Learning from the Future of Component Repositories*

Pietro Abate
abate@pps.jussieu.fr

Roberto Di Cosmo
roberto@dicosmo.org

Ralf Treinen
treinen@pps.univ-paris-diderot.fr

Stefano Zacchiroli
zack@pps.univ-paris-diderot.fr

Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126, CNRS, F-75205 Paris, France

ABSTRACT

An important aspect of the quality assurance of large component repositories is the logical coherence of component metadata. We argue that it is possible to identify certain classes of such problems by checking relevant properties of the possible *future repositories* into which the *current* repository may evolve. In order to make a complete analysis of all possible futures effective however, one needs a way to construct a finite set of representatives of this infinite set of potential futures. We define a class of properties for which this can be done.

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of "forced upgrades" induced by introducing new versions of existing components in a repository, and (ii) identifying outdated components that need to be upgraded in order to ever be installable in the future. For both applications we provide experience reports obtained on the Debian distribution.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*Software quality assurance (SQA)*; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*Software maintenance*

Keywords

component repository, quality assurance, speculative analysis

1. INTRODUCTION

As a consequence of the fact that software systems must undergo continuing evolution [10], 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 [8]. The advent of component-based software systems [14] 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 [16]. Partly because their components—called *packages* in this context—are freely available to study. Partly because some of them, such as the Debian distribution, are among the largest coordinated software collections in history [5]. Software packages share important features with *software component models* [9], but exhibit also some important differences. On one side, packages, like components, are 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 interrelationships between packages expressed in their metadata.

```
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)

Figure 1 shows an example of the metadata of a package of 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.) 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 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 some more interesting types of metadata, like virtual packages, 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.

*Work performed at IRILL `http://www.irill.org`, center for Free Software Research and Innovation in Paris, France.

CBSE'12, June 26–28, 2012, Bertinoro, Italy.

```
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`

An important feature of component architectures is that individual components may be upgraded independently of other components. In 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 ("unstable") of the Debian distribution with its more than 35,000 packages (as of February 2012) receives each single day about 150 upgrades.

Previous work [11] has focused on analyzing the metadata contained in a snapshot of a package repository. For instance, chasing uninstallable packages is a common quality assurance activity for distributions [15] and efficient tools to attend it exist, despite the NP-completeness of the underlying 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 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 make some packages uninstallable), we might want to prevent or delay such evolutions. We use future analysis to develop 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 uninstallable, no matter how p's metadata may have changed. The number of packages that are challenged by an upgrade to version v of p tells us how disruptive for the repository is the transition to the new v version of p.

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` non installable. We say that all upgrades to versions in the range $3.0 < \cdot < 5.0$ of package `bar` *challenge* package `foo` and `baz` as in any futures evolution of this repository containing the package `bar` with any version between 3.0 and 5.0 will make packages `foo` and `baz` uninstallable.

