UMONS
Faculté des Sciences
Département d'Informatique

Addressing Maintainability Issues in Open Source Component-based Software Ecosystems

Maëlick Claes

A dissertation submitted in fulfillment of the requirements of the degree of *Docteur en Sciences*

Advisors

Dr. Tom Mens
Université de Mons, Belgium

Dr. Philippe Grosjean
Université de Mons, Belgium

Jury

Dr. Olivier Delgrange
Université de Mons, Belgium

Dr. Roberto Di Cosmo
Université de Paris Diderot, France

Dr. Philippe Grosjean
Université de Mons, Belgium

Dr. Tom Mens
Université de Mons, Belgium

Dr. Bruno Quoitin
Université de Mons, Belgium

Dr. Gregorio Robles
Universidad Rey Juan Carlos, Spain

July 2016

Acknowledgments

I would like to thank all the people without who this work would never have been possible. First and foremost, Dr. Tom Mens and Dr. Philippe Grosjean. I thank them for having proposed me to work with them and for their support, help and guidance throughout these four years. I also thank them for the good moments we shared.

I would like to thank the members of my jury: Dr. Olivier Delgrange, Dr. Roberto Di Cosmo, Dr. Gregorio Robles and Dr. Bruno Quoitin. They all influenced my work either through their own research work or by providing me support.

I would like to thank my co-authors and colleagues at the Software Engineering Lab: Dr. Alexandre Decan and Dr. Mathieu Goeminne. I also thank my co-authors, Dr. Roberto Di Cosmo and Dr. Alexander Serebrenik. Alongside my two thesis advisers, they are the people who had the most influence on this research work.

Even though we never co-authored a paper, there are two other people who left an important trace on my research. I would also like to particularly thank Dr. Bram Adams for having invited me to Montréal to collaborate for a month with him. This collaboration served as a foundation for an important part of this thesis. I also thank Dr. Bogdan Vasilescu for the collaboration and influence he had in my early years as a PhD student.

I thank all my present and past colleagues from the University of Mons and the Software Engineering Lab. In particular I would like to thank all the professors that taught me computer science and without whom I would not have been able to go into this research. I also thank all researchers I interacted with. This includes those I met at conferences and seminars and the anonymous ones who reviewed the papers I co-authored. They all had a positive influence on my research work.

Finally I would like to thank my family and all my friends for having been there throughout the years since I started to study computer science, but also for the general influence they have on me.

Abstract

The advent of the Internet and Open Source Software led to the emergence of the *software ecosystems*, consisting of a large number of interconnected software projects. In order to keep a trace of the relations between the different parts of a software ecosystem, one solution is to divide the ecosystem in different components, such as packages, and use dedicated software to manage the relationships between them. While component manager software helps both users and developers dealing with relationships such as dependencies and conflicts, it has its own kind of maintainability issues.

The goal of this thesis is to understand how dependency relations between software components, which are maintained by different people, impact software maintainability issues. By understanding how constraints on components can detect, prevent or reduce these issues, tools can be built to provide support for maintainer and user communities of component-based ecosystems and distributions.

Using two case studies, the Debian and R ecosystems, we

1. Define a conceptual framework in order to be able to extract data from different component repositories, analyze it to study maintainability issues, and interpret the results;

2. Use this framework to understand dependency relationships between components and how to better deal with them;

3. Present a tool for supporting developer and user communities of component-based software ecosystems.

Résumé

L'avènement d'Internet et des logiciels open source a mené à l'émergence des écosystèmes logiciels qui consistent en de nombreux projets logiciels interconnectés. Pour garder une trace des relations entre les différents projets d'un écosystème logiciel, une solution est de diviser l'écosystème en différents composants, telles que des paquets, et d'utiliser un logiciel spécialisé pour gérer les relations entre composants. Bien que des logiciels gestionnaires de composants aident à la fois les utilisateurs et développeurs à gérer les relations tels que dépendances et conflits, ils ont leur propres problèmes de maintenances.

Le but de cette thèse est de comprendre comment les relations de dépendances entre composants logiciels, maintenus par différentes personnes, impactent les problèmes de maintenabilité logicielle. En comprenant comment les contraintes sur des composants peuvent détecter, empêcher or réduire ces problèmes, des outils peuvent être construits pour fournir un support pour les mainteneurs et utilisateurs de communautés d'écosystèmes et de distribution de composants.

Sur base de deux cas d'utilisation, les écosystèmes Debian et R, nous

1. définissons un framework conceptuel pour extraire des données provenant de différents dépôts de paquets, les analyser pour étudier les problèmes de maintenabilité et interpréter les resultats;

2. utilisons ce framework pour comprendre les relations de dépendances entre composants et comment les gérer;

3. présentons un outil pour supporter les communautés de développeurs et utilisateurs d'écosystèmes logiciels basés sur des composants.

Contents

CHAPTER 1

Introduction

With the advent of the Internet and Open Source Software, people living all across the world started working together to build software artifacts at a scale never seen before. This led to the emergence of the so-called *software ecosystems* consisting of a large number of interconnected software projects [117]. In order to keep a trace of the relations between the different parts of a software ecosystem and limit the increase of complexity these relations could generate, one solution is to divide the ecosystems in different components, such as packages, and use a piece of software to manage the relationships between them. While component manager software helps both users and developers dealing with relationships such as dependencies and conflicts, it has its own kind of maintainability issues.

This introductory chapter presents this research context and how the thesis contributes to solving maintainability issues in open source component-based software ecosystems.

This chapter is partly inspired by our book chapter "Inter-component Dependency Issues in Software Ecosystems" [34].

1.1 Research Context

The advent of the Internet and Open Source Software led to the emergence of *software ecosystems*, consisting of a large number of interconnected software projects. In order to keep a trace of the relations between the different parts of a software ecosystem, one solution is to divide the ecosystem in different components, such as packages, and use a piece of software to manage the relationships between them. This thesis dissertation lies at the intersection of multiple domains of study: open source software development, repository mining, software evolution and component-based software development. In this first section we present these research domains.

1.1.1 Free, Libre and Open Source Software

In this dissertation we focus on the study of *Free, Libre and Open Source Software*. The main reason for this is the availability of the source code in online repositories. Very often these repositories also contain the development history alongside other data related to development activity such as issue trackers and mailing lists.

While Free, Libre and Open Source Software (FLOSS) was a common practice in the early days of computing, things started to change in the 1970s with the increasing cost of software development. Software manufacturers started to distribute software under restrictive licenses. With the appearance of the Internet and the World Wide Web in the 1980s and then its spread in the 1990s, the free software movement started to appear as a reaction to the limits imposed to software users by software manufacturers.

Different definitions of what FLOSS is exist. The term *free software* was first coined by Richard Stallman in 1985 in the GNU Manifesto [143]; he founded the same year the *Free Software Foundation* (FSF). In 1986 free software was defined with four basic liberties:

- Freedom 0: The freedom to run the program for any purpose.
- Freedom 1: The freedom to study how the program works, and change it to make it do what you wish.
- Freedom 2: The freedom to redistribute copies so you can help your neighbor.
- Freedom 3: The freedom to improve the program, and release your improvements (and modified versions in general) to the public, so that the whole community benefits.

Following the publication of Eric Raymond's *The Cathedral and the Bazaar* [134] and the release of the source code for *Netscape Navigator* (the ancestor of today's Mozilla Firefox web browser), the term Open Source was coined to focus on the technical aspects of FLOSS rather than its ethical ones. It led to the creation of the *Open Source Initiative* (OSI) in 1998 and the Open Source Definition [127]. It has been acknowledged that definitions of free and open source software are equivalent in most cases [142], thus differences between the two terms are mainly philosophical.

It is important to note that the term free software is ambiguous as free has two meanings in English: it can relate to both gratuity and freedom. Because of that the term *Libre Software*, inspired from French and Spanish, has also been employed since the beginning of the 2000s to resolve this ambiguity. Similarly the term FLOSS has been employed to avoid taking sides in the debate between the FSF and the OSI.

Even though we acknowledge that software ecosystems would not be possible without the ethical aspect of libre software, in the context of this dissertation we are mainly interested in the technical aspects of FLOSS. Thus in the remainder of the dissertation we will solely use the term Open Source.

1.1.2 Software Ecosystems

This dissertation focuses on a particular type of open source software: component-based software ecosystems. Software ecosystem is a term that appeared during the last decade in the fields of software evolution and software repository mining. While no consensus exists on what a software ecosystem is, multiple definitions have been proposed by researchers. In this dissertation we will stick to the one proposed by Lungu [104, 105] who said that a software ecosystem is "a collection of software projects which are developed and evolve together in the same environment". Other definitions have been proposed by researchers [78, 79, 109] and will be presented in more detail in Chapter 2

There are multiple well-known examples of open source software ecosystems:

- The *GNU* project aims at providing a fully Open Source operating system. It consists of many smaller software projects each providing a piece of that operating system.

- Linux is a kernel originally designed for the GNU operating system. While not technically an ecosystem as it is a single project, it contains thousands of drivers which form an ecosystem.

- Gnome and KDE both provide to users a desktop environment for operating systems such as GNU/Linux and derivatives of BSD.
- The *Apache Foundation* is a community of developers gathered around the *Apache License*. It develops and maintains many software projects such as the well-known *Apache HTTP Server*.

The research work presented in this dissertation was funded by a research project "Ecological Studies of Open Source Software Ecosystems". This dissertation focuses on a particular type of software ecosystem where each project is distributed as one or multiple distinct components. A component can be for example a package or a plugin, and often shares some kinds of relationships with other components of the ecosystem. One of these relationships are dependencies that require a package to only work in the presence of another package.

All the previously mentioned software ecosystems are not component-based ecosystems. While the software projects composing them may share some dependency relationships, in particular between end-user programs and libraries, it is not trivial to know which they are. Moreover, each software project generally consists of a source code repository, which may contain multiple components. While builds might be available for some of the projects, they will often be distributed alongside their dependencies and not as independent components.

One example of a component-based software ecosystem is *Debian*. It is a large collection of software programs distributed through its package manager *APT*. It is most particularly known as one of the oldest GNU/Linux distributions still being actively maintained. It is important to note that software projects distributed in Debian are not necessarily developed and maintained by Debian's developers. For example, Debian contains packages for most of the software projects developed in the previously mentioned ecosystems. The role of Debian package developers is to bundle appropriate versions of the different software projects in packages and make sure that these are stable enough in the specific context of a user's workspace, taking into account the other thousands of packages that she may have installed.

Other examples of package-based software ecosystems include collections of library packages for programming languages, like *CRAN* for *R*, *CPAN* for *Perl*, *npm* for *JavaScript*'s *Node.js*, *PyPI* for *Python* or *OPAM* for *OCaml*.

Apart from a package-based software ecosystems, other software ecosystems make use of components such as plug-ins (e.g., the *Eclipse* software development environment [159]), modules (e.g., the NetBeans software development environment), libraries [48], extensions and add-ons (e.g. the Firefox web browser), and mobile app stores [12, 120].

1.2 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 and discuss the possible problems caused by such inter-dependencies.

1.2.1 Terminology

Several researchers have proposed general models to study inter-component dependencies [41, 61, 106]. Based on these models, we use the following vocabulary to describe the different types of inter-component 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 is actually used by a particular user is called its **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. Examples 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. A component's **reverse dependencies** are the components that depend on it.

Figure 1.1 provides two concrete examples 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 a version of *R* greater or equal to 2.6 as well as on packages *stats*, *grDevices*, *graphics*, *MASS* and *ellipse*. 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 1.1: 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 1.1 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 1.1 Debian package *xul-ext-adblock-plus* requires version 8.0 or higher of *iceweasel*.

Figure 1.2 shows an example of a graph showing the aforementioned relationships. Components are visualized 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 on two components c and d that are in mutual conflict. Component f is also in conflict with version 2.1 or superior of component d. Component e depends on a version lower than 3.0 of component f.

In addition to abstract dependencies and optional dependencies, one can consider the **stage** at which they are needed (build-time, testing, installation or run-time) and the way in which these dependencies are intended to be used (e.g., as stand-alone programs, middleware, plug-ins or linkable libraries).

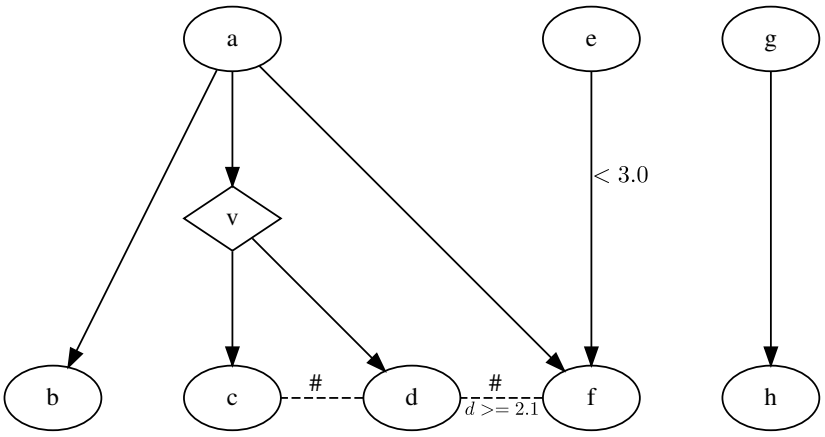


Figure 1.2: Example of a component dependency graph.

1.2.2 Issues in component-based software ecosystems

Identifying and retrieving dependency information

In *package-based* ecosystems, each package is generally required to provide *metadata* specifying package dependencies. Two examples of this were given in Figure 1.1. Sometimes, however, the metadata can be incomplete or inconsistent, or even entirely missing. In particular, constraints on dependency versions can be missing or inaccurate, because the component metadata is not always updated if the source code of components has been modified.

Moreover, even when one is able to have an accurate list of dependencies, it is usually not enough. Indeed, dependencies have their own dependencies which are consequently indirect dependencies of the initial component one wants to use. This type of recursive dependencies is known as **transitive dependencies**.

Satisfying dependencies and conflicts

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 and conflicts can make some dependencies unsatisfiable. Being unable to satisfy dependencies is generally highly undesirable because it would prevent a user from using a component.

Even when dependencies of a component can be satisfied, problems may arise because of conflicts. Conflicts can be declared explicitly in the component metadata if the format used allows it. However, they can also result from limits of the system. Indeed, when only one version of a component is allowed

to be used at the same time, two components can be implicitly in conflict if they both depend on different versions of the same package.

However, this explicit declaration of dependencies and conflicts is only the beginning of the story. Even if two packages P and Q do not declare a conflict, it may very well happen that they cannot be installed together. For example, P may (strongly) depend on some P_2 and Q on some Q_2, with P_2 and Q_2 in *declared conflict*.

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. This problem is known for package-based ecosystems as the problem of **co-installability** [10, 46, 47, 155]. 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.

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 1.1, 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 strong conflicts may arise during component evolution without the maintainers being aware of them. Because this may break an important part of a user system, this type of undesired strong conflicts are the ones that need to be studied and for which there is a need to develop tools to detect their cause.

Component upgrade

When developing software components, errors may be inadvertently introduced when changes occur in the software components one depends upon [1, 10, 135]. When changes to a component cause the software to fail, it puts a heavy burden on the maintainers of the components that depend on this failing component. This is especially true in large ecosystems where thousands of components are interdependent, and a single failure may affect a large fraction of the ecosystem [1, 75, 135].

Inter-project cloning

One way to avoid problems due to component dependencies could 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.

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. While this may provide an efficient solution to dependency problems on the short term, it can also introduce other issues. Indeed it will create software clones between different components which are known to increase maintainability effort if not managed properly [83].

1.3 Thesis statement

In the light of all the above, there is a need to understand how dependency relations between software components, which are maintained by different people, impact software maintainability issues. By understanding how constraints on components can help detect, prevent or reduce these issues, tools can be built to provide support for maintainer and user communities of component-based ecosystems and distributions.

The dissertation focuses on the following objectives:

1. Defining a conceptual framework in order to be able to extract data from different component repositories, analyze it to study maintainability issues, and interpret the results.

2. Understanding dependency relationships between components and how to better deal with them.

3. Supporting developer and user communities of component-based software ecosystems.

To achieve these goals we study two active and long-lived component-based software ecosystems. Using our conceptual framework we extract data from these two ecosystems and empirically analyze it.

We first study a package distribution for which a lot of effort has been put on ensuring stability, and for which numerous tools have already been developed based on previous research. We show how existing tools can discover problems quickly and how a historical analysis of their results can help to make new tools or improve existing ones.

Our second and main study relates to the opposite case of a package-based software ecosystem whose package manager does not make use of previous results on component-based software ecosystem research. For this case study we:

- Describe how the ecosystem is structured across multiple repositories and how this structure impacts studying them;
- Study how dependencies impact the maintainability of the oldest and biggest repository;
- Study the presence of code duplication inside this repository and give the proportion of code that could have been avoided by relying upon dependencies;
- Present a prototype of a web-based dashboard for component maintainers reporting to them all the previous results, and how it could be extended with the help of our conceptual framework to support other component-based ecosystems.

1.4 Structure

The remainder of this dissertation starts by presenting a state of the art of the research context in Chapter 2. It presents previous results found in the scientific literature: generally on software ecosystems, specifically on component-based software ecosystems and on both Debian and R ecosystems. Then Chapter 3 present the two case studies used throughout this dissertation.

It is followed by Chapter 4 which presents our conceptual framework for analyzing component-based software ecosystems. The design of this framework is driven by the nature of the data required to perform an analysis of issues in component-based software ecosystems.

Then we present the results of our empirical analysis on two ecosystem case studies. First, Chapter 5 presents the results of a historical analysis of co-installability conflicts between Debian packages. We show how fast the Debian community is generally solving problems caused by co-installability conflicts but also how our analysis allows to detect additional problems previously undiscovered by existing tools.

The remaining chapters present the main results on the empirical analyses of the R package-based ecosystem. Chapter 6 describes the main different R package repositories and how they are related to each other. Chapter 7 presents an analysis over more than two years of history of results of the continuous integration process of *CRAN* and how dependencies impact package maintainability. Chapter 8 investigates the problem of identical function clones inside R packages. Chapter 9 presents the web-based dashboard we designed to report our results on the R ecosystem to the R community.

Finally chapter 10 concludes by summarizing our main contributions, their threats and limitations, and future work.

CHAPTER 2

State of the Art

Component-based software ecosystems have gained a lot of interest from the software engineering community in the past two decades. We present in this chapter a non-exhaustive summary of the state of the art of existing studies in the field of software ecosystem evolution. The first section presents how research in software evolution has led to empirical research on software ecosystems. We present the different existing definitions of software ecosystem found in the literature and the different studies conducted on ecosystems. Then the second section presents how other researchers have analyzed and addressed the different issues related to component-based software ecosystems presented in Chapter 1.

This chapter is mainly inspired by some parts of the two published book chapters [34, 115] I co-authored. Some smaller parts of the current chapter are also inspired by our different conference publications on Debian and R [35–38, 43, 44]

2.1 Software Evolution and Ecosystems

Software systems are among the most complex artifacts ever created by humans. In the 1970s Manny Lehman studied software systems and proposed a series of properties of such systems known as *Lehman's laws of software evolution* [100]. In the following years multiple other studies gave additional evidence for these laws in multiple proprietary software systems.

With the advent of the Internet, the possibility to confirm or refute Lehman's laws arose. Indeed software development has become increasingly popular over the last two decades and large collections of software development history are now available online. In particular multiple studies tried to confirm Lehman's laws on open source software. Confirmed by replication studies [73, 136], Godfrey et al. [63] showed that the Linux kernel experienced a super-linear increase in size between 1994 and 1999. While confirming the law of *continuing growth* it challenges other laws such as *increasing complexity*. Indeed, previous studies showed that development seems to slow down as a system grows in size [58, 101]. Additional counter-examples to Lehman's laws have empirically shown that all open source software ecosystems evolution cannot be modeled and predicted by relying on Lehman's laws [31–33, 73].

In the case of Linux, Godfrey et al. [63] identified that drivers are mostly responsible for both the size and growth of the kernel code base. They gave multiple reasons to explain this super-linear growth. For example drivers are generally independent of each other and thus an increasing number of drivers can be added to the system without impacting its overall complexity. Also drivers are generally tied to hardware but are kept in the systems as "legacy" drivers in case some users still use some old hardware.

From this stems the idea of software ecosystems. With the advent of large scale collaboration over the Internet, it became possible to build software ecosystems growing at a super-linear pace while limiting the increase in complexity by subdividing software systems in multiple software projects loosely coupled together. To reflect this increase in complexity and scale, the term *software ecosystem* has been coined by Messerschmitt and Szyperski [117] to refer to such systems.

The term *ecosystem* comes from the field of ecology. According to [96], *ecology* is *the scientific study of the interactions that determine the distribution and abundance of organisms.* Typically, the dynamics of these interactions are studied in the context of an ecosystem. The term *ecosystem* was originally coined in 1930 by Roy Clapham, to denote the physical and biological components of an environment considered in relation to each other as a unit [161]. In other words, an ecosystem combines all living organisms (plants, animals,

microorganisms) and physical components (light, water, soil, rocks, minerals) that interact with each other.

Software ecosystems have now become a very active area of research, as can be seen in a recent systematic literature review [108, 109]. Unfortunately, in contrast to natural ecosystems, there is no common definition of software ecosystem. It can be defined and interpreted in different ways, depending on the point of view.

Business-centric viewpoint One of the first occurrences of the term software ecosystem can be found in [23] where it is used to refer to the way in which software suppliers, vendors, competitors, users, and third-party developers interact in software product lines. This view emphasizes the *business* perspective of a software system. A similar view, including the socio-economic environment and regulatory framework is adopted by Jansen et al. [78, 79], who define a software ecosystem as *"a set of actors functioning as a unit and interacting with a shared market for software and services, together with the relationships among them."* This view is schematically presented in Figure 2.1. An entire book is devoted to this perspective of software ecosystems [80]. A typical, but not exclusive, characteristic of these types of software ecosystems is the *competitive* aspect. The different projects in the ecosystem are in competition, either because they target the same end-users or offer the same type of service.

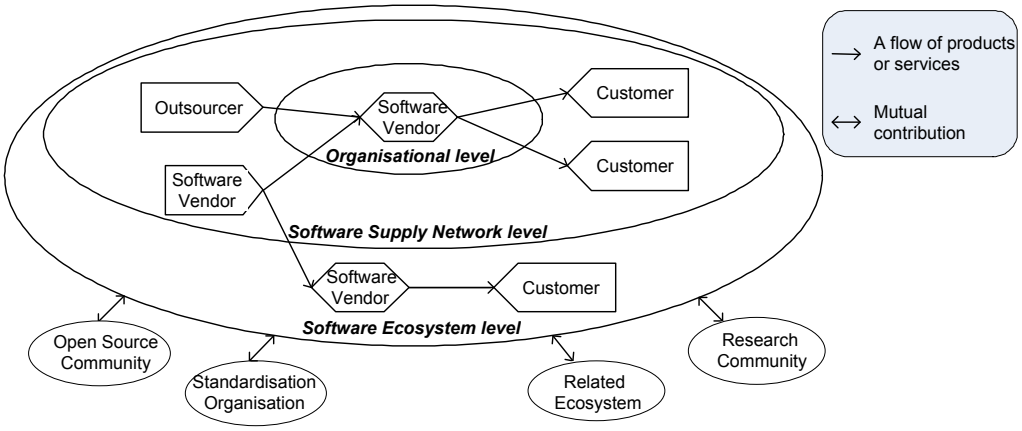


Figure 2.1: Actors in a software ecosystem. Figure reproduced from [154].

Since, as illustrated above, business-centric software ecosystems often constitute a core strategic asset for its contributors and supporting companies,

it is crucial to gain more insight in how ecosystems evolve and can be maintained successfully over time.

Development-centric viewpoint An alternative, more fine-grained definition of software ecosystem is provided in the seminal work of Messerschmitt [117] to refer to *"a collection of software products that have some given degree of symbiotic relationships."* A similar definition is given by Lungu [104, 105], who defines a software ecosystem as *"a collection of software projects which are developed and evolve together in the same environment."* This environment refers to the development environment, *i.e.* the software and hardware tools used during the development process.

We extend these definitions to take into account the *collaborative* and *social* aspects as well, by explicitly considering the communities involved (e.g., user and developer communities) as being part of the software ecosystem. Like software projects, the communities evolve over time (users and developers come and go). In addition, there is a high degree of interaction, even some kind of symbiosis, between the software projects and the communities of the ecosystems. This viewpoint is adopted by [60, 64, 65, 116, 128, 137, 152] that focus both on the technical aspects of the software produced and the social aspects of the communities producing and using this software.

It is especially in ecosystems where the community works towards a common goal that the collaborative nature wins over the competitive nature. Typically, software ecosystems consist of a relatively closed core software system that provides the basic functionality and that is developed by a more or less stable core team of developers, surrounded by a large collection of contributions provided by peripheral developers or even end-users [122, 130, 137].

We can provide numerous examples of software ecosystems, and many of them can be interpreted from both the business-centric and the development-centric viewpoint.

Mobile app stores Commercial or free application repositories for mobile operating systems (such as *iOS*, *Android* and *Windows 8*), form a business-centric ecosystem. While these operating systems are provided by Apple, Google and Microsoft, respectively, the SDKs and APIs allow third-party developers to build mobile applications on top of these operating systems. The mobile app ecosystems consist of the users, developers, managers of the mobile OS and the third-party mobile applications built on top of them. The official mobile app stores allow for applications to be sold to end-users, with

a shared profit. For Android, there is also a free and open source software repository of applications, called *F-Droid*[1].

The empirical study of the evolution of mobile applications is an emerging area of research. For example, Battacharya et al. [13] carried out an empirical study on the evolution of bug-related issues in 24 widely-used open source Android apps, while Basole et al. [12] studied the emergence and growth of mobile app stores in the mobile service ecosystem. McDonnell et al. [112] studied the rapid evolution of APIs and their adoption by client apps in the Android ecosystem.

IDEs Development environments for programming languages such as *Java* (e.g., *Eclipse* and *NetBeans*) or *Smalltalk* (e.g., *Squeak* and *Pharo* [135]) can be seen from a business-centric viewpoint. For example, the non-profit Eclipse Foundation is involved in the strategic direction, marketing and promotion of Eclipse and contains representatives of different companies such as IBM (the founder of Eclipse), Google, OBEO, Oracle, SAP, Talend. Eclipse is supported by numerous software vendors, and each of these vendors may provide different plugins with similar functionality, that are in direct competition with one another.

From a development-centric viewpoint, the Eclipse ecosystem is the universe of Eclipse *plugins* [39] together with the developers of these plugins. Studying the evolution of plugins is an active area of research [25–27, 159, 160]. All different Eclipse plugins rely on a common underlying architecture, platform and set of libraries without which they are unable to function correctly. The community of plugin developers therefore shares the common goal of improving a complete integrated software development environment. NetBeans, the main open source competitor for Eclipse, has a similar modular architecture with a common core.

A single GNU/Linux distribution Distributions, such as *Debian*, *Ubuntu* or *Red Hat*, are generally based on a packaging tool such as *APT* or *RPM*. Here the ecosystem's projects are not necessarily programs. For example they can be the set of packages and their building and configuration files.

All GNU/Linux distributions They form an ecosystem comprising several hundreds of actively competing Linux distributions, that are all based on a common core (the kernel of the Linux operating system [77] and a set of GNU libraries and utilities). The distributions vary in the system they target (e.g., desktop computers, laptops, tablets, smartphones, embedded

[1] `https://f-droid.org/`

systems) and the applications that are bundled with the distribution. Some distributions are commercially driven (e.g., Red Hat, SUSE, and Ubuntu), while others are entirely community-driven (e.g., Debian and Fedora). An excerpt of the evolution of Linux distributions is shown in Figure 2.2. While the family of all Linux distributions is an ecosystem, each of the distributions that belong to this family can also be considered as an ecosystem of its own, composed of the packages (together with the necessary building and configuration files) contained in the distribution. Gonzalez et al. [69] have taken a closer look at the evolution of the Red Hat and Debian distributions.

Forges and archive networks Open Source Software (OSS) repositories, commonly known as *forges*, can be considered as business-centric, since there is no control on the governance of the projects hosted in the forge. Examples of such forges are SourceForge, GitHub, Bitbucket, Launchpad and Savannah. There are also many forges that are dedicated to particular programming languages, such as the CCAN archive network for the C programming language, the CPAN archive network for Perl, RubyGems for the Ruby language, CTAN for all kinds of material around TEX, the Python Packaging Index for Python programs, and so on. Because of the lack of control, within and across these forges there are often different projects with similar functionality between which the users can freely choose.

Capiluppi and Beecher [29] performed an empirical study in which they studied the type of software forge (they refer to them as FLOSS repositories) and their mode of governance on the projects they host. They compared SourceForge (which they consider to be an *open* repository) with Debian (which they consider to be a *controlled* repository). They concluded that Debian hosted larger, more active and more complex structures. As a side-effect, more effort is needed to maintain these projects.

In [16] they explored how the structure, complexity and decay of open source projects may be influenced by the repository in which they are retained (e.g., SourceForge, Savannah, Debian, RubyForge, GNOME, KDE). They concluded that membership of a particular repository may depend on the maturity and quality of the project. For example, SourceForge tends to host more early inceptors and immature projects, while Debian tends to hosts high-quality mature projects.

