Science of Computer Programming 90 (2014) 93–115

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Learning from the future of component repositories [☆]

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

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

HIGHLIGHTS

- We consider quality issues that can arise in future evolutions of software repositories.
- We define a class of such issues that is amenable to automatic verification.
- Two relevant instances of such a class are identified: outdated and challenged components.
- We validate our findings on real-world repositories.

ARTICLE INFO

Article history:
Received 23 November 2012
Received in revised form 14 June 2013
Accepted 26 June 2013
Available online 23 July 2013

Keywords:
Component repository
Quality assurance
Software evolution
Speculative analysis
Open source

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of "forced upgrades" induced by introducing new versions of existing components in a repository, and (ii) identifying outdated components that are currently not installable and need to be upgraded in order to become installable again.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

☆ This paper is an extended version of [1], appeared in the 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.
* Corresponding author.
E-mail addresses: abate@pps.univ-paris-diderot.fr (P. Abate), roberto@dicosmo.org (R. Di Cosmo), treinen@pps.univ-paris-diderot.fr (R. Treinen), zack@pps.jussieu.fr (S. Zacchiroli).

0167-6423/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.scico.2013.06.007

```
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)
```

Fig. 1. Example of Debian meta-data (excerpt).

partly because some of them, such as the Debian distribution,[1] are among the largest coordinated software collections in history [6].

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.

Fig. 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, 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.

1.1. 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 algorithms 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.

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