Inter-component Dependency Issues in Software Ecosystems

Maelick Claes, Alexandre Decan, Tom Mens
COMPLEXYS Research Institute
University of Mons, Belgium

Abstract

Component-based software reuse has lead to the emergence of numerous open source software ecosystems. Such ecosystems offer the user a wide and diverse collection of software components that are interconnected by dependency relationships and maintained by large communities of developers. While developers can reuse the work of others by depending on their components, such dependencies give rise to many issues that impact maintenance effort if not properly addressed. This chapter discusses some commonly encountered dependency issues, and illustrates them through two case studies of popular open source package-based software ecosystems: Debian and R. For both of them, we present the limitations of current tool support for dependency management, and we provide results of empirical analyses that highlight how such tool support could be improved.

1 Introduction

Software engineering research has traditionally focused on studying the development and evolution processes of individual software projects. With the omnipresence of the Internet, an entire range of collaborative software development tools have become available and widely used, especially in the open source development scene. This has lead to bigger and more geographically distributed communities of developers, and made it possible to develop more complex software systems. It also gave rise to so-called *software ecosystems*, i.e., "collections of software products that have some given degree of symbiotic relationships" [31].

Analysing software projects from such an ecosystemic perspective can reveal new insights into why and how they evolve. Projects that are part of an ecosystem tend to be interdependent, and developers contributing to this ecosystem may be involved in multiple projects and share implicit or explicit knowledge across these projects. Hence, the evolution of a project may be affected to a certain degree by the changes in connected projects. This implies that project evolution should be studied in the context of its surrounding ecosystem. This view is shared by Lungu [28], who defined a software ecosystem as "a set of software projects that are developed and evolve together in the same environment".

One of the main reasons for dependencies between components in an ecosystem is software reuse, a basic principle of software engineering [37]. Software components often rely on (i.e., reuse) the functionality offered by other components (e.g., libraries), rather than reimplementing the same functionality. While this tends to reduce the effort from the point of view of a single component, it increases the overall complexity of the ecosystem, through the need to manage these dependencies. This complexity can become the cause of many maintainability issues and failures in component-based software ecosystems [9]. For this reason it is important to study dependency-related issues and to provide tools that would allow ecosystem maintainers to deal with these issues.

This chapter therefore discusses different types of maintenance issues related to component-based software ecosystems and how these issues impact maintainers and users of the ecosystem. We illustrate how these issues have been studied on two well-known *package-based* software ecosystems, Debian and R, both containing thousands of packages.

We show how analyzing these issues from the ecosystem point of view may help the ecosystem's maintainers to detect these issues better. This will allow them to decide more easily if and when the observed issues become problematic, and to take decisions to fix the issue or prevent it from reappearing in the future.

2 Problem overview

This section presents different types of issues related to inter-component dependencies that can happen during the development and evolution of components of a software ecosystem. We provide a common vocabulary of the inter-component dependency relationships we are interested in, discuss the possible problems caused by such interdependencies, and provide a summary of the state-of-the-art of proposed solutions.

2.1 Terminology

Several researchers have proposed general models to study intercomponent dependencies [15, 20, 29]. Based on these models, this chapter uses the following vocabulary to describe the different types of intercomponent relationships that are relevant.

Components act as the basic software unit that can be added, removed or upgraded in the software system. They provide the right level of granularity at which a user can manipulate available software. Components are typically organized in coherent collections called **distributions**, **repositories** or **archives**. The set of components of a distribution that are actually used by a particular user is called her **component status**. To modify the component status, for example by upgrading existing components or installing new ones, the user typically relies on a tool that is called the **component manager**. This manager uses **component metadata** in order to derive the context in which components may or may not be used. Exemples of such metadata are **component dependencies and**

conflicts. Component dependencies represent positive requirements (a component needs to be present for the proper functioning of another component), while component conflicts represent negative requirements (e.g., certain components or component versions cannot be used in combination). One of the most generic ways to express dependencies (though not supported by every component manager) is by means of a conjunction of disjunctions, allowing a choice of which component can satisfy a dependency.

Figure 1 provides a concrete example of how component dependencies and conflicts can be specified for packages in the Debian and R ecosystems, respectively. The Debian package xul-ext-adblock-plus depends on one of the three packages iceweasel, icedove or iceape. This is expressed by a disjunction (vertical bar |) of packages. The package conflicts with mozilla-firefox-adblock. The R package SciViews depends on version 2.6 of the R language as well as on packages stats, grDevices, graphics and MASS. The notion of conflicts and the ability to express disjunctions of dependencies are not explicitly supported by R package metadata.

```
Package: xul-ext-adblock-plus
Description: Advertisement blocking extension
             for web browsers
Source: adblock-plus
Version: 2.1-1+deb7u1
Replaces: adblock-plus (<< 1.1.1-2)
Provides: adblock-plus, iceape-adblock-plus,
    icedove-adblock-plus, iceweasel-adblock-plus
Depends: iceweasel (>= 8.0) | icedove (>= 8.0)
                            | iceape (>= 2.5)
Enhances: iceape, icedove, iceweasel
Conflicts: mozilla-firefox-adblock
```

```
Package: SciViews
Title: SciViews GUI API - Main package
Imports: ellipse
Depends: R (>= 2.6.0), stats,
         grDevices, graphics, MASS
Enhances: base
Version: 0.9-5
```

Figure 1: Two concrete examples of component metadata. Left: the Debian package xul-ext-adblock-plus. Right: the R package SciViews.

Some ecosystems allow components to depend on, or conflict with, an **abstract component**. In that case, the dependency (or conflict) is satisfied (or violated) by any component that **provides** features of that abstract component. For example, in Figure 1 the Debian package xul-ext-adblock-pls provides the features of the following abstract packages: adblock-plus, iceape-adblock-plus, icedove-adblock-plus, iceweasel-adblock-plus. Any dependency on adblock-plus would be satisfied if xul-ext-adblock-plus, or any other package providing adblock-plus, was installed.

Dependencies and conflicts can be restricted to specific versions of the target component. This is usually represented by a constraint on the version number. For example, in Figure 1 Debian package xul-ext-adblock-plus requires version 8.0 or higher of iceweasel. R package metadata does not support depending on specific package versions.

Figure 2 shows an example of a graph showing the aforementioned relationships. Components are visualised by ellipses and abstract components by diamonds. Edges represent component dependencies and dashed lines represent component conflicts. Constraints on the component version are depicted by edge labels. For example, abstract component v depends

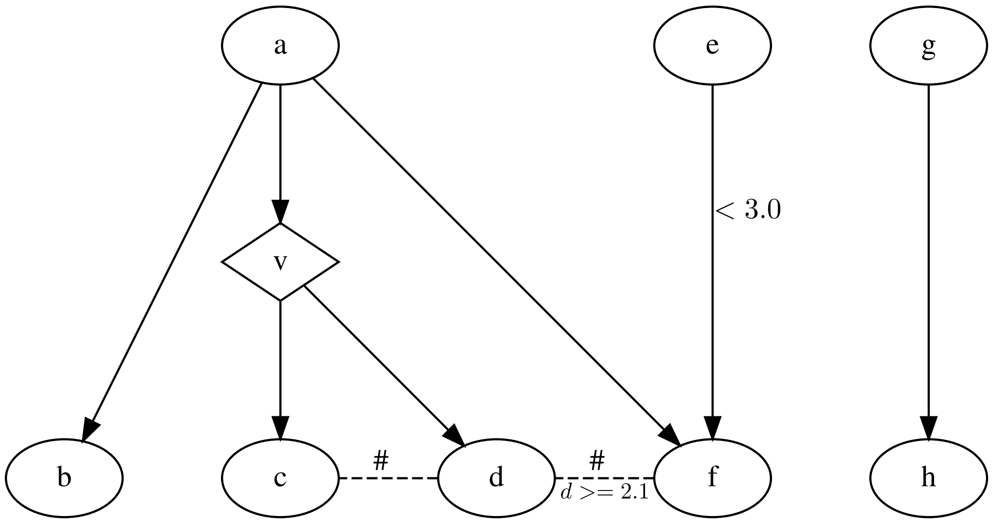

Figure 2: Example of a component dependency graph.

on two components c and d that are in mutual conflict. Component d is also in conflict with version 2.1 or superior of component f. Component e depends on a version lower than 3.0 of component f.

2.2 Identifying and retrieving dependency information

A particular type of software ecosystems where dependencies play a central role are *package-based* ecosystems. Such ecosystems generally consist of collections of software projects bundled in packages that need to rely on other packages in order to function correctly. Well known examples of such package-based ecosystems are Debian and R, both containing thousands of packages. LaBelle et al. [27] showed that the package dependency graphs for the open source Debian and FreeBSD distributions form a complex network with small-world and scale-free properties.

Extensive research has been conducted on the Debian package-based ecosystem [2, 3, 6, 14, 15, 17, 18, 40, 42] and the R package-based ecosystem [12, 22]. Other software ecosystems make use of components comparable to packages such as plug-ins (e.g., the Eclipse software development environment [43]), modules (e.g., the NetBeans software development environment), libraries [19], extensions and add-ons (e.g. the Firefox web browser), mobile app stores [7, 32].

In *package-based* ecosystems, each package is generally required to provide *metadata* specifying package dependencies. Two examples of this were given in Figure 1. Sometimes, however, the metadata can be incomplete or inconsistent, or even entirely lacking. In particular, constraints on dependency versions are often missing or inaccurate, because the component metadata is not always updated if the source code of components is being modified. In those cases, it may still be possible to retrieve the information using automated configuration tools such as make, cmake, autoconf, ant and maven.

Another way to retrieve the necessary metadata is through *static code analysis*. The source code of a software project usually contains the necessary information about which library or module is imported and which part of it is being used. A static analyser can use this information to obtain all dependencies across components at the ecosystem level. This solution does have its limits though, since there is no guarantee that dependencies discovered in the source code will actually be used at runtime. For this, dynamic code analysis would be required. This is particularly so for dynamically typed languages where it is much harder to derive the call dependencies statically.

In order to facilitate retrieval of component dependencies, Lungu et al. [29] proposed *Ecco*, a framework to generically represent dependencies between software projects. It models an ecosystem as a set of projects containing entities which are classes or methods. Entities can be of type **provided**, **called** or **required**. Lungu et al. [29] used *Ecco* to compare different strategies to extract dependencies statically from dynamically typed Smalltalk source code. While some methods are more efficient than others, none is able to successfully recover the list of all existing dependencies.

As explained by Abate et al. [3], a direct dependency graph obtained by identifying the list of components required by each component of the ecosystem, is not enough to characterize package interactions because those other components may have dependencies themselves. Because of this, they introduced the notion of **strong dependencies** of a component, which are the components that are always required, directly or indirectly, in order to successfully use the component depending on them. On top of this they introduced a measure of component **sensitivity** in order to determine, by means of the strong dependency graph, how much a change to a component may impact the ecosystem. In the context of Debian for example, they noticed that the most extreme cases of sensitive packages would go unnoticed when relying solely on direct dependencies. A sensitivity metric based on strong dependencies can be used by maintainers to decide which component should be or should not be upgraded or removed.

2.3 Satisfying dependencies and conflicts

Satisfying dependency constraints Once the dependencies of each component of an ecosystem have been identified, one needs to verify if they can be satisfied. The presence of dependency constraints can make some dependencies unsatisfiable. Being unable to satisfy dependencies will prevent a user from using a component, which would be highly undesirable. Based on the strong dependency graph, tools like `distcheck` have been developed to detect those components that cannot satisfy their dependencies. Such tools have been used successfully in different ecosystems such as Debian, OPAM and Drupal and have been shown to be useful to developers [1].

Satisfying component co-installability Conflicts may prevent a component to be used in a given context. If a component is in conflict with one of its strong dependencies, it will be unusable. When a conflict is declared (directly or indirectly) between two components, all components that strongly depend on both of them will not be able to work either. In a system where only one version of a component can be used at the same time, when two components need to depend (directly or indirectly) on two different versions of the same component, they will be unusable together. This problem is known for package-based ecosystems (and more particularly Debian) as the problem of **co-installability** [6, 17, 18, 40]. It can be generalized as the ability for two components to be used together.

We refer to **strong conflicts** as all components that are known to be always incompatible together. Just like the strong dependency graph can be used to satisfy dependency constraints, a strong conflict graph can be used to detect co-installability problems between components. Such a graph enables to identify the most problematic components of an ecosystem.

It is important to stress that components may be in strong conflict "by design": they cannot be installed together because they were never meant to work together. If this is the case, developers and users can be made aware of this impossibility by documenting such "known" conflicts explicitly in the component metadata. An example of this is shown in Figure 1, where package xul-ext-adblock-plus is declared to be in direct conflict with mozilla-firefox-adblock.

In addition to such known conflicts, new and unexpected indirect strong conflicts may arise during component evolution without the maintainers being aware of them. These conflicts require specific tool support to cope with them, as will be explained in Section 3.

2.4 Component upgrade

When developing software components, errors may be inadvertently introduced when changes occur in the software components one depends upon. When changes to a component cause the software to fail, it puts a heavy burden on the maintainers of the component that depend on this failing component. This is especially true in a large ecosystem where thousands of components are interdependent, and a single failure may affect a large fraction of the ecosystem.

This problem of component upgrades has been studied by many researchers. Nagappan et al. [34] effectively showed that software dependency metrics can be used to predict post-release bugs. Robbes et al. [35] studied the ripple effect of API method deprecation and revealed that API changes can have a large impact on the system and remain undetected for a long time after the initial change. Hora et al. [21] also studied the effects of API method deprecation and proposed to implement rules in static analysis tools to help developers adapt more quickly to a new API. McDonnell et al. [30] studied the evolution of APIs in the Android ecosystem. They found that, while more popular APIs have a fast release cycle, they tend to be less stable and require more time to get adopted. Bavota et al. [8] studied the evolution of dependencies between Apache software projects and found that developers were reluctant to upgrade the version of the software they depend upon. In [9] they highlighted that dependencies have an exponential growth and must be taken care of by developers.

All these studies indicate that component upgrade is often problematic and that contemporary tools provide insufficient support to cope with them. One of the solutions to detect errors during the development process is continuous integration [39]. However, while continuous integration can help to detect changes that break the system, it does not provide information on which components can be safely upgraded. Developers would benefit from recommender tools specifically designed to help them making such decisions.

In the context of *package-based* ecosystems, Di Cosmo et al. [15] highlighted peculiarities of package upgrades and discussed that current techniques are not sufficient to overcome failures. They proposed solutions to this problem [2, 14] and built a tool called comigrate to efficiently identify sets of components that can be upgraded without causing failures [42]. Similarly, Abate et al. [5] proposed a proof-of-concept package manager designed to allow the use of difference dependency solvers as plugins in order to better cope with component upgrade issues.

2.5 Inter-project cloning

One solution to avoid problems due to component dependencies would be to reuse code through copy-paste rather than depending on it. Indeed, some ecosystems consisting of distributed software for a specific platform do not allow components to depend one upon another. For example, the component manager for Android mobile apps only allows for apps to depend on the core Android platform, forcing app developers to include third-party libraries inside their own package. Mojica et al. [32] showed that this gives rise to very frequent code reuse across mobile apps.

Similarly, in ecosystems with inter-component dependencies, developers may decide to reimplement (part of) a component they need in order to avoid depending on it. In some cases, the effort needed to reimplement the component may be smaller than if developers have to fix errors caused by dependency changes. Especially for open source software, the development time can be significantly reduced by directly cloning the existing code as long as it does not violate software licenses.

In the context of a single software projects, the presence of software clones has been extensively studied and have shown to be beneficial or detrimental to software maintenance [23, 24, 25, 36]. While there have been recent studies on inter-project cloning [26, 38], insight on the causes and implications of inter-project software clones is still lacking. Although using cloning instead of a component manager to manage dependencies may help to avoid dependency upgrade problems from a user point of view, it forces each developer to choose which version of all their strong dependencies to include in their own component.

Additionally we previously studied functions that were duplicated between different *CRAN* packages [13] and showed that most clones could not have been avoided by relying upon dependencies. While there is still a non negligible amount of cloned functions that could be removed, further work is required to understand why these functions have been cloned.

3 First case study: Debian

This section presents some of the inter-component dependency issues raised in the previous section, for the concrete case of the Debian package-based ecosystem.

3.1 Overview of Debian

Debian is an open source package distribution of the GNU/Linux operating system. Debian aims at providing an operating system that is as stable as possible, and uses a software package management system with a strict policy to achieve these goals (see `www.debian.org/doc/debian-policy`). Having existed for more than two decades, Debian is one of the oldest Linux distributions that is still maintained today. It contains several tens of thousands of packages, and its community spans over a thousand distinct developers! The development process of Debian is organised around three main package distributions: stable, testing and unstable.

stable corresponds to the latest official production distribution, and only contains stable, well-tested packages. Table 1 summarises the characteristics of the different releases of the stable distributions.

Version	Name	Freeze date	Release date	# packages
3.1	sarge	N/A	2005-06-06	about 15K
4.0	etch	N/A	2007-04-08	about 18K
5.0	lenny	2008-07-27	2009-02-15	about 23K
6.0	squeeze	2010-08-06	2011-02-06	about 28K
7.0	wheezy	2012-06-30	2013-03-04	about 36K
8.0	jessie	2014-11-05	2015-04-26	about 43K

Table 1: stable releases of Debian since 2005.

testing contains package versions that should be considered for inclusion in the next stable Debian release. A stable release is made by *freezing* the testing release for a few months to fix bugs and to remove packages containing too many bugs.

unstable is a rolling release distribution containing packages that are not thoroughly tested and that may still suffer from stability and security problems. These releases contain the most recent packages but also the most unstable ones.

A major problem when analysing strong package conflicts is the sheer size of the package dependency graph: there are literally thousands of different packages with implicit or explicit dependencies to many other packages. Vouillon et al. [41] addressed this problem by proposing an algorithm and theoretical framework to compress such a dependency graph to a much smaller *co-installability kernel* with a simpler structure but equivalent co-installability properties. Packages are bundled together into an equivalence class if they do not have a strong conflict with one another, while the collection of other packages with which they have strong conflicts is the same.

As an example, the Debian i386 testing distribution on 1 January 2014 contained >38K packages, >181K dependencies, 1,490 declared conflicts and >49K strong conflicts. The co-installability kernel for the same data resulted in 994 equivalence classes and 4,336 incompatibilities between these equivalence classes. The coinst tool (coinst.irill.org) was developed specifically for extracting and visualizing such coinstallability kernels.

Based on this tool and related research advances on strong dependency and strong conflict analysis [2, 3, 6, 14, 15, 17, 18, 40], other tools have been created to determine appropriate solutions to package co-installation problems. For example, comigrate (coinst.irill.org/comigrate), coinst-upgrade, distcheck, and the dose tools for Debian Quality Assurance (qa.debian.org/dose/). These tools are actively being used by the Debian community. These solutions, however, do not take into account the evolution over time of strong conflicts.

In [10], we used coinst to study the evolution of strong conflicts on a period of 10 years for the Debian i386 testing and stable distributions. We aimed to determine to which extent this historical data provides additional information to understand and predict how strong conflicts evolve over time, and to improve support for addressing package co-installation problems.

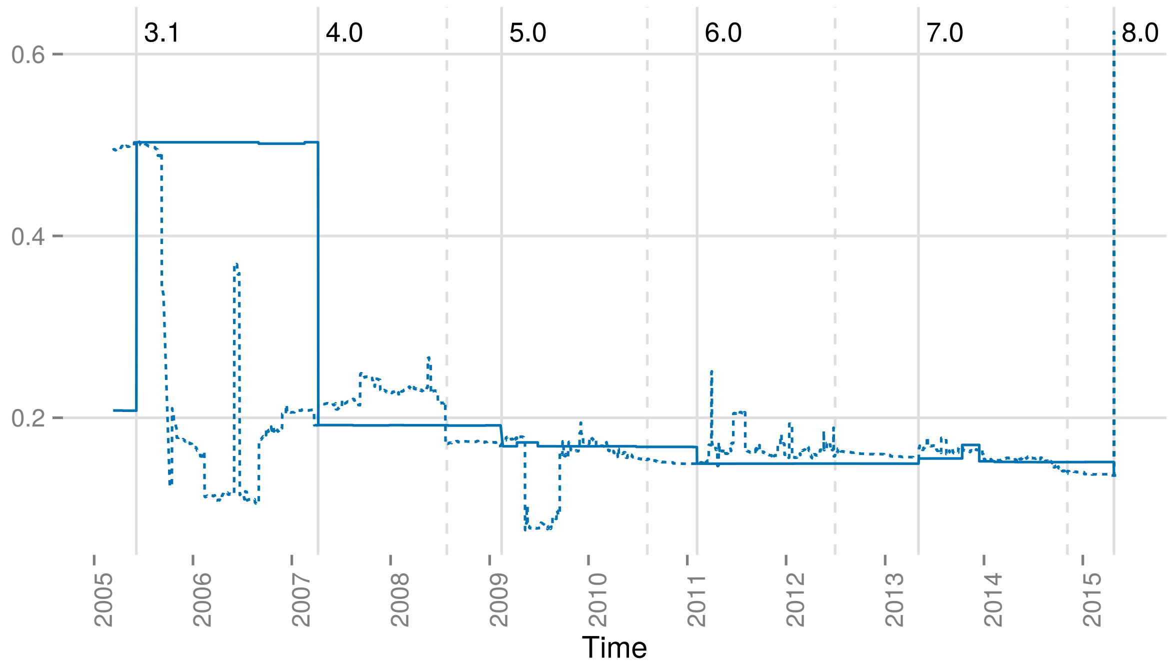

Figure 3: Ratio of strong conflicting packages in snapshots of Debian's testing distribution (dotted blue line) and stable distribution (solid blue line). The vertical lines correspond to the freeze date (dashed lines) and release date (straight lines) of each major stable release.

Figure 3 shows the evolution over time of the ratio of strong conflicting packages in a snapshot over all packages in that snapshot. We observe that starting from 2007 and with only a few exceptions, the ratio of the testing distribution remains between 15% and 25%.

We also observe a slight decrease over time, despite the fact that the number of packages increases with each new major release. This shows that the Debian community actively strives to keep strong conflicts at a minimum. The stable distribution follows a comparable evolutionary behaviour, combined with the presence of "plateaus" corresponding to different public releasess of Debian. Finally, the testing distribution reveals quite a number of "trend breaks", i.e., sudden increases in the number or ratio of strong conflicts that appear suddenly and disappear after some time.

3.2 Aggregate Analysis of Strong Conflicts

Some of the conflicts are present in the distribution "by design", but others may be harmful. Distinguishing the good from the bad ones is a complex task that has traditionally required a lot of manual investigation, with many issues going unnoticed for quite an extensive amount of time. A natural approach to identify potentially problematic packages is to look for trend breaks in the evolution of the absolute or relative number of strong conflicting packages in the distribution. Sudden increases hint that some problematic package(s) may have appeared, and sudden decreases indicate that some problematic package(s) have been fixed. Many such discontinuities are clearly visible in Figure 3, with peaks ranging from a few hundreds to over 4,000 strong conflicts.

Using the coinst-upgrade tool [40] that identifies the root causes for the changes in conflicts between two repositories, we retrieved all trend breaks that added at least 500 strong conflicts. We manually inspected each trend break, and checked it against the information available from the Debian project, to determine the nature of the problematic packages and the degree of seriousness of the problem, and paired the events where each problematic package was first introduced and then removed.

We observed that a few trend breaks were *day flies* that were fixed the day after their introduction, while several took a few weeks, three took hundreds of days to fix, two have been fixed in several phases, and two still remain unfixed today. Most of these issues would have been captured by the comigrate [42] tool if it would have been available at that time, and one issue could have been anticipated using the challenged [4] tool.

Interestingly, a few relevant trend breaks *are not identifiable by any of the existing tools*, while an inspection of the aggregate analysis (as presented here) would have drawn attention to them. This illustrates that there is a clear opportunity for improving current automated tool support.

3.3 Package-level Analysis of Strong Conflicts

Once a trend break has been spotted, one still needs to identify manually what are the packages in the snapshot that are the root causes of the trend break. Some of these problematic packages are shown in boldface in Table 2.

This process can be automated by studying the characteristics of each package related to strong conflicts by resorting to three simple metrics:

- the *minimum* number of strong conflicts
- the *maximum* number of strong conflicts
- the number of *conflicting days over mean*, i.e., the number of days the package has more strong conflicts than $\frac{\text{maximum}+\text{minimum}}{2}$

These metrics allow one to focus on packages with a significant amount of strong conflicts, while at the same time ignoring those packages that have such a large number of conflicts only for a short period of time. The latter case usually corresponds to transient problems, like the *day flies* that we were able to identify in the previous aggregate analysis.

Potentially problematic package	minimum conflicts	maximum conflicts	conflicting days over mean
libgdk-pixbuf2.0-0	0	675	1349
libgdk-pixbuf2.0-dev	0	3320	915
liboss4-salsa-asound2	2963	3252	891
liboss-salsa-asound2	1741	2664	862
klogd	3	502	709
sysklogd	3	719	639
ppmtofb	0	719	639
selinux-policy-default	0	719	633
aide	0	719	633
libpam-umask	0	720	546

Table 2: Top 10 of potentially problematic packages identified by three simple metrics. Packages shown in boldface were manually identified as root causes of trend breaks during the aggregate analysis.

After ordering the packages with respect to the above three metrics, we obtain a list of potentially problematic packages, of which the top 10 are presented in Table 2. Interestingly, most of the packages that we manually identified as root causes during the aggregate analysis (shown in boldface) are also revealed by the metrics, with the important advantage that the metrics-based approach can be automated and requires much less manual inspection.

4 Second Case Study: The R Ecosystem

4.1 Overview of R

There are many popular languages, tools and environments for statistical computing. On the commercial side, among the most popular ones are SAS, SPSS, SPSS, Statistica, Stata and Excel. On the open source side, the R language and its accompanying software environment for statistical computing (`www.r-project.org`) is undeniable a very strong competitor, regardless of how popularity is being measured [33].

R forms a true *software ecosystem*, through its package management system that offers an easy way to install third-party code and datasets alongside tests, documentation and examples. The main R distribution installs a few *base* packages and *recommended* packages. The exact number of installed packages depends on the chosen version of R. (For R 3.2.2 there are 16 *base* packages and 15 *recommended* packages.) In addition to these main R packages, thousands of additional packages are developed and distributed through different repositories.

Precompiled binary distributions of the R environment can be downloaded from the Comprehensive R Archive Network (*CRAN*, see `cran.r-project.org`). *CRAN* constitutes the official R repository, containing the broadest collection of R packages. It aims at providing stable packages compatible with the latest version of R. Quality is ensured by forcing package maintainers to follow a rather strict policy. All *CRAN* packages are tested daily using the command-line tool `R CMD check` which automatically checks all packages for common problems. The check is composed of over 50 individual checks carried out on different operating systems. It includes tests for the package structure, the metadata, the documentation, the data, the code, etc. For packages that fail the check, their maintainer is asked to resolve the problems before the next major R release. If this is not done, problematic packages are archived, making it impossible to install them automatically, as they will no longer be included in *CRAN* until a new version is released that resolves the problems. However, it remains possible to install such archived packages manually.

Every R package needs to specify in its *DESCRIPTION* file the packages it depends upon (see Figure 1 for an example). We consider as dependencies the packages that are listed in the *Depends* and *Imports* fields of the *DESCRIPTION* file, as these are the ones that are required to install and load a package.

We have conducted multiple studies on the R ecosystem, focused on problems related to inter-component dependencies mentioned in Section 2 [11, 12, 13, 16]. We summarize our main findings in the following subsections. First, we present the main repositories containing R packages and the difficulties encountered when trying to manage dependencies across these different repositories. Next, focusing on the *CRAN* repository, we show how a part of package maintenance effort needs to be dedicated to fixing errors caused by dependency upgrades. Finally, we study the presence of identical cross-package clones in *CRAN* packages and investigate their reason of existence.

4.2 R Package Repositories

Besides *CRAN*, R packages can also be stored on, and downloaded from, other repositories such as *Bioconductor* (`bioconductor.org`), *R-Forge* (`r-forge.r-project.org`), and several smaller repositories such as *Omegahat* and *RForge*. Many R packages can also be found on "general-purpose" web-based version control repositories such as *GitHub*, a web platform for Git version control repositories. Table 3 provides a brief comparison of the four of the biggest R package repositories. It also provides an indication of the size of each repository, expressed in terms of the number of provided R packages.

Table 3: Characteristics of considered R package repositories

Repository	Since	# packages [Date]	Role	Package versions
CRAN	1997	6411 [19-03-2015]	Distribution	Stable releases
BioConductor	2001	997 [19-03-2015]	Distribution only	Stable releases
R-Forge	2006	1883 [18-03-2015]	Mainly development	*SVN* version control
GitHub	2008	5150 [17-02-2015]	Mainly development	*Git* version control

Bioconductor focuses on software packages and datasets dedicated to bioinformatics. *Bioconductor* packages are not installed by default: users must configure their R installation with a *Bioconductor* mirror. As in *CRAN*, packages that fail the daily check will be dropped from the next release of *Bioconductor*. ***R-Forge*** is a software development forge specialized at hosting R code. Its main target is to provide a central platform for the development of R packages, offering SVN repositories, daily built and checked packages, bug tracking, and so on. ***GitHub*** is becoming increasingly popular for R package development. Both *R-Forge* and *GitHub* differ from *CRAN* and *Bioconductor* because they do not only *distribute* R packages, they also facilitate the *development* of R packages thanks to their integrated version control system.

Support for multiple repositories is built deeply into R. For example, the R function install.packages can take the source repository as an optional argument, or can be used to install older versions of a given package. While this works well for repositories such as *CRAN* and *Bioconductor*, it does not for development forges such as *GitHub* due to the lack of central index of all packages.

One of the easiest ways to install packages hosted on forges is by using the devtools package. It provides various functions to download and install a package from different sources. For example, the function install_github allows the installation of R packages directly from *GitHub*, while the function install_svn allows the installation from an SVN repository (such as the one used by *R-Forge*). By default, the latest package version will be installed, but optional parameters can be used to install a specific version. As such, there is theoretically no longer a strict need to rely on package distributions. Therefore, development forges must be considered an important and integral part of the R package ecosystem.

Figure 4 shows the overlap of R packages on different distributions and development forges. **Between *Bioconductor* and the other package repositories, the overlap is very limited**. A negligible amount of *Bioconductor* packages is present on *CRAN* or *R-Forge*, which can be explained by the highly specialised nature of *Bioconductor* (focused on bioinformatics) compared to the other package repositories.

Around 18% of the *CRAN* packages are hosted on *GitHub*, while 22.5% of all R packages on *GitHub* are also present on *CRAN*. This overlap can be explained by the fact that both repositories serve different purposes (distribution and development, respectively). ***Many* R packages are developed on *GitHub*, while stable releases of these packages are published on *CRAN***.

We observe that *R-Forge* has 12.3% of its packages in common with *GitHub*, while as

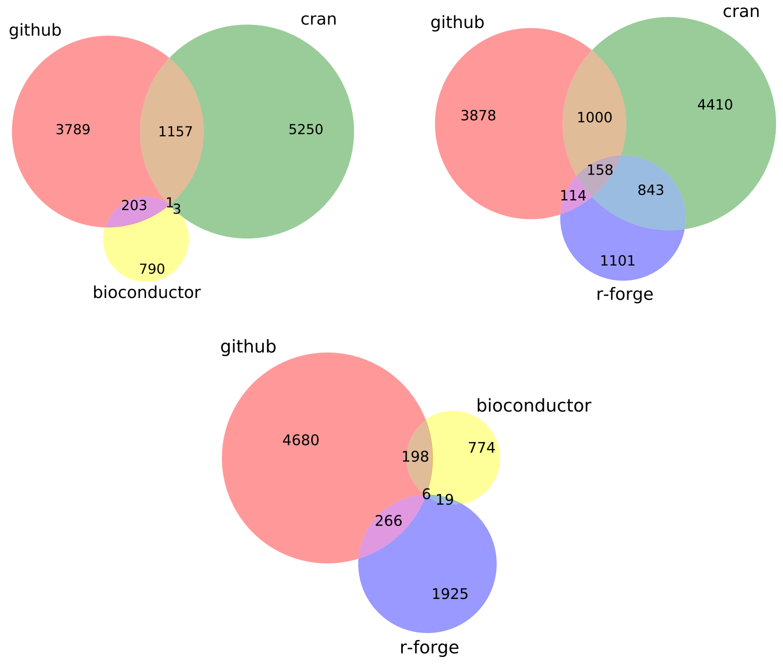

Figure 4: Number of R packages belonging to *GitHub*, *CRAN*, *Bioconductor* and *R-Forge* (counted during the first trimester of 2015).

much as 45.2% of its packages are in common with *CRAN*. This shows that ***R-Forge* serves as a development platform for *some* of the packages that get distributed through *CRAN***.

While the usage of devtools and similar tools potentially provides a way to use forges as a rolling release distribution, there are limitations to such a solution. First, there might be no central listing of packages available on these forges. For *R-Forge* the problem could easily be solved as it contains relatively few SVN repositories. *GitHub*, however, contains millions of Git repositories filled with content from various programming languages. Even if we limit *GitHub* repositories to those tagged with the R language, the vast majority does not contain an R package. The lack of a central listing of packages prevents devtools to automatically install dependencies. An additional problem is that the same package can be hosted in multiple repositories, making the problem of dependency resolution even more difficult. To summarise, **R package users and developers would benefit from a package installation manager that relies on a central listing of available packages on different repositories.** It is definitely feasible to achieve such a tool, since popular package managers for other languages such as JavaScript (e.g., bower and npm) and Python (e.g., pip) also offer a central listing of packages, facilitating their distribution through several repositories including *GitHub*.

4.3 Inter-repository Dependencies

How can we quantify the dependency resolution problem in the R package ecosystem by analysing the extent of inter-repository package dependencies? We previously observed that an R package may belong to different repositories (for example, *GitHub* may store the development version while *CRAN* may contain the stable release version of the package). Figure 5 summarises the inter-repository package dependencies. An arrow $A \xrightarrow{x\%} B$ means that $x\%$ of the packages in repository A have a *primary* dependency belonging to repository B. This *primary* dependency is computed by privileging the distributed version of a package over its development version. For example, if package p_1 on *GitHub* depends on package p_2 belonging to both *CRAN* and *GitHub*, it will be counted as a primary dependency from *GitHub* to *CRAN*.

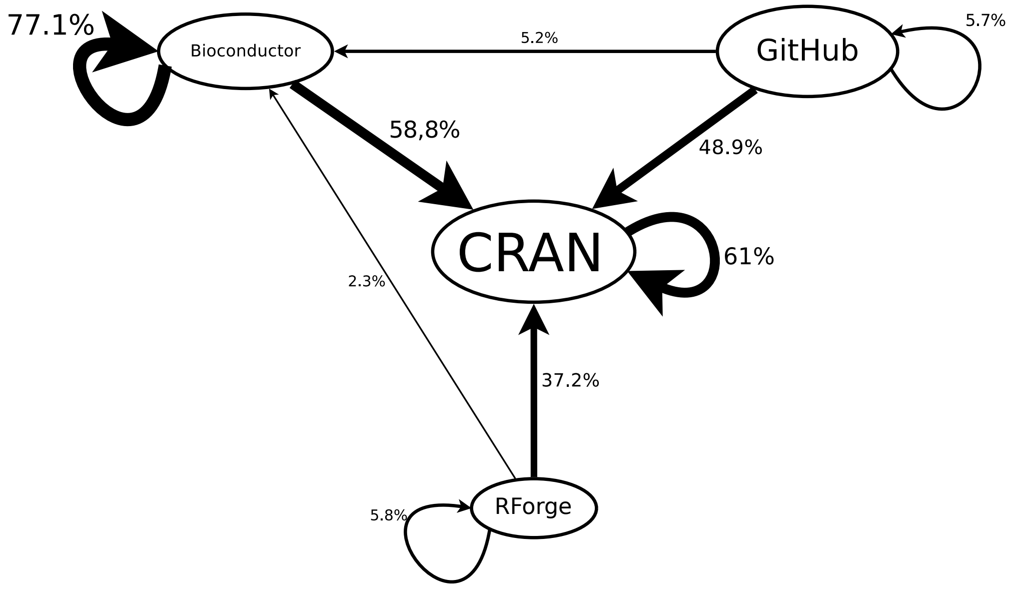

Figure 5: Percentage of packages per repository that depend on at least one package from another repository. The font size of the repository name is proportional to the number of packages it hosts.

We observed that ***CRAN* is self-contained**: only 61% of *CRAN* packages have dependencies, and all those dependencies are satisfied by *CRAN* because this is imposed by *CRAN*'s daily `R CMD check`. ***Bioconductor* depends primarily on itself and on *CRAN***: 58.8% of all *Bioconductor* packages depend on *CRAN* packages, while 77.1% of all *Bioconductor* packages depend on other *Bioconductor* packages. The situation for *GitHub* and *R-Forge* is very different : 48.9% of *GitHub* packages and 37.2% of *R-Forge* packages depend on a package from *CRAN*. This represents 87.1% (resp. 86.4%) of all the dependencies in *GitHub* (resp. *R-Forge*). Interestingly, the number of *GitHub* and *R-Forge* R packages having an intra-repository dependency is very low (less than 6%).

These observations strongly suggests that ***CRAN* is at the center of the ecosystem** and that it is nearly impossible to install packages from *GitHub*, *Bioconductor* or *R-Forge* without relying on *CRAN* for the package dependencies. Because of *CRAN*'s central position, its longevity and its important size in terms of packages, one might choose to distribute a new package only on *CRAN*, and to depend only on *CRAN* packages. However, we observed that more and more packages are developed and distributed on *GitHub*. We believe that **inter-repository dependencies will become a major concern for the R community, that could be addressed by a multi-repository package dependency manager.**

4.4 Intra-repository dependencies

Because *CRAN* is self-contained, it does not suffer from inter-repository dependency problems. This does not mean, however, that *CRAN* does not suffer from package dependency upgrades. Inter-repository package dependency upgrades are the cause of many errors, and therefore put a heavy burden on package maintainers. Despite the presence of continuous integration processes at the repository level (for example, the R CMD check tool in *CRAN* or *Bioconductor*), a lot of maintenance effort remains required to deal with such errors.

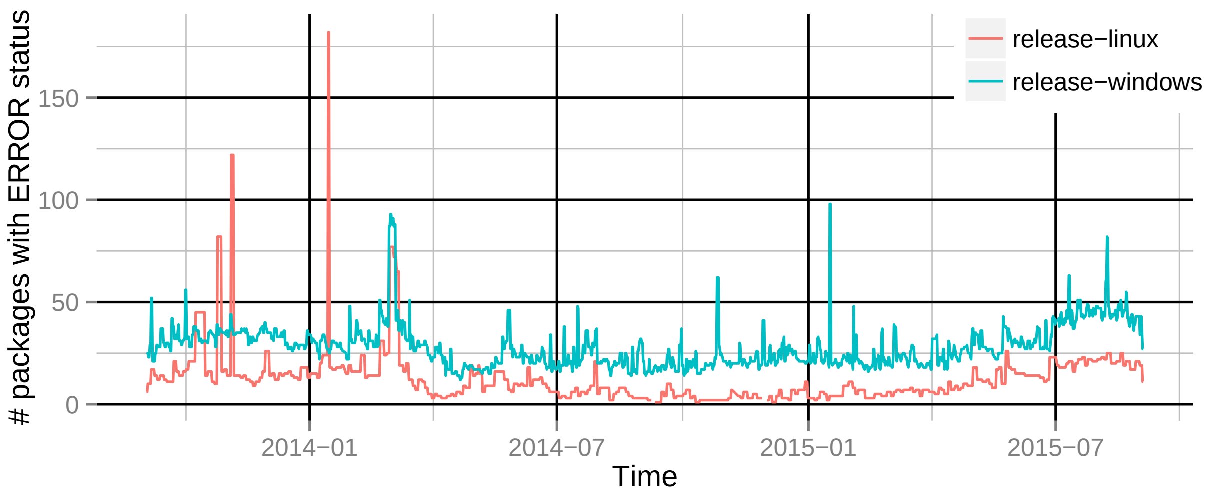

Figure 6: Evolution of the number of *CRAN* packages with ERROR status for two different flavors of the R CMD check.

For *CRAN* packages, the R CMD check is run on every package for different *flavors* of R. Each flavor corresponds to a combination of the operating system, compiler and R version being used. While the check results can vary a lot depending on the chosen flavor, we concentrate our next analyses on the flavor based on stable releases of R for *Debian*. This choice of flavor avoids the noise introduced by portability issues or changes occurring in R itself. This flavor is among the best supported ones, it also contains the most *CRAN* packages and it exhibits a less error-prone environment (see Figure 6).

On every day from 2013-09-03 to 2015-09-03, we took a snapshot of the results of the R CMD check (see cran.r-project.org/web/checks/). The snapshots associate to each *CRAN* package its reported status, which is either OK, NOTE, WARNING or ERROR. There were 4,930 available packages on *CRAN* at the beginning of this period, and 7,235 at the end of this period. During the whole period, 19,517 pairs package-version were available for a total of 7,820 different packages. We were only interested in the ERROR status because it can provoke package archival. Each time we found an ERROR, we identified the reason of this change among the following ones: because the package itself gets upgraded (*PU*); because of a strong dependency upgrade (*DU*); or due to other external factors (*EF*). In total we identified 1,320 occurrences of a status change to ERROR, and 1,288 occurrences of a status change from ERROR back to some other status. The results are summarized in Table 4.

status changes	number	package update (*PU*)	strong dependency upgrade (*DU*)	external factor (*EF*)
from ... to ERROR	1,310	30 (2.3%)	541 (41.3%)	739 (56.4%)
from ERROR to ...	1,288	346 (26.85%)	293 (22.75%)	649 (50.4%)

Table 4: Identified reasons for status changes to and from ERROR.

We observe that most ERRORs are introduced (56.4%) and fixed (50.4%) without a version update from the package (*PU*) or an upgrade of one of its dependencies (*DU*). Looking at the ERRORs that were caused by a package update to a new version, we see that, while very few packages (2.3%) failed when a new version of the package itself was released, more ERRORs were removed by the update of a package version (26.85%) than by the upgrade of a strong dependency (22.75%).

Out of the 1,288 ERRORs that were removed, 26 were introduced before we started extracting data. Only for the remaining 1,262 ERRORs we could identify both the cause of their introduction and disappearance. We observed that most (45.7%) of the 514 ERRORs that were introduced by a *DU* were removed by another *DU*, while 32.7% disappeared because of a *PU*. We also observed that among the 334 ERRORs fixed by a *PU*, 50.3% were introduced by a *DU* and 44.9% by an *EF*. This is, **more than half of the errors fixed by the package maintainers were introduced by changes in their package dependencies.**

From the above, we can conclude that **breaking changes in packages force dependent packages to be updated.** This can require an important maintenance effort. A way to reduce this effort would be to allow package maintainers to specify the required versions of their dependencies. This is currently not possible because the *CRAN* policy imposes package maintainers to support the latest available version of each dependency. **The R community would benefit from allowing packages to depend on different versions of other packages.** It would give them the time needed to perform appropriate dependency upgrades without impacting the *validity* of other packages, and without impacting end users. The community could also benefit from specific tools that predict in advance what could become broken if a specific dependency were to be upgraded in incompatible ways.

5 Conclusion

This chapter presented different issues that are commonly encountered in evolving software ecosystems involving a large number of software components and interdependencies. We provided a common vocabulary inspired by the state-of-the-art in this research domain. We discussed how each issue impacts component developers and presented solutions have been proposed in the research literature. We illustrated these issues in practice, through two case studies carried out on two very popular open source package-based software ecosystems.

For the Debian package ecosystem we showed that, despite the existence of multiple tools to solve many of the issues related to component dependencies, an historical analysis of strong conflicts allowed us to discover problems that could not be identified by current tools.

For the R package ecosystem we presented the main repositories where R packages are developed and distributed and showed that, despite the rising popularity of *GitHub*, *CRAN* remains the most important package repository. Focusing on *CRAN*, we found that an important number of package erros are caused by dependency upgrades and developers need to fix the error by releasing a new version of their package. The R community would therefore benefit from more advanced tools that recommend package maintainers and users how to overcome problems related to package upgrades.

To conclude, while previous research has lead to the creation of efficient tools to cope with dependency issues in component-based software ecosystems, there is still room for improvement. By historically analysing the component dependency graph, more precise information can be obtained and used to detect the root causes of dependency issues, and to provide better automated tool support for dependency management.

References

[1] P. Abate, R. Di Cosmo, L. Gesbert, F. Le Fessant, R. Treinen, and S. Zacchiroli. Mining component repositories for installability issues. In *Int'l Conf. Mining Software Repositories (MSR)*, pages 24–33, 2015.

[2] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. Dependency solving: A separate concern in component evolution management. *J. Systems and Software*, 85(10):2228–2240, 2012.

[3] Pietro Abate, Roberto Di Cosmo, Jaap Boender, and Stefano Zacchiroli. Strong dependencies between software components. In *Int'l Conf. Empirical Software Engineering and Measurement (ESEM)*, pages 89–99. IEEE Computer Society, 2009.

[4] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. Learning from the future of component repositories. In *Int'l Conf. Component-Based Software Engineering (CBSE)*, pages 51–60. ACM , 2012.

[5] Pietro Abate, Roberto DiCosmo, Ralf Treinen, and Stefano Zacchiroli. Mpm: A modular package manager. In *Int'l Conf. Component-Based Software Engineering (CBSE)*, pages 179–188. ACM, 2011.

[6] Cyrille Artho, Kuniyasu Suzaki, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. Why do software packages conflict? In *Int'l Conf. Mining Software Repositories (MSR)*, pages 141–150, 2012.

[7] Rahul C. Basole and Jürgen Karla. Value transformation in the mobile service ecosystem: A study of app store emergence and growth. *Service Science*, 4(1):24–41, 2012.

[8] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. The evolution of project inter-dependencies in a software ecosystem: the case of Apache. In *Int'l Conf. Software Maintenance (ICSM)*, 2013.

[9] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. How the Apache community upgrades dependencies: an evolutionary study. *Empirical Software Engineering*, 20(5):1275–1317, 2015.

[10] Maelick Claes, Tom Mens, Roberto Di Cosmo, and Jérôme Vouillon. A historical analysis of debian package incompatibilities. In *Int'l Conf. Mining Software Repositories (MSR)*, pages 212–223. IEEE Press, 2015.

[11] Maëlick Claes, Tom Mens, and Philippe Grosjean. maintaineR: A web-based dashboard for maintainers of CRAN packages. In *Int'l Conf. Software Maintenance, Reengineering, and Reverse Engineering (ICSME)*, pages 597–600, 2014.

[12] Maëlick Claes, Tom Mens, and Philippe Grosjean. On the maintainability of CRAN packages. In *Int'l Conf. Software Maintenance, Reengineering, and Reverse Engineering (ICSME)*, pages 308–312, 2014.

[13] Maëlick Claes, Tom Mens, Narjisse Tabout, and Philippe Grosjean. An empirical study of identical function clones in CRAN. In *Int'l Workshop on Software Clones (IWSC)*, pages 19–25, 2015.

[14] Roberto Di Cosmo, Davide Di Ruscio, Patrizio Pelliccione, Alfonso Pierantonio, and Stefano Zacchiroli. Supporting software evolution in component-based FOSS systems. *Sci. Comput. Program.*, 76(12):1144–1160, 2011.

[15] Roberto Di Cosmo, Stefano Zacchiroli, and Paulo Trezentos. Package upgrades in FOSS distributions: details and challenges. *CoRR*, abs/0902.1610, 2009.

[16] Alexandre Decan, Tom Mens, Maelick Claes, and Philippe Grosjean. On the development and distribution of r packages: An empirical analysis of the R ecosystem. In *European Conf. on Software Architecture Workshops (ECSAW)*. ACM, 2015.

[17] Roberto Di Cosmo and Jaap Boender. Using strong conflicts to detect quality issues in component-based complex systems. In *Indian Software Engineering Conf.*, pages 163–172, 2010.

[18] Roberto Di Cosmo and Jérôme Vouillon. On software component co-installability. In *ESEC/FSE*, pages 256–266. ACM, 2011.

[19] Danny Dig and Ralph Johnson. How do APIs evolve? A story of refactoring. *J. Software Maintenance and Evolution: Research and Practice*, 18(2):83–107, 2006.

[20] D.M. German, J.M. Gonzalez-Barahona, and G. Robles. A model to understand the building and running inter-dependencies of software. In *Working Conf. Reverse Engineering (WCRE)*, pages 140–149, Oct 2007.

[21] André Hora, Romain Robbes, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, and Marco Tulio Valente. How Do Developers React to API Evolution? The Pharo Ecosystem Case. In *Int'l Conf. Software Maintenance, Reengineering, and Reverse Engineering (ICSME)*, page 10, Bremen, Germany, September 2015.

[22] Kurt Hornik. Are there too many R packages? *Austrian Journal of Statistics*, 41(1):59–66, 2012.

[23] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *Int'l Conf. Software Engineering (ICSE)*, pages 485–495, 2009.

[24] Cory Kapser and Michael W. Godfrey. 'Cloning considered harmful' considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, 2008.

[25] Miryung Kim, Vibha Sazawal, David Notkin, and Gail C. Murphy. An empirical study of code clone genealogies. In *ESEC/FSE*, pages 187–196. ACM, 2005.

[26] Rainer Koschke. Large-scale inter-system clone detection using suffix trees and hashing. *Journal of Software: Evolution and Process*, 26(8):747–769, 2014.

[27] Nathan LaBelle and Eugene Wallingford. Inter-package dependency networks in open-source software. *CoRR*, cs.SE/0411096, 2004.

[28] Mircea Lungu. Towards reverse engineering software ecosystems. In *Int'l Conf. Software Maintenance (ICSM)*, pages 428–431, 2008.

[29] Mircea Lungu, Romain Robbes, and Michele Lanza. Recovering inter-project dependencies in software ecosystems. In *Int'l Conf. Automated Software Engineering (ASE)*, pages 309–312. ACM, 2010.

[30] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of api stability and adoption in the android ecosystem. In *Int'l Conf. Software Maintenance (ICSM)*, pages 70–79. IEEE Computer Society, 2013.

[31] D.G. Messerschmitt and C. Szyperski. *Software ecosystem: Understanding and indispensable technology and industry.* MIT Press, 2003.

[32] Israel J. Mojica, Bram Adams, Meiyappan Nagappan, Steffen Dienst, Thorsten Berger, and Ahmed E. Hassan. A large scale empirical study on software reuse in mobile apps. *IEEE Software*, 31(2):78–86, March 2014.

[33] Robert A. Muenchen. The popularity of data analysis software. http://r4stats.com/articles/popularity/, 2015.

[34] Nachiappan Nagappan and Thomas Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *Int'l Conf. Empirical Software Engineering and Measurement (ESEM)*, pages 364–373, 2007.

[35] Romain Robbes, Mircea Lungu, and David Röthlisberger. How do developers react to API deprecation? The case of a Smalltalk ecosystem. In *Int'l Symp. Foundations of Software Engineering (FSE).* ACM , 2012.

[36] Ripon K. Saha, Muhammad Asaduzzaman, Minhaz F. Zibran, Chanchal K. Roy, and Kevin A. Schneider. Evaluating code clone genealogies at release level: An empirical study. In *Working Conf. Source Code Analysis and Manipulation (SCAM)*, pages 87–96, 2010.

[37] Johannes Sametinger. *Software Engineering with Reusable Components.* Springer, 1997.

[38] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal Kumar Roy, and Mohammad Mamun Mia. Towards a big data curated benchmark of inter-project code clones. In *Int'l Conf. Software Maintenance, Reengineering, and Reverse Engineering (ICSME)*, pages 476–480, 2014.

[39] Bogdan Vasilescu, Stef van Schuylenburg, Jules Wulms, Alexander Serebrenik, and Mark G. J. van den Brand. Continuous integration in a social-coding world: Empirical evidence from GitHub. In *Int'l Conf. Software Maintenance, Reengineering, and Reverse Engineering (ICSME)*, pages 401–405, 2014.

[40] Jérôme Vouillon and Roberto Di Cosmo. Broken sets in software repository evolution. In *Int'l Conf. Software Engineering (ICSE)*, pages 412–421, 2013.

[41] Jérôme Vouillon and Roberto Di Cosmo. On software component co-installability. *ACM Trans. Software Engineering and Methodology*, 22(4):34, 2013.

[42] Jérôme Vouillon, Mehdi Dogguy, and Roberto Di Cosmo. Easing software component repository evolution. In *Int'l Conf. Software Engineering (ICSE)*, pages 756–766, 2014.

[43] Michel Wermelinger and Yijun Yu. Analyzing the evolution of Eclipse plugins. In *Int'l Conf. Mining Software Repositories (MSR)*, pages 133–136. ACM Press, 2008.