Outdated packages. If a quality problem is observed in the current repository (e.g. a package is uninstallable due to unsatisfiable dependencies), 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.

```
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

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.

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 these packages then `foo` will not be installable since its first dependency clause will not be satisfied. Upgrading `baz` alone will not work since `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. Our tool does this analysis automatically.

The problem in answering all sorts of questions about the future of 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 binary packages that stem from a single source package (and hence inherit its version), or synchronization requirements on component versions that are known to work well together.

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 it 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 to check a general class of admissible properties, it is enough to explore 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 Debian distribution are discussed in Section 7.

2. APPROACH

All investigations of the futures of a repository are faced with the problem of dealing with potentially infinitely many futures. In this section we illustrate our approach at hand of one specific example, that is the problem of checking whether package p in version n is outdated with respect to a current repository R. The ideas illustrated here apply to other properties of futures, as will be shown in the remainder. The approach consists of two steps:

1. First we will 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 universe, which then can be efficiently checked by an automatic tool. This second step, however, does not apply if we need a finer analysis telling us which particular kind of future poses which particular problem (as it is the case of finding challenging upgrades).

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.

Optimistic Futures.

The first insight is that when we advance a package q to a newer version then we may assume that this new version behaves as nicely as possible, that is it does not depend on any other packages and does not conflict with any packages. We call such a future *optimistic*. 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: 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 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.

Conservative futures.

Next, we cope with the problem that when moving to a future of a repository we may arbitrarily remove packages or introduce new packages. One sees easily that we may ignore package removals from R since every installation w.r.t. a future F of R in which we removed a package q is also an installation w.r.t. $F \cup \{q\}$. 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. 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). However, one can show that the only new packages that are essential for us 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. 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.

Observational equivalence.

So far we have seen that we may restrict our analysis to futures that are both optimistic and conservative. However, 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.

It is possible to compute an over-approximation of representatives of these equivalence classes by simply taking all version numbers that are mentioned in constraints on q (restricted to numbers that are greater than the version of q in R, if q already exists in R), plus one between any two succeeding values. For instance, if we currently have in R package q in version 5.1, and furthermore we have other packages declaring dependencies on $q > 5.3$ and $q \neq 6.0$, then $\{5.2, 5.3, 5.4, 6.0, 7.0\}$ is an over-approximation of a representative set of future versions of q. Some of these versions are redundant, though: for example 5.2 and 5.3 behave the same w.r.t. the constraints > 5.3 and $\neq 6.0$, as do 5.4 and 7.0. By dropping all redundant versions, we can get a minimal representative set for all the future versions of q, which is $\{5.2, 5.4, 6.0\}$.

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.

Folding many futures into one.

Computing outdated packages still requires to explore all the future 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 35,000.

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 package universe 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 universe 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.

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

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

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

Package: green
Version: 5.2
Source: color

Package: red
Version: 5.2
Source: color
```

Figure 4: {white, black} and {green, red} are clusters

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. We call a *cluster* a set of packages stemming from the same source and for which the current versions are similar. The problem now, when investigating installations w.r.t. the universe 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 universe U.

This approach is illustrated in Figure 4. First, package `white` is currently not installable since any installation of it would require `black`, which in its current version requires a version of `green` that does not yet exist. If we ignore source information then we would construct a universe U containing in addition to the original repository one future version per package without any conflicts and dependencies. In that case, `white` in version 2.1 becomes installable by installing `black` in its future version without dependencies. If we use source information, however, we find by inspecting source and current versions that `black` and `white` form one cluster, as do `red` and `green`. Due to the added conflicts in this case, 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.

3. PACKAGES AND REPOSITORIES

In this section we formalize a model of packages and repositories that is general enough to capture the relevant metadata present in the large majority of FOSS distributions. We will build on it in later sections to capture the notion of future repository evolutions.

Packages.

Let N be a set of names and V be an ordered set of version numbers. Mimicking the characteristics of existing packaging systems, we further 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 $v_1 > v_2$), and *dense* (i.e. for any two versions $v_1 < v_3$ there exists v_2 such that $v_1 < v_2 < v_3$).

A version *constraint* is a unary predicate on versions: for any $v \in \mathrm{V}$ we may write constraints $= v$, $\neq v$, $< v$, $> v$, $\leq v$, and $\geq v$. The set of constraints is denoted by CON. The *denotation* $[[c]]$ of a constraint $c \in \mathrm{CON}$ is the set of versions that satisfy the constraint, defined by

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

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 $D \in \mathscr{P}(\mathscr{P}(\mathrm{N} \times \mathrm{CON}))$,
- a set of conflicts $C \in \mathscr{P}(\mathrm{N} \times \mathrm{CON})$.

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.

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.

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 binary packages stemming from the same source package and (currently) having similar 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 $p.v \in [[c]]$.

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 $p \in I$.

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. 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 [11].

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

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.

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.

4. 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 $F \in futures(R)$, 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 $R \in futures(R)$, 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 change their cluster.

Our goal is to prove properties that hold for any installation w.r.t. any possible future. We now formally define two such properties that we have been using as motivating examples: outdated and challenging packages.

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 $R \in futures(R)$) and if it has to be upgraded to make it ever installable again. We discussed the interest of this property in Section 1, let us just observe that automatically finding outdated packages enables repository maintainers to spot packages that are in need of manual intervention.

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 its cluster to a new version v without touching other packages makes q not installable, no matter how the dependencies and conflicts of the packages in the cluster of p change. 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.

The outdated and challenging properties 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 *admissible* properties. To define such a class, we first need to introduce the set of properties that can be established by only looking at the packages whose names already exist in repository R:

Definition 9. A property ϕ of a package set is called *R-focused* if for all package sets P_1 and P_2 (not necessarily subsets of R)

$$\{(p.n, p.v) \mid p \in P_1, p.n \in R.n\} = \{(p.n, p.v) \mid p \in P_2, p.n \in R.n\} \text{ implies } \phi(P_1) = \phi(P_2)$$

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

Definition 10. Let R be a repository. A property ψ of futures of R is called *admissible* if there is an R-focused property ϕ s.t. for all futures F of R:

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

Trivially, admissible predicates are closed under boolean combinations.

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. $\forall F \in futures(R) : \big(\phi_1(F) \rightarrow \forall I \in Inst(F) : \phi_2(I)\big)$
2. $\forall F \in futures(R), \forall I \in Inst(F) : (\phi_1(I) \rightarrow \phi_2(I))$

(proof in Appendix)

In other words, we can under certain circumstances fold restrictions on futures into restrictions on installations. As a consequence, both the properties of a package p being outdated and a pair (p,v) challenging a package q are admissible predicates that we will write $\psi_{outdated}$, resp. $\psi_{challeng.}$. The interest of admissible properties is that we can actually *build* a finite set of futures of R such that the property holds for *all* futures of R iff it holds for all futures in such a set.

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 admissible properties can be answered by just looking at this set. We proceed by iteratively narrowing down the set of futures to consider, following the same schema as in Section 2.

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.

Definition 11. 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 give an over-approximation of the future where all constraints associated to a package disappear. We can now build an *optimistic* vision of the future.

Definition 12. 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 13. 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 a synchronization constraint. A simple kind of optimistic future can be obtained by replacing a package by a blank package with a newer version:

Definition 14. 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 in Appendix)

Conservative futures.

As a second step of our reduction we address the problem that new packages may be arbitrarily introduced in the future. As we will show, we may assume that this happens only for packages that are already *known*, i.e. packages whose names appear in a dependency of some package in the repository. It will also be established that package removals can be ignored because any package that is installable in a future repository is also installable in a future where no package has been removed.

Definition 15. $F \in \mathit{futures}(R)$ 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 F' of R and $I' \in \mathit{Inst}(F')$ such that $P(I) = P(I')$ for any R-focused property P.

(proof in Appendix)

Proving admissible properties.

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

THEOREM 1. *Let R be a repository, and ψ an* admissible *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.

Observational equivalence.

We have shown that we can restrict our search, without loss of generality, to conservative futures of a repository R which contain exactly the same package names as R. We also know that optimistic futures provide the best possible approximation. But there is still one important obstacle: the number of possible *future versions* to examine, which 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 16. Let $n \in \mathrm{N}$ and $v, v' \in \mathrm{V}$. The pairs (n, v) and (n, v') are R-observationally equivalent, noted $(n, v) \simeq (n, v')$, when for all packages $p \in R$ and all $(n, c) \in \mathit{dep}(p) \cup p.C$ we have that $v \in [[c]]$ iff $v' \in [[c]]$.

This definition can be used directly to check equivalence of two versions v and v' of a package p: just collect all constraints mentioning p in R, which are a finite set, and evaluate them on v and v'; 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 4. 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 17. For any package p such that $p.n \in \mathit{depnames}(R) \cup R.n$ 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 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.

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

Algorithm 1 Computing discriminants

```
function DISCRIMINANTS(S : package set, R : repository)
    C ← {all constraints in R mentioning a package name in S}
    V ← {all versions appearing in C}
    D ← {}
    E ← {}
    for all v ∈ interpolate(V) (in descending order) do
        valuation ← map(fun c → v ∈ [[c]], C)
        if valuation ∉ E then
            D ← D ∪ {v}
            E ← E ∪ {valuation}
        end if
    end for
    return D
end function
```

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 test 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 $v \in [[c]]$). So we end up with the discriminants $\{4, 5.0, 10\}$.

When performing upgrades of clusters of packages, to upgrade them in a synchronised 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.

Definition 18. 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 2. *Let $F, G \in futures(R)$, and ψ and admissible 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.

Algorithm 2 Computing prediction maps

```
PM ← []
for all C ∈ Clusters(R), at version cv do
    for all v ∈ Discriminants(C, R) with v > cv do
        F = R[C ↦ v]
        for all q ∈ R do
            if ¬checkinstall(F, q) then
                PM[(C, cv)] ← PM[(C, cv)] ∪ {q}
            end if
        end for
    end for
end for
return PM
```

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.

Challenging upgrades.

We can collect information about all the upgrades of a repository R in a single data structure, that we call a *prediction map*. This 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 be *surely* broken by upgrading the cluster C to version v, no matter the actual dependency and conflicts.

Definition 19. 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. 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 broken 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). In the actual implementation there is a further optimization that allows to check only a subset of R for broken packages in case a cluster contains only one package p. Indeed, if q is installable in R, but breaks in an updated repository $R[P \mapsto v]$, this means that 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 as defined in [1]. In practice this optimization is quite effective since impact sets are small, can be computed in a few minutes, and many clusters contain only one package.

Algorithm 3 Computing outdated packages

```
OutDatedPackages ← []
U ← R
for all p ∈ R do
    for all v ∈ Discriminants(q) \ p.v do
        U ← U[p ↦ v]
    end for
end for
for all p ∈ R do
    if ¬checkinstall(synch(U), p) then
        OutDatedPackages ← OutDatedPackages ∪ {p}
    end if
end for
return OutDatedPackages
```

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 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*:

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

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, an $\mathit{synch}(U)$-installation cannot contain two packages that are in the same cluster but have different version. This leads to our final Algorithm 3.

7. EXPERIMENTAL EVALUATION

We have implemented the algorithms presented in the previous section in a tool suite that finds challenging upgrades and outdated packages in component repositories, and we have performed an extensive experimental evaluation of our tools on the Debian distribution,[1] one of the largest known software repositories.

In the Debian repository, 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 can be large: the one corresponding to `gcc-4.3` contains 44 different packages including `libgcc1` and `gcc-4.3-base`.

All our experiments have been performed on a commodity 2.4GHz quad-core desktop computer running Debian. Finding challenging upgrades on a full Debian distribution takes 20 minutes, with a parallel version of Algorithm 2 that uses the Parmap library [4] to analyze different chunks of the repository independently. Finding all outdated packages takes less than 1 minute, and we did not need to parallelize the algorithm.

Since our tools will be incorporated in the quality assurance process 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 in fact adopt slightly different versioning schemes. In particular, we have implemented heuristics to perform version synchronization even in the presence of minor differences in the versioning scheme. These heuristics are specific to Debian and not of interest for this paper.

Challenging upgrades.

We run our algorithm on the 5.0 "Lenny" Debian release that contains 22,314 packages. Table 1 shows the top 25 challenging cluster upgrades sorted by the number of packages that would be broken by the upgrade.

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. 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.

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 inter-dependent 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 `edos-distcheck` tool on the resulting repository to find the broken packages.

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 `edos-distcheck` tool reported 386 of these as not installable in the current repository, 110 of which our tool found to be outdated.

It turns out that a Python transition was going on at the time,[2] which means that many packages were being upgraded from a dependency on Python version 2.6 to Python version 2.7. Since transitions like this are closely monitored by the Debian release team we choose to exclude these packages from our analysis. To do this,

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

[2] `http://release.debian.org/transitions/`

Table 1: Top 25 challenging upgrades (clustered)

Source	Version	Target Version	Breaks
python-defaults	2.5.2-3	≥ 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

Source	Version	Target Version	Breaks
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

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 `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          Package: binutils
Version: 0.1-12          Version: 2.21.90.20111004-1
Depends: binutils (< 2.21.53.20110923)
```

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

Once reported by us, this bug was then promptly acknowledged and fixed by changing the conflict to `cyrus-common-2.2 (< 2.4)`.[3] Bugs filed as consequence of our analysis are available for public inspection[4] and seem to have provided useful feedback to the respective maintainers.

8. RELATED WORK

Brun et al. [3] 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. Our approach can also be seen as a form of speculative analysis, with the remarkable fact 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 [11], some of the authors have given a formalization of the installation problem, various complexity results, and an efficient algorithm to check for installability; in [1], 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 [2]; 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 [12, 13, 17]. 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 [6]. Social network methods [7] have also been used to validate and predict the list of *sensitive* components in the Windows platform [17].

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 that need to be manually addressed by component maintainers—such as outdated packages—or 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 ap-

[3] `http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=644503`

[4] `http://bugs.debian.org/cgi-bin/pkgreport.cgi?tag=edos-outdated;users=treinen@debian.org`

plications: 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.

Data availability.

All data presented here are available, in greater detail, from `http://data.mancoosi.org/papers/cbse2012/`. The tools used to conduct the experiments are Free Software and available from `http://mancoosi.org/software/`.

10. REFERENCES

[1] P. Abate, J. Boender, R. Di Cosmo, and S. Zacchiroli. Strong dependencies between software components. In *International Symposium on Empirical Sofware Engineering and Measurement*, pages 89–99. IEEE, 2009.

[2] P. Abate and R. Di Cosmo. Predicting upgrade failures using dependency analysis. In *HotSWup'11*, 2011.

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

[4] M. Danelutto and R. Di Cosmo. A "minimal disruption" skeleton experiment: seamless map & reduce embedding in OCaml. *Procedia CS*, 2012. To appear.

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

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

[7] R. A. Hanneman and M. Riddle. *Introduction to social network methods*. University of California, Riverside, 2005.

[8] H. H. Kagdi, M. L. Collard, and 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):77–131, 2007.

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

[10] M. M. Lehman and L. A. Belady, editors. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[11] F. Mancinelli, J. Boender, R. Di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the complexity of large free and open source package-based software distributions. In *ASE*, pages 199–208, 2006.

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

[13] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In *ACM Conference on Computer and Communications Security*, pages 529–540, 2007.

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

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

[16] J. Whitehead and T. Zimmermann, editors. *7th International Working Conference on Mining Software Repositories, MSR 2010*. IEEE, 2010.

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

APPENDIX

Definition 20. 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.

Lemma 1.

PROOF. Observe that futures are closed under subset, that is if $I \subseteq F \in \mathit{futures}(R)$ then $I \in \mathit{futures}(R)$. We show that if (1) is false then (2) is false.

Assume that $F \in \mathit{futures}(R)$, $\phi_1(F)$, $I \in \mathit{installations}(F)$, and $\neg\phi_2(I)$. Since futures are closed under subset we also have that $I \in \mathit{futures}(R)$, $I \in \mathit{installations}(I)$, $\phi_1(I)$ (since ϕ is downward closed) and $\neg\phi_2(I)$.

Finally, we show that if (2) is false then (1) is false. Assume that $F \in \mathit{futures}(R)$, $I \in \mathit{installations}(F)$, $\phi_1(I)$ and $\neg\phi_2(I)$. Since futures are closed under subset we also have that $I \in \mathit{futures}(R)$, $\phi_1(I)$, $I \in \mathit{installations}(I)$, and $\neg\phi_2(I)$. □

For instance, we obtain that (p,v) challenges q if and only if $\forall F.\forall I \in \mathit{Inst}(F).\phi_{\mathit{challeng.}}(I)$ where

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

Lemma 2.

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 $p' \in I$ with $p.D \subseteq p'.D$ and $p.C \subseteq p'.C$: Since peace and abundance are satisfied for p' in I they are also satisfied for p in $\omega_R(I)$. □

Lemma 3.

PROOF. 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 construct

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

By construction, F' is an optimistic and conservative future of R. Let $I' = \{p \in \omega_R(I) \mid p.n \in R.n \cup \mathit{depnames}(R)\}$ Obviously, $\pi_R(I) = \pi_R(I')$. It remains to show that I' is an installation.

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