted down the total number of packages in the repository, calculated the set of not installable packages using `dose-distcheck`, and further extracted the set of outdated packages using the same tool of Section 7.1.2.

A sampling period of 15 days, we found, gives enough samples to visualize trends but not too many to make outdated analysis unwieldy: we took a total of 80 repository snapshots (2 sample per month over 20 months of 2 repositories), and analyzed them all. The results are shown in Figure 7.

Discussion. At first glance one can observe that, at least in this release cycle, there does not appear to be a significant correlation between the number of packages (which is steadily growing over time) and the number of not installable or outdated packages. Similarly, there does not seem to be correlation between the two quantities either: we can observe that the evolution of outdated packages is much smoother than the evolution of the non installable packages. This can be explained by the fact that, due to nature of this property, once a package is outdated it remains outdated regardless of how the other packages evolve, which make this property insensitive to day-to-day fluctuations of the repository.

Looking at both measures, we see a common pattern of software life cycles. The beginning of the cycle is pretty good, as it shows a low number of not installable, as well as outdated packages. The central part of the development cycle is more hectic, with an higher average number of not installable packages and several peaks of outdated packages like the one in October 2011 that we discussed in Section 7.1.2. On June 30th, Wheezy has been declared frozen[7], meaning that at that moment the automatic flow of packages from unstable to testing has turned into a semi-automatic process, vetted by explicit "freeze exceptions". It also means that at that point the work on stabilizing the repository for release, rather than including new software and features, has started. We

[6] `http://anonscm.debian.org/gitweb/?p=mirror/snapshot.debian.org.git;a=blob;f=API;hb=HEAD`, retrieved November 2012

[7] `https://lists.debian.org/debian-devel-announce/2012/06/msg00009.html`

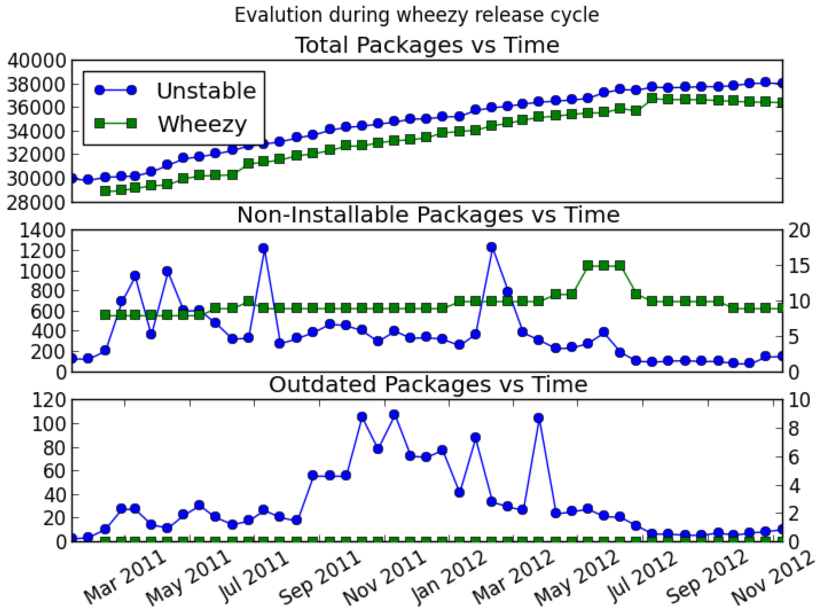

Figure 7: Total (at the top), not installable (middle), and outdated (bottom) packages during the release cycle of Debian "Wheezy", up to date. For not installable and outdated packages, unstable and Wheezy are shown using different scales on the y axis: scale for unstable is shown on the left of the graph, scale for Wheezy on the right.

can observe significant trend changes in all plotted metrics. Finally, towards the end of the cycle—which we are approaching, according to the average 2 years release cycle of Debian over the past 7 years—the number of not installable packages approaches zero.

It is worth noting that the number of not installable packages in Wheezy is always very low, and the number of outdated packages always zero. That seems to validate the testing/unstable split as a good approach to shield testing users from transient phases where a high number of packages are not installable.

Finally, it is interesting to look at the reasons behind the various peaks in the number of outdated packages present in the unstable repository. We have manually inspected the 4 highest peaks of Figure 7:

date	*outdated*
October 10, 2011	106
November 10, 2011	108
January 25, 2012	88
March 25, 2012	105

The first snapshot is just a few days apart from the case we have discussed in Section 7.1.2 and the reason for most outdated packages is the same: an ongoing transition of the Python interpreter from version 2.6 to version 2.7. The subsequent snapshot still shows a high number of Python packages (about 60) that are still outdated due to unsatisfiable dependencies on a Python version strictly less than 2.7. This confirms well-known folklore in the Debian community that large transitions, such as the Python 2.7 one that involved 160 packages, can take several weeks, sometime months, to complete. The small fluctuations in outdated packages between October and November 2011 seem to be done by other, smaller, ongoing transitions such as the first part of a large transition related to the Mono interpreter (completed shortly after the first snapshot) and one related to the Samba platform (started shortly before the second snapshot).

The last two snapshots confirm that the main contributions to the number of outdated packages stem from ongoing transitions from the testing to the unstable repository. In the third snapshot (January 25, 2012) 47 outdated packages out of 88 are part of a transition from version 2.10.5 to version 2.10.8 of the Mono bytecode interpreter. In the fourth snapshot (March 25, 2012) 81 outdated packages out of 104 are involved in a Iceowl/Icedove products of the Mozilla suite. The reader interested in inspecting the actual metadata used by package maintainers to enforce transition constraints is referred to our full data set (see Section 9 for data availability).

7.2. Opam

Opam[8] is a source-based oriented package manager which aims to facilitate the deployment and installation of software written in OCaml [15]. The main difference between Opam and binary-based package manager, like those found in Debian and other FOSS distributions, lays in the nature of the packages being installed. In Debian, there is a clear distinction between source and binary packages, where binary packages are *built* from source packages and only the latter are effectively distributed via repositories and installed on the users' machines. On the contrary, an Opam repository only contains source packages. The task of the Opam package manager is to download and compile the source package and subsequently install the binaries obtained, on the user machine, at the end of the compilation process.

In this context, Opam dependencies and conflicts are effectively compile-time relationships instead of run-time relationships. Nonetheless it is possible to use our tools to identify outdated and challenging source packages.

Opam repositories are several orders of magnitude smaller then Debian repositories. As of May 2013, the official Opam repository contains 850 packages, but only 390 in their most up-to-date version, out of which 42 packages are not installable. In contrast to Debian repositories, Opam repositories contain all past versions of all packages in one repository. Users can choose to install a specific version of a library with the package manager taking care of re-compiling all related components. Table 3 shows the top 25 challenging packages.

Basic libraries like `ocamlnet` or `type_conv` break a large number of packages. In particular, older versions of `type_conv` which is part of a larger software collection, are

[8] http://opam.ocamlpro.com/, retrieved June 2013

Table 3: Top 25 challenging upgrades in Opam

Source	**Version**	**Target version**	**Breaks**
ocamlnet	3.6.0	3.6.0 < .	26
type_conv	108.00.02	.	23
type_conv	108.08.00	108.08.00 < . ≤ 109.20.00	20
type_conv	108.07.00	108.08.00 ≤ .	14
type_conv	108.07.01	108.08.00 ≤ .	14
bin_prot	108.08.00	108.08.00 < .	12
comparelib	108.08.00	109.14.00 ≤ .	12
sexplib	108.08.00	109.14.00 ≤ .	12
pipebang	108.08.00	108.08.00 < .	12
variantslib	108.08.00	108.08.00 < .	12
pa_ounit	108.08.00	109.14.00 < .	12
fieldslib	108.08.00	109.14.00 < .	12
ocamlgraph	1.8.2	1.8.2 < .	11
ospec	0.3.0	0.3.0 < .	9
core	108.08.00	108.08.00 < .	8
oasis	0.3.0	0.3.0 < .	8
pa_ounit	108.07.00	.	6
comparelib	108.07.01	109.14.00 < .	6
comparelib	108.07.00	109.14.00 < .	6
fieldslib	108.07.00	109.14.00 < .	6
bin_prot	108.07.00	108.08.00 < .	6
fieldslib	108.07.01	109.14.00 < .	6
sexplib	108.07.00	109.14.00 ≤ . < 109.15.00	6
variantslib	108.07.01	108.08.00 ≤ .	6
pipebang	108.07.01	108.08.00 < .	6
pipebang	108.07.00	108.08.00 < .	6
variantslib	108.07.00	108.08.00 ≤ .	6

not compatible any more with a large number of components and should be removed from the repository.

For outdated packages the analysis shows only one outdated package (more precisely two versions of the same package), namely `ocp-build` which is made outdated by a strict dependency on the package `typerex`. Since outdated packages are a subset of the not installable packages, this information is still important to pin-point two packages (out of 42 not installable packages) that need manual intervention or that should be removed from the archive.

Our analysis shows that despite the fact that the Opam repository contains only a limited number of packages and that the dependency structure is slowly evolving, we can already identify those problematic packages and allow package maintainers to either fix the meta-data of their packages or remove older version from the repositories tout-court.

8. Related work

Brun et al. [16] have proposed the idea of applying *speculative analysis* to software artifacts in order to guide software engineering decisions. They try to predict changes to the source code in integrated development environments to guide programmers towards the most likely solution to compilation errors. The approach presented in this paper can also be seen as a form of speculative analysis, with the remarkable difference that we are able to consider all possible future repository states, rather than only some of them.

This work complements previous studies in the area of component quality assurance. In [10], some of the authors have given a formalization of the installation problem, various complexity results, and an efficient algorithm to check for installability. In [12], the notions of *strong dependency* and *impact set* have been proposed as a way to measure the relevance of a package with respect to other components in the distribution. The framework presented in this paper is based on a formal model similar to those used in these studies. The main difference is in the explicit syntactic treatment of constraints, which is needed to formalize futures.

Of the two applications of the framework we propose, the search for challenging packages has been introduced before in [17]. It is here reinstated in the general formal framework that allows to prove it correct.

In the area of quality assurance for large software projects, many authors have correlated component dependencies with past failure rates, in order to predict future failures [18, 19, 20]. The underlying hypothesis is that software "fault-proneness" of a component is correlated to changes in components that are tightly related to it. In particular, if a component A has many dependencies on a component B and the latter changes a lot between versions, one would expect that errors propagate through the "component network" reducing the reliability of A.

A related interesting statistical model to predict failures over time is the *weighted time damp model* that correlates recent changes to software fault-proneness [21]. Social network methods [22] have also been used to validate and predict the list of *sensitive* components in the Windows platform [20].

It would be interesting to enrich our prediction model by correlating package dependencies, which are formally declared and can be assumed trustworthy, with upgrade failures, but this is not yet possible, as current FOSS distributions still lack the data to correlate upgrade failures with dependencies.

9. Conclusions

Studying the future of component repositories can reveal important facts about their present. In particular, it can be used as a technique to pinpoint installability problems—such as outdated packages—that need to be manually addressed by component maintainers to watch out for challenging upgrade paths that, if followed, would break a significant amount of existing components.

In order to be effective, though, analyses about component evolution need to take into account all possible repository evolutions, which are infinite in all non-trivial repository models. This work presents a formal model of component repositories and identifies the class of future-looking properties that can be established by considering only a finite number of repository evolutions. We have applied the formal framework to two specific quality assurance applications: finding challenging upgrades and outdated packages. We have validated the framework by implementing quality assurance tools that have been run on the Debian distribution.

Our results show that investigating the future of component repositories is not only feasible but may also be beneficial. It can be used as a sound formal basis for tools that help repository maintainers spotting defects in their software *assemblies*, a welcome help when the sizes of those assemblies ramp up to hundreds of thousands components.

As future work, our approximation of the possible future evolutions of a repository could be improved, by identifying inter-package relationships that are likely to persist across package upgrades. Such a refined model could yield to a more precise approximation than blank packages, which are stripped of all inter-package relationships. Also, we have for now only scratched the surface of the kind of analyses that future-looking properties can be used for. We plan to extend our tooling to better support long-running analyses, such as the release cycle view we have presented of Debian Wheezy, in an attempt to provide better guidance to QA teams, release teams, and more generally decision makers in the area of software life cycle management.

Data availability. All experimental data discussed in this paper are available, in greater detail, from `http://data.mancoosi.org/papers/scp2012/`. The tools used to conduct the experiments are released under free software licenses and are available from `http://mancoosi.org/software/`.

References

[1] P. Abate, R. Di Cosmo, R. Treinen, S. Zacchiroli, Learning from the future of component repositories, in: CBSE 2012: 15th International ACM SIGSOFT Symposium on Component Based Software Engineering, ACM, 2012, pp. 51–60. doi:10.1145/2304736.2304747.

[2] M. M. Lehman, L. A. Belady (Eds.), Program evolution: processes of software change, Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[3] H. H. Kagdi, M. L. Collard, J. I. Maletic, A survey and taxonomy of approaches for mining software repositories in the context of software evolution, Journal of Software Maintenance 19 (2) (2007) 77–131.

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

[5] J. Whitehead, T. Zimmermann (Eds.), MSR 2010: 7th International Working Conference on Mining Software Repositories, IEEE, 2010.

[6] J. Gonzalez-Barahona, G. Robles, M. Michlmayr, J. Amor, D. German, Macro-level software evolution: a case study of a large software compilation, Empirical Software Engineering 14 (3) (2009) 262–285.

[7] K.-K. Lau, Z. Wang, Software component models, IEEE Trans. Software Eng. 33 (10) (2007) 709–724.

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

[9] I. Jackson, C. Schwarz, Debian policy manual, http://www.debian.org/doc/debian-policy/, retrieved November 2012 (2012).

[10] F. Mancinelli, J. Boender, R. Di Cosmo, J. Vouillon, B. Durak, X. Leroy, R. Treinen, Managing the complexity of large free and open source package-based software distributions., in: ASE 2006: 21st IEEE/ACM International Conference on Automated Software Engineering, 2006, pp. 199–208.

[11] R. Treinen, S. Zacchiroli, Solving package dependencies: from EDOS to Mancoosi, in: DebConf 8: proceedings of the 9th conference of the Debian project, 2008.

[12] P. Abate, J. Boender, R. Di Cosmo, S. Zacchiroli, Strong dependencies between software components, in: International Symposium on Empirical Sofware Engineering and Measurement, IEEE, 2009, pp. 89–99.

[13] M. Danelutto, R. Di Cosmo, A ”minimal disruption” skeleton experiment: Seamless map & reduce embedding in ocaml, Procedia CS 9 (2012) 1837–1846.

[14] P. Abate, R. Treinen, The dose-debcheck primer, https://gforge.inria.fr/docman/view.php/4395/8241/debcheck-primer.html, retrieved November 2012 (October 2012).

[15] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, J. Vouillon, The OCaml system release 4.00, Documentation and user’s manual, Projet Gallium, INRIA (Jul. 2012).

[16] Y. Brun, R. Holmes, M. D. Ernst, D. Notkin, Speculative analysis: exploring future development states of software, in: FoSER Workshop on Future of Software Engineering Research at FSE 2010, ACM, 2010, pp. 59–64.

[17] P. Abate, R. Di Cosmo, Predicting upgrade failures using dependency analysis, in: HotSWup’11: 3rd ACM Workshop on Hot Topics in Software Upgrades, 2011.

[18] N. Nagappan, T. Ball, Using software dependencies and churn metrics to predict field failures: An empirical case study, in: ESEM 2007: International Symposium on Empirical Sofware Engineering and Measurement, 2007, pp. 364–373.

[19] S. Neuhaus, T. Zimmermann, C. Holler, A. Zeller, Predicting vulnerable software components, in: ACM Conference on Computer and Communications Security, 2007, pp. 529–540.

[20] T. Zimmermann, N. Nagappan, Predicting defects using network analysis on dependency graphs, in: ICSE'08: 30th International Conference on Software Engineering, ACM, 2008, pp. 531–540.

[21] T. L. Graves, A. F. Karr, J. S. Marron, H. Siy, Predicting fault incidence using software change history, IEEE Trans. Softw. Eng. 26 (7) (2000) 653–661.

[22] R. A. Hanneman, M. Riddle, Introduction to social network methods, University of California, Riverside, retrieved November 2012 (2005).
URL http://www.faculty.ucr.edu/~hanneman/nettext/