GitHub has become an even more popular subject of research. Without aiming to be complete, we point to some relevant references here. For example, Gousios et al. [70, 71] provide *GHTorrent*, a scalable way to analyze *GitHub* data. Dabbish et al. [42] focused on the social and community aspects of *GitHub*. Blincoe et al. [21] used user-specified cross-references between

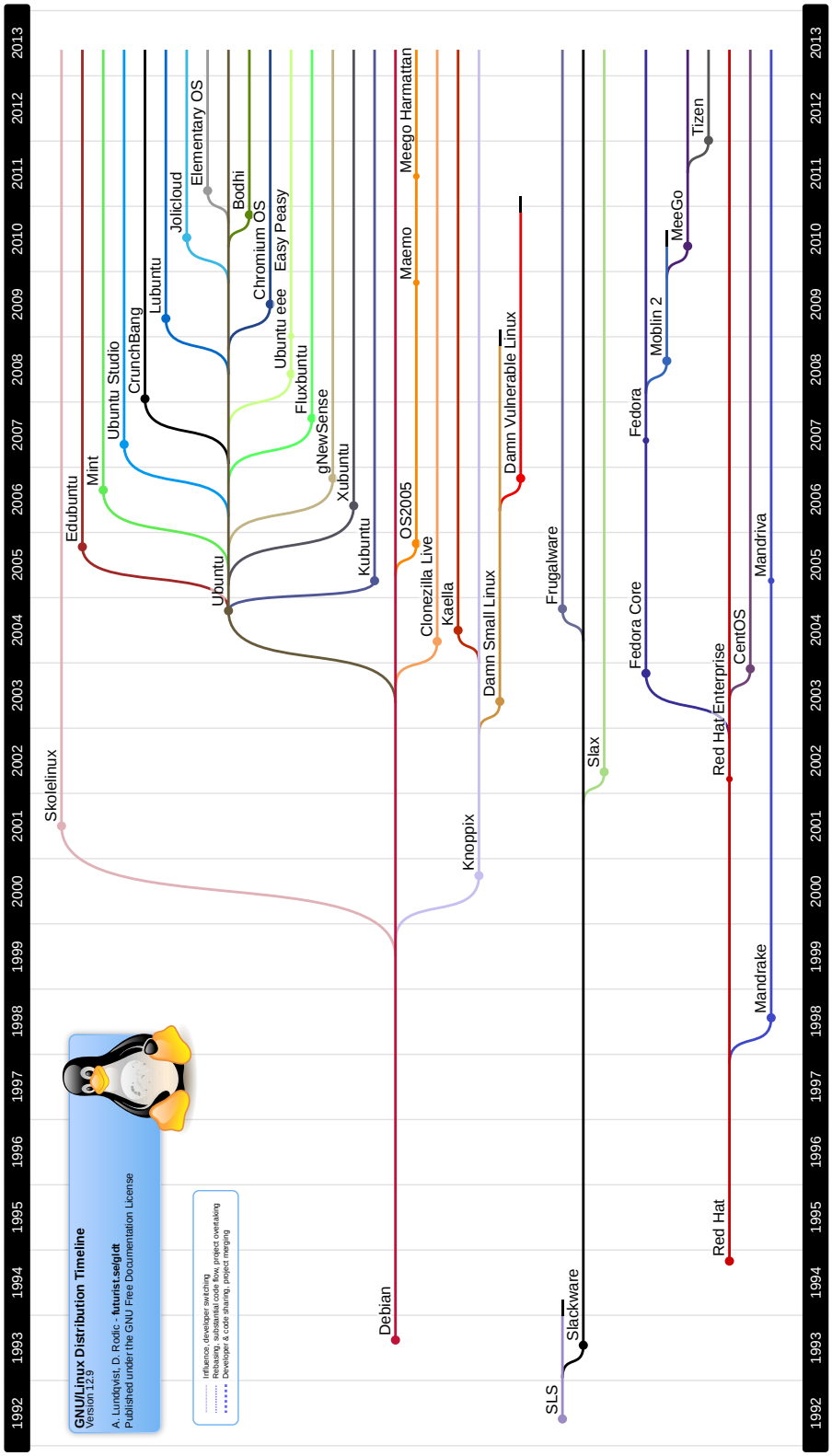


Figure 2.2: Linux distribution timeline (simplified version based on `http://futurist.se/gldt`)

projects to identify ecosystems in *GitHub*. Vasilescu et al. [151] compared the involvement on *GitHub* with the activity on Stack Overflow, a Q&A website for software developers. Vasilescu et al. [153] studied a large sample of *GitHub* projects developed in Java, Python and Ruby. They compared direct code modifications (commits) with indirect ones (pull requests) and related this to success or failure of continuous integration with TRAVIS-CI. Thung et al. [148] used the PageRank algorithm to identify influential developers and projects on a sub-network of *GitHub*. Yu et al. [163] studied patterns of communities found in *GitHub*'s social network.

Social networks *Facebook*, *LinkedIn*, *MySpace* and *Google+* can also be regarded as business-centric software ecosystems. They allow application developers to develop and integrate third-party applications, through a well-defined API. This provides significant added value to both the social network and the application providers.

GNU This project aims to provide a full free operating system based on the GNU General Public License (GPL) and the principles of UNIX. It is composed of GNU projects which are often ecosystems themselves. Examples of such sub-ecosystems are *R* and GNOME.

Graphical desktop environments GNOME and KDE are two full desktop environments for Linux and BSD operating systems. Both are based on a specific graphic toolkit (respectively GTK+ and Qt4). The developer communities share the common goal of delivering a complete user-friendly desktop environment. Because of these properties, if one wants to study the evolution of the ecosystem it is not enough to study the evolution of each individual projects. GNOME has been the topic of study for many researchers [59, 103, 123, 152].

Apache It is an ecosystem with a community of developers based around the Apache Software Foundation and the Apache License. One of its most famous projects is the Apache HTTP server. Apache is a decentralized community that uses a consensus-like development process. The aim is to provide stable, open and quality software developed by technical experts. Mockus et al. compared the Apache development process with the one of Mozilla [119]. Bavota et al. [14] studied the evolution of the dependencies between projects constituting the Apache ecosystem. Weiss *et al.* [158] studied the e-mails exchanged by the contributors of the Apache projects for discovering simple

migration patterns between projects and from the outside to a project. Gala-Pérez *et al.* observed that the ratio of e-mail messages in public mailing lists to versioning system commits has remained relatively constant along the history of Apache, and therefore advocate this ratio as a way to measure healthiness of an ecosystem's evolution [56].

Collaborative and socio-technical aspects of software ecosystems

From the two aforementioned definitions of software ecosystems we have seen that projects belonging to a software ecosystem can vary in a continuum ranging from highly *competitive* (if the business-centric viewpoint prevails) to highly *collaborative* (if the sense of community is very strong and there is strong incentive to work together towards a common goal). Many ecosystems fall somewhere in between, with some degree of collaboration and some degree of competition. It is clear that the competitiveness will have an important influence on the way the ecosystem will evolve over time.

Technical aspects Software ecosystems need to rely on a sophisticated software and hardware infrastructure and tools needed for their proper functioning, distribution, development, maintenance and evolution. Typical support that is provided are SDKs, APIs, download repositories, package management, dependency management and installation tools, version control systems, tools for change tracking, bug tracking and defect management, mailing lists, websites and other communication fora.

Social aspects Communication between the members of the software development team are at least as important as the technical aspects for the success of any software project [17, 45, 52, 150]. This is especially true for OSS projects where it is, in most cases, easier to become involved in the development team. This implies that the team structure needs to be more flexible in order to accommodate the easy integration of newcomers and to deal with the frequent departure of developers.

Fitzgerald [54] coined the term OSS 2.0 to reflect the new generation of OSS ecosystems that significantly "evolved" over the last decade or so from its single-project antecedents. Empirical results and insights obtained for individual OSS projects do not necessarily apply to projects that are part of a bigger, highly collaborative ecosystem of interacting parts. For example, the usage of APIs by Eclipse developers and how this relates to their level of experience was studied by Businge et al. [27]. Nakakoji et al. [122] distinguished between different types of OSS community members: developers, bug fixers, bug reporters, readers and passive users. They further

subdivided developers into peripheral developers, active developers, core members and project leaders. They proposed a so-called *onion model* for the OSS community structure, suggesting that there are very few project leaders, a bit more core members, even more active developers, and so on, and that promotion and migration of contributions tend to follow the layers of this model. Jergensen contested this onion model in an OSS 2.0 setting [81], by showing that contributor migrations do not tend to follow this model in many cases. Many other empirical studies have studied the activity patterns of, and differences between, core developers and peripheral developers [30, 49, 130, 137, 146, 162]. For a more detailed discussion we refer the interested reader to [66].

In their replication study, Dinh-Trong and Bieman [49] studied similar hypotheses on the FreeBSD system. They provided evidence that FreeBSD includes a small set of core developers involved in few parts of the system, and a larger set of top developers implementing 80% of the system. Yu and Ramaswamy [162] also made a distinction between core and associate project members, but unlike Nakakoji et al. [122] their approach infers roles automatically by clustering developers based on the frequency of their interaction. Capiluppi et al. [30] analyzed 400 open source projects, their evolution, and the developer communities responsible for their maintenance. They distinguished between stable and transient developers based on the amount of changes they perform and concluded that most of studied projects have no real developer community but only few regularly contributing developers. Focusing on the core teams of developers, Robles et al. [137] visually studied their activity patterns to identify 'code gods' projects in which the core developers hardly change during the entire project lifetime. Terceiro et al. [146] observed that core developers introduce less structural complexity than peripheral developers in general, implying that a stable and healthy core team contributes to the sustainability of open source projects. Still related to developer communication, Abreu and Premraj [8] studied the correlation with software quality. They observed a statistically significant correlation between communication frequency and number of injected bugs in the software. Through mining the source code repository and mailing lists of the well-known Apache and Mozilla OSS projects, Mockus et al. [119] investigated the roles and responsibilities of developers, and observed a set of implicit conventions among developers that implies an intensive communication. Because the communication is not scalable (one cannot linearly increase the communication intensity without adding more human resources), a strategy is needed to restrain the number and the size of communications. Apache seems to have a very efficient approach that consists of a minimal server core with a well-defined interface. Madey et al. [55, 150] analyzed

the social networks involved in OSS development and observed power laws at many scales. Martinez-Romo et al [111] went further and provided a methodology to analyze open source social networks to assess the relation between an open source project community and a company. Studer et al [60] extended their research by analyzing the KDE ecosystem and obtained the same results. Bird et al. [19] analyzed social networks emerging from mailing lists discussions and observed a Pareto distribution. Mailers tend to form a small-world network [118] from several points of view; for instance, few mailers received messages from an important number of persons while most of mailers received messages from few senders. A strong correlation between mailing and coding activities was found and evidence was provided that the role of developers in mailing lists is more important than one of the other mailers.

2.2 Component-based Software Ecosystems

This section presents the current state of the scientific literature regarding component-based software ecosystems. Particularly we look at how the different issues introduced in Chapter 1 have been studied and addressed by previous research.

2.2.1 Identifying and retrieving dependency information

While many component-based ecosystems require each component to provide *metadata* specifying dependencies, sometimes the metadata can be incomplete or inconsistent, or even entirely lacking [106]. 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 analyzer 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 dependencies, Lungu et al. [106] 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*. Provided entities are for example classes and methods declared in the source code. Called entities represent calls to a method (or a class). Required entities are called by the project but not provided by it. Lungu et al. [106] 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. [4], 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.

2.2.2 Satisfying dependencies and conflicts

Satisfying dependency constraints Since the pioneering work of Mancinelli et al. [107] we know that, despite the NP-compeleteness of the general problem of identifying uninstallable components, efficient tools can be developed for identifying them. Galindo et al. [57] even propose to use software product line tools for this task.

Based on the 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 Like direct dependencies are not enough to know the list of all components required by another, direct conflicts are not enough to know what are all components incompatible with another. Just like strong dependencies indicate are the components always required for another one to work, strong conflicts are the components that can never be used alongside another one. A strong conflict graph can be used

to detect co-installability problems between components [10, 46, 47, 155, 156]. These studies led to the creation of coinst, an tool to efficiently reduce a co-installability graph to a manageable size.

2.2.3 Component upgrade

This problem of component upgrades has been studied by many researchers. Robbes et al. [135] studied the ripple effect of API method deprecation in the Squeak/Pharo ecosystem. They showed 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. [75] 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. [113] 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. [14] 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 [15] 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 [153]. 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. [41] highlighted peculiarities of package upgrades and discussed that current techniques are not sufficient to overcome failures. They proposed solutions to this problem [2, 40] and built a tool called comigrate to efficiently identify sets of components that can be upgraded without causing failures [157]. Similarly, Abate et al. [3, 6] proposed a package manager designed to allow the use of different dependency solvers as plugins in order to better cope with component upgrade issues.

2.2.4 Inter-project cloning

Code cloning is an active research topic of the software engineering community. A comprehensive overview of software cloning literature can be found online[2].

[2] `http://students.cis.uab.edu/tairasr/clones/literature/`

Most research focuses on how to detect clones (e.g., [90, 92, 95, 98, 138]), while some articles focus on how to remove clones (e.g., [74, 91, 93]).

Code clones can be classified in four mutually exclusive types [18, 138]. Type-1 clones are syntactically identical code snippets at the abstract syntax level (i.e., ignoring differences in white space, layout and comments). Type-2 clones additionally differ in identifier names and literal values. Type-3 clones syntactically differ by having some statements added, modified and/or removed with respect to each other. Type-4 clones implement the same functionality while being syntactically dissimilar.

In the context of a single software project, the presence of software clones has been extensively studied and has shown to be beneficial or detrimental to software maintenance [83,86,87,139]. Code clones are often considered harmful because they led to redundancy due to code duplication. This makes software maintenance more difficult. For example, Jürgens et al. found inconsistent changes to code clones to be very frequent and a significant number of defects are introduced by such changes [83]. On the other hand, many situations have been reported in which clones are not considered harmful, are impossible or impractical to remove, or are even beneficial [86, 87, 139].

The bottom-line is that, if you really have to clone some code, you need to do it safely. Proactive tool support can be very beneficial to help detect the presence of clones, to propagate changes across clones, to assess the risk or benefits of clones, and to help remove clones if needed. Many tools have been proposed for detecting clones, including CCFinder, [85] CBCD, [102] CloneDR, CPMiner, Dup, [11] Duploc, [51] iClones, KClone [82] and NiCad. For a qualitative comparison of clone detection techniques and tools, see [138].

While there have been recent studies on inter-project cloning [94, 144], insight on the causes and implications of inter-project software clones is still lacking. Although using cloning may help to avoid dependency upgrade problems from a user point of view, it forces each developer to choose which version of all their transitive dependencies to include in their own component.

The only study on code duplication between components with an ecosystemic point of view we are aware of is by Mojica et al. [120]. They showed that Android mobile apps contains a lot of very frequent code reuse across them. They explain that it is mainly because 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.

CHAPTER 3

Selected Case Studies

This chapter briefly presents the two case studies used throughout this dissertation. First the GNU/Linux package distribution Debian, followed by the ecosystem consisting of the different package repositories for the statistical environment R. For both ecosystems we give the context and history of the ecosystem, the specificities of each of them, and pointers to different studies that have been previously conducted on them.

This chapter is partly based [34, 35, 43]

3.1 Introduction

In this section we present the two ecosystems that are used as case studies in this dissertation. The first one is the Debian package distribution and the second one is the package-based ecosystem of the statistical environment R. The reason we use them as case studies is that they are both composed of thousands of packages and have existed for more than 15 years, making them large enough and mature enough for being empirically studied.

The Debian ecosystem has been studied by many researchers and an extensive collection of tools based on research results have been produced and used by Debian maintainers [2, 4, 10, 40, 41, 46, 47, 155, 157]. We use this ecosystem to show how existing tools based on research results efficiently help the community to solve their problems. We also present how they can be improved by analyzing their result on the available history of Debian.

On the other hand, R packages do not benefit from as much advanced tools to support package maintainers. One of the reasons of this can stem from a community that is largely composed of people without a strong background in computer science or software engineering, such as statisticians, biologists, economists, etc. Studying the R ecosystem is first interesting because it does not benefit as much from existing results on package management as other ecosystems such as Debian do. However it is still a successful ecosystem because R was ranked as the sixth most popular programming language in 2015 by IEEE[1]. Thus it is also important to study it and create new tools based on these studies to support its package maintainers.

3.2 Debian

The Debian distribution is a coherent collection of free software packages, initially announced in 1993, with a first stable release in 1996. To facilitate maintenance and collaborative work, Debian is built out of individual *packages* maintained by independent developers. Over time, Debian has undergone an impressive growth, and today it contains tens of thousands of different packages, with over a thousand developers. While it has been ported to a multitude of architectures (see `www.debian.org/ports`), and supports several kernels, this dissertation focuses on the GNU/Linux distribution for the i386 architecture only. This architecture is historically the first one for which Debian has been made available, and the most popular over time.

[1] `http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages`

The development process of the Debian distribution is mainly organized around three collections of packages, called *releases*: stable, testing and unstable. stable corresponds to the latest official production release (see Table 3.1), and only contains stable, well-tested packages. The testing distribution contains package versions that should be considered for inclusion in the next stable Debian release. Since version 5.0, each 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 contains packages that are not thoroughly tested and that may still suffer from stability and security problems. This release contains the most recent packages but also the most unstable ones.

Version	Name	Freeze date	Release date	# packages
1.0			Never released	
1.1	buzz		1996-06-17	474
1.2	rex		1996-12-12	848
1.3	bo		1997-06-05	974
2.0	hamm		1998-07-24	about 1.5K
2.1	slink		1999-03-09	about 2.2K
2.2	potato		2000-08-15	about 2.6K
3.0	woody		2002-07-19	about 8.5K
3.1	sarge		2005-06-06	about 15K
4.0	etch		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 29K
7.0	wheezy	2012-06-30	2013-03-04	about 36K
8.0	jessie	2014-11-05	2015-04-26	about 43K

Table 3.1: Stable production releases of Debian

The Debian free software distribution is one of the largest organized collections of software packages today, and the availability of the full history of its evolution has made it an ideal object of study over the last few years, to the point that several infrastructures have been built to facilitate the extraction of information from this historical data: the Ultimate Debian Database[2] (UDD) described by Nussbaum et al. in [125], and the Debsources archive[3] described by Zacchiroli et al. in [28].

At the macro level, several characteristics of the Debian package repositories have been discussed in the literature. The small-world structure of the repositories is shown in [97] and [22]. The growth of the distribution,

[2] `https://udd.debian.org`
[3] `https://sources.debian.net`

according to its package size and programming language usage has been first analyzed in [68] and more recently in [28]. Changes in package characteristics such as age, maintainers, bugs and popularity are charted in [124].

Multiple tools based on the work by Di Cosmo et al. have started to be used by the Debian community. distcheck can detect packages that are uninstallable and report the reason why they can't be installed. coinst detects strong conflicts and simplify the dependency graph to give the sets of packages that can't be installed together. comigrate identifies packages that can be upgraded without causing failures.

3.3 R

There are many popular languages, tools and environments for statistical computing. On the commercial side, among the most popular ones are SAS, 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 undeniably a very strong competitor, regardless of how popularity is being measured [121].

R forms a *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 version R 3.2.2 there were 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.

CRAN, the Comprehensive R Archive Network, 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.

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* (`omegahat.org`) and *RForge* (`rforge.net`). 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. While *CRAN* is the official and most well-known package repository for R, it is not enough to only consider *CRAN* to fully study the R ecosystem because of the existence of many other package repositories.

Hornik [76] studied the evolution of *CRAN* packages. He showed that like other package distributions such as Debian it experienced a super-linear increase in the number of packages and file size. He also looked at the dependencies of packages and showed that around three quarters of all packages do not depend on any other *CRAN* packages and that approximately only 10% of all packages have at least one reverse dependency.

CRAN requires developers to make their package work with the last stable version of R but does not impose compatibility with previous versions. Older package versions are not removed but archived. It means that someone relying on an older version of R does not have the possibility to automatically install an old package version alongside the correct versions of its transitive dependencies. Ooms [126] discussed this problem and proposed two solutions inspired by Debian and *npm* to overcome it.

Germán et al. [60] studied the evolution of *CRAN* by comparing the characteristics, growth, dependencies and community structure of core packages and user-contributed packages. They also analyzed the user and developer communities by studying mailing list traffic. Zagalsky et al. [164] studied the difference of practice in the R community between its two main communication channels, official mailing lists and *StackOverflow* questions related to R.

We are not aware of any studies taking into account other R package distributions (such as *Bioconductor*) or development forges (such as *R-Forge* or *GitHub*). All related research seems to be restricted to the official package distribution *CRAN*.

Although *CRAN* has a super-linear growth [76], *GitHub* has an even higher growth rate and caught up and overtook *CRAN* in terms of number of packages as seen in Figure 3.1. Through e-mail interviews [114] with R package maintainers, we identified that one of the problems they encounter is the lack of version constraints on dependencies: *"Especially with respect to package dependencies, the risk of things breaking at some point due to the fact that a version of a dependency has changed without you knowing about it is immense. That actually cost us weeks and months in a couple of*

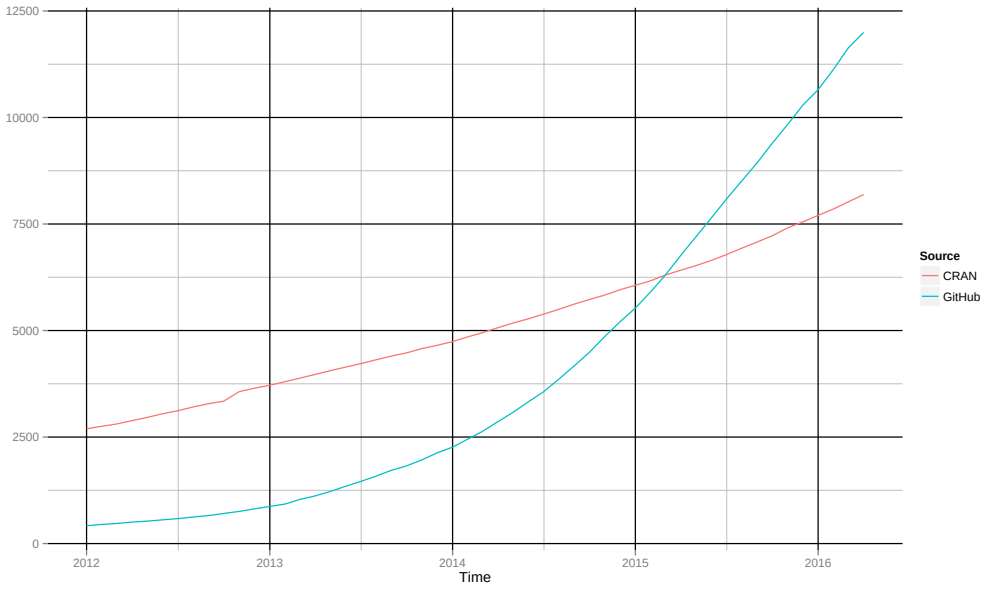


Figure 3.1: Evolution over time of the number of R packages in *CRAN* and *GitHub*

professional projects I was part of. While it's rather a philosophical than a technical question how package dependencies are/should be handled in R, I personally think it's really relevant to at least be able to be very specific and rigid with regard to your dependencies. And I think the R universe could provide better tools to fit the needs of developers and professionals out there in a better way." [9] The increase in popularity of *GitHub* and its lack of generalized quality checking process, stress out the importance of managing maintainability issues for the R community.

CHAPTER 4

A Conceptual Framework for Analyzing Package-based Software Ecosystems

The remainder of this dissertation is focused on the empirical analysis of two case studies. We strive to ensure that our analysis can be reproduced on other software ecosystems with minimal effort. In order to fulfill this goal we propose in this chapter a conceptual framework to analyze package-based software ecosystems. The framework consists of a general workflow for data extraction, processing and analysis. It defines a common terminology in order to be able to study and compare different package repositories. Then this chapter details the various steps of the framework and finally presents how it has been applied on the two case study ecosystems.

4.1 Introduction

We define a general framework to serve as foundation for the empirical analysis conducted in the subsequent chapters. Its goal is to provide a common terminology, organize software tools and scripts, data and meta-data required for the analysis of a package-based software ecosystem. The goal of this framework is to provide a way to gather, store and analyze data coming from different sources concerning the ecosystem. Because we are interested in performing data analysis of ecosystems, the framework is designed around a data format intended to be used by analysis tools and scripts providing an output such as charts, reports like scientific papers or end-user dashboards.

At high level our framework is designed based on a three step workflow. These three steps are data extraction, data analysis and reporting. This workflow revolves around a terminology to abstract packages and package distributions.

This chapter will first define the terminology used by the framework. Next, the different steps of the framework workflow are presented: data extraction, data analysis and reporting. Then, different formats available to represent data are presented. Finally, concrete examples are given to illustrate how the framework can be used.

4.2 Terminology

In this section we define an abstract terminology for the different concepts used by the framework. We define a package as a set of files that can contain, among others, meta-data, source code, documentation or configuration files that will be installed on users' systems. Characteristic of a package is that it can be stored online at different places and its content evolves over time.

In order to take these particularities in consideration we extend Lungu's definition of an ecosystem [104, 105]: *"a collection of software projects which are developed and evolve together in the same environment"*. We define an ecosystem as a set of distributions containing different projects. A project is a physical location where the content of a package is stored and may change over time.

Notation 4.2.1 (Ecosystem, distributions and projects)**.**

- *A package-based ecosystem E is composed of different package distributions.*
- *A package distribution (or repository) $D \in E$ is a set of project histories.*

- *A project history $P \in D$ consists of a set of project states $s \in P$.*
- *$Projects(E) = \{P \in D \mid D \in E\}$ is the set of all project histories belonging to the ecosystem E.*
- *$States(D) = \{s \in P \mid P \in D\}$ and $States(E) = \bigcup_{D \in E} States(D)$ are the sets of all project states belonging respectively to the distribution D or the ecosystem E.*

Each project state s has a release time, package name and version. The package version is a tag generally used by the package manager to identify a version of a package.

Notation 4.2.2 (Packages, versions and release times)**.**

- *A project state $s \in States(E)$ is defined as a couple $((p, v), t)$ where (p, v) is itself a couple of the package name p and the version tag v, and $t \in T$ the release time of the project state.*

  The time domain T is a totally ordered set. $package : P \to Name$, $version : P \to Versions$ and $time(s) = t$ are defined as projection functions and respectively abbreviated to p_s, v_s and t_s for a given state s.
- *$States(p, D) = \{s \in States(D) \mid p_s = p\}$ is the set of project states from distribution D whose package name is p.*
- *$States(p, v, D) = \{s \in States(p, D) \mid v_s = v\}$ is the set of project states from distribution D whose package name is p and version tag is v.*
- *$Packages(D) = \{p_s \in Name \mid s \in States(D)\}$ and $Packages(E) = \{p_s \mid s \in States(E)\}$ are the sets of all package names belonging respectively to the distribution D and the ecosystem E.*
- *$Versions(D) = \{(p_s, v_s) \mid s \in States(D)\}$ is the set of all package versions belonging to the distribution D.*
- *$Versions(E) = \{(p_s, v_s) \mid s \in States(E)\}$ is the set of all package versions belonging the ecosystem E.*

Figure 4.1 shows a possible theoretical structure of a package-based ecosystem. While in some cases it can happen that a package is found in a single project with each state matching a different version of that package, the use of this terminology allows to deal with ecosystems where it is not necessarily the case.

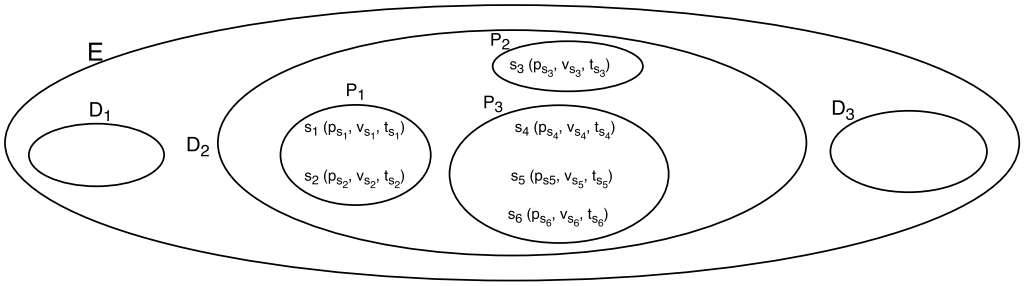


Figure 4.1: Example of a possible ecosystem topology

When one of the ecosystem's package distributions is a source code repositories, such as *GitHub*, each *GitHub* project corresponds to a project history in our framework and each commit of the project's git repository corresponds to a single project state. Hence, the projects are not directly related to packages. The package name and the version tags that are associated to each state can be inferred from meta-data found in the git repositories themselves. Thus it is common for multiple states (i.e., git commits) to have the same version tag, in particular for successive states for which package meta-data is not updated.

Similarly, package names can be different between two states of a single project, and be the same between states of different projects. In the context of packages hosted on *GitHub*, this can happen when the author of a project changes the name of its package without creating a new *GitHub* project, or when two different projects define the same package.

In addition to package name and version tag, each project state can have meta-data associated to it. We consider meta-data as a dictionary associating keys to values of different types. Meta-data can also include, among others, the official maintainer name, a list of authors, a list of relationships with other packages such as dependencies, copyright information, description, tags, etc., all stored in structured or unstructured format.

Notation 4.2.3 (Meta-data)**.**
$md : States(E) \times Key \to Metadata : (s, k) \mapsto md(s, k)$ represents the meta-data value for project state s and key k.

Snapshots represent the state of a distribution at a given point in time. It is the set of the latest project states that were available in that distribution at that time.

Notation 4.2.4 (Snapshot)**.** *For a given $t \in T$, a package snapshot is defined as*

$D_t = \{s \in P \mid P \in D, t_s < t, \forall s' \in P, s' \neq s \wedge t_{s'} < t \Rightarrow t_{s'} < t_s\}$, *the set of the latest project states, released before time* t*, for each project of* D*.*

Finally packages have relationships such as dependencies or conflicts that can be defined between them. It is important to note that these relationships are not defined between states but between a state and a package version. Indeed, while dependencies are defined in the project history, they reference package names rather than projects.

Notation 4.2.5 (Relationships)**.**

- $Dep \subseteq States(E) \times Versions(E)$ *is the dependency relation between project states and package versions.*
- $dep(s, t, D) = \{s' \in D_t \mid (s, (p_{s'}, v_{s'})) \in Dep\}$ *is the set of all direct dependencies of project state* s *that are included in a given package snapshot.*
- $revdep(s, t, D) = \{s' \in D_t \mid (s', (p_s, v_s)) \in Dep\}$ *is the set of all direct reverse dependencies of project state* s *that are included in a given package snapshot.*
- $dep^*(s, t, D) = dep(s, t, D) \cup \bigcup_{s' \in dep(s,t,D)} dep^*(s', t, D)$ *is the set of all transitive dependencies of* s *in a given package snapshot.*
- $revdep^*(s, t, D) = revdep(s, t, D) \cup \bigcup_{s' \in revdep(s,t,D)} revdep^*(s', t, D)$ *is the set of all transitive reverse dependencies of* s *in a given package snapshot.*
- $Conf \subseteq States(E) \times Version(E)$ *is the conflict relation between project states and packages. It states which package versions are incompatible with a given project state.*
- $conf(s, t, D) = \{s' \in D_t \mid (s, (p_{s'}, v_{s'})) \in Dep\}$ *is the set of all declared conflicts of project state* s *included in a given package snapshot.*

Notation	Description
E	an ecosystem
D	a distribution
P	a project history
s	a project state
(p_s, v_s)	the package name and the version tag of a state s
t_s	the release time of a state s
D_t	the snapshot of a distribution at time t
$dep(s, t, D)$	all dependencies in distribution D of project state s at time t
$dep^*(s, t, D)$	all transitive dependencies in distribution D of project state s at time t
$revdep(s, t, D)$	all reverse dependencies in distribution D of project state s at time t
$revdep^*(s, t, D)$	all transitive reverse dependencies in distribution D of project state s at time t
$conf(s, t, D)$	all declared conflicts in distribution D of project state s at time t

Table 4.1: Summary of introduced notation

4.3 Data extraction

Data extraction consists of fetching data from different sources and converting it to a data format used by our framework. Most of the data used for our analysis is generally fetched from various online sources. Examples of data sources are open source code repositories, web servers and web pages. Even though we do not rely on them, ideally one should be able to extend our framework using other data sources such as mailing list repositories or bug tracking systems.

One of the goals of the framework is to easily update data. This means that ideally one should not download again data of packages previously downloaded. Data must be updated differently depending on its nature. Some data sources can consist of individual pieces of information that are not supposed to change over time while other data sources can change.

For one time data retrieval, it is only necessary to download a piece of data from the server if it does not already exist in the local database. An example of one time data retrieval is a package version distributed on a web server as a tarball. Each package version is supposed to be immutable through time once released and thus does not need to be downloaded again if it was

already retrieved in the past.

Secondly some data sources can be downloaded incrementally. For example all the versions of a package can be downloaded incrementally. When local data needs to be updated it is only necessary to download the versions that were released since the last data extraction. If the content and history is stored in a source code management tool like *git* or *Subversion* this is done automatically using these tools.

While there already exists tools to extract history information from source code repositories, they are not necessarily suited for our analysis. For example *CSVAnalY*[1], from the *MetricsGrimoire* [67] toolset, works by creating a database schema summarizing the source code repository history. It is however not well suited to retrieving the content of specific file versions. Moreover it is also tied to source code repositories, such as *git* and *Subversion* repositories. In our case we are interested in packages that may be stored inside such repositories but not necessarily. It is thus simpler and more reliable to directly use *git* to retrieve the information we need.

Finally some data pieces must be downloaded completely each time the local data is updated. One particularity of this kind of information is that it can be stored historically or not. In the first case it means the local data will contain different versions of the information representing its state at a given point in time. An example is the results of the *R CMD check*. It is a web page[2] containing the output of the tool run regularly on the ecosystem and for which there is no available public archive of previous results.

In the second case only the last version of the information is kept locally if there is no need to keep it historically. It could also be an issue to retain its history if it would use too much disk space due to the size or update frequency of the information. An example of such a data type is a list of repositories available for download.

4.4 Data analysis

Data analysis consists of all transformations that need to be applied to the data after extraction in order to produce meaningful results. These transformations are independent from the data source and must only rely on the data extracted during the extraction step. These transformations can require an important amount of computation and can output some additional data containing the result of these computations. This ensures that the original data is always available and not overwritten. Multiple iterations of

[1] `https://github.com/MetricsGrimoire/CVSAnalY`

[2] `https://cran.r-project.org/web/checks/check_summary.html`

data analysis can be implemented in order to allow an analysis to rely on data produced by another transformation.

In the context of this dissertation various kind of data analysis have to be considered. First simple computations such as data aggregation have to be performed. For example this can consist in aggregating the value for a metric over time for different snapshots of a package distribution. Secondly we can also perform more complex operations such as parsing the code of all package versions of the ecosystem and analyzing it; or running an external tool on each package.

Finally, a common type of analysis is statistical methods and tests. Because many of the subsequent research questions require the study of time-dependent data, we resort to the statistical technique of *survival analysis* [88, 89, 140] to be able to answer research questions related to the introduction and survival of a given event in the ecosystem.

Survival analysis is used to create a model estimating the survival function, which is the probability that an event does not occur after a certain point in time, in a given population. This technique takes into account the fact that the studied event might not occur during the observation period. It models the time it takes for events to occur and allows to take into account so-called *right-censored* data, for which it may be unknown whether the event occurred or not because it has not yet occurred or the subject has "disappeared". For example, if we study the survival of all packages during a given period, we do not know which of these packages may have become inactive after the end of the period of study. Survival analysis allows to take into account those packages for which we don't know whether the event we are interested in happened.

The *survival function* models the probability of an arbitrary subject in the dataset to survive t units of time after the start of the study. In order to approximate the survival function, we use the Kaplan-Meier estimator [53]. The Kaplan-Meier curves visualize the cumulative probability to survive from time zero. As a result, these curves start at value 1 (100% probability of survival at time zero) and continue to decrease monotonically over time. The probability $\hat{S}(t)$ that an individual is still alive at time t is defined as the probability that it survived at time t_1, at time t_2, and so on until time t. Therefore it is defined as

$$\hat{S}(t) = \prod_{t_i<t} \frac{n_{t_i} - d_{t_i}}{n_{t_i}}$$

where n_{t_i} is the number of individuals for which the studied event have not occurred yet before time t_i, and d_{t_i} the number of individuals for which the

event occurred at time t_i. When right-censoring happens for an individual before a given t_i, it is simply ignored when counting n_{t_i}, the number of individuals still at risk before t_i.

We make use of survival analysis and Kaplan-Meier estimator in Chapters 5 and 7.

4.5 Reporting

Reporting aims at producing meaningful information about the ecosystem. This can simply be generating a few charts, numbers or statistical results. However it can also be much more elaborate like a complete application such as a dashboard. In that case it allows a user to browse data and decide which analysis to perform or results to display. An example of such a dashboard is given in Chapter 9. Reporting can also consist in exporting data to a particular data format such as *CSV*.

This step can also process data but contrarily to the data processing step, the result won't be saved in a data format defined in the framework. Thus deciding if a computation must be included in the data processing step or in the reporting step depends on the type of result output, the size of the processed data and the time required to compute it.

For example complex computations that require aggregating data at the level of the ecosystem generally fit better in the data processing step. In some cases, like a dashboard, it is often not conceivable to include this kind of computation in the dashboard itself. It should instead rely on previously cached data. Another aspect to take into account is the volume of data that can reasonably be stored. Saving the result of a computation for all components might considerably increase the volume of data stored on disk. Deciding whether a computation or analysis better fit in the result output or data processing step requires to make a compromise between longer result output generation time or a bigger volume of data to store.

4.6 Data representation

This section presents the different data formats that our framework has to deal with in the context of the two case studies of this dissertation: R and Debian. These formats can be encountered during data extraction, during data analysis as the input or output of a tool or even during reporting.

4.6.1 Debian control files

An example of meta-data format used by package management systems is the Debian control files. Each Debian package contains a file containing meta-data such as package name, version or dependencies stored as a dictionary with key separated from values by a colon. An example of such a file is provided in Figure 4.2. The different types of keys that can be used and the format used by their respective values is extensively described in the Debian documentation [131].

```
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
```

Figure 4.2: An example of Debian control file from the package xul-ext-adblock-plus.

4.6.2 R DESCRIPTION files

R packages make use of a format inspired by Debian control files. Each R package contains a *DESCRIPTION* file containing meta-data stored in a similar way but with different key names and different conventions for the values. One major difference with Debian control files are that dependencies cannot contain disjunctions of packages. Also, while Debian provides a formal definition for package version numbers, R package version numbers don't. An example of a *DESCRIPTION* file is provided in Figure 4.3.

4.6.3 CUDF

Common Upgradeability Description Format (CUDF) is a file format defined by Treinen et al. [149] to describe update scenarios for package-based software distributions. While it focuses on update scenarios it can represent a list of package versions in a similar way to Debian control file and R DESCRIPTION

```
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 4.3: Example of R DESCRIPTION file from the package SciViews.

files. However contrary to the two previous formats it is not tied to a specific software distribution or language and tries to generalize concepts found across multiple package management systems. One of the strengths of CUDF is that it can be used with existing tools such as comigrate.

```
package: wesnoth
version: 1
depends: libc6 >= 8, libfreetype6 >= 4, libfribidi0 >= 1,
 libgcc1 >= 6, libsdl-image1.2 >= 2, libsdl-mixer1.2 >= 1,
 libsdl-net1.2, libsdl1.2debian >= 3, libstdc++6 >= 5,
 libx11-6 , zlib1g >= 5, wesnoth-data = 1
```

Figure 4.4: Excerpt of a CUDF file for Debian package *wesnoth*

4.6.4 *devtools* remotes

Recent releases of *devtools* have a mechanism[3] to specify the location of a package in a *DESCRIPTION* file either as a tarball hosted on web servers or as a git, Subversion, *GitHub* or Bitbucket repository. This is achieved by specifying the type of the remote followed by a URI identifying the repository and possibly a version.

Here is a list of remotes, taken from the official *devtools* documentation, for different types of supported repositories:

```
Remotes: git::https://github.com/hadley/ggplot2.git
Remotes: bitbucket::sulab/mygene.r@default, dannavarro/lsr-package
Remotes: svn::https://github.com/hadley/stringr
Remotes: url::https://github.com/hadley/stringr/archive/master.zip
Remotes: local::/pkgs/testthat
```

[3]https://github.com/hadley/devtools/blob/master/vignettes/dependencies.Rmd

```
Remotes: gitorious::r-mpc-package/r-mpc-package
```

4.6.5 Output formats

Multiple data formats can be used by the tools we design to export data. Because we perform data analysis, one data format we will often use are tables of data, also known as data frames. This kind of data can easily be exported in a *Comma Separated Values* (*CSV*) file which can easily be imported by external tools. A *CSV* can easily be imported by a relational database system such as *MySQL*. While rather versatile, *CSV* has the disadvantage of not specifying the type of table columns.

A recent initiative to facilitate sharing data frames between different data analysis environments is *feather*[4]. It provides an R package and a Python module for efficiently reading and writing data frames in binary files.

For complex data that can't be formatted as a table, we use compressed *JSON* and *rds* files. *JSON* is a human readable standard for storing objects and *rds* is the format used by R to serialize its objects. While we use *JSON* as much as possible for interoperability, *rds* is used to increase the performance of our tools written in R.

4.7 Examples

In this section we illustrate how our conceptual framework has been implemented to analyze both case studies presented in Chapter 3.

4.7.1 Debian

In Chapter 5 we base our analysis on public archives[5] of daily snapshots of all available Debian packages. For a given distribution and date, a snapshot contains the concatenated control files of the latest version of all packages that were available for that date. Because each package is identified as a project in the distribution and each package version is a single project state, there is no particular need to make a distinction between packages and projects.

A replication package containing the data, scripts and results of the empirical analysis from Chapter 5 is available on *GitHub*[6]. This replication package contains two R packages and a set of scripts. The first package *DebianEvolData* contains code responsible for data extraction alongside the

[4] `https://github.com/wesm/feather`

[5] `http://snapshot.debian.org/archive/debian`

[6] `https://github.com/ecos-umons/DebianCoinstEvol`

most intensive part of the data analysis. The package *DebianEvolAnalysis* contains the least computation-intensive part of the data analysis alongside reporting.

Moreover, to allow one to easily reproduce the analysis without having to extract and process the large volume of data from the Debian archives, we provide in the *GitHub* repository aggregated data files. These data files can be directly used by *DebianEvolAnalysis* to generate the results from Chapter 5.

4.7.2 R

Chapters 6, 7, 8 and 9 all make use of data extracted with *extractoR*, a series of R packages implementing the data extraction and analysis for *CRAN* and *GitHub*.

extractoR.cran is the package responsible for extracting R packages from *CRAN*. Content of all *CRAN* package versions are available at `https://cran.r-project.org/src/contrib`. Similarly to Debian, we define a *CRAN* project as a single package and associate with each version a single state. However because of *GitHub* we have to make a distinction between packages and projects.

Indeed each commit of all *GitHub* repositories may contain one package, multiples packages stored in sub-directories or no package at all. Because package names are defined inside the *DESCRIPTION* file, it is possible for a single *GitHub* repository to have its package name changed over time. Moreover a single package name, as identified by the dependency relationships from the *DESCRIPTION* files, might be found in different *GitHub* repositories.

Given a list of *GitHub* repositories, package *extractoR.github* is responsible for downloading or updating *git* repositories from *GitHub* that contain a package at the root of their master branch. *extractoR.github* also browses the history of all the *git* repositories to find all commits which updated the *DESCRIPTION* file. Thus, each project represents a single *GitHub* repository and each project state a commit that modified the *DESCRIPTION* file inside the associated *git* repository.

Additionally to extracting data, these two packages are also used each time another package needs to access the content of a project. For example *extractoR.description* is responsible for parsing *DESCRIPTION* files. When it parses a *CRAN DESCRIPTION* file of a given project state, it relies on *extractoR.cran* to read this file from the appropriate package version. When it parses a *GitHub DESCRIPTION*, it relies on *extractoR.github* to first checkout the appropriate *git* commit and read the file. Similarly, given a project state, *extractoR.namespace* parses the *NAMESPACE* files of that state and *extractoR.content* parses the R code contained in the project state.

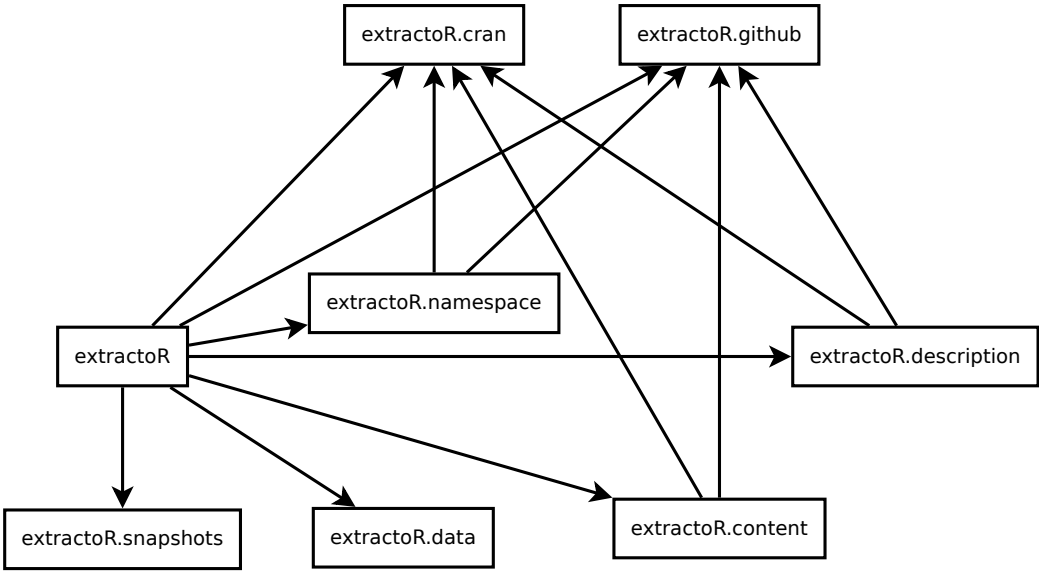


Figure 4.5: Dependency graph of the different packages contained in *extractoR*

Additional packages are contained inside *extractoR*. *extractoR.snapshots* is used to manipulate the daily snapshots extracted from the *CRAN R CMD check*[7]. *extractoR.data* facilitates importing and exporting to various data formats. Figure 4.5 shows how the different packages of *extractoR* depends upon each other.

[7] `https://cran.r-project.org/web/checks/check_summary.html`

CHAPTER 5

Addressing Co-Installability Issues in the Debian Ecosystem

Users and developers of software distributions are often confronted with installation problems due to conflicting packages. A prototypical example of this are the Linux distributions such as Debian. Conflicts between packages have been studied under different points of view in the literature, in particular for the Debian operating system, but little is known about how these package conflicts evolve over time.

This chapter presents an extensive analysis of the evolution of package incompatibilities, spanning a decade of the life of the Debian stable and testing distributions for its most popular architecture, i386. Using the technique of survival analysis, this empirical study sheds some light on the origin and evolution of package incompatibilities, and provides the basis for building indicators that may be used to improve the quality of package-based distributions.

This chapter is mainly inspired by a conference paper [35] presented at the MSR 2015 conference.

5.1 Introduction

An important part of the medatata declared by components, and particularly packages, are the *declared dependencies*, which describe the other packages immediately necessary for its installation and execution. Another important part of the metadata are *declared conflicts* of each package, which describe the immediate incompatibilities with other packages.

In principle one would like all packages to be installable together, and the packaging guidelines of distributions like Debian clearly suggest that conflict declarations should be used sparingly [132]. Nevertheless, there are still many conflicts that may arise [10]: two packages may want to control the same common resource (a library, a configuration file, a network port), two or more packages may provide incompatible implementations of the same functionality, and one can even find special packages (such as Debian package harden-servers) that are used to implement security policies by declaring conflicts with all other packages whose functionality may be abused.

Unfortunately, while the interplay between *declared dependencies* and *declared conflicts*, taken in isolation, makes perfect sense, it may end up breaking many packages, preventing a user from installing together a set of software packages that he needs to use simultaneously [46]. Identifying and resolving these issues is very important when maintaining a package repository. Unfortunately, detecting such incompatibilities due to the interplay between declared dependencies and conflicts is algorithmically hard.

Only recently, efficient algorithms and tools have been proposed for detecting these incompatibilities [156]. One of these tools, known as comigrate has been specifically developed to prevent to a large extent the introduction of such incompatibilities [155]. Nonetheless, after a set of incompatible packages has been spotted, a distribution maintainer is still left with the complex and time-consuming task of finding the right course of action to resolve it: which of the hundreds of dependencies and conflict relations involved in the incompatibility needs to be modified? In which package metadata should one look to find it?

To provide help in this difficult and crucial task, we perform an extensive analysis of a large package-based repository over a significant period of time, and study how package incompatibilities are introduced, evolve, and may get removed. By mining the *history* of the repository, and comparing some of the results with known issues, we are able to provide insight into the characteristics that are statistically significant to pinpoint the packages that are most likely to be problematic.

With this study, we aim to provide a basis for building future indica-

tors and tools that may be used to improve the quality of package-based distributions. To this extent, we focus on the following questions. How can we identify potentially problematic packages in the distribution? When are incompatibilities introduced in, or removed from, packages? What causes (dis)appearance of package incompatibilities?

The case study that we have chosen to carry out such an empirical analysis contains two Debian Linux distributions (stable and testing) for the i386 architecture, over a 10-year time period (starting from January 2005). To the best of our knowledge, this is the first study focusing on the long-term evolution of package incompatibilities in the Debian distribution. We chose this ecosystem as case study because it contains a large number of packages and an extensive history. A replication package containing the data, scripts and results of our analysis is available online[1].

The remainder of this chapter is structured as follows. Section 5.2 presents the research questions and methodology, Section 5.3 reports on our empirical analysis, and Section 5.4 discusses the results. Section 5.5 presents some threats to validity of our research. Finally Section 5.6 concludes.

5.2 Methodology

As mentioned in chapter 3 there are three distinct distributions of Debian: stable, testing and unstable. Because we are interested in studying the evolution of Debian development activity, our empirical study will primarily consider the testing release, as well as its impact on the stable release that is derived from it. Debian unstable is a rolling release distribution containing the most up to dates packages. The testing release corresponds most closely to a development version: package versions contained in it are candidates for the next stable production release.

5.2.1 Mining Strong Conflicts

The Debian package management system relies on metadata stored in control files. Among others, the control file of each package P describes the *direct* relationships with other packages: *dependencies* indicate which other packages are directly needed to perform the installation of P, and *declared conflicts* indicate the packages for which it is explicitly known that they cannot be installed together with P.

In the literature, as well as in this thesis dissertation, the term *strong conflict* is used to indicate that two (or more) packages can never be installed

[1] `https://github.com/ecos-umons/DebianCoinstEvol`

together, independently from what is explicitly declared as a conflict in their metadata [46]. In addition, we use the term *conflicting package* to refer to a package that has at least one *strong conflict* with another package.

It is important to stress that *strong conflicts* are not necessarily "bad": many packages may not be installable together "by design". But if such conflicts are not reported explicitly as *declared conflicts*, they should still be considered as "problematic": a user may be unaware of the impossibility to install both packages together, and during package evolution new and unexpected indirect *strong conflicts* may arise without the package maintainers being aware of them.

For the Debian i386 testing and stable distributions we have extracted daily snapshots during the almost 10-year period from 12 March 2005 (>14K packages) until 6 January 2015 (>42K packages). For each daily snapshot, we only considered packages included in the official Debian distribution. We excluded from our analysis those packages that belong to the contrib or non-free category. These repositories contain packages that are not fully compliant with the *Debian Free Software Guidelines* or depend upon such package.

A major problem when analyzing package *strong conflicts* is the sheer size of the package dependency graph: there are literally tens of thousands of different packages with implicit or explicit dependencies to many other packages. As an example, the full graph for the Debian i386 testing distribution on 1 January 2014 contained 38,411 packages, 181,265 dependencies, 1,490 *declared conflicts* and 49,026 *strong conflicts*.

Vouillonet al. [156] addressed this problem by proposing an algorithm and theoretical framework to compress such a dependency graph to a much smaller one with a simpler structure, but with equivalent co-installability properties, which is called a *co-installability kernel*. The idea is that sets of packages are bundled together into an equivalence class if all packages in the set do *strongly depend* with one another, while the collection of other packages, with which they *strongly conflict*, is the same. Applying this algorithm to the Debian i386 testing distribution on 1 January 2014 results in 994 equivalence classes, and 4,336 incompatibilities between these equivalence classes.

The coinst tool (`http://coinst.irill.org`) was developed specifically for extracting and visualizing co-installability kernels for GNU/Linux distributions. We used the output of this tool as the basis of our analysis.

For each daily snapshot, we used R scripts to browse and extract all names of packages contained in the *main* archive area (i.e., belonging to the official Debian distribution)[2]. To retrieve the information about the co-installation

[2]The information for a given snapshot date <*DATE*> (using the format *YYYYMMDD*) is available on `http://snapshot.debian.org/archive/debian/<DATE>T060000Z/`

conflicts of these packages we used JSON output files generated by coinst with the command

```
coinst -conflicts conflicts -stats -o graph.dot Packages.bz2 >& log
```

Previous research used *strong conflict* graphs to determine appropriate solutions to package co-installation problems. These solutions, however, did not take into account the evolution over time of these *strong conflicts*. In our current work, we aim 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.

5.2.2 Research Questions

We address each of the following research questions, in separate subsections. Answers to these questions can help, at the longer term, to come up with quality indicators and tool support for dealing with *strongly conflicting* packages.

RQ_1 How can we identify problematic packages in the distribution?

RQ_2 How long does it take before a *strong conflict* is introduced in a package?

RQ_3 What is the effect of *strong conflicts* on the longevity of packages?

RQ_4 How long does it take before all conflicts get removed from a *strongly conflicting* package?

RQ_5 What causes frequent appearance and disappearance of *strong conflicts*?

All survival analysis results produced in this chapter were obtained using R scripts that relied on the R package *survival* for computation and on the R package *ggplot2* for visualization.[3]

5.3 Results

5.3.1 Overall Characterization

Let us start by presenting some plots and descriptive statistics characterizing the evolution of *strongly conflicting* packages belonging to the Debian stable and testing distributions.

dists/testing/~\main/binary-i386/Packages.bz2

[3]See cran.r-project.org/web/packages/survival and cran.r-project.org/web/packages/ggplot2.

Figure 5.1 compares the daily evolution of the total number of packages (in blue) against the number of *strongly conflicting* packages (in red). The evolution of the stable distribution (solid lines) clearly shows "plateaus" that start at the moment of a major public release of a new Debian version. This is quite normal, as the stable version of Debian is only allowed to incorporate security-critical changes after a release.

The testing distribution (dotted lines) is more interesting: the development process leads to a general linearly increasing trend, with some periods of stability or light decrease that start at the official freeze date of the testing distribution (dotted vertical lines), and end at the official date of the next stable public release (solid vertical lines). During these freeze periods only bug fixes are allowed or packages can be removed, while it is generally forbidden to add any new package or package version to the testing distribution.

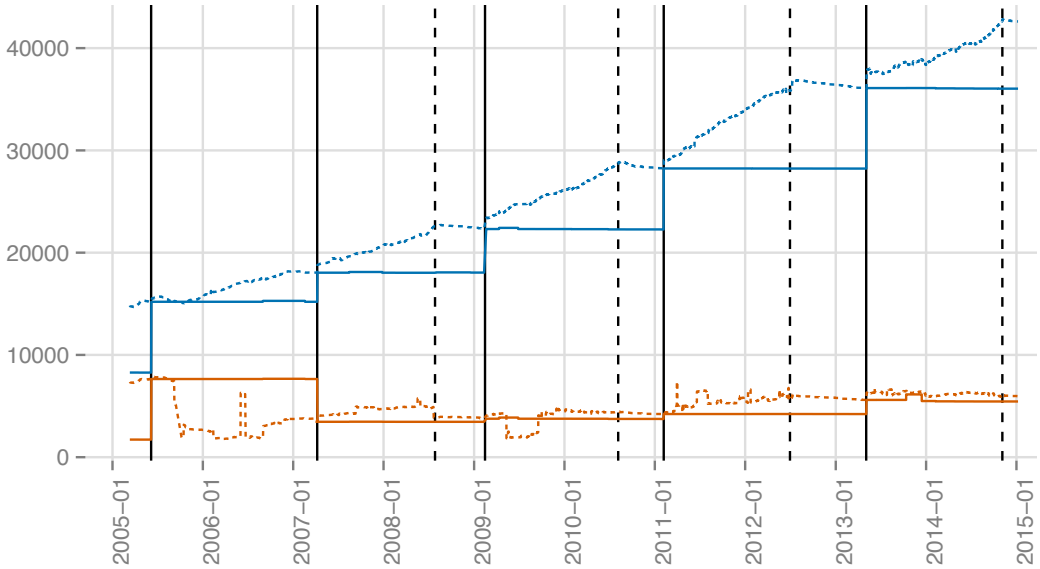


Figure 5.1: Daily evolution of the total number of packages (in blue) and *strongly conflicting* packages (in orange) for the testing distribution (dotted lines) and stable distribution (solid lines) of Debian. Solid vertical black lines correspond to official dates of a stable public release. Dotted vertical black lines correspond to the freeze dates of the testing distribution preceding the stable release.

Figure 5.2 shows the evolution over time of the ratio of the number of *strongly conflicting* packages in a snapshot over all packages in that snapshot. For the testing distribution (dotted blue lines) we observe that, starting from 2007 and with only a few exceptions, this ratio remains between 15% and 25%. We also observe a slight decrease over time, despite the fact that the

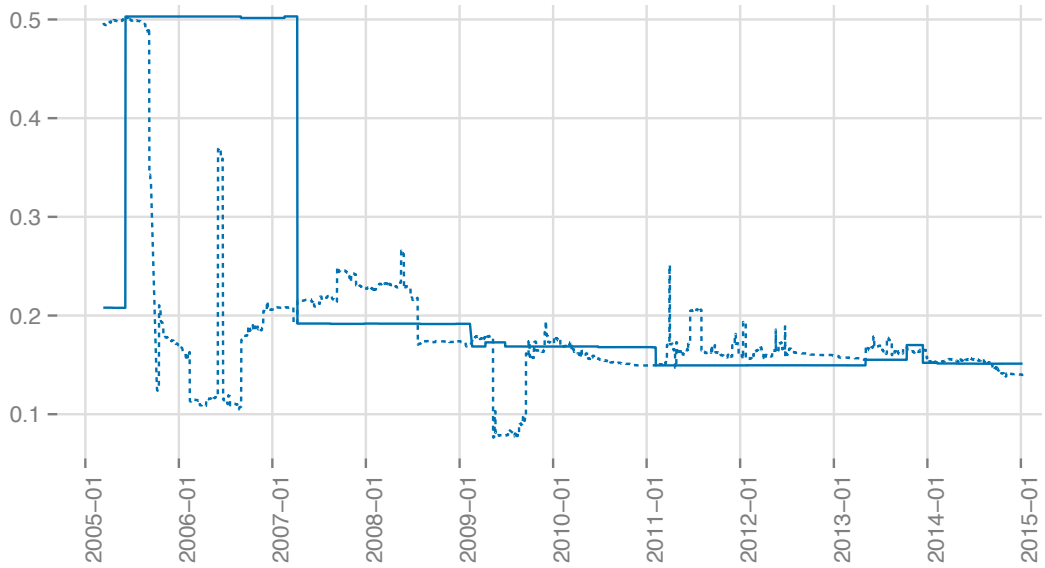


Figure 5.2: Ratio of *strongly conflicting* packages in snapshots of the testing distribution (dotted blue lines) and the stable distribution (solid blue lines).

number of packages continues to increase with each new major release: this corresponds to the fact that the Debian community actively works to keep *strong conflicts* at a minimum. For the stable distribution (solid blue lines) we observe the same evolutionary behavior, combined with the presence of the "plateaus" corresponding to different public releases of Debian that were also found in Figure 5.1. Finally, for the testing distribution we observe 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. This will be the subject of deeper investigation in RQ_1.

Figure 5.3 displays, daily snapshots of the testing distribution, the relative number of *strong conflicts* per package. Most of the time there are between 2,000 and 3,000 packages with exactly one *strong conflict*. This corresponds to a ratio of about 50% of all *strongly conflicting* packages. There are much less packages having two *strong conflicts*, and even less with three *strong conflicts* or more.

Figure 5.4 displays the same information but for the stable distribution. Again we observe the familiar "plateaus" and a ratio of between 50% and 70% of all conflicting packages that had only one *strong conflict* for the considered daily snapshots.

Figure 5.5 visualizes the age of the packages present in the Debian testing distribution on 6 January 2015. There are in total 42,603 such packages (out of a total of 67,748 packages that existed at some time during the entire

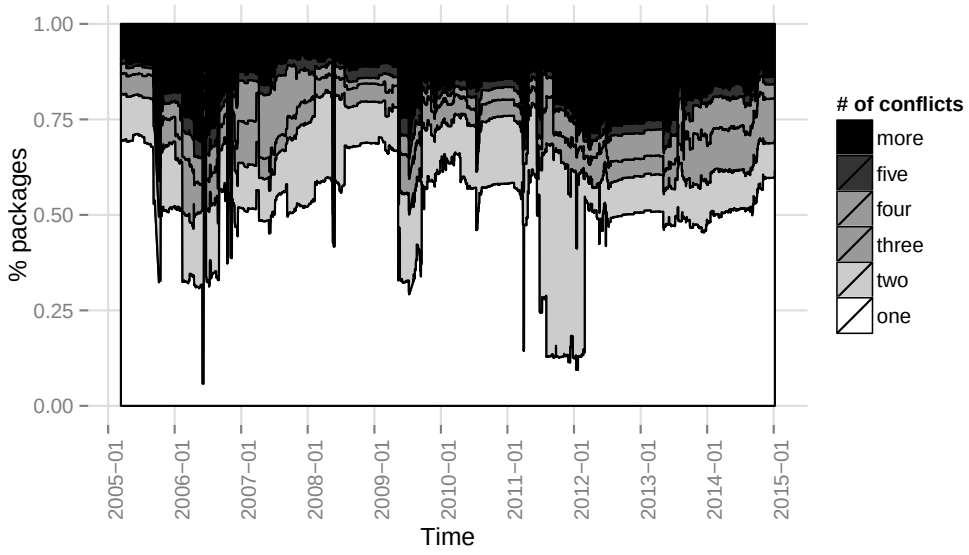


Figure 5.3: Daily evolution of the number of packages in the testing distribution having a *strong conflict* with 1, 2, 3, 4, 5 or >5 packages.

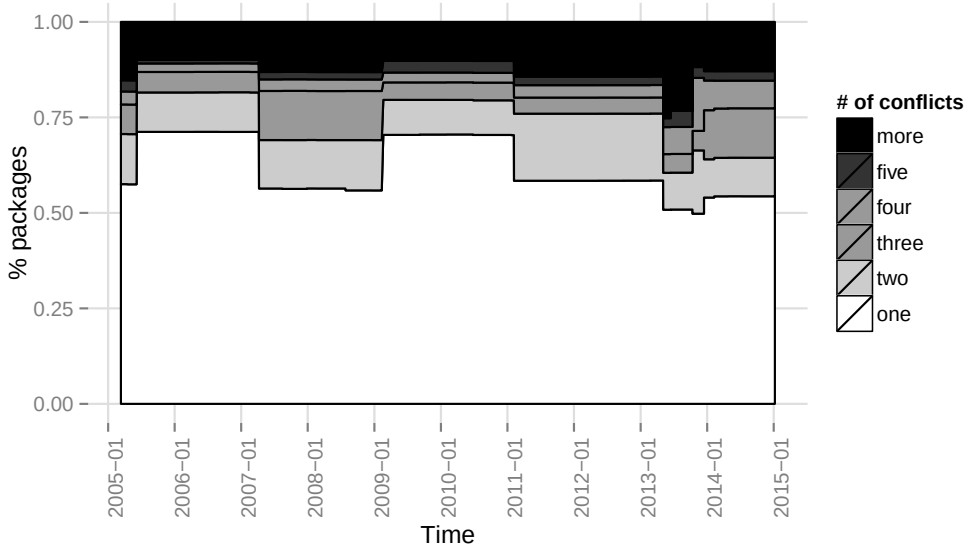


Figure 5.4: Daily evolution of the number of packages in the stable distribution having a *strong conflict* with 1, 2, 3, 4, 5 or >5 packages.

considered period). Gaps in the histogram are caused by the freeze periods during which addition of new packages is not allowed. The peak on the right represents all packages that have been there since the beginning of the considered period. It corresponds to 15.8% of all packages in the distribution as of 6 January 2015.

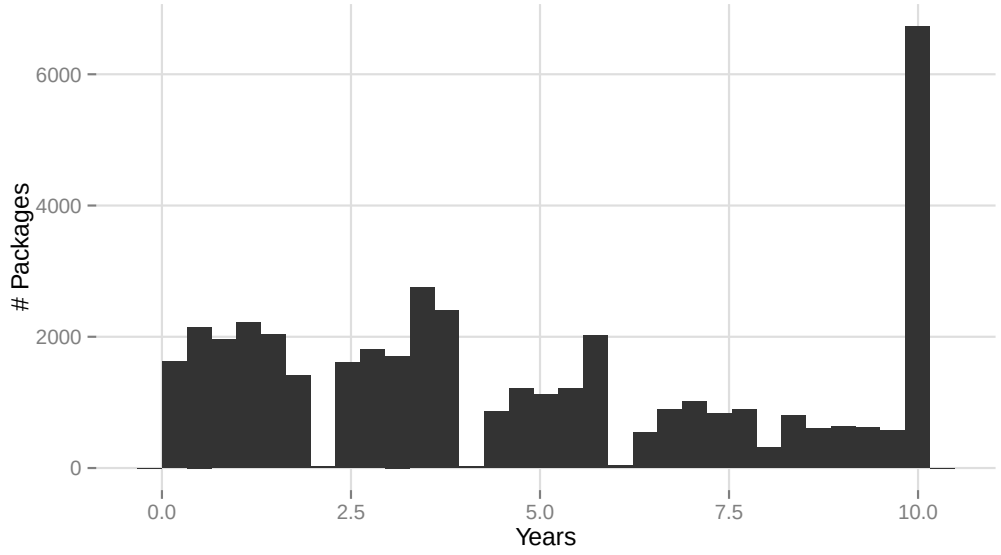


Figure 5.5: Age (in years) of packages that were present in the Debian testing distribution on 2015-01-06.

Among all packages considered in Figure 5.5, let us focus on only those 16,101 packages that had a *strong conflict* at least once in their lifetime. Figure 5.6 visualizes the number of conflicting days for these packages as a percentage of their total lifetime. We observe that 6,063 (i.e., 37.66%) packages were almost never conflicting (<5% of the time). Another peak is observed at the other side of the spectrum, were we find 21.28% of all packages (3,427 in total) that had at least one *strong conflict* >95% of the time. More specifically, 18.7% of all considered packages (3,009 in total) had *strong conflicts* during their entire lifetime.

Figure 5.7 shows the same information as Figure 5.6, but for the stable distribution. Unsurprisingly, because packages in the stable distribution tend to be stable, *strongly conflicting* packages in this distribution tend to remain in conflict during their entire lifetime.

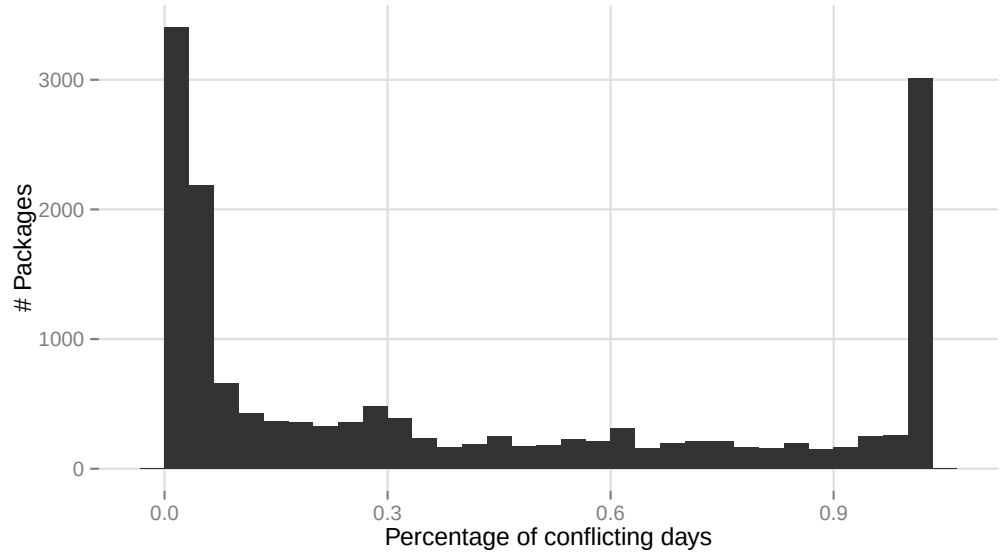


Figure 5.6: Ratio of days that *strongly conflicting* packages in the Debian testing distribution on 2015-01-06 were in conflict previously.

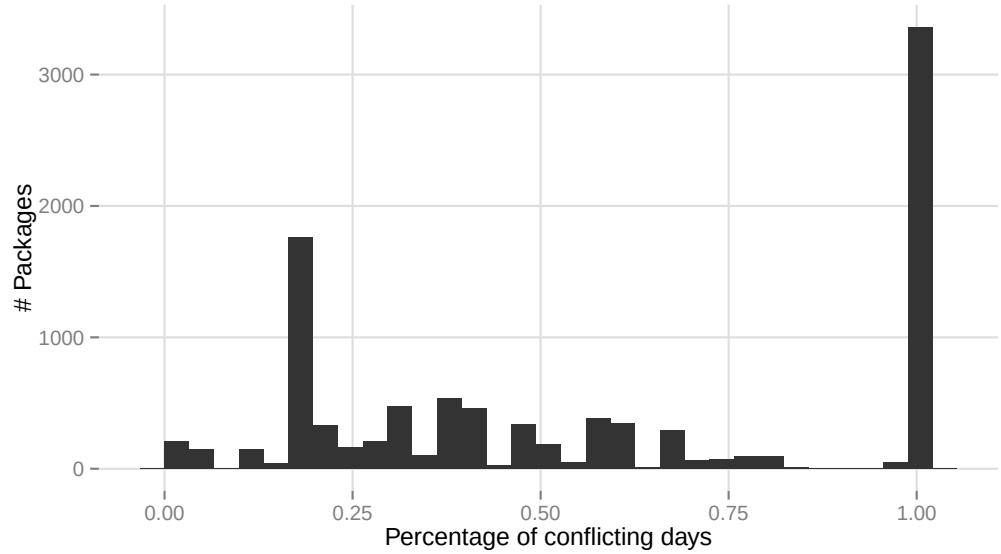


Figure 5.7: Ratio of days that *strongly conflicting* packages in the Debian stable distribution on 2015-01-06 were in conflict previously.

5.3.2 How can we identify potentially problematic packages?

As previously discussed, some of the conflicts present in the repository are there by design, but others are unjustified and 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 an extensive amount of time. With this research question, we look for a way of automating the detection of potentially problematic packages, and reducing the amount of effort needed to nail down real issues.

a) Aggregate Analysis

A natural approach to identify potentially problematic packages is to look for trend breaks in the evolution of the absolute or relative number of *strongly conflicting* packages in the distribution: sudden increases in their number is a clear hint that a problematic package has appeared, and sudden decreases indicate that a problematic package has been fixed. Many discontinuities are clearly visible in Fig. 5.1 and 5.2, with peaks ranging from a few hundreds to over 4000 *strong conflicts*.

We retrieved all trend breaks that added at least 500 *strong conflicts*, using the coinst-upgrade tool described in [155] that is able to identify the root causes for the changes in conflicts between two repositories. We then 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.

The result of this analysis is summarized in Table 5.1. For each problematic trend break, we report the date of the trend break, the number of new *strong conflicts* that were introduced at that date, the main root cause of the problem, the number of days it took to fix the problem, and the number of *strong conflicts* that were resolved by the fix. We also report whether the root cause of the problem would have been prevented by using one of the more recent tools comigrate [157] and challenged [5] that have been developed to improve the quality assurance process.

From Table 5.1 we observe 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 tool if it would have been available at that time, and one issue could have been anticipated using the challenged tool.

Interestingly, a few relevant trend breaks *are not identifiable by any of*

the existing tools, while a check for trend breaks in the aggregate analysis (as done here) would have drawn attention to them. This provides evidence of the added value of our approach.

b) Individual Analysis

Once a trend break has been spotted, one still needs to identify manually which are the potentially problematic packages. This process can be automated by studying their characteristics related to *strong conflicts* by resorting to three simple metrics for each package:

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

The motivation for choosing these simple metrics is that one should 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. Indeed, the latter case usually corresponds to transient problems, like the *day-flies* that we were able to identify in the previous aggregate analysis.

After ordering the packages with respect to these three metrics, we obtain a list of potentially problematic packages, of which we presented the first lines in Table 5.2. Interestingly, we find back most of the packages that were already identified during the aggregate analysis (see Table 5.1), with the important advantage that the proposed metrics can be computed fully automatically, and do not require any manual inspection.

Trend breaks	Start date	Days to fix	Main root cause (manually identified)	Tool able to detect	Relevance
+4379/-4201	2006-06-02	19	updated x11-common conflicts with videogen	comigrate	medium
+2364/-2371	2011-03-30	1	new libgdk-pixbuf* conflicts with libgtk2.0-0	*this chapter*	medium
+1658	2009-09-16	*not fixed yet*	new liboss-salsa-asound2 conflicts with all alsa tools	*this chapter*	minor
+1279/-809	2005-10-15	120	reinstallable cdebconf conflicts with debconf	*this chapter*	**serious**
+1268/-1270	2012-01-12	10	updated initscripts conflicts with sysklogd	comigrate	**serious**
+1188/-2442	2006-09-01	984	updated python conflicts with ppmtofb	challenged	minor
+1025/-1282	2011-06-19	45	updated initscripts conflicts with selinux-policy-default	comigrate	**serious**
+859/-1126	2012-06-23	1	new libopenblas-base conflicts with libatlas3gf-*	*this chapter*	medium
+763	2011-04-26	*not fixed yet*	updated libsdl1.2debian conflicts with liboss-salsa-asound2	comigrate	minor
+758/-756	2012-05-18	1	updated netbase conflicts with ifupdown	comigrate	**serious**
+727	2013-05-05	*multiple dates*	new libopenmpi1.6 conflicts with libopenmpi1.3	comigrate	medium
same	*same*	*multiple dates*	less conflicts with man	comigrate	**serious**
+706/-732	2008-05-17	11	updated libldap-2.4-2 conflicts with libldap2	comigrate	minor
+682/-1074	2007-09-10	316	updated libpam-modules conflicts with libpam-umask	comigrate	minor
+633/-577	2013-07-26	19	updated initscripts conflicts with bootchart	comigrate	minor
+632	2007-04-08	*multiple dates*	new package libgif4 conflicts with libungif4g	*this chapter*	minor
+536/-558	2011-03-21	31	new packages libhttp-* conflicts with libwww-perl	*this chapter*	medium

Table 5.1: Aggregate analysis of trend breaks and their manually identified root cause. The first column displays +n/-m where n is the number of conflicts introduced by the trend break and m the number of resolved conflicts when the root cause is fixed.

Potentially problematic package	minimum conflicts	maximum conflicts	conflicting days over mean
libgdk-pixbuf*2.0-0*	0	675	1349
libgdk-pixbuf*2.0-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
libldap2	0	719	546
libaws2.2	0	719	546
libaws-bin	0	2247	315
libhugs-ldap	0	2620	44
bootchart	0	598	31
libopenblas-base	0	1171	28
systemd-sysv	5	1166	28
qtchooser	0	1166	28
libopenblas-dev	0	1166	28
ifupdown	0	598	26

Table 5.2: Top 20 of potentially problematic packages identified by three simple metrics. Packages listed in **boldface** also appear as a root cause in Table 5.1.

5.3.3 How long does it take before a *strong conflict* is introduced in a package?

For our second research question, we are interested in the first time a *strong conflict* appears in a package. We hypothesize that newly introduced packages have a high likelihood of introducing *strong conflicts*.

To verify this, we have to exclude all packages that were already present at the first day of the considered period for which we have data, since we have no way of knowing when a *strong conflict* first appeared in them. This leaves us with 54,988 packages that are newly introduced somewhere during the considered time-frame.

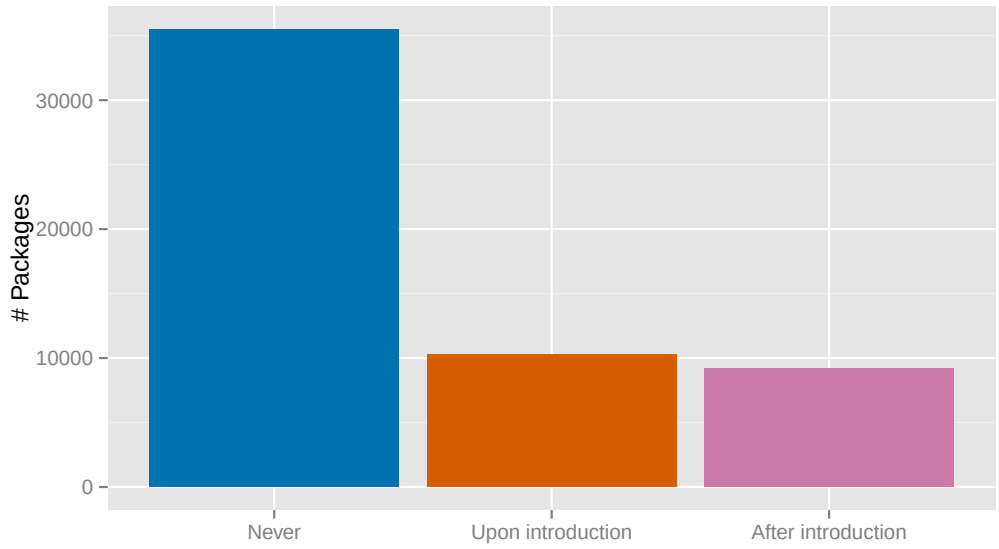


Figure 5.8: Number of newly introduced Debian packages, classified according to when the first *strong conflict* was introduced for that package: never, upon package introduction, or after package introduction.

These packages can be classified into three different categories, summarized in Figure 5.8 and discussed below.

1. Most new packages (64.59%, corresponding to 35,516 packages) **never** encounter a *strong conflict*.

2. For the 19,472 packages (i.e., 35.41%) that do encounter a *strong conflict*, in the majority of the cases (52.91%, corresponding to 10,302 out of 19,472 packages) a *strong conflict* is already present **at the moment of introduction of the package**.

3. For the remaining 9,170 *strongly conflicting* packages, a *strong conflict* was introduced at least one day (but often much later) **after package introduction**. The distribution of the number of days before the first *strong conflict* is introduced has a median value of slightly below one year (326 days to be precise) and follows a decreasing trend (see Figure 5.9).

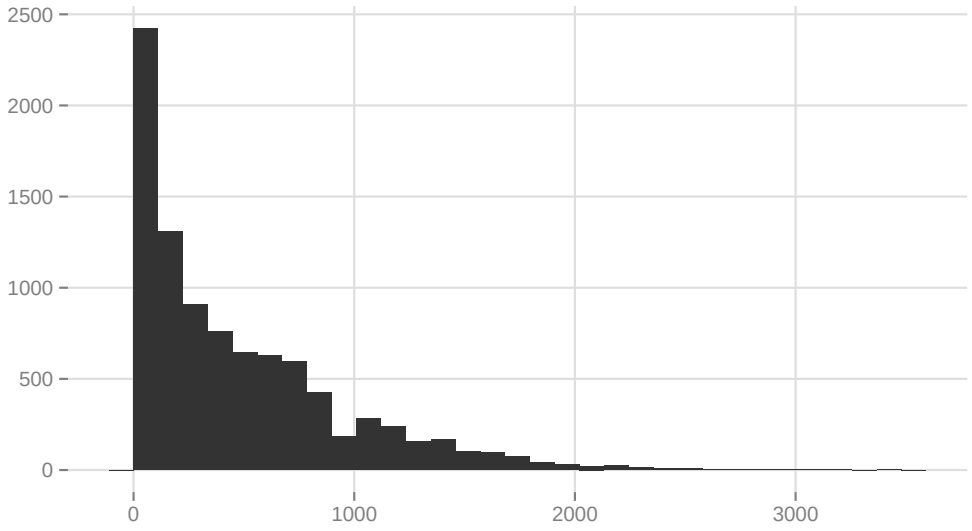


Figure 5.9: Frequency distribution of the number of days (x-axis) before *strong conflicts* arise in newly introduced packages. Packages without *strong conflicts* or containing *strong conflicts* at the day of their creation are excluded.

It is important to note that the results in Figure 5.9 are an under-approximation, since packages that have not encountered a *strong conflict* during the considered period may still become *strongly conflicting* in the future. Survival analysis takes into account this probability. Figure 5.10 shows the Kaplan-Meier curve. It shows the cumulative probability $S(t)$ that a package stays without conflicts for at least t years. The curve shows that a package has around 80% of chance of never gaining any conflicts in its first 10 years of existence. Moreover, as the curve appears to converge and because of its shape, the longer a package has survived without *strong conflicts*, the less likely it becomes that a *strong conflict* will appear.

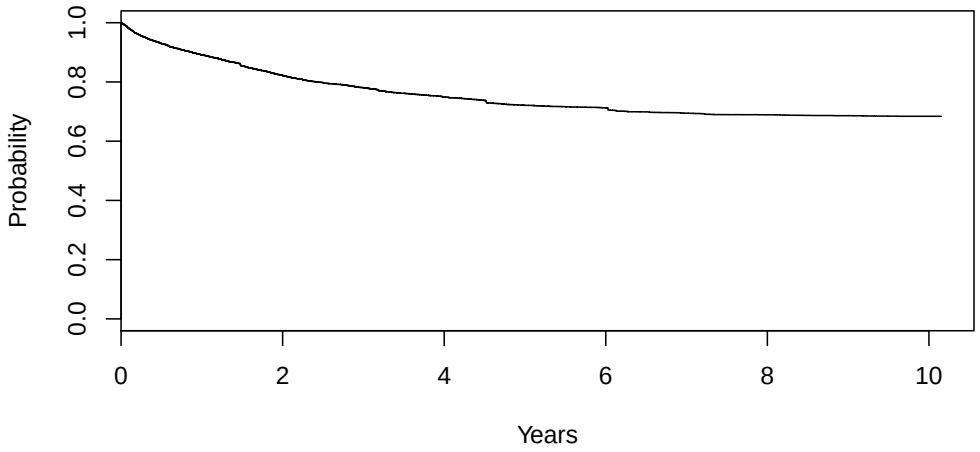


Figure 5.10: Kaplan-Meier curve for the introduction of *strong conflicts* in non-conflicting packages. The time scale on the x-axis is expressed in number of years.

5.3.4 What is the effect of *strong conflicts* on the longevity of packages?

First, we study whether the *absence* of *strong conflicts* upon *introduction* of a package has an effect on its longevity. For the same reason as in RQ_2 we use survival analysis to answer this question. We analyze only those 54,988 packages that are newly introduced after the beginning of the considered period, because we cannot know the age of the other packages. Figure 5.11 shows the Kaplan-Meier curves for the cumulative probability of the survival function. The green curve shows the survival probability for packages without *strong conflicts* upon introduction, the red curve shows the probability for packages *strong conflicting* at the time of package introduction.

We used the survdiff function from the R package survival to test for difference with statistical significance between two survival distributions. This function implements the G^ρ family of non-parametric tests [72]. If $\rho = 0$ (as in our case), this becomes a log-rank test, also known as a Mantel-Haenszel test [110, 129]. Using this test, we found that packages for which a *strong conflict* has been introduced after introduction of the package live longer than packages that already had a *strong conflict* upon introduction. When looking at the figure, however, the difference is fairly small, and becomes smaller as

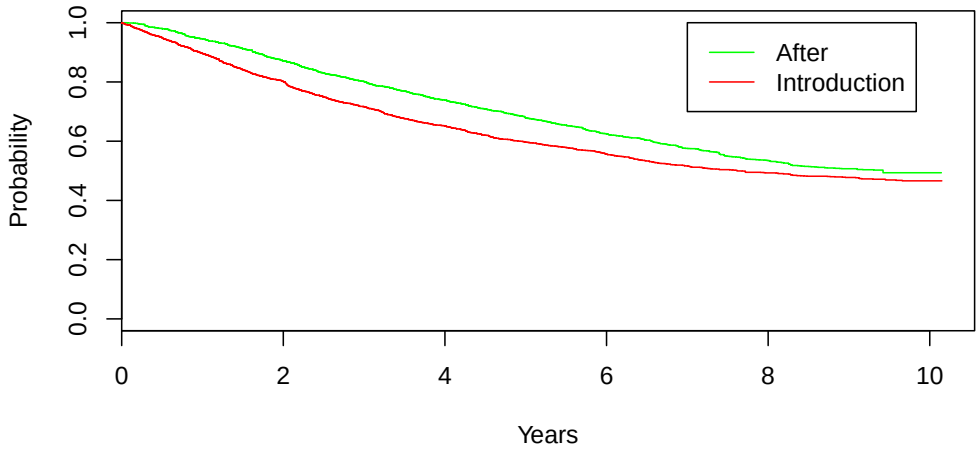


Figure 5.11: Kaplan-Meier curves of the longevity (in years) of Debian testing packages with *strong conflicts* upon (in red) or after (in green) the time the package got introduced.

the package survives longer.

Secondly, we study whether the *absence* of *strong conflicts* during the entire observed lifetime of a package has an effect on its longevity. Figure 5.12 shows the Kaplan-Meier curve for the survival probabilities. Again, a log rank test reveals a difference with statistical significance: packages suffering from *strong conflicts* during their lifetime tend to live longer than packages without *strong conflicts*. This difference is in the opposite direction of what one would intuitively expect. When looking at the figure, however, the observed difference appears to be negligible.

Thirdly, we compare the longevity of packages that were *strongly conflicting* during their *entire* lifetime (i.e., 100% of the time) with packages that only had *strong conflicts* occasionally (<100% of the time). Figure 5.13 shows the Kaplan-Meier curve for the survival probability. Again, a log rank test reveals a difference with statistical significance: packages that are *strongly conflicting* occasionally tend to live longer than packages that are *strongly conflicting* during their entire lifetime. In this case, the difference is much more pronounced. Nevertheless, a package which is conflicting during its entire lifetime has still more than 25% probability to survive for more than 10 years.

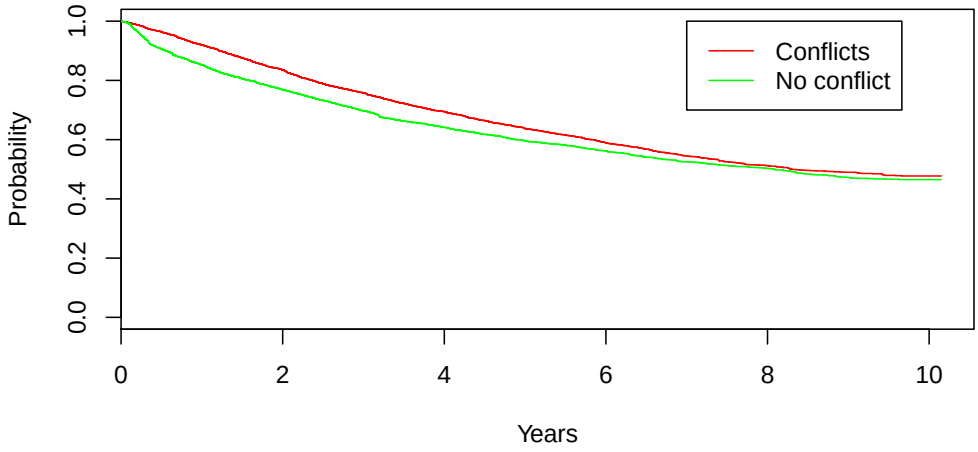


Figure 5.12: Kaplan-Meier curves of the longevity (in years) of Debian testing packages without (in green) or with at least one *strong conflict* (in red) during their lifetime.

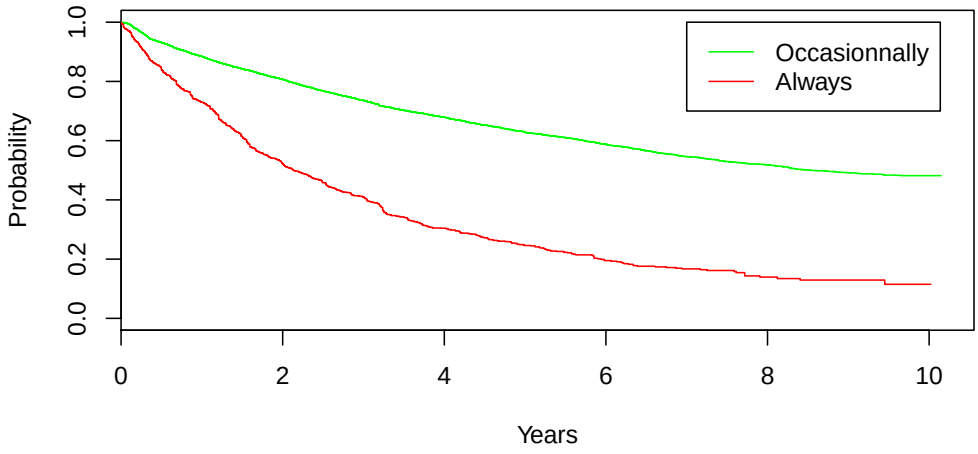


Figure 5.13: Kaplan-Meier curves of the longevity (in years) of Debian testing packages with occasional *strong conflicts* (green) versus packages with *strong conflicts* during their entire lifetime (red).

RQ_4 How long does it take before all conflicts get removed from a *strongly conflicting* package?

This question is the counterpart of question RQ_3 where we studied how long packages survive. With RQ_4 we analyze how long *strong conflicts* survive. For this analysis, we do not include those packages that were already *strongly conflicting* at the beginning of the considered period. We therefore exclude 220 packages that already existed at the beginning of the studied period, that still existed at the end of the considered period, and that contained *strong conflicts* all their lifetime. Because of this exclusion, we might slightly underestimate the probability for a *strong conflict* to be long-lived.

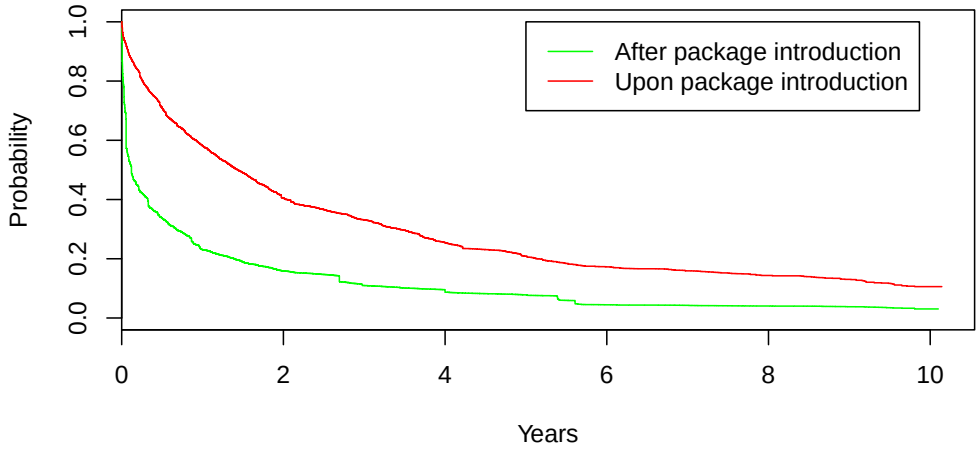


Figure 5.14: Kaplan-Meier curve of the probability (over time) for all *strong conflicts* to get removed from packages.

Figure 5.14 presents the Kaplan-Meier curve of the probability $S(t)$ of a package to remain *strongly conflicting* for at least t years. We make the distinction between *strong conflicts* that were introduced *upon* package introduction and those that were introduced *after* package introduction. The survival probability for the latter starts with a steep descent. Indeed, most *strong conflicts* introduced after package introduction do not last very long: 50% of them stay less than 24 days. In contrast, 50% of the *strong conflicts* that were already present upon package introduction stay more than 11 months! Similarly, *strong conflicts* added upon package introduction have a 15% probability to survive at least 10 years, while those added after package

introduction have less than 5% probability of surviving 10 years or more.

Even if most *strong conflicts* are short-lived, some packages might continue to have *strong conflicts* for a long time, and it may not be possible to remove these conflicts. An example of such a package is courier-imap, which provides an IMAP mail server and which is in conflict with any other package providing an IMAP server.

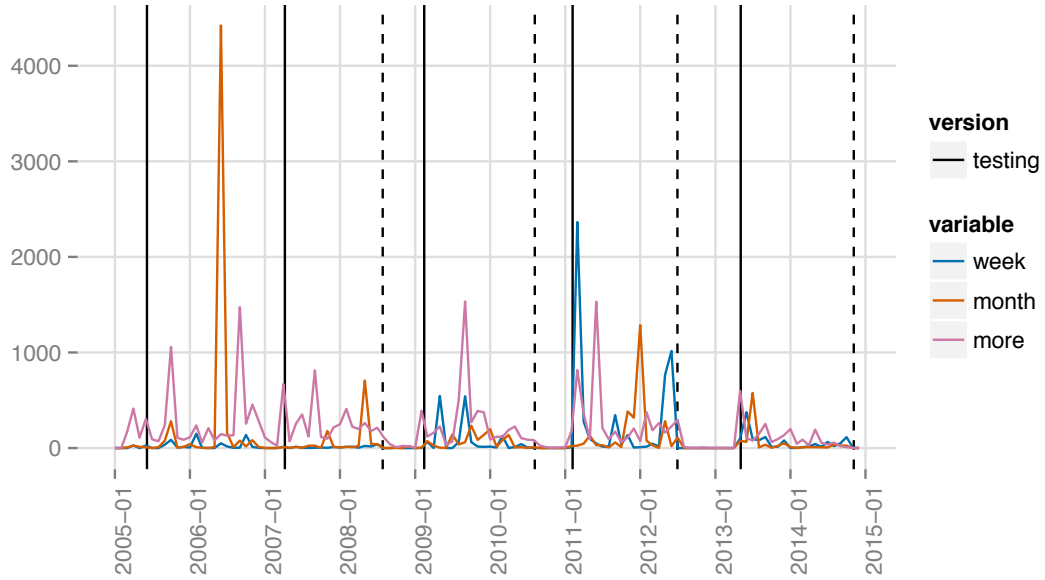


Figure 5.15: Number of Debian testing packages for which at least one *strong conflict* got introduced and for which all *strong conflicts* were removed after, respectively: less than one week (blue); between one week and a month (red); more than a month (purple).

Because of the short-lived nature of *strong conflicts*, we analyze the history of the conflict resolution times in Figure 5.15. As in Figure 5.1, vertical lines indicate the start date and end date of each *freeze* period. Regardless of the resolution time, we observe that *strong conflicts* do not get introduced during freeze periods. This is indeed what one would expect, since the freeze periods are meant to fix bugs and resolve problems, rather than introducing new problems. When comparing the dates of *strong conflict* introduction for those packages with short resolution times (less than a week) to those packages with longer resolution times (more than a week), we cannot reveal any specific pattern. Except perhaps for the fact that, since 2011, the introduction of *strong conflicts* in packages with short resolution times tends to be concentrated just before or just after a *freeze* period.

RQ_5 What causes frequent appearance and disappearance of *strong conflicts*?

We now focus on the events that cause a package to become *strongly conflicting* or to loose all of its *strong conflicts*.

During the considered period, there were 26,266 packages that became *strongly conflicting* 49,768 times. Similarly, there were 25,178 packages that lost all their *strong conflicts* 51,248 times.

$< 50\%$	60%	70%	80%	90%	100%
1	2	2	3	4	20

Table 5.3: Distribution of the number of times each package became *strongly conflicting*.

$< 50\%$	60%	70%	80%	90%	100%
1	2	2	3	4	21

Table 5.4: Distribution of the number of times each package lost all its *strong conflicts*.

Tables 5.3 and 5.4 show that most packages became *strongly conflicting* or lost all their *strong conflicts* only once, while for only very few packages this happened many times (up to respectively 20 and 21 times). We manually analyzed the packages with most repeated *strong conflict* additions and removals: erlang, openoffice.org-thesaurus-en-us and a few related packages. The explanations we found for these frequent state changes are twofold.

A first reason is that new versions of related packages can get introduced in the testing distribution at slightly different times. This introduces temporary incompatibilities because there is no explicit dependency between the involved related packages. The old Debian migration tools could not cope with these situations, while the more recent comigrate tool would prevent this. This happened twelve times for the packages erlang and erlang-doc-html, and four times for the packages openoffice.org-thesaurus-en-us and openclipart-openoffice.org (later renamed openclipart-libreoffice).

A second reason for repeated addition and removal of *strong conflicts* is that some packages have a large number of dependencies, and are hence more likely to be impacted. This was especially the case for OpenOffice packages, but also happened for erlang that depends on initscripts which got transient *strong conflicts* three times.

5.4 Discussion

With RQ_1 we have shown that a simple approach based on monitoring trend breaks in the number of *strong conflicts* present in the distribution is able to identify several significant disruptions in the past history of Debian packages. Manual inspection of these issues revealed that most of them uncover medium to serious issues in the quality of the repository, as summarized in Table 5.1. Many of these issues would have been prevented by using recent tools like `comigrate` [157] and `challenged` [5], which are now being gradually introduced in the Debian QA process. This constitutes strong evidence of the relevance of these tools, which may be adapted to other kinds of repositories. We also showed that some of the uncovered issues would not have been captured by any of the existing tools, while a simple check for sudden increases in the number of *strong conflicts* would spot them. This provides strong motivation for adding such a check in Debian's QA process, and more generally to the QA process for all GNU/Linux distributions.

For questions RQ_2, RQ_3 and RQ_4 we studied the relation between the presence of *strong conflicts* on the longevity of packages. To this extent we made use of the statistical technique of survival analysis.

RQ_2 revealed that, for all packages in the Debian `testing` distribution that were newly introduced during the considered analysis period, *strong conflicts* only occurred in about one third of them (35.41%). We also observed that, the longer a package has survived without *strong conflicts*, the less likely it becomes that *strong conflicts* will appear.

With RQ_3 we assessed the effect of *strong conflicts* on the longevity of packages. Packages that were introduced conflict-free tend to live longer than packages that already had a conflict at the moment they were introduced, but the observed difference is small. For those packages where *strong conflicts* did occur, in roughly half of the cases *strong conflicts* were already present at the moment of package introduction.

Occasionally conflicting packages tend to live longer than packages that are always in conflict, with a clear observed difference. Hence, it makes sense to focus on packages that are always conflicting, to detect as early as possible those that need to be dropped.

With RQ_4 we studied the time it takes for all *strong conflicts* in a package to disappear. We observed that for those packages that already had *strong conflicts* upon package introduction, it takes much longer (if at all) before all these *strong conflicts* get removed than for packages that started off without any *strong conflicts*. Although this may seem contradictory at first, it is consistent with the intuition that a *strong conflict* present at the

moment of package introduction may be actually needed to express intended incompatibilities, and does not necessarily represent a real defect. This also explains why many *strong conflicts* never get removed.

We also observed that, if a previously existing package becomes *strongly conflicting*, it often does not take a long time before these conflicts get removed (less than 24 days in half of the cases), which is strong evidence that these conflicts are not intended incompatibilities, but defects in the repository that need to be fixed. Their presence is a clear indication of the need of incorporating better tools in the QA process.

Finally, our analysis of the packages that most frequently switched from conflicting to non conflicting (RQ_5) showed again clearly the need for modern tools like comigrate or an improved version thereof that are able to prevent the appearance of new incompatibilities. Without such tools, several packages get impacted and fixed over and over again in every new version.

5.5 Threats to Validity

The foremost threat to validity relates to *generalizability*. We have restricted ourselves to Debian in this chapter, but the lessons learned from our study of the evolution of package incompatibilities could be applied to other package-based software distributions as well. Such insights, as well as the tools and best practices used for reducing the extent of the problem (e.g., comigrate in the context of Debian) could help maintainers of other Linux distributions to improve upon their practices and increase the quality of their repositories.

In most of our analysis, we had to exclude those packages that already existed before the considered 10-year period, because earlier data is unfortunately no longer available, and those packages that continue to exist after the considered period. If we could include these packages, the obtained results might change. We are fairly confident, however, that the main conclusions of our analysis will remain the same, given the fact that the evolution history over time remained fairly stable.

Our analysis is based on the output produced by the coinst tool. The risk that possible bugs in this tool may affect the outcome of our results is limited because the algorithms underlying coinst have been formally verified in Coq [147], and it has been used repeatedly in the past by different researchers. Moreover, conflicts identified by coinst can be independently checked using other existing tools, like dose-deb-coinstall from the Dose suite used regularly on Debian repositories [7].

Finally, the scripts that we have developed for our empirical analysis may still contain some bugs, and the obtained results may be biased by some

simplifying assumptions we have made during our analysis.

5.6 Conclusion

The incompatibilities among packages known as *strong conflicts* are an important problem in package-based distributions, and have been studied in a series of recent research works [155–157]. Leveraging the coinst, coinst-upgrade and comigrate tools issued from this research work, we empirically analyzed the evolution of *strong conflicts* among packages for all the available history of the Debian package-based software distribution for the i386 architecture, which spans a decade.

While the number of packages in the Debian testing distribution increases linearly, the ratio of packages with *strong conflicts* stays more or less constant, with occasionally important decreases or increases in the number of *strong conflicts*. This reflects the fact that Debian maintainers make a specific effort to reduce *strong conflicts* as much as possible, which must be accepted only when they describe component incompatibilities that cannot be otherwise eliminated.

Using the statistical technique of survival analysis, we investigated the moment and cause of introduction and removal of *strong conflicts* in Debian packages, as well as the relation with the packages' longevity. We found limited evidence that packages containing *strong conflicts* live longer than those without. We also found evidence that:

- packages that are always in *strong conflict* have a smaller survival probability than those who are not;
- the longer a package has survived without *strong conflicts*, the less likely it is that a *strong conflict* will appear;
- *strong conflicts* that are already present upon package introduction tend to stay present much longer than *strong conflicts* that are added later;
- half of the *strong conflicts* that appear after package introduction stay a short amount of time (< 1 month).

These findings confirm the importance of adopting tools and techniques that prevent the introduction of *strong conflicts*. Without these tools, the historical analysis reveals that a lot of defects get regularly reintroduced, with peaks reaching tens of times for the same package.

Using metrics related to the presence, amount and duration of *strong conflicts*, we could identify several packages that have been reported as

problematic by the Debian community in the past. We have shown how various of these issues would have been prevented by using recently developed tools, but several issues spotted by our metrics are not captured by any existing tool. This is a strong motivation for introducing these metrics in the future into the repository quality assurance process. As an added bonus, the simplicity of our metrics makes them easily transposable to other package repositories.

CHAPTER 6

Analyzing the Topology of the R Ecosystem

This chapter explores the ecosystem of software packages for R, one of the most popular environments for statistical computing today. We empirically study how R packages are distributed on the different major repositories: *CRAN*, *BioConductor*, *R-Forge* and *GitHub*. We also explore the role and size of each repository, the inter-repository dependencies, and how these repositories grow over time. With this analysis, we provide a deeper insight into the extent and the evolution of the R package ecosystem.

This chapter is mainly inspired by a workshop paper [43] presented at IWSECO-WEA 2015 and a conference paper [44] presented at the SANER 2016 conference.

6.1 Introduction

The R package management system provides an easy way to install third-party code and datasets alongside tests, documentation and examples [76]. The main R distribution installs a few *base* and *recommended* packages.

Thousands of additional packages are developed and distributed through different repositories. *CRAN*, the *Comprehensive R Archive Network* (see `cran.r-project.org`), constitutes the official R repository offering both source and precompiled stable packages compatible with the latest version of the R environment. Because of *CRAN*'s longevity, its size and its historical role of being the official distribution platform for R packages, developers often choose to distribute their packages on *CRAN*. As reported by Karl Broman in his insightful R package tutorial [24], *"The main advantage to getting your package on* CRAN *is that it will be easier for users to install (with* `install.packages`*). Your package will also be tested daily on multiple systems."*

Some aspects of *CRAN*'s package policy, however, turn out to be quite restrictive in practice. For example, *CRAN* imposes cross-platform compatibility, it discourage packages to have dependencies outside of *CRAN*, it imposes packages to stay up-to-date with *CRAN*'s most current environment and with the latest version of R. This refrains, or even prohibits, certain package developers of getting their packages on *CRAN*:

- *"It can be a painful process, so you want to get your package in order before you submit."* [24]
- *"Even with our current policy of aiming for back-compatibility we get a lot of complaints that we are asking too much."* [133]
- *"The non-transparent nature of the CRAN submission / rejection process is particularly at issue."* [62]

Moreover, the argument of getting a significantly increased visibility when distributing one's package on *CRAN* is questionable [24]: *"It used to be that putting your package on CRAN also gave it some exposure, but with >6000 packages, that's no longer quite true. To get the word out about your package, I'd recommend twitter, writing a blog, or writing a paper [...]."* As such, there is no longer a strict need to rely on *CRAN* as the official platform for distributing R packages.

R packages can also be distributed on other repositories such as *Bioconductor* (`bioconductor.org`) and several smaller repositories such as *Omegahat*.

There are also commercial R packages being sold by companies such as *Revolution Analytics*. Finally, many open source R packages are being developed on public version control repositories such as *R-Forge* (dedicated to R packages) and *GitHub* (general purpose). Other minor development repositories exist, such as *rforge*[1], but often far less packages and/or few active ones.

Given the increasing popularity of *GitHub* as a platform for distributed software development ($>$ 35 million repositories as of April 2016, of which several thousands of actively maintained R packages) this chapter sets out to explore how *GitHub* is used for developing and distributing R packages. Leek's blog [99] summarizes some of the concerns: *"one of the best things about the R ecosystem is being able to rely on other packages so that you don't have to write everything from scratch. But there is a hard balance to strike with keeping the dependency list small."*

In this chapter we study the topology of the R ecosystem in order to have a better picture of how the different package repositories are structured together. In particular, we wish to get insight in the effect of other package repositories on *CRAN*.

Additionally, we wish to know whether the increasing popularity of *GitHub* as a host for developing many R packages has become a "game changer". Indeed, it is quite straightforward to develop and install packages directly from *GitHub*, possibly avoiding the need for having one's package distributed on *CRAN* or *BioConductor*.

The R community has raised concerns about the way R packages are currently distributed, what problems current package management systems suffer from, and how they could be solved [126]. In addition, the large number of available non-archived packages ($>$ 8,400 in April 2016) is becoming a bottleneck [76] and leads to package dependency problems: *"the number of packages on* CRAN *and other repositories has increased beyond what might have been foreseen, and is revealing some limitations of the current design. One such problem is the general lack of dependency versioning in the infrastructure."* [126]

6.2 Methodology

Taking the point of view of a R package user, we first focus on the following two research questions regarding the general topology of the major R package distributions:

1. Where and how are packages developed and distributed?

[1] `https://rforge.net/`, not to be confused with *R-Forge*

2. How do packages depend on one another?

To answer these questions we will study the size of the different major R package distributions and how they depend upon them. For that we extend our conceptual framework from Chapter 4 with a relation of dependency between distributions.

Because the same package may belong to different repositories[2], we define a total order $>$ on the set $E = \{CRAN, Bioconductor\ \text{software}, Bioconductor\ \text{datasets}, GitHub, R\text{-}Forge, \mathsf{Unknown}\}$ such that $CRAN > Bioconductor$ software $> Bioconductor$ datasets $> GitHub > R\text{-}Forge > \mathsf{Unknown}$. This total order privileges the distributed version of a package over its development version. For example, if a package p_1 on *GitHub* depends on a package p_2 that belongs to both *CRAN* and *GitHub*, it will be counted as a dependency from *GitHub* to *CRAN*.

Using this total order we can introduce some new dependency terminology.

Notation 6.2.1 (Dependencies between distributions)**.** *Let $\alpha, \beta \in E$ be two R distributions. We define.*

$$\begin{aligned}\mathit{deps}_t(\alpha,\beta) = \{(s_1,s_2) \in \alpha_t \times \beta_t \mid \\ s_2 \in dep(s_1,t,\beta) \ \wedge \nexists \gamma \in E : (s_2 \in \gamma_t \wedge \gamma > \beta)\}\end{aligned}$$

$$\mathit{dependsOn}_t(\alpha,\beta) = \frac{|\{s_1 \in \alpha_t \mid \exists s_2 \in \beta_t : (s_1,s_2) \in \mathit{deps}_t(\alpha,\beta)\}|}{|\alpha|}$$

$$\mathit{requiredBy}_t(\alpha,\beta) = \frac{|\{s_1 \in \alpha_t \mid \exists s_2 \in \beta_t : (s_2,s_1) \in \mathit{deps}_t(\beta,\alpha)\}|}{|\alpha|}$$

$|\mathsf{deps}_t(\alpha,\beta)|$ counts all dependency relationships from packages in α to packages in β, $\mathsf{dependsOn}_t(\alpha,\beta)$ gives the fraction of distinct packages in α depending on at least one package in β, and $\mathsf{requiredBy}_t(\alpha,\beta)$ the fraction of distinct packages in α on which at least one package in β depends.

We look more deeply at *GitHub* and how it is influencing the R ecosystem. In particular we want to measure to which extent do R developers distribute their packages on *GitHub*.

To analyze the extent to which *GitHub* is used as a package distribution platform, we intend to show that *GitHub* is becoming more and more important, and that despite the fact *GitHub* packages distributed on *CRAN* are

[2]For example, *GitHub* may store the development version while *CRAN* may contain the stable release version of the package

generally older than other *GitHub* packages, there are numerous packages including instructions to install them from *GitHub*. In order to achieve this goal we study the following questions:

- *How important has* GitHub *become for R packages?* We provide evidence that the number of new R packages on *GitHub* is growing faster than the number of new R packages on *CRAN*.
- *How old are* GitHub *R packages distributed on* CRAN*?* We provide evidence that *GitHub* packages that are not distributed on *CRAN* are younger than those distributed on *CRAN* and that they form a distinct population. We also show that the age of a package cannot be used as a major discriminant in a model to predict the distribution of a package.
- *Which* GitHub *R packages are distributed on* GitHub*?* We show that many *GitHub* packages contain instructions to install them from *GitHub* and that thus many of them are expected to be distributed on *GitHub*.

6.3 Results

6.3.1 Topology of major R package distributions

Where and how are R packages developed and distributed?

Figure 6.1 shows the overlap of R packages on the different major distributions. **The overlap between *CRAN* and *Bioconductor* is very limited**. Both package distributions only have 4 packages in common, corresponding to 0.4% of all considered *Bioconductor* packages and only 0.06% of all considered *CRAN* packages. This is expected as *Bioconductor* is a package distribution similar to *CRAN* but specialized in bioinformatics.

Many packages on *CRAN* also appear on *GitHub*, since both repositories serve different purposes (distribution and development, respectively). We also observe in Figure 6.1 that the intersection of packages that can be found on *CRAN* and *GitHub* is non negligible. 18.1% ($\frac{1157+1}{6411}$) of all considered *CRAN* packages can also be found on *GitHub*. 22.5% ($\frac{1157+1}{5150}$) of all R packages on *GitHub* are also present on *CRAN*. This relatively large proportion of overlap can be explained by the fact that *CRAN* is only a distribution platform, and cannot be used for collaborative development. Hence, ***many* R packages are developed on *GitHub*, while stable releases of these packages are published on *CRAN***. We observe something similar when comparing *Bioconductor* with *GitHub*, and for the same reasons. Indeed,

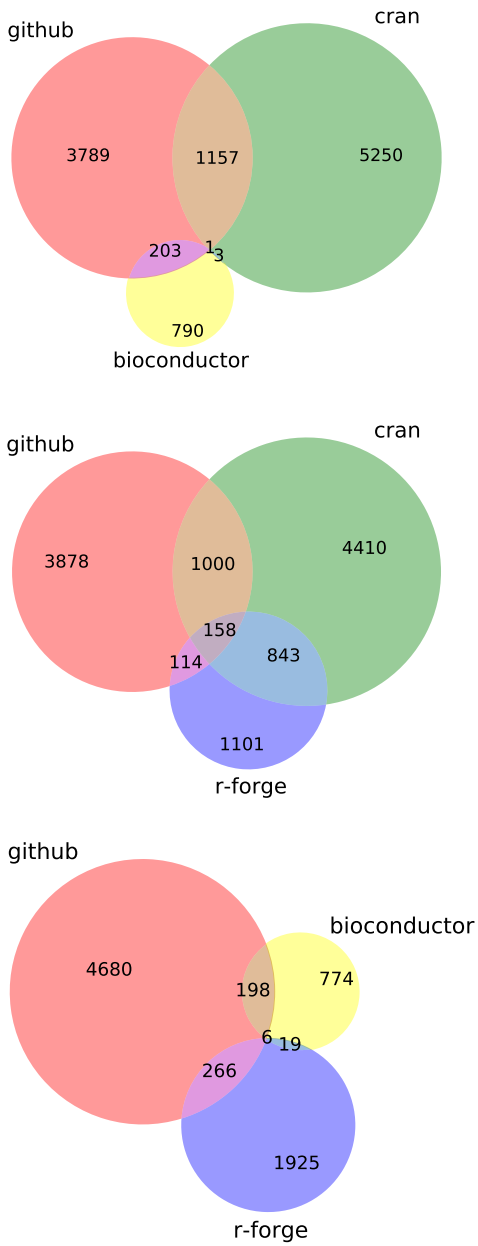


Figure 6.1: Intersections of R packages belonging to *GitHub*, *CRAN*, *Bioconductor* and *R-Forge* in March 2015

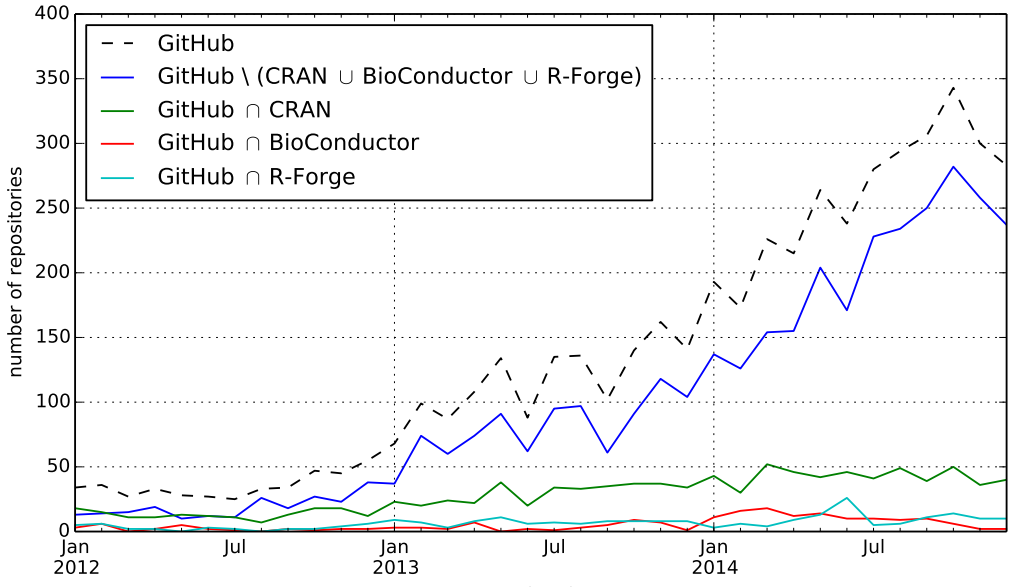


Figure 6.2: Monthly number of newly created repositories on *GitHub* containing R packages.

20.6% ($\frac{203+1}{203+1+3+790}$) of all packages on *Bioconductor* have a counterpart on *GitHub*.

R-Forge has 12.3% ($\frac{266+6}{2216}$) of its packages in common with *GitHub*, while as much as 45.2% ($\frac{158+843}{2216}$) 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***. This is not true for *Bioconductor*: only 1.1% ($\frac{6+19}{2216}$) of all *R-Forge* packages are also available on *Bioconductor*.

One of our goals is to study whether development forges like *GitHub* are overtaking *CRAN* and *Bioconductor* as a primary source of R packages. To do so, we consider the set of all R packages that are available on *GitHub* on 17 February 2015, and study since when they were created and had a counterpart in other R package repositories.

Figure 6.2 suggests that the monthly number of newly created repositories for *CRAN* and *Bioconductor* packages on *GitHub* is slightly increasing over time. This seems to imply that, over time, **developers of packages that are distributed on *CRAN* and *Bioconductor* decide to use *GitHub* as a host for developing their packages**. This does not seem to affect the growth of the packages in the *CRAN* and *Bioconductor* distributions. Figure 6.3 shows that the number of packages in *GitHub* ∩ *CRAN* grows faster than the number of packages distributed on *CRAN*.

We did not find evidence of packages disappearing from *CRAN* or *Bioconductor* due to their migration to *GitHub*. As shown in Figure 6.1, in March 2015 *CRAN* and *Bioconductor* still remain the primary sources for the *distribution* of stable R packages. As such, **development of R packages through *GitHub* seems to complement distribution of packages through *CRAN* and *Bioconductor***, and perhaps even has a catalyst effect.

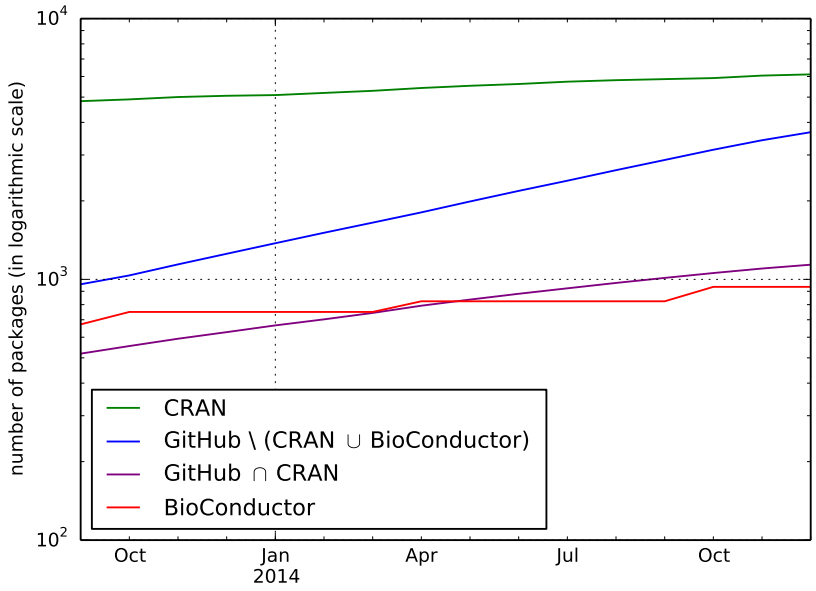


Figure 6.3: Evolution of the number of R packages in *CRAN*, *GitHub* and *Bioconductor*.

Figures 6.2 and 6.3 also reveal that ***GitHub* is increasingly hosting R packages that do not have a counterpart in *CRAN* or *Bioconductor***. Many of these packages are no longer actively maintained today. Those that do, may be developed for personal use only, or could still be unstable but at some point in the future may turn into stable packages that could become distributed in *CRAN* or *Bioconductor*.

How do packages depend on one another?

If an R package depends on another, do these packages belong to the same repositories, or do we observe many inter-repository dependencies? We expect most of such inter-repository dependencies to go towards *CRAN* since it is the official R package distribution. We also expect many dependencies from other repositories towards *Bioconductor* since it is an active package distribution

that offers the same quality checks as *CRAN*, and also because it contains many useful datasets. Therefore, when considering dependencies to packages belonging to *Bioconductor*, we also count dependencies to dataset packages belonging to *Bioconductor*.

Table 6.1 presents metrics for all pairs of considered R package repositories. Unknown represents those dependencies for which we did not find a matching package name in any of the considered repositories. This value was especially high for *GitHub* (140 packages with 156 dependencies to 89 unknown packages).

***CRAN* is self-contained**: the majority of dependencies of its packages stay within *CRAN*: 61% of all *CRAN* packages depend on another *CRAN* package. This is expected, since otherwise the packages would not pass the R CMD check. Note that only 24.9% of all *CRAN* packages are required by other *CRAN* packages.

***Bioconductor* depends primarily on *CRAN* and on itself**: 58.8% of all *Bioconductor* packages depend on *CRAN* packages, while 77.1% of all *Bioconductor* packages depend on other *Bioconductor* packages. Similar to *CRAN*, 26.5% of all *Bioconductor* packages are required by other *Bioconductor* packages. We also observe that 9.3% of *Bioconductor* software packages depend on *Bioconductor* datasets.

GitHub* and *R-Forge* depend primarily on *CRAN: 87.1% of *GitHub* dependencies and 86.4% of *R-Forge* dependencies go to *CRAN* packages.

This shows that **CRAN is still at the center of the ecosystem** and that it has a minority of packages forming a core required by other packages both from CRAN and other sources.

6.3.2 To which extent do R package developers distribute their packages on *GitHub*?

How important has *GitHub* become for R packages?

According to `githut.info`, in the last quarter of 2014, R was the 12th most represented language on *GitHub* (in terms of number of active repositories). This is a major increase with regards to. the last quarter of 2013, when R was only ranked 22nd.

Figure 6.4 shows the number of *GitHub* and *CRAN* packages on June 1, 2015. At this date, ***CRAN* still hosts more packages than *GitHub*.** There are 910 packages belonging to both package repositories. This represents 14.0% of all *CRAN* packages that are also available on *GitHub*, and 20.2% of

Table 6.1: Number of packages primarily belonging to repository α that depend on or are needed by at least one package primarily belonging to repository β.

Metrics	α	β					
		CRAN	*Bioconductor* software	*Bioconductor* datasets	*GitHub*	*R-Forge*	Unknown
$\mathsf{dependsOn}_t(\alpha,\beta)$	*CRAN*	**61.0%**	2.1%	0.1%	0%	0%	0%
$\mathsf{requiredBy}_t(\alpha,\beta)$		**24.9%**	5.8%	0.1%	15.7%	8.6%	–
$\lvert\mathsf{deps}_t(\alpha,\beta)\rvert$		**10,560**	212	4	1	1	0
$\mathsf{dependsOn}_t(\alpha,\beta)$	*Bioconductor* software	**58.8%**	**77.1%**	9.3%	0.2%	0.1%	1.1%
$\mathsf{requiredBy}_t(\alpha,\beta)$		6.5%	**26.5%**	2.8%	12.6%	4.8%	–
$\lvert\mathsf{deps}_t(\alpha,\beta)\rvert$		1,615	**2,748**	133	2	1	13
$\mathsf{dependsOn}_t(\alpha,\beta)$	*Bioconductor* datasets (1,115 packages)	15.2%	**77.1%**	17.5%	0%	0%	0%
$\mathsf{requiredBy}_t(\alpha,\beta)$		0.2%	6.2%	4.3%	1.9%	0.3%	–
$\lvert\mathsf{deps}_t(\alpha,\beta)\rvert$		333	**1,567**	204	0	0	0
$\mathsf{dependsOn}_t(\alpha,\beta)$	*GitHub*	**48.9%**	5.2%	7.4%	5.7%	0.3%	2.7%
$\mathsf{requiredBy}_t(\alpha,\beta)$		0%	0%	0%	4.9%	0.1%	–
$\lvert\mathsf{deps}_t(\alpha,\beta)\rvert$		**8,614**	684	37	386	15	156
$\mathsf{dependsOn}_t(\alpha,\beta)$	*R-Forge*	**37.2%**	2.3%	0.1%	0.7%	5.8%	1.5%
$\mathsf{requiredBy}_t(\alpha,\beta)$		0.1%	0.1%	0%	0.6%	5.0%	–
$\lvert\mathsf{deps}_t(\alpha,\beta)\rvert$		**1,830**	93	4	19	136	36

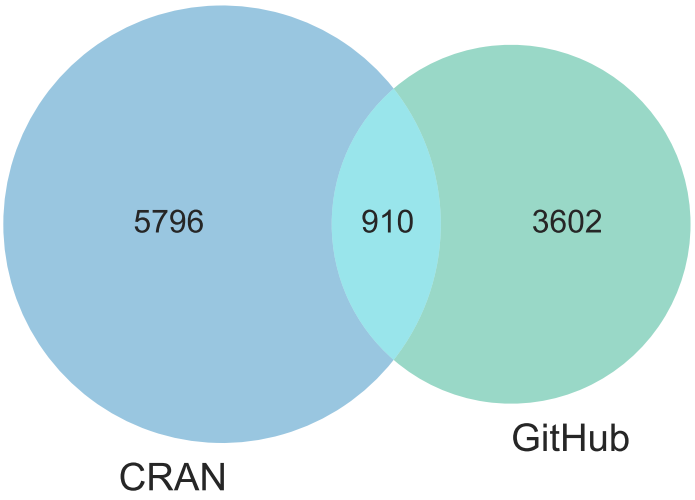


Figure 6.4: Number of R packages by source in June 2015

all *GitHub* packages that are also distributed on *CRAN*. This large overlap can be explained by the fact that both repositories still serve different purposes for a part of the R packages. One can expect that those R packages are developed on *GitHub*, while their stable releases are published on *CRAN*. This is for example the case for Amelia, ggplot2, dplyr, ... Our interviews [114] with R package maintainers that were active on both *GitHub* and *CRAN* confirmed this way of working.

Figure 6.5 shows the number of newly created R packages, by month, on each platform. For *CRAN*, we see a more or less stable trend for the monthly number of new packages, with an exception of a big peak in the second half of 2012, due to the introduction of package namespaces in R. In contrast, for *GitHub* we see an increasing trend in the monthly number of new packages. Even more, since July 2014 **the number of new R packages on *GitHub* appears to be surpassing those on *CRAN*.** Since early 2015, the number of newly created packages is even more than three times higher on *GitHub* than on *CRAN*.

In summary, *GitHub* already hosts many R packages, and there is an important acceleration of the number of new packages appearing on *GitHub* each month.

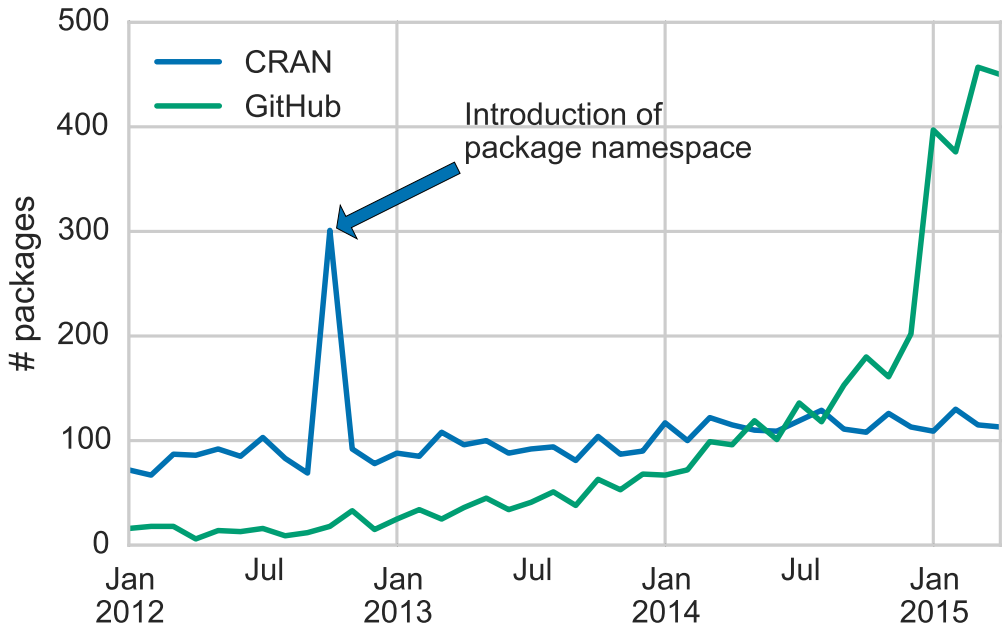


Figure 6.5: Number of new R packages, by month

How old are *GitHub* R packages distributed on *CRAN*?

It is difficult to identify if a *GitHub* package is still in its development stage or if it is ready to be distributed.

We expect *GitHub* to be used as a development platform and *CRAN* as a distribution platform, so many packages will only end up in *CRAN* after some time, if they are considered to be sufficiently stable for being distributed and pass all necessary checks. This is, we expect that *GitHub* packages that are distributed on *CRAN* are older than *GitHub* packages that are not distributed on *CRAN*.

We define the *age* of a *GitHub* package as the time between its very first version and the latest known commit. We compared the age of *GitHub* packages to see if this criterion can be used to distinguish packages that are already distributed on *CRAN* from those which are not.

Figure 6.6 shows the distribution of the age of all 4,512 *GitHub* R packages, as well as the distribution of those 3,602 R packages not found on *CRAN*, and the distribution of the 902 R packages also available on *CRAN*. We statistically compared the age of the sample of *GitHub* packages that are distributed on *CRAN* with the age of the sample of those that are not. Since the samples were not normally distributed, we carried out a one-sided non-parametric Mann-Whitney-U test.

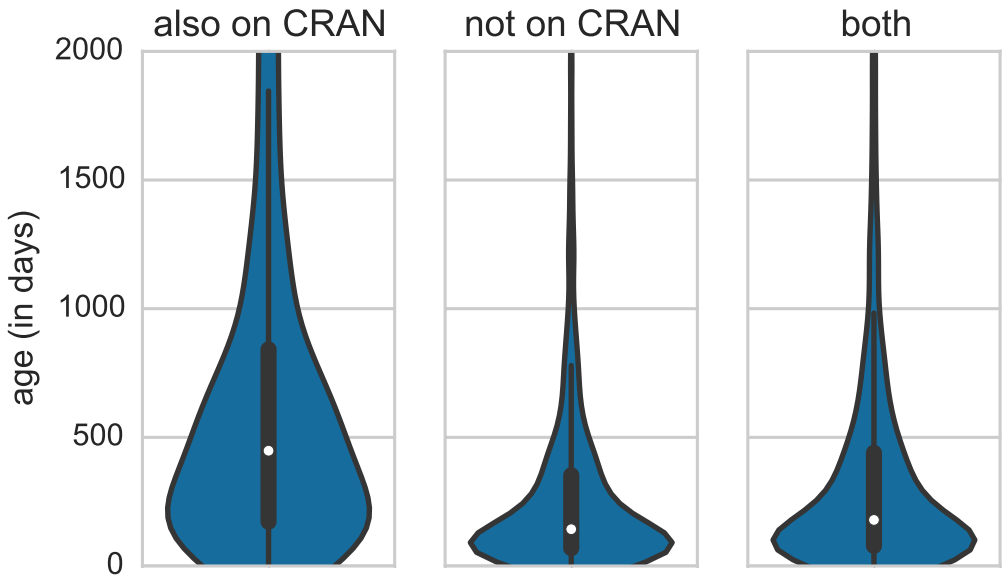


Figure 6.6: Violin plot (using a kernel density estimate) of the distribution of the age of *GitHub* packages.

It is used to test the null hypothesis that the distribution of both populations are equal, or alternatively, whether observations in one population tend to be larger than observations in the other. We choose as alternative hypothesis that the population of *GitHub* packages distributed on *CRAN* is *older* than the population of *GitHub* packages that are not distributed on *CRAN*. The motivation behind this choice is logical: we expect *GitHub* to be used as a development platform and *CRAN* as a distribution platform, so many packages will only end up in *CRAN* after some time, if they are considered to be sufficiently stable for being distributed and pass all necessary checks.

As expected, the null hypothesis was rejected with significance level $\alpha = 0.01$. More specifically, we observed that **the majority of the packages (>75%) that are *not* distributed on *CRAN* are younger than the median of *GitHub* packages that are distributed on *CRAN*.** While the median age of *GitHub* packages distributed on *CRAN* is 448 days old, only 42.5% out of the 1,107 *GitHub* packages that are older than 448 days are actually distributed on *CRAN*. This is, the age of a package cannot be used as a major discriminant in a model to predict its distribution on *CRAN*.

In summary, *GitHub* packages distributed on *CRAN* are older than *GitHub* packages not distributed on *CRAN*, and constitute a distinct population.

However, the age of a *GitHub* package cannot fully explain its distribution status.

Which *GitHub* R packages are distributed on *GitHub*?

We would like to study how many R packages are expected to be distributed and installed from *GitHub*. It is, however, difficult to identify if a *GitHub* package is still in its development stage or if it is ready for distribution. As far as we know, there is no sound and complete characterization of when an R package is ready to be distributed on *GitHub*.

As an approximation, we looked for specific installation instructions from *GitHub* within all of the README files at the root of their *GitHub* repositories. We analyzed these README files using a regular expression corresponding to the use of the function install_github. Our results could include false positives (e.g., "this package cannot be installed with install_github(...)"). However, a manual verification of our results for many packages did not reveal such false positives.

We are also aware that our approach may be inaccurate, since R packages being distributed on *GitHub* may not necessarily have a README file that mentions install_github. Therefore, we can only compute a lower bound of the proportion of R packages that are distributed on *GitHub*. We obtained that **40.9% of all R packages on *GitHub* have a README file that contains instructions to install the package from *GitHub***. These packages correspond to 44.9% of the *GitHub* packages that are also on *CRAN*, and to 39.9% of the *GitHub* packages that are not on *CRAN*.

In summary, many *GitHub* packages are intended to be distributed on *GitHub* as they contain instructions to install them from *GitHub*.

6.4 Discussion

The success of *GitHub* as a *development platform* for software packages seems obvious. It is more difficult, however, to quantify the use of *GitHub* as a *distribution platform*. Many package developers are already resorting to *GitHub* to distribute their software or libraries using tools like NPM or Bower. In addition to these tools, many packages can be downloaded directly from *GitHub* without having to rely on a package manager.

For R packages in particular, however, there is no commonly accepted package manager for *GitHub*. There are neither automatic processes nor absolute assertions that can be used to identify which of the R packages on *GitHub* are intended to be distributed.

Nevertheless, the interviews with R package maintainers confirmed that they actively use *GitHub* for distributing packages, for a variety of reasons. For example, some R package maintainers decide to host their packages on *GitHub* rather than *CRAN*, because their packages depend on external packages that are not accepted by *CRAN*. This is the case for the ANTsR package (`stnava.github.io/ANTsR/`) that depends on cmake, as well as some packages that depend on commercial packages not available in *CRAN*.

From a quantitative point of view, we found many R packages that are distributed on *GitHub*, providing specific installation instructions. We also found that packages that are only available on *GitHub* are younger than the *GitHub* packages that are also distributed on *CRAN*.

6.5 Threats to Validity

We only considered a subset of the R ecosystem, consisting of only 4 package repositories, but covering more than 12,000 distinct R software packages. While other R package repositories exist, given their small size we have not included them in our analysis. The Omega Project for Statistical Computing (`www.omegehat.org`) hosts around one hundred R packages. *RForge* (`rforge.net`), not to be confused with *R-Forge*, provides a collaborative environment for R package developers based on SVN repositories, and contains less than two hundred packages, many of which are no longer active. *GitHub* competitors like *BitBucket* (`bitbucket.org`), *Gitorious* (`www.gitorious.com`) and *Gitlab* (`gitlab.com`) are considerably less frequently used for hosting R package development.

While for *Bioconductor* we explicitly excluded (or treated differently) the packages containing datasets, we were not able to do the same for the other repositories, since we found no automated way to distinguish "ordinary" software packages from datasets. If an R package contains both data and functions, it is hard to decide whether it should be regarded as a software package or a dataset.

For part of our analysis, we relied on information extracted from SVN or Git, or from hosting services like *GitHub*. There are many potential perils and pitfalls that should be taken into consideration when doing so [20, 84]. Some of them can be avoided, others are inherent to the limitations of the considered version control systems or hosting services. For example, how should forking be taken into account? In our analysis, we excluded all forks. We also relied on *GithubArchive* as a proxy for *GitHub* data to extract events, but we cannot guarantee that this data is fully consistent and complete. We also based ourselves on the packages still existing in *GitHub*. We were not

able to extract historical information from *GitHub* repositories that have been removed before.

For R packages hosted on *GitHub*, we assumed that their *DESCRIPTION* file always resides in the root directory of each Git repository, because this is where functions like devtools::install_github expect packages to be located, and because this avoided inclusion of repositories containing R code but that are not R packages. It may, however, have led to the exclusion of some R packages. We also found that some *GitHub* accounts containing R packages (in particular, accounts cran and rpkg) actually served as partial mirrors of *CRAN*, or as a mean to expose R code to *GitHub*. These accounts were excluded from our analysis, but we have no guarantee that other accounts may also be mirrors of R packages developed or distributed elsewhere.

The chosen date of the R package ecosystem snapshot, and the chosen duration for the historical analysis may influence our results. Repeating the same analysis for other dates would allow us to confirm the observed results. For the historical analysis of the *GitHub* data, we based ourselves on the packages still existing in *GitHub* in February 2015. We were not able to extract historical information from *GitHub* repositories that have been removed before that date.

It is not trivial to determine whether an R package available on *GitHub* is ready for distribution. For all identified R packages we checked whether they were ready for release by verifying the presence of a README file with specific installation instructions. Although a manual verification did not reveal any false positives, there may have been false negatives that we have not considered.

6.6 Conclusion

In this chapter, we studied the ecosystem of R packages beyond the official *CRAN* repository. We also considered the *BioConductor* package distributions, and we explored two R package development forges: *GitHub* and *R-Forge*. In total, we analyzed the origin and the dependencies of more than 12,000 packages that were still available in March 2015.

We observed that *CRAN* remains the center of the R package ecosystem, since its packages do not depend on external packages, while *BioConductor*, *R-Forge* and *GitHub* strongly depend on *CRAN* packages. *BioConductor* also contains many packages required by the others, but with an order of magnitude difference compared to *CRAN*.

We also observed that *GitHub* is becoming increasingly used as a collaborative development platform for R packages, both for packages already

distributed on *CRAN* and *BioConductor*, as well as for new packages that do not have any counterpart in the considered distributions or forges. We did not observe any positive or negative effect of this increased use of *GitHub* on the growth of the number of *CRAN* or *BioConductor* packages.

Driven by its increasing popularity, we empirically studied more closely the use of *GitHub* as an alternative or complement to *CRAN* for R package development and distribution. We observed that more and more R packages are hosted on *GitHub*. While the *GitHub* packages distributed on *CRAN* tend to be older than those that are not, their age cannot fully explain whether they are distributed through *CRAN*. Additionally, many R package developers make use of *GitHub* as a distribution platform. Their packages contain instructions to be installed from *GitHub*, and are often exclusively distributed through *GitHub*.

CHAPTER 7

Analyzing the Maintainability of R Packages

When writing software, developers are confronted with a trade-off between depending on existing components and re-implementing similar functionality in their own code. Errors may be inadvertently introduced because of dependencies to unreliable components, and it may take longer to fix these errors. In the previous chapter we studied the general shape of the R ecosystem and showed that distribution of packages mainly happens on two repositories: *CRAN* and *GitHub*. In this chapter, we study how dependencies and package updates impact maintainability in these two major distributions.

First, based on an analysis of package dependencies and package status, we present results on the causes of errors in *CRAN* packages, and the time that is needed to fix these errors. Secondly we show that packages hosted on *GitHub* depend on most active *CRAN* packages. We conjecture that *GitHub* packages might experience even more maintainability problems than *CRAN* packages.

This chapter is mainly inspired by two conference papers [37,44] presented at the CSMR-WCRE 2014 and SANER 2016 conferences.

7.1 Introduction

When writing software, developers may want to reuse code that has been written by someone else rather than writing it again. Developers can do this by directly copying the code or by depending on it (e.g., using a software library). However, when there is not enough coordination between the developers of dependent software components, maintainability problems may arise: components may cease to function correctly because of changes made to the component(s) they depend upon. This can become particularly problematic in large software ecosystems containing thousands of different components maintained by thousands of different maintainers, some of which being considerably less active and responsive than others.

The CRAN package repository, which is the primary source of packages used by the R community, is experiencing such problems.

As seen in Chapter 6, while *GitHub* is growing more rapidly as a R package hosting platform than *CRAN*, the latter is still the center of the R ecosystem. *CRAN* size is considered by some as "too many" [76]. In addition, problems with the dependency versioning system of R have been reported and possible directions for improvement have been proposed [126].

Again, some aspects of *CRAN*'s package policy, however, turn out to be quite restrictive in practice. This refrains, or even prohibits, certain package developers from getting their packages on *CRAN*.

This chapter aims at understanding the impact of errors spreading across dependent packages in *CRAN*, and the influence of the target operating system on this. Moreover because the number of packages hosted on *GitHub* is growing so fast, we try to estimate how much backward incompatible updates might impact maintainability of *GitHub* packages.

7.2 Methodology

As stated in the introduction, our goal is to gain a better insight in the characteristics that impact the maintainability of CRAN packages, by analyzing the errors introduced in them, and the time needed to fix these errors.

To achieve this goal we answer the three following research questions:

- RQ_1: What is the source of errors in CRAN packages, and how are these errors fixed?
- RQ_2: How long does it take to fix an error?

- RQ_3: Are *CRAN* packages more frequently updated than *GitHub* packages?

While *GitHub* packages don't benefit from a daily checking process as *CRAN* does, by answering RQ_3 we can have an idea whether or not maintainability issues related to dependencies are more problematic than for *CRAN* packages.

To answer these research questions we extracted historical metadata for all current and archived CRAN packages in April 2013, available online[1]. We automated the extraction and processing of packages by implementing a set of R tools, available online for replication purposes[2]. These tools download the CRAN source packages, extract their content and store the content of the metadata of each package (stored in a DESCRIPTION file).

Next to the metadata of each package, we also extracted content of the CRAN package checks, on a daily basis, since September 2013. Overall, the results reported here are based on the data extracted from 2013-09-03 to 2016-04-26. The extracted package checks report the daily *status* of each package for each flavor, based on the output of *R CMD check* tool: OK, NOTE, WARNING or ERROR. Packages with an ERROR status are the most relevant, as these are the ones that will be archived upon release of the next non-minor R version.

7.2.1 *R CMD check* and flavors

CRAN packages are checked using the *R CMD check* tool and multiple *flavors*. A flavor is the combination of an operating system, a version of R, a C compiler and hardware architecture. Because CRAN packages are checked for each *flavor*, we wish to know more about the differences between these flavors.

As seen in Figure 7.1, the different flavors on which *CRAN* packages are tested vary over time. Some of them didn't exist at the beginning of the considered period. This is the case for the MacOS X flavors using the development version of R. Other flavors, such as the *r-release-linux-ix86* (Linux on 32 bits Intel CPU), stopped being used at some point. Finally many flavor names were changed over time. For example the *r-devel-linux-x86_64-debian* was renamed to *r-devel-linux-x86_64-debian-gcc* and many flavors based on the *patched* versions of R were temporarily renamed to *prerel* before the release of R 3.1 and 3.2.

[1] `http://cran.r-project.org/src/contrib/`
[2] `http://github.com/maelick/extractoR`

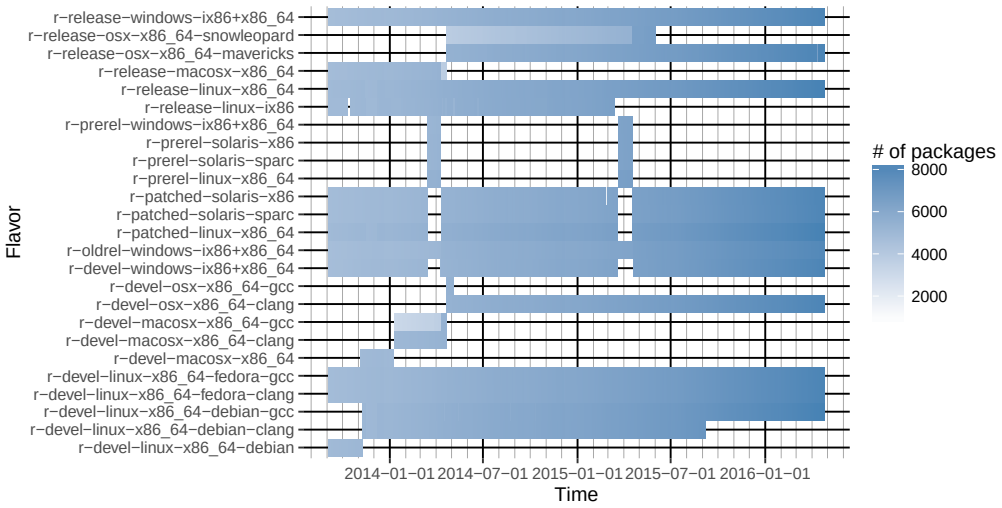
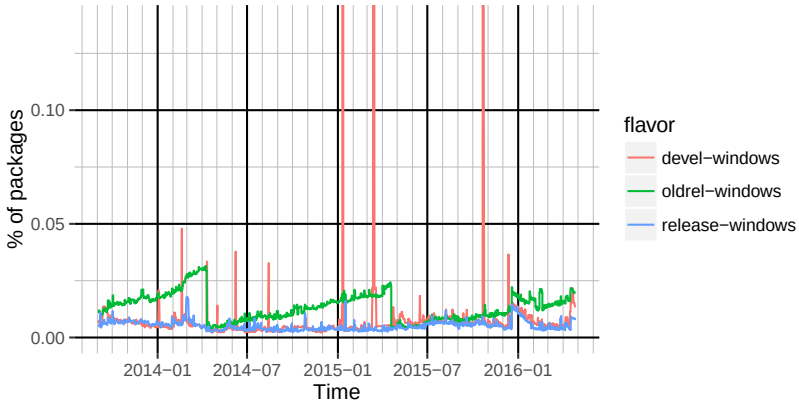


Figure 7.1: History of the number of CRAN packages checked, on each flavor, by the *R CMD check* tool

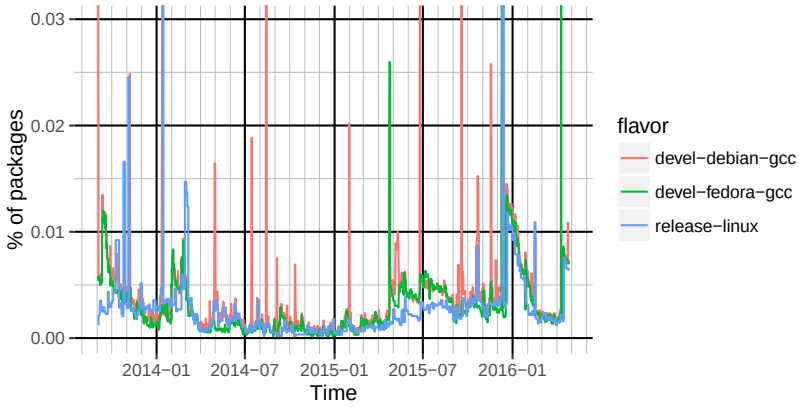
In order to avoid having to analyze many different flavors with different periods, we grouped flavors based on their similarities. Therefore, we focus on a subset of all flavors, keeping the more important ones. Table 7.1 list the different flavors taken into account into the remainder of this chapter.

Flavor name	Actual *CRAN* flavor names
devel-debian-gcc	*r-devel-linux-x86_64-debian* *r-devel-linux-x86_64-debian-gcc*
devel-fedora-clang	*r-devel-linux-x86_64-fedora-clang*
devel-fedora-gcc	*r-devel-linux-x86_64-fedora-gcc*
devel-osx	*r-devel-macosx-x86_64* *r-devel-macosx-x86_64-clang* *r-devel-osx-x86_64-clang*
devel-windows	*r-devel-windows-ix86+x86_64*
oldrel-windows	*r-oldrel-windows-ix86+x86_64*
patched-solaris	*r-patched-solaris-x86* *r-prerel-solaris-x86*
release-linux	*r-release-linux-x86_64*
release-osx	*r-release-macosx-x86_64* *r-release-osx-x86_64-mavericks*
release-windows	*r-release-windows-ix86+x86_64*

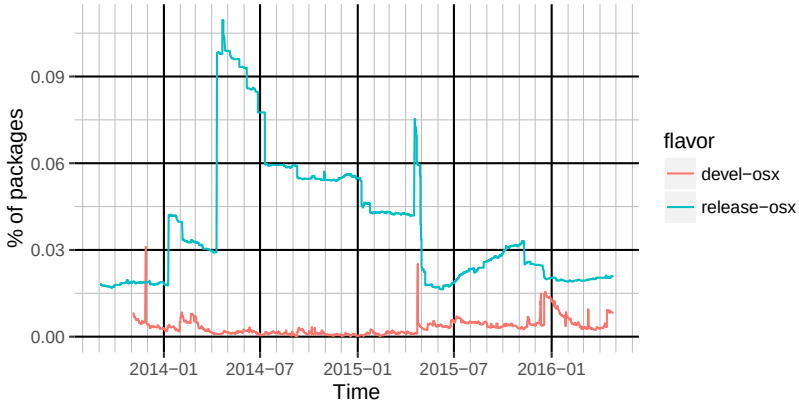
Table 7.1: Definition of the flavor names used in this chapter.



(a) Windows-based flavors



(b) Windows-based flavors



(c) MacOS X based flavors

Figure 7.2: Evolution of the percentage of *CRAN* packages with an ERROR status for the considered flavors.

Figure 7.2 shows the evolution of the percentage of packages with an ERROR status in each considered flavor. These figures show that both the number of errors and the variations of this number is specific to the considered flavor. Because we are interested in looking at the impact of dependency relationships on package maintainability, we want to focus on flavors for which packages are less prone to errors unrelated to dependencies. In particular, flavors that use the development version of R are more prone to errors introduced by changes introduced in R itself.

Moreover, as seen in Figure 7.3 and Table 7.2, the Linux flavor is the one that is the most suited to study. Indeed, it has less errors and less variations of errors than the Windows and MacOS X based flavors. Thus in the remainder of this chapter, we will solely focus on the *release-linux* flavor.

Flavor	# new	# fixed	# full resolution
release-linux	2203	2156	2150
release-osx	4758	4715	4690
release-windows	2417	2329	2250

Table 7.2: Number of newly introduced ERROR status, number of fixed ERROR status and number of "fully resolved" ERROR status (ERROR status that has been introduced and fixed) during the considered time period for each flavor using the released version of R.

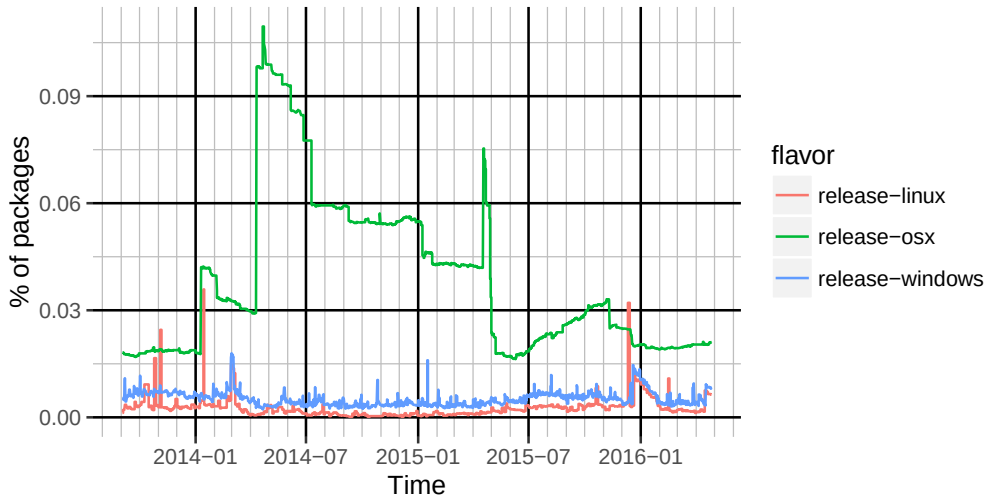


Figure 7.3: Evolution of the percentage of available *CRAN* packages with an ERROR status for flavors based on the released version of R.

7.3 Results

7.3.1 What is the source of errors in CRAN packages, and how are these errors fixed?

The status of a CRAN package, as reported by the daily check, may change to an ERROR status, or may be fixed from ERROR to one of the other status (OK, NOTE or WARNING). In Table 7.3, we identified different reasons for such a package status change.

type	description
PA	Package Archived: ERROR status disappears because the package has been archived
PU	Package Update: ERROR status change coincides with the release of a new package version
DA	Dependency Archived: ERROR status appears because one of its dependencies has been archived or can disappear because one of its (direct or transitive) dependency has been unarchived
DDU	*Direct* Dependency Update: ERROR status change coincides with the release of a new package version of a *direct* dependency
TDU	Transitive Dependency Update: ERROR status change coincides with the release of a new version of a *transitive* dependency
EF	External Factors: ERROR status changed without a new package version or a new version of any dependency

Table 7.3: Types of changes that may change the status of a package to ERROR (or that may fix the ERROR status).

Metric	*PA*	*PU*	*DA*	*DDU*	*TDU*	*EF*
# introduced errors	0	99	107	519	419	1059
% introduced errors	0	4.49	4.86	23.56	19.02	48.07
# fixed errors	121	516	57	195	328	939
% fixed errors	5.61	23.93	2.64	9.04	15.21	43.55

Table 7.4: Absolute and relative number of introduced and fixed ERROR statuses for each causes identified in Table 7.3

We computed the CRAN package dependency graph each time the ERROR

status of a package changed. For each of these packages we computed the list of all direct and transitive dependencies. Table 7.4 show, for each identified cause, the absolute and relative number of ERROR status which got introduced and fixed.

We observe that the main cause of both error introduction and resolution is external factors. Almost one out of two errors is introduced without any change occurring within the package or one of its dependencies. Most of the other errors were introduced by an update of either a direct dependency or a transitive dependency. However, when it comes to error resolution, the main cause, which involves changes in the package or its dependencies, is package update. While 23% of introduced errors were *caused* by a direct dependency update, only 9% are *fixed* by it.

Metric	*PA*	*PU*	*DA*	*DDU*	*TDU*	*EF*
# errors caused by *DDU*	60	247	0	102	16	90
% errors caused by *DDU*	11.65	47.96	0	19.81	3.11	17.48

Table 7.5: Resolution causes of errors introduced by direct dependency.

Metric	*PA*	*PU*	*DA*	*DDU*	*TDU*	*EF*
# errors fixed by *PU*	0	41	16	247	14	197
% errors fixed by *PU*	0	7.96	3.11	47.96	2.72	38.25
# errors fixed by *PA*	0	11	14	60	3	32
% errors fixed by *PA*	0	11.67	9.17	50	2.5	26.66

Table 7.6: Introduction cause of errors fixed by package update and errors fixed by package archived.

For introduced errors that were fixed during the considered period[3], we looked at both the cause of the error introduction and resolution. Table 7.5 reports how errors introduced by *DDU* were fixed. We see that only 20% of the errors caused by dependency updates are fixed by dependency update. On the other hand around half of these errors were fixed by the package maintainer itself.

Table 7.6 reports how those fixed by *PU* and *PA* were introduced. We see that in both cases, half of the time the error was caused by changes in direct dependencies. This means that package maintainers need to focus their maintenance effort on fixing errors introduced by changes in depending

[3]This amounts to 97.6 % of all introduced errors and 99.7% of all fixed errors in the considered period.

packages. Moreover, while not many errors disappear from *CRAN* because the package containing it was archived, half of those packages had an error caused by *DDU*.

7.3.2 How long does it take to fix an error?

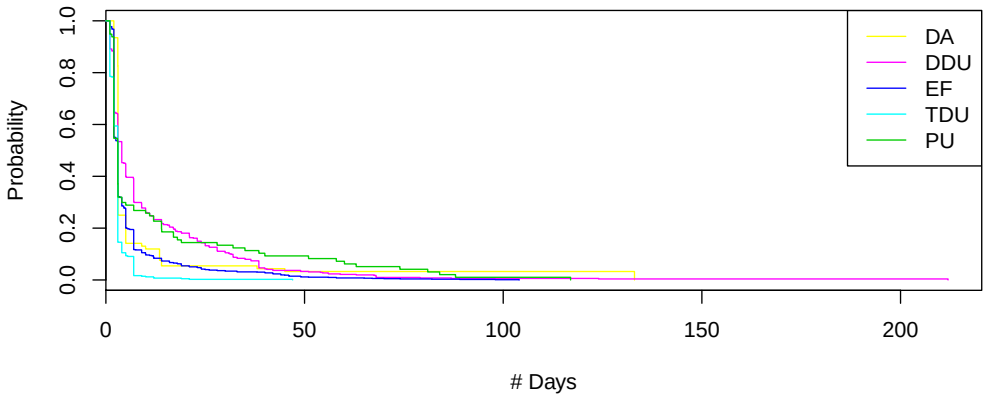


Figure 7.4: Kaplan-Meier for the probability that an error is still present after some time based on its introduction cause.

In order to answer this question we used the statistical technique of *survival analysis* to estimate the probability of an error not being fixed after a certain amount of time. Figure 7.4 shows that errors introduced by *PU* or *DDU* have a slightly higher probability to stay longer. However, Figure 7.5 shows that most errors are fixed very quickly, with the exception of errors fixed by *PA* and *PU*. Errors that were fixed by *PU* are more likely to require more time to be fixed. Moreover, packages with errors that can't be fixed will eventually be archived.

7.3.3 Which CRAN packages are more frequently updated?

From Chapter 6 we know that around 50% of all *GitHub* packages depend on at least one *CRAN* package. So how likely is it that a *CRAN* package gets updated? And does this differ between packages that are required by other *CRAN* packages and packages that are required by *GitHub* packages?

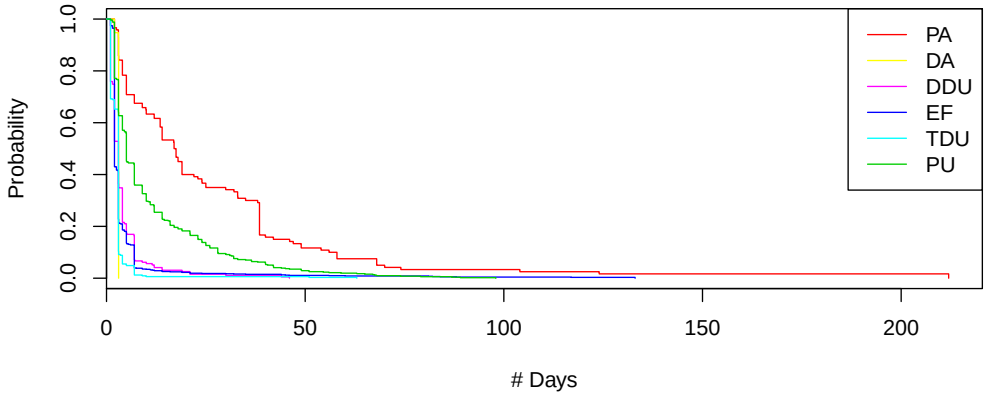


Figure 7.5: Kaplan-Meier for the probability that an error is still present after some time based on its resolution cause.

To respond to these questions, we use the statistical technique of *survival analysis* to estimate the probability of not updating a *CRAN* package for a certain amount of time. In our case, the observed event is the moment on which an R package gets updated. The survival curve is shown in Figure 7.6 using a Kaplan-Meier estimator.

For this survival analysis we considered all *CRAN* package updates over a six month period (from December 2014 to June 2015), representing a total population of 3,740 packages. We observed that, starting from 36 days, those packages that are at least required by *GitHub* packages are more likely to be updated than the others. We confirmed this hypothesis using a one-sided non-parametric log-rank statistical test with significance level $\alpha = 0.05$. This test compares whether the generation process of observed events of the two populations are equal.

The null hypothesis states that both samples have identical survival and hazard functions[4]. The null hypothesis was rejected with p-value $= 0.03$ when comparing "*GitHub* only" and "*CRAN* only" populations, and was rejected with p-value $<$ 0.001 when comparing "Both" and "*CRAN* only" populations.

This provides statistical evidence that the population of *CRAN* packages that are at least required by *GitHub* packages is significantly more prone to be updated than the population of *CRAN* packages that are only required by *CRAN* packages.

[4]The hazard function $\lambda(t)$ is the probability that an individual having survived until time t, it survives for an additional time dt.

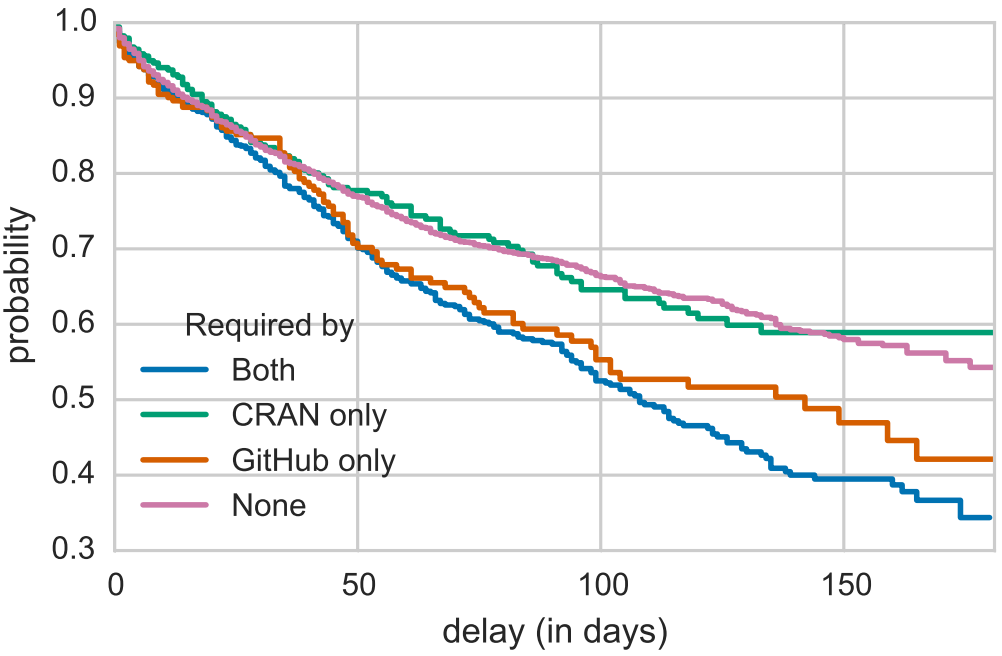


Figure 7.6: Probability that a *CRAN* package is not updated

7.4 Discussion

The *CRAN* community has identified dependency updates as a concern that needs to be addressed:

- *"One recent example was the forced roll-back of the ggplot2 update to version 0.9.0, because the introduced changes caused several other packages to break."* [126]
- *"It is more and more of a pain if the package I'm depending on breaks. If it is just something I was doing for fun, it's not that big of a deal. But if it means I have to rewrite/recheck/rerelease my R package than that is a much bigger headache."* [99]
- *"[...] If I have to load a Depends package, it adds a significant burden: I have to check for conflicts every time I take a dependency on a new package. With Imports, the package is free of side-effects [...]"* [141]

All interviewed R package maintainers active on *GitHub* share this concern:

- *"[...] the risk of things breaking at some point due to the fact that a version of a dependency has changed without you knowing about it*

is immense. That actually cost us weeks and months in a couple of professional projects I was part of." [9]

- *"A better systematic for dependency management together with making your codebase more robust against changes in dependencies is the thing that I actually spend most of my time cracking my brain about [...]"* [9]

- *"I had one case where my package heavily depended on another package and after a while that package was removed from* CRAN *and stopped being maintained. So I had to remove one of the main features of my package. Now I try to minimize dependencies on packages that are not maintained by "established" maintainers or by me [...]"* [9]

- *"There have actually been a few times when I have rewritten a function in my own package because of that difficulty, especially with packages that themselves have many dependencies. The biggest issue we have is multiple layers of dependencies, some of which are on* CRAN *and some of which are not. That can be difficult to keep in sync, but usually, if your package is not on* CRAN*, you can just keep it using the older dependency for a while until you have time to sort that issue."* [9]

- *"It's a bit of a hassle when your package depends on other development versions, but there are changes in the latest version of* devtools *to make this easier."* [9]

Since more and more packages are being developed and distributed outside of *CRAN*, we argue that inter-repository dependency management will become a major concern for the R community.

R package users and developers could benefit from a package installation manager that relies on a central listing of available packages. 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*.

To reduce the problem of backward incompatible changes in R packages hosted on *GitHub* due to *CRAN* package updates, package maintainers could deploy continuous integration processes (such as Travis CI) on their *GitHub* projects, in order to benefit from an equivalent of *CRAN*'s daily R CMD check. However, we observed that only 23.6% of the considered R packages hosted on *GitHub* have defined a configuration for Travis CI. Even with such a continuous integration process, the lack of a built-in support for dependency constraints satisfaction in R still forces developers to react to

all the backward incompatible changes in each dependency of their package, even if the dependent package is stable or no more under development.

R package maintainers active on *GitHub* confirm the need for more systematic package dependency management: *"I personally think it's REALLY relevant to at least be ABLE to be very specific and rigid with regard to your dependencies. And I think the R universe could provide better tools to fit the needs of developers and professionals out there in a better way. But in that regard I like efforts such as* Packrat *and* checkpoint *very very much."* [9]

People from the R community have started to explore ways to improve how *CRAN* and the current R package management system work together [126]. Inspired by the way in which Debian Linux and npm manage their package distributions, two solutions are proposed. The first solution consists in having a testing and a development branch of the *CRAN* distribution. The development distribution contains the most recent but also more unstable packages, while the testing distribution is regularly frozen in order to release a stable snapshot of *CRAN* with each new version of R. While this solution would certainly benefit *CRAN*, it does not solve the problems for R packages hosted on *GitHub*. R package maintainers are already adopting such a solution, but at an individual level: *"Yes. I usually have a* cran *branch which matches the* CRAN *version. A* master *branch (or one that I set as default) would be the current development version. I might have some experimental branches as well."* [9] Having standardised support for this would benefit the community as a whole.

A second, more general, solution would consist in fundamental changes to the way in which R installs and loads packages: each package should be allowed to specify the version of its dependencies it requires, and provide a way to install and load multiple versions of the a package at the same time.

7.5 Threats to Validity

Our results may be biased by limitations of the CRAN check tool and the other extraction and measurement tools that we have developed and used. We know that the CMD check on many R packages on *CRAN* results in an error status without an update of any of their dependencies (for example, if the language R itself or one of the base R packages evolves). This could influence the results for packages with many dependencies, as there are potentially more situations in which at least one dependent package gets broken at the same time of an update of its dependency.

Our analysis relied on information extracted from Git and *GitHub*. Many pitfalls should be taken into consideration when doing so [20, 84]. Some of

them can be avoided, others are inherent to the limitations of the considered version control systems or hosting services. For example, how should forking be taken into account? In our analysis, we excluded all forks. We based ourselves on the packages still existing in *GitHub* in June 2015. We were not able to extract historical information from *GitHub* repositories that have been removed before that date.

For R packages hosted on *GitHub*, we assumed that their *DESCRIPTION* file always resides in the root directory of each Git repository, because this is where functions like devtools::install_github expect packages to be located, and because this avoided inclusion of repositories containing R code but that are not R packages. It may, however, have led to the exclusion of some R packages. We also found that some *GitHub* accounts containing R packages (in particular, accounts cran and rpkg) actually served as partial mirrors of *CRAN*, or as a mean to expose R code to *GitHub*. These accounts were excluded from our analysis, but we have no guarantee that other accounts may also be mirrors of R packages developed or distributed elsewhere.

7.6 Conclusion

With the aim to assess the maintainability of packages belonging to the CRAN archive network, we analyzed the introduction and fixing of errors in packages. We collected daily information from the CRAN website of the automated check of each package for different flavors from September 2013 to April 2016. Because some flavors are more error prone than others, we analyzed a flavor that is less prone to errors caused by changes unrelated to changes occurring in dependencies.

While the majority of errors appear and are resolved within a few days without any developer intervention, the other errors requiring intervention from the package maintainers are primarily errors introduced when a package they rely upon was modified. Maintenance effort hence needs to be focused on fixing errors caused by others. Moreover, those errors that stay for a longer time, might eventually end up being archived. This may become detrimental to package maintainability in the long run for CRAN packages.

We showed that R packages hosted on *GitHub* are very likely to suffer from inter-repository dependency problems. While most of the dependencies are provided by *CRAN*, many *GitHub* packages cannot be installed automatically due to a lack of central listing of all available packages. We showed that *CRAN* package updates cause backward incompatible changes. Because *CRAN* packages that are required by *GitHub* packages are more prone to be updated, this problem is potentially worse for *GitHub* packages. We showed

how this problem is currently addressed on *CRAN*, and how it affects *GitHub* packages.

While the current role of the automated error check for CRAN packages is to inform developers about whether their package conforms to the CRAN quality policy, we believe that R maintainers and developers could benefit from a more specific tool giving less general information than the current check and more information about the implications and problems raised by dependency changes.

In order to be able to install an R package, its dependent packages need to be available as well. While R provides an easy built-in way to install packages, this solution, in combination with the way in which *CRAN* distributes packages, is neither sufficient nor satisfactory to handle *GitHub* package dependencies [126]. Through interviews with R package maintainers on *GitHub*, we observed that the lack of support for dependency constraints in R is already a major concern. Contrarily to *CRAN*, distributed *GitHub* packages are not systematically monitored by a continuous integration process like the R CMD check in *CRAN*.

Thus, the R package-based software ecosystem would strongly benefit from an automatic package installation and dependency management tool, like the ones that are currently available for other package-based software ecosystems. The need for such automated tools for the R ecosystems was relatively low a few years ago, since *CRAN* was, together with *Bioconductor*, the major distribution platform for R packages. Today, this is no longer the case, since development platforms play an increasingly important role in the distribution of R packages.

CHAPTER 8

Analyzing Code Cloning in *CRAN* Packages

Code clone analysis is a very active subject of study, and research on inter-project code clones is starting to emerge. In the context of software package repositories specifically, developers are confronted with the choice between depending on code implemented in other packages, or cloning this code in their own package. This chapter presents an empirical study of identical function clones in the CRAN package archive network, in order to understand the extent of this practice in the R community.

Depending on too many packages may hamper maintainability as unexpected conflicts may arise during package updates. Duplicating functions from other packages may reduce maintainability since bug fixes or code changes are not propagated automatically to its clones. We study how the characteristics of cloned functions in CRAN snapshots evolve over time, and classify these clones depending on what has prevented package developers to rely on dependencies instead.

This chapter is mainly inspired by a workshop paper [38] presented at the International Workshop on Software Clones (IWSC) 2015.

8.1 Introduction

Analyzing the impact (whether it be harmful or beneficial) of inter-project code cloning is an emerging topic of research in the code cloning community [94,144]. Developers are often confronted with the difficult choice between depending on existing functions developed in other libraries, or copy-pasting or re-implementing similar functions in their own code. In the case of depending upon a third-party library, errors may be introduced inadvertently when upgrading to a newer version of the library one depends upon. Finding and fixing these errors can be cumbersome. Duplicating functions across different libraries is an alternative but may be detrimental to the maintainability in the long run.

In this chapter, we study the extent of the practice of cloning functions between packages contained in *CRAN*. Identical cloned functions across packages appear to be omnipresent in *CRAN*. Many reasons can justify the presence of these clones.

We have seen in Chapter 7 that the *CRAN* policy puts a heavy burden on package maintainers, especially if their packages fail due to an update of some dependent package over which the maintainer has no control. We know from interviews of R package maintainers that this phenomenon happens in practice: *"Nothing that would be related to* GitHub*. I had one case where my package heavily depended on another package and after a while that package was removed from CRAN and stopped being maintained. So I had to remove one of the main features of my package. Now I try to minimize dependencies on packages that are not maintained by "established" maintainers or by me ;-) Nowadays it has become a standard that PhD students in statistics develop an R package as a byproduct of their PhD thesis. But for students who leave academia or move to other projects it's often a low priority if at all to maintain their package."* [9]

Thus R package maintainers might sometimes resort to copy-and-paste reuse to reduce the extent of this problem. Indeed, copying the code of a function they want to reuse from another package requires less effort (at least at the short term) than explicitly depending on that package and take the risk that future changes in that package may lead to conflicts in ones own package.

We also suspect that some function clones between R packages exist because the declaration of an explicit package dependency does not allow to access the required function. This is for example the case if the function is local to the package, or if it is anonymous, or if it is a global function that is not exported in the package's namespace.

With our study, we aim to verify whether this is really the case and how this practice evolves over time. In particular, we are interested in Type-1 clones [18, 138]. We are interested in Type-1 clones, which are syntactically identical code fragments, because we are interested in code that is duplicated to avoid dependencies. We want to understand to which extent functions are cloned across packages, why R package developers clone functions, and if clones could be avoided by the introduction of explicit dependencies. From now on, when we use the term *clone* it will implicitly refer to Type-1 function clones, i.e., functions that share the same code body. Our longitudinal empirical study of inter-package function clones in *CRAN* focuses on the following research questions:

1. How prevalent are clones in *CRAN*, and how does this evolve over time?
2. How and why did clones appear?
3. Is it possible to remove clones and how?

First Section 8.2 extends the conceptual framework introduced in Chapter 4 to take into account clones, introduces related metrics and describes how we computed clones between CRAN packages. Then Section 8.3 presents interesting observed clone cases from CRAN and answers, based on empirical results, to the previous research questions. Finally Section 8.4 presents threats to validity and Section 8.5 concludes.

8.2 Methodology

8.2.1 Terminology

Before answering the research questions introduced in Section 8.1, let us extend the terminology introduced in Chapter 4 with the necessary terminology and notations.

Each project state $s \in State(E)$ is characterized by a number of *function* definitions, that contain source code.

Notation 8.2.1 (Functions)**.** *Let $s \in State(E)$.*

- *F_s is the set of all functions belonging to a project state s. A function $f \in F_s$ is a triple $(f_{args}, f_{body}, f_{env})$ where f_{args} denotes the set of function parameters, f_{body} the function body, and f_{env} the environment of the function.*
- *$F_E = \cup_{s \in States(E)} F_s$ denotes the set of all functions in the ecosystem E.*

- $F_s = F_s^G \cup F_s^L$ *partitions all functions of* F_s *into* local *functions* F_s^L *(that are defined internally to another function) and* global *functions* F_s^G.
- $contains \subseteq F_s^L \times F_s$ *determines inside which other function of the project state a local function is defined.*
- $F_s = F_s^N \cup F_s^A$ *partitions all functions of* F_s *into* named *functions* F_s^N *(that have an explicit name) and anonymous functions* F_s^A *(that do not have an associated name).*
- $name : F_s^N \to \Sigma^* : s \mapsto name(s)$ *provides the name of each named function as a sequence of symbols belonging to some alphabet* Σ.
- $time : F_s \to T : s \mapsto time(s)$ *provides the release time of the project state in which a function comes from.*

```
ReadJSONFiles <- function(files) {
  lapply(files, function(f) rjson::fromJSON(file=f))
}
```

Figure 8.1: Example of a global and named function *ReadJSONFiles* which contains a function that is both local and anonymous.

Notation 8.2.2 (Snapshots and functions)**.** *Let* $t \in T$.

- $F^t(D) = \{f \in F_s \mid s \in D_t\}$ *is the set of all functions in snapshot* D_t
- $F_s^t(D) = F^t(D) \cap F_s$ *is the set of all functions in project state* s *belonging to snapshot* D_t.

Clones may appear either within the same package, within two versions of the same package, or belong to two different packages. *Clone sets* represent groups of identical clones.

Notation 8.2.3 (Clones and clone sets)**.** *Let* $t \in T$.

- $C^t(D) = \{(f, g) \in F^t(D) \times F^t(D) \mid f \neq g \wedge f_{body} = g_{body}\}$ *denotes the clone relation between functions. Let* $f \in F^t(D)$*, then we define* $clones^t(f) = \{g \in F^t(D) \mid (f, g) \in C^t(D)\}$.

- *$C^t(D)$ forms a partial equivalence relation (i.e., it is symmetric and transitive). The quotient set $F^t(D)/C^t(D)$ is the set of all clone sets. A* clone set *$C \subseteq F^t(D)/C^t(D)$ is an equivalence class of (function) clones defined by this partial equivalence relation $C^t(D)$.*

- *For each* clone set *C we define its* origin(s) *as the function(s) representing the oldest incarnation(s) of the clone:*

$$origin(C) = \{f \mid \exists g \neq f, (f,g) \in C \wedge \forall g, (f,g) \in C : f \neq g \Rightarrow time(f) < time(g)\}$$

While theoretically a clone set can have multiple origins, in practice this occurs very rarely. And even if it does, these multiple origins tend to belong to the same package version (i.e., the multiple origins are internal clones of one another in the same package). For example, for snapshot of *CRAN* D_t at date t =2014-12-01 we found exactly 1 origin package for all 3,184 detected clone sets.

This study focuses on inter-package clones, i.e., clones across *different* packages, as they are subject to the maintenance problem described in Chapter 7. We will not consider clones between different versions of the *same* package.

Notation 8.2.4 (Inter-package function clones)**.**

- *$C^t_{Inter}(D) = \{(f,g) \in C_t(D) \mid \exists v, w \in D_t : v \neq w, f \in F_v(D), g \in F_w(D)\}$ denotes all clones between functions belonging to different packages. Like $C^t(D)$ it forms a partial equivalence relation that allows us to define clone sets.*

- *$clones^t_{Inter}(f, D) = \{g \in F^t \mid (f,g) \in C^t_{Inter}(D)\}$.*

For example, *CRAN* package biotools 1.2 has one identical clone with package soilphysics 1.1. The function body is

```
if (is.null(text))
  text <- "Welcome to the statistical software revolution!"
if ( !inherits(text,"character") || length(text) != 1)
  stop("'text' must be a character vector of length 1!")
vec <- strsplit(text, "")[[1]]
lab <- c(vec, "\n")
for (i in 1:length(lab)) {
  setTxtProgressBar(txtProgressBar(char = lab[i]), 0.01)
  Sys.sleep(0.1)
}
```

Notation	Description
F_s	all functions belonging to project state s
$F^t(D)$	all functions belonging to packages in snapshot D_t
$F^t_s(D)$	all functions belonging to project state s in snapshot D_t
$C^t(D)$	all pairs of function clones for snapshot D_t
$C^t_{Inter}(D)$	all pairs of external clones for snapshot D_t

Table 8.1: Summary of introduced notation

8.2.2 Metrics

In this section, we define the metrics that we will use for the empirical analysis in Section 8.3.

Size metrics

Let $t \in T$, D_t the corresponding snapshot from a distribution D, $s \in D_t$ a project state, and $f \in F_s$ a function.
$LoC(f)$ = number of lines of code of f.
$AST(f)$ = size of the abstract syntax tree (AST) of f. It computes the number of nodes in the AST returned by the R function `parse`. This AST is similar to a LISP AST. Each R language operator is a call to a function. Thus all symbols that look like an operator, whether they are "`for`", "`function`", "`+`" or the assignment operator "`<-`", are treated as functions. Moreover the enclosing curly braces ("{}", working like in C) are also treated as functions. Because we consider `if (x == 0) x + 1` being the same code as `if (x == 0) { x + 1 }`, we simplified the AST by removing the "{" nodes.

We can define size metrics at the level of a project state as follows:
$LoC(s) = \sum_{f \in F_s} LoC(f)$ = number of lines of code of s.
$AST(s) = \sum_{f \in F_v} AST(f)$ = AST size of s.
$NoF(s) = \mid F_s \mid$ = number of functions belonging to s.

We define size metrics at the snapshot level as follows:
$NoP(t, D) = \mid D_t \mid$ = number of packages in snapshot D_t.
$LoC(t, D) = \sum_{s \in D_t} LoC(s)$ = lines of code for snapshot D_t.
$AST(t, D) = \sum_{s \in D_t} AST(s)$ = AST size for D_t.
$NoF(t, D) = \mid F^t \mid = \sum_{s \in D_t} NoF(s)$ = number of functions in D_t.

Dependency metrics

$Out(s, t, D)$ = out-degree of project state $s = \mid dep(s, t, D) \mid$
$In(s, t, D)$ = in-degree of project state $s = \mid revdep(s, t, D) \mid$

$Out^*(s,t,D) = \mid dep^*(s,t,D) \mid$ = number of transitive dependencies of project state s
$In^*(s,t,D) = \mid revdep^*(s,t,D) \mid$ = number of transitive reverse dependencies of project state s

Clone metrics

$NoC(f) = \mid clones^t_{Inter}(f) \mid$ = number of inter-package Type-1 clones of function f in the same snapshot D_t.

Let $Clones(s) = \{f \in F_s : NoC(f) > 0\}$ the set of function clones contained in project state s.
$NoCF(s) = \mid Clones(s) \mid$ = number of cloned functions of project state s.
$LoCC(s) = \sum_{f \in Clones(s)} LoC(f)$ = number of lines of cloned code in project state s.
$RoCF(s) = \frac{|Clones(s)|}{NoF(s)}$ = ratio of externally cloned functions in project state s.
$RoCS(s) = \frac{LoCC(s)}{LoC(s)}$ = ratio of cloned code in project state s.

We can aggregate these clone metrics at the snapshot level as follows.
$NoCP(t,D) = \mid \{s \in D_t : NoCF(s) > 0\} \mid$ = number of packages containing clones in snapshot.
$NoCF(t,D) = \sum_{s \in D_t} \mid Clones(s) \mid$ = number of cloned functions in snapshot.
$NoCS(t,D)$ = number of clone sets of snapshot D_t = number of classes defined by the partial equivalence relation C^t_{Inter}.
$LoCC(t,D) = \sum_{s \in D_t:\ NoCF(s)>0} LoCC(s)$ = number of lines of cloned code in snapshot D_t.
$LoCCP(t,D) = \sum_{s \in D_t:\ NoCF(s)>0} LoC(s)$ = number of lines of code in all packages containing clones.
$RoCF(t,D) = \frac{NoCF(t)}{NoF(t)}$ = ratio of external function clones.
$RoCP(t,D) = \frac{NoCP(t)}{NoP(t)}$ = ratio of packages with clones.
$RoCC(t,D) = \frac{LoCC(t)}{LoC(t)}$ = ratio of cloned lines of code.
$RoCCP(t,D) = \frac{LoCCP(t)}{LoC(t)}$ = ratio of cloned lines of code in packages containing code.

All these metrics can be qualified by an extra parameter n representing a minimal threshold on the function size expressed in lines of code, i.e., we restrict the functions under consideration to $\{f \in F^t \mid LoC(f) > n\}$.

If the threshold $n = 0$, we obtain the same values as the original metrics.

For example, $RoCF(s, 10)$ will compute the ratio of cloned functions in package s, taking only into account those functions that have more than 10 lines of code.

8.2.3 Type-1 function clone extraction

To analyze the history of clones in CRAN, we have proceeded as follows to extract and identify Type-1 function clones. First, we parsed the source code of each version of each CRAN package using built-in R functions. More specifically, we used the R base function parse to construct the abstract syntax tree (AST) of the body of each function of each package. Working with the AST allows us to ignore all code comments and differences in code indentation between otherwise identical function bodies. Next, we computed a hash value for each function body using the SHA-1 cryptographic secure hash algorithm. Two functions that have the same hash value for their function body can be considered as identical functions with a negligible probability ($< 10^{-18}$) of false positives.

For the purpose of the empirical analysis we excluded all functions whose body contains less than 6 lines of code. Through manual inspection we found that this value allows us to avoid most of the small code fragments leading to "accidental clones".

We also excluded all intra-package clones, i.e., clones that appear within the same package. For the empirical analysis, only those clones that appear between different packages (i.e., belonging to the clone relation C^t_{Inter}) are of interest to us.

8.3 Results

8.3.1 Observed Clone Cases

Before delving into an empirical analysis, we focus on a limited subset of "interesting" CRAN packages with respect to their cloning behavior. In particular, we considered those packages for which there is an unusually high number of clones, or an unusually high number of packages that have cloned functions belonging to the considered package. The aim of this section is not to provide a representative classification but is rather indicative about some interesting cases of clones found in CRAN. We present these observed cases of cloning behaviour below.

Coexisting package versions

In some CRAN snapshots, two different "versions" of the same package may coexist. While these packages have a different name, one of them can be regarded as the new version of the other. Needless to say, the majority of functions from the old package will be cloned in the new package. A valid

reason for this clone case is to allow existing packages to continue to depend on the old version, while already exposing the new version with extended functionality.

One occurrence of this type of clone behavior was found for packages *plyr* 1.8.3 and *dplyr* 0.4.3. They are maintained by the same person, and both allow to manipulate R data structures more easily and more efficiently but in a different way. They share 3 identical function clones totalling 48 lines of code.

Another occurrence are packages *lme* and *nlme* that have coexisted for some time. Both packages fit the same goal of providing statistical model functions. *nlme* adds non-linear models to *lme* and actually replaced it in later snapshots of CRAN. In April 2016 they still shared more than 600 identical function clones totalling over 7,000 lines of code.

A third example is the pair of packages *np* and *npRmpi*. The latter package is a version of *np* that uses MPI (Message Passing Interface) to distribute computation. In April 2016, both were still available on CRAN, maintained by the same person and share more than 10,000 lines of code.

The *forked* package

Related to the previous case, *forked* packages continue to coexist with the package they have forked from. An example is package *Rcmdr*, offering a graphical interface to use R statistical functions. Package *QCAGUI* provides a graphical interface for the *QCA* package, and can be considered as a fork of *Rcmdr* with most of the statistical related features removed. In April 2016, *Rcmdr* and *QCAGUI* shared more than 8000 lines of code.

The frequently cloned package

For some packages, most functions have been cloned by other packages. An example is *distr* 2.5.3, which contains 182 lines of code, and all its global functions have been cloned by different packages.

The utility package

We refer to an utility package as a package that bundles together a lot of functions that are cloned from many other packages. An example is package *DescTools*, which gathers functions for basic statistics that are scattered across different packages, and bundles them together into a single package. *DescTools* 0.99.15 copied 52 functions (totalling 1,419 lines of code) from 27 different packages. Some of them are public functions while others are local functions meaning that they are probably used in wrapper functions.

The popular package

A popular package contains specific functions that are cloned by a lot of other packages. An example of such a package is *MASS*, a well-known and widely reused statistical package. Its version 7.3-15 has 16 functions that have been cloned by 16 different packages for a total of 180 code lines.

The popular function

A popular function is a function that is cloned by a lot of different packages, while the other functions of the same package are not. An example is the package *combinat* 0.0-8, whose function *permn* of 151 lines of code is cloned by 7 different packages.

8.3.2 How prevalent are clones in CRAN?

In order to assess the importance of the cloning phenomenon across CRAN packages we computed the snapshot for each day t from January 2000 to December 2014. For each snapshot D^t we computed the snapshot-level metrics related to clones.

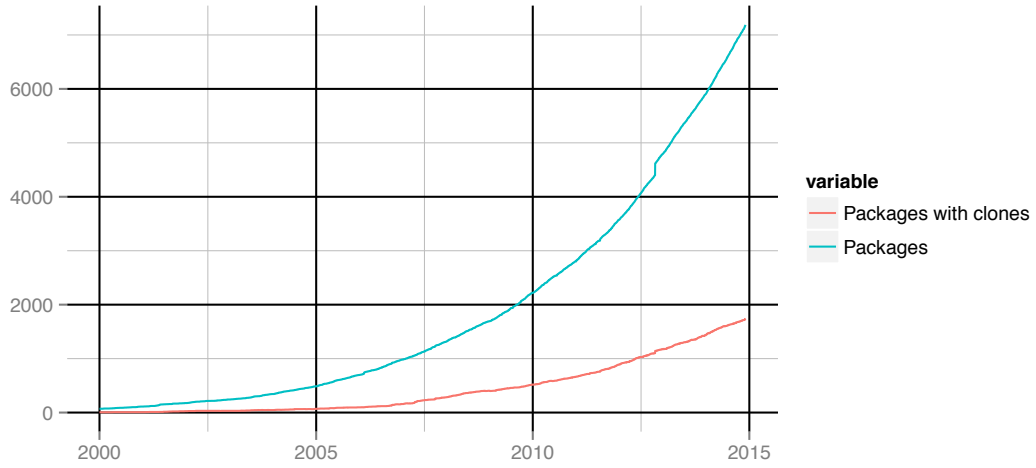


Figure 8.2: Evolution over time of $NoP(t)$ (in blue) and $NoCP(t)$ (in red).

Figure 8.2 shows the evolution in CRAN of the number of packages $NoP(t)$ and number of packages containing clones $NoCP(t)$. The general trend is that the number of packages containing clones increases over time, up to 2,000 packages containing clones today. This corresponds to 24.2% of all packages. The trend follows the overall exponential growth trend of the number of available CRAN packages.

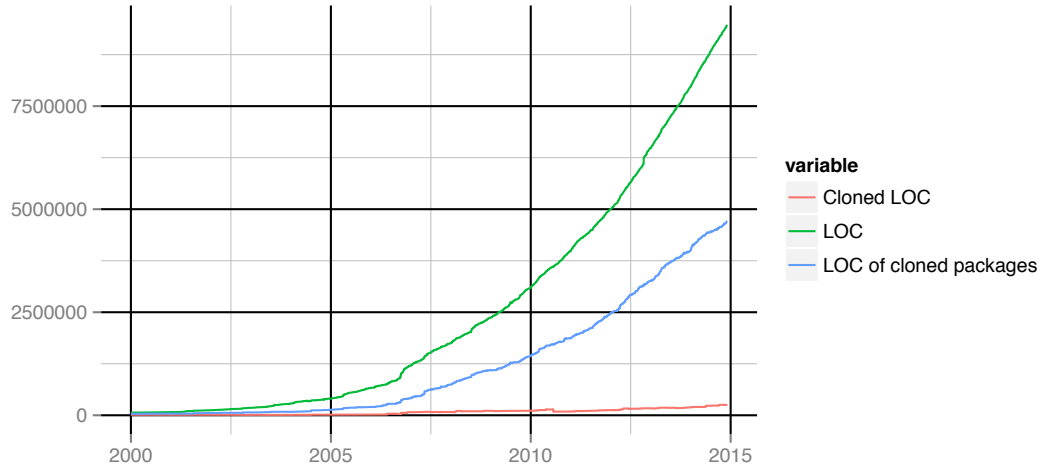


Figure 8.3: Evolution over time of $LoC(t)$ (in green), $LoCC(t)$ (in red) and $LoCCP(t)$ (in blue) in CRAN.

The evolution of the number of lines of cloned code $LoCC(t)$ (Figure 8.3) also follows an increasing trend. We observe that $LoCC(t)$ is much smaller than $LoCCP(t)$, the total number of lines of code of the packages containing these clones. The ratio amounts to 2.6% of all lines of code in CRAN and 5.3% of all lines of code of packages containing clones. This is much less than the ratio observed in Figure 8.2 of 24.2% of packages containing clones. Nevertheless, these cloned functions are included in packages that, together, represent 49,7% of all lines of code in CRAN!

Figure 8.4 shows how these ratios evolve over time. The ratio $RoCC(t)$ of lines of cloned code decreases over time (starting from around 20% initially to less than 5% today). The ratio $RoCP(t)$ of packages containing cloned code has the opposite behaviour, with a higher percentage of packages is containing cloned code. In both cases, the ratio seems to stabilize during the last 8 years of CRAN. We hypothesize that this is because CRAN has become more mature.

From these findings we can conclude that

- The cloning phenomenon in CRAN impacts quite a lot of packages (up to 25%). However it does not impact the majority of CRAN packages.
- The ratio of packages impacted by cloning appears to have stabilized.
- While cloning impacts very few lines of code compared to the overall size of CRAN, it still impacts more than 250,000 code lines. Moreover, those lines are included in packages that represent around 50% of all code lines in CRAN.

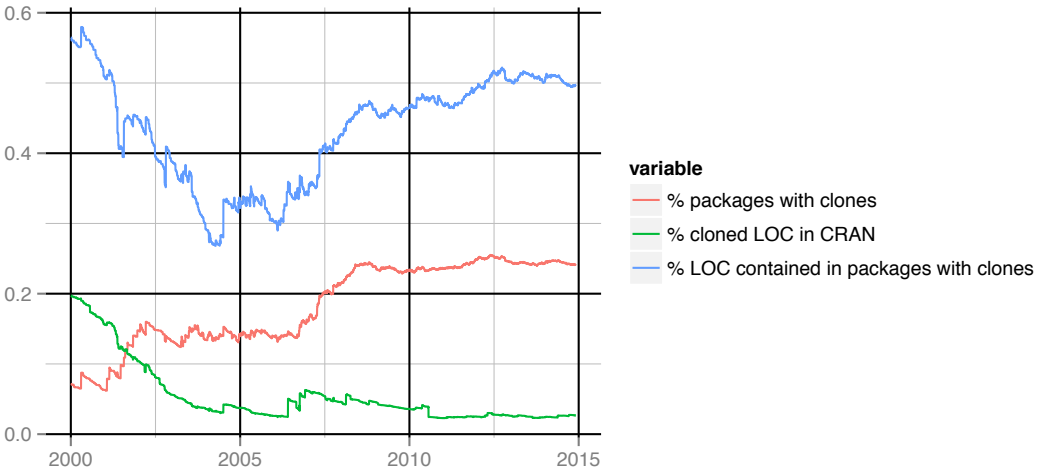


Figure 8.4: Evolution over time of $RoCP(t)$ (in red, the ratio of packages containing clones), $RoCC(t)$ (in green, the ratio of cloned lines of code) and $RoCCP(t)$ (in blue, the ratio of lines of code in those packages containing cloned code).

8.3.3 Why did clones appear?

We have seen previously that cloning potentially impacts hundreds of thousands of lines of code. Our goal is to understand the reason of existence for those clones and whether these clones could have been avoided.

To fulfill this goal we study in more detail snapshot D_t corresponding to date t =2014-12-01. We limit ourselves here to those packages that are not archived at date t. In the previous research question we did not exclude archived packages because there is no history available online to know which packages were archived or not at a certain point in time.

We counted $NoP(t) = 6,253$ non-archived packages. The clone relation C^t_{Inter} resulted in $NoCS(t) = 3,184$ clone sets, involving 7,366 function clones in $NoCP(t) = 1,409$ distinct packages. In total, this amounts to $LoCC(t) = 162,327$ lines of cloned code out of $LoC(t) = 8,338,417$ in total (i.e., $< 2\%$).

In order to understand why these clones exist we studied $origin(C)$, the origin function of each identified clone set C. The origin corresponds to the function with the oldest date, implicitly assuming that all other clones belonging to the same clone set were copied from it. We found exactly 1 origin package for all 3,184 considered clone sets.

For the origin of each clone set, we try to answer the following questions:

- Is the origin *anonymous* (i.e., not stored in any variable)?

- Is the origin declared *locally* (i.e., declared inside another function)?
- Is the origin available as a *public* function to the package users?
- Does the origin still exist in the most recently available package version?

Among the 3,184 considered origin functions (one for each clone set), we identified 796 functions (i.e., 25%) that were either anonymous or local. 250 of these were both anonymous and local.

For the 2,388 (3,184 - 796) origin functions that were globally visible (i.e., not local) in the origin package version, 202 (8.45% of all global origin functions) were no longer available in the latest considered version of the same package, either because the function has been removed or changed somehow over time.

The current CRAN policy requires packages to define a "NAMESPACE" file that lists which functions or objects are exported by the package. Those exported functions are all the functions that the package user is allowed to use[1]. Because NAMESPACE files can use regular expressions and because package environments can be modified dynamically, we extracted the list of exported objects by loading each package in a virtual machine containing a snapshot of CRAN corresponding to the release date of the package.

Out of the 2,186 (2,388 - 202) origin functions that still existed at 2014-12-01, we weren't able to retrieve the list of exported functions for 287 of them (i.e., 13%). Of the remaining 1,899 origin functions, 673 were exported while 1,226 were not.

In summary, it turns out that, for the considered snapshot, cloning cannot be avoided for the majority of clones in each identified clone set. Of the 3,184 origin clones, 25% (796) were local functions that cannot be depended upon no matter whether they are exported or not by their containing package. Of the remaining 1,899 global functions, only 35% (673) were public ones.

Figure 8.5 presents the distribution of number of lines of code $LoC(f)$ for each clone set origin function f. The function size varies a lot, and while most origin functions tend to be small (less than 20 LoC for more than 50% of them), their size can increase up to 1,000 LoC. We also observe that function size tends to be bigger for global than for local origins, and bigger for public clones than for private origins.

[1]Although, technically, it is still possible to call non-exported functions using syntactic sugar, this is strongly discouraged by the CRAN check process.

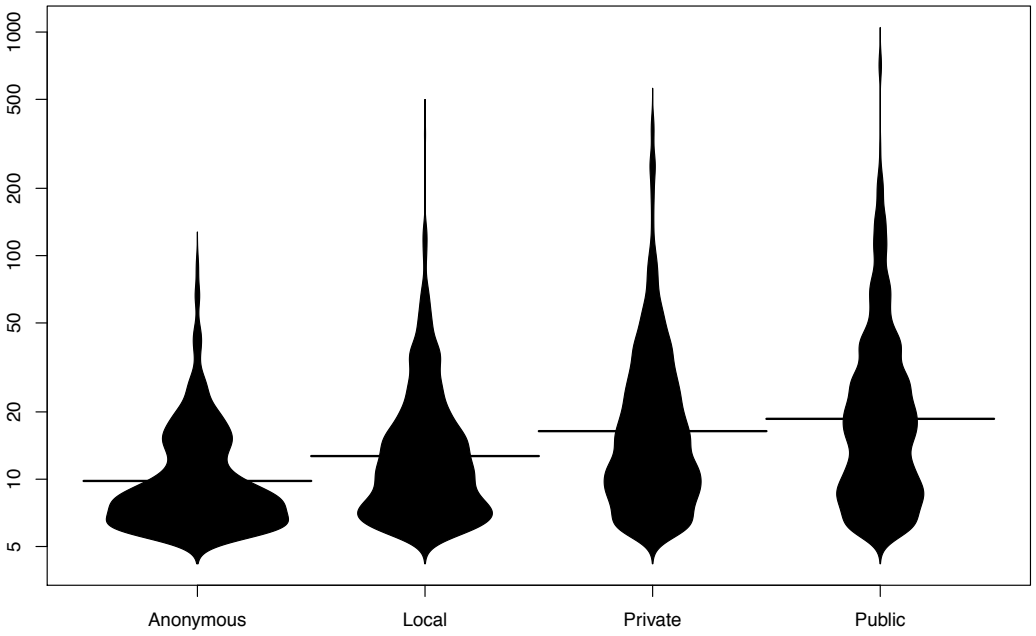


Figure 8.5: Beanplots showing the distribution of $LoC(f)$ for the clone set origins, classified according to their visibility: anonymous, local, (global) private or (global) public.

8.3.4 Is it possible to remove clones? How?

Removing clones by adding a dependency to the origin We have seen that it was only possible for a small fraction of the clone sets to remove identical clones by adding a dependency to the origin function. However, we still need to check whether this dependency already exists or if it could be added. This dependency cannot be added if the package containing the origin (directly or indirectly) depends on the package containing the clone, since otherwise a cyclic dependency would be introduced.

To the previously identified 673 public (and hence, potentially refactorable) origin functions correspond 782 clones in 332 packages. Of these, there were 49 packages with an existing direct dependency to the origin package. In the opposite direction we found 20 packages for which their origin package directly depends upon them and only one for which the origin package indirectly depends upon it.

Removing clones by adding a dependency to a clone of the origin function While the majority of clones cannot be removed by depending upon the package that contains the origin function, perhaps a dependency

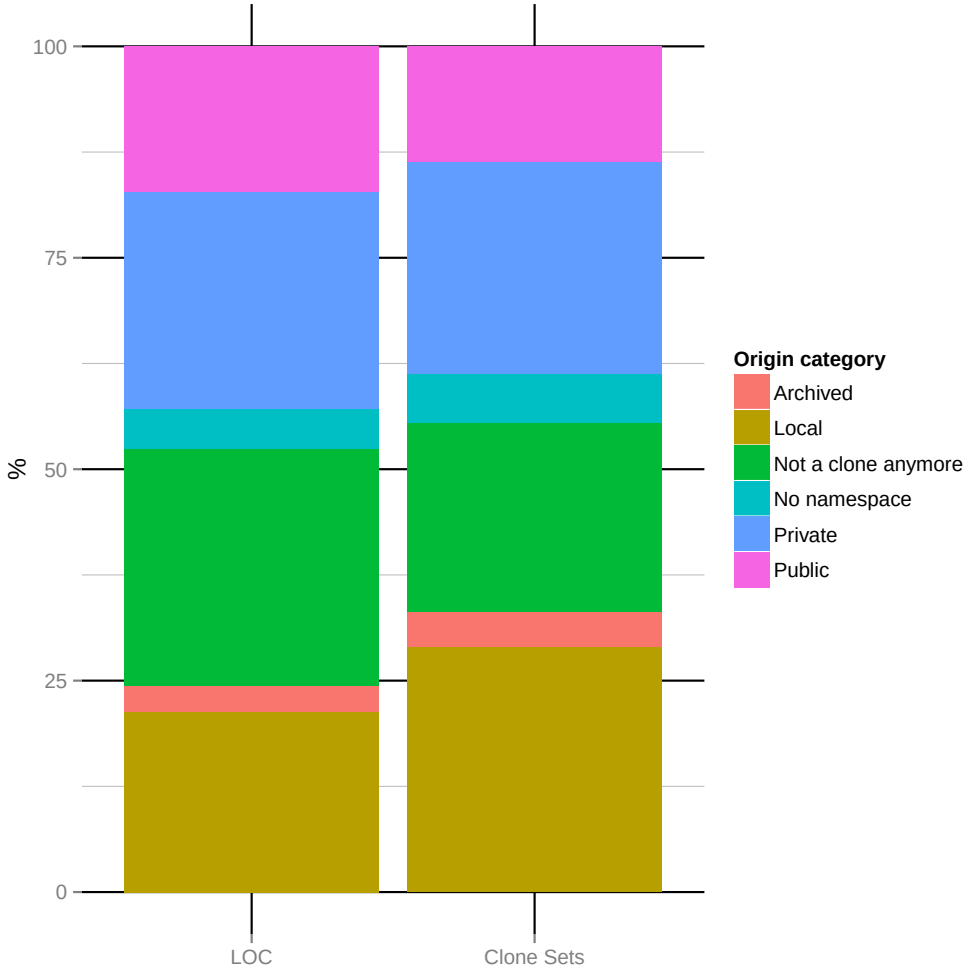


Figure 8.6: Percentage of clone origin functions and clone origin number of lines of code in each category.

can be added to another package containing a clone of the origin.

Let us consider the 3,458 non-origin clones of the 2,511 clone sets for which the origin was not refactorable by adding a dependency. Is one of the non-origin clones refactorable instead? For 801 clone sets, all non-origin clones are all local functions, and for 194 clone sets we were not able to retrieve the list of exported objects by the package. For the remaining global functions for which we could retrieve exported objects, 1,266 were private functions and only 250 were public ones.

For the 250 clone sets containing at least one public clone, there was a total of 317 clones that could potentially be removed by adding a dependency

to the package with this public clone. 31 already have this dependency and 18 have a reverse dependency to the package declaring the public clone, making it impossible to add the dependency.

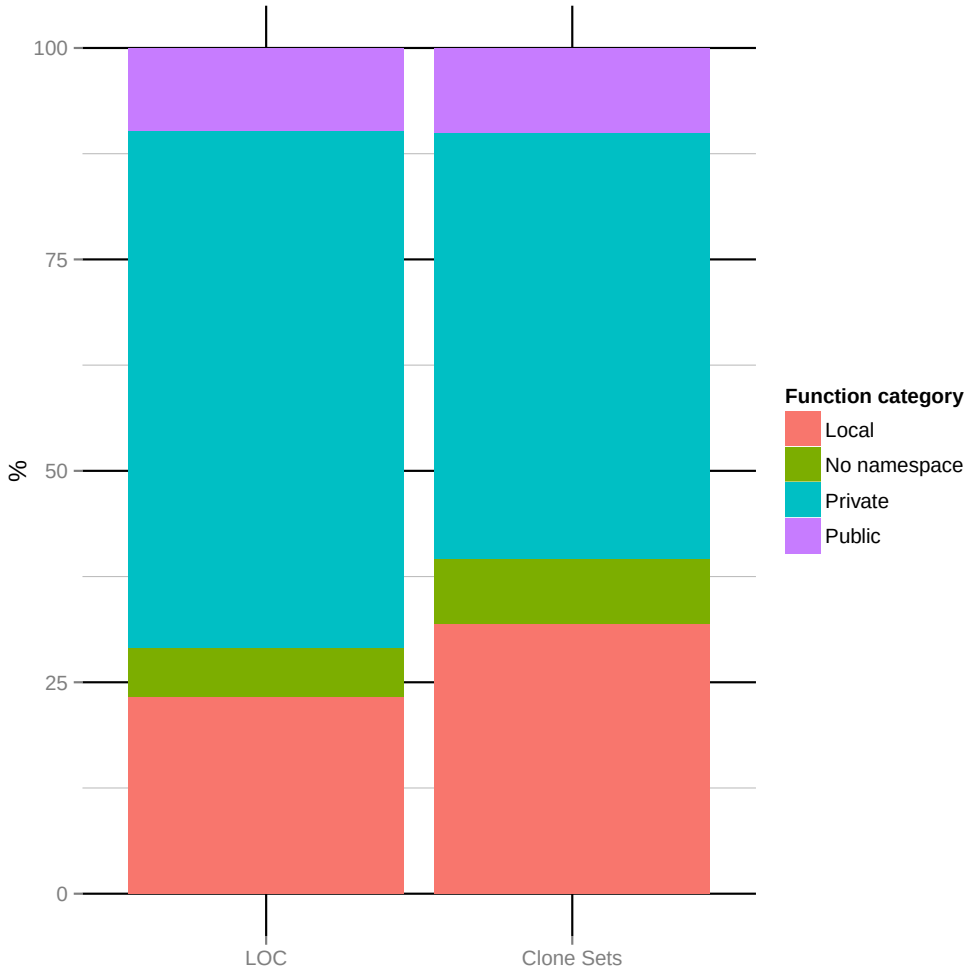


Figure 8.7: Percentage of functions and lines of code in each category for non-origin clones.

8.4 Threats to Validity

Our study has several potential threats to validity. Since we have restricted ourselves to R packages, our results do not necessarily generalize to other package-based systems.

For our analysis we used mainly tools and scripts that we developed ourselves and which could still contain bugs. We also relied on data available on CRAN web site and some of this information could be unreliable. In particular we cannot be sure that the release date of packages is the actual one as it can be misestimated by a few days.

For snapshots before September 2013 there is no way of knowing which packages were present on CRAN at that time. The history of when package versions were archived is not available and we have to rely on data we started to extract since September 2013.

The threshold of at least 6 code lines we have used to consider identical functions to be clones is arbitrary and could lead to over- or underestimations of the number of clones.

8.5 Conclusion

This chapter studied the problem of inter-project software clones from an ecosystemic point of view. To this extent, we carried out an empirical study of Type-1 function clones across R packages contained in the CRAN package repository over a 15-year time period. Our goal was to understand to which extent functions are cloned across packages, why R package maintainers clone functions, and if clones could be avoided.

While identical cloned functions of at least 6 code lines appear to be present in a rather small portion of the code of all packages, they still represent hundred of thousands of lines of code. Moreover, they are present in one out of four packages that together make up half of all CRAN code.

We were able to identify an important number of clones that could theoretically have been avoided by introducing explicit dependencies to another package containing the function clone. Only in very few cases it was not possible to add such a dependency because it would give rise to a cyclic dependency.

We also found valid reasons why cloned functions appeared. In the majority of cases, cloning could not be avoided because the original function being cloned was local or private. This made it technically impossible to reuse the function by simply depending upon the package defining it.

Hence, the problem of identical cloned functions in CRAN appears to be less problematic than what one could expect at first. Nevertheless, we believe that R package maintainers still lack information about, and could benefit from, feedback on the presence of clones in their package and dedicated tools to help them deal with it.

CHAPTER 9

maintaineR: a Dashboard for Analyzing Maintainability Issues

The *R* development community maintains thousands of packages through multiple package repositories. Its two most popular repositories, *CRAN* and *GitHub*, are totaling almost 20 thousand different packages. The growth and evolution of these archives makes it more and more difficult to maintain packages and their inter-dependencies, and the existing tools that aim to help developers in this process no longer suffice. We propose *maintaineR*, a web-based dashboard that allows *CRAN* and *GitHub* package developers to understand and deal with the implications and problems raised by package updates. The dashboard complements existing analysis tools by providing additional support such as the visualization of package dependencies and reverse dependencies, cross-package function clones, and so on.

This chapter is mainly inspired by a conference paper [36] presented at the ICSME 2014 conference.

9.1 Introduction

Many generic web-based dashboards exist to help developer communities with specific maintenance activities. For dedicated software developer communities involved in a specific software ecosystem (including specific programming languages, development processes tools, guidelines, rules and hardware infrastructure), targeted web-based dashboards are however not always available, or need to be extended to accommodate the specific needs of developers.

In this chapter we present *maintaineR*, a dashboard focused towards the *R* project community. In Chapters 6, 7 and 8, we showed that the number of available packages on *CRAN* and *GitHub* is growing rapidly and that package maintainers face issues with package maintenance. In addition, limitations of *R*'s dependency versioning system have been reported and possible directions for improvement (such as staged package distributions and versioned package management) have been proposed [126].

Therefore, there is a need for more specific tools dedicated for R package developers, that allow them to gain insight in, and deal with, the implications and problems raised by dependency updates and dealing with multiple repositories. We developed *maintaineR*, a web-based dashboard to alleviate the above problems. It can be downloaded from `http://github.com/maelick/maintaineR`, together with clear installation instructions. *maintaineR* is more specific and fine-grained than what is currently available to *CRAN* maintainers. It helps them to identify and avoid problems that could break their own package or those of others. The tool is based on a fine-grained function-level analysis of dependencies, conflicts and clones (copy-paste reuse of code) between packages.

Several tools have been proposed to analyze, understand and visualize software ecosystems and their evolution. For example, Neu et al. [123] developed *Complicity*, a web-based application aiming to support software ecosystem analysis through interactive visualisations. Perez et al. [128] presented *SECONDA*, a software ecosystem analysis dashboard.

Many generic web-based tools are available that provide insight in the evolution of software products by analyzing historical data extracted from version control repositories using a combination of metrics, visualization and statistics. Well-known examples of these are SonarQube™ (`http://www.sonarqube.org`) that provides an extensible open source platform for managing code quality, and *GitHub* that includes a variety of views on the version control activity of ongoing projects, including network visualizations and historical visualizations. The main difference is that *maintaineR* offers dedicated support for *CRAN*, taking into account the specificities of the *R*

language and the processes and tools used by *CRAN* package maintainers.

Section 9.2 presents the dashboard's main architecture, and the *R* packages that have been developed and reused for creating it. Section 9.3 presents the tool itself and finally Section 9.4 concludes.

9.2 Overall architecture

Since *maintaineR* targets *R* maintainers, its implementation relies mainly on *R* technologies and *CRAN* packages.

Figure 9.1 presents the overall architecture of the tool, which is divided in two parts. The back-end (left part of figure) fetches, processes and writes data to a database. It serves the purposes of retrieving packages from *CRAN* and *GitHub*, processing them and storing the result of large computations inside a database. The front-end (on the right) consists of a *Shiny* web application. It is used to render and control data queried from the database.

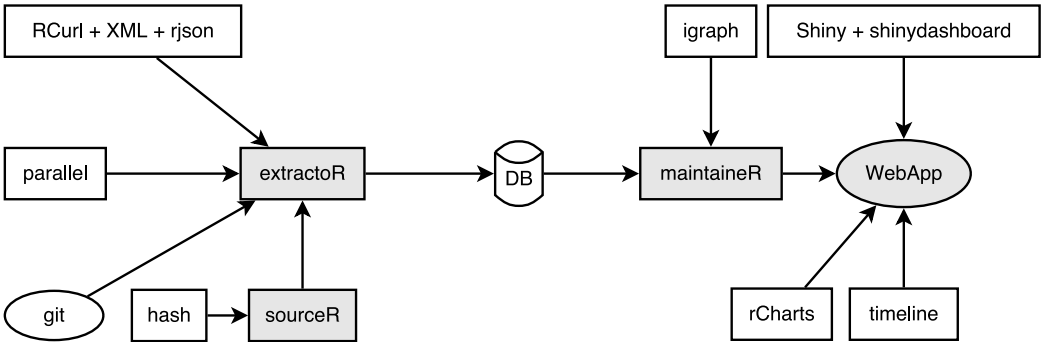


Figure 9.1: Overall architecture of the components required for the web-based dashboard. Rectangles represent *R* packages and circles other tools. Grey elements represent tools developed ourselves; white ones are third-party tools.

The back-end contains different components. Package metadata and content have been extracted with *extractoR*[1], a bundle of *R* packages we initially developed for the purpose of empirically analyzing R packages. We implemented a package *sourceR*[2] to manipulate R code, and more particularly find duplicate code. We enabled *extractoR* to run *sourceR* on R packages. The back-end relies on multiple existing R packages. *RCurl*, *XML*, *rjson* are used to fetch and parse web content. *hash* is used to compute SHA-1 hashes for Type-1 clone detection. Finally *git* is used to fetch and browse the history of *GitHub* repositories containing R packages.

[1] `https://github.com/ecos-umons/extractoR`
[2] `https://github.com/ecos-umons/sourceR`

The front-end is a web-based dashboard[3] built using *Shiny* [4], a web application framework for *R*, and *shinydashboard* [5], an extension of *Shiny* to make web dashboards. For graph manipulations we used the package *igraph*. For visualization inside the dashboard we used packages *timeline* and *rCharts*, which relies on the *d3.js* Javascript library.

9.3 Tool presentation

Executing the *Shiny* web application opens in a web browser the page `http://127.0.0.1:3000`, where 3000 is replaced by the port number on which *Shiny* runs. A list of all available *R* packages appears (as shown in Figure 9.2), and selecting one of these will get the user to a package view. This view is divided in five tabs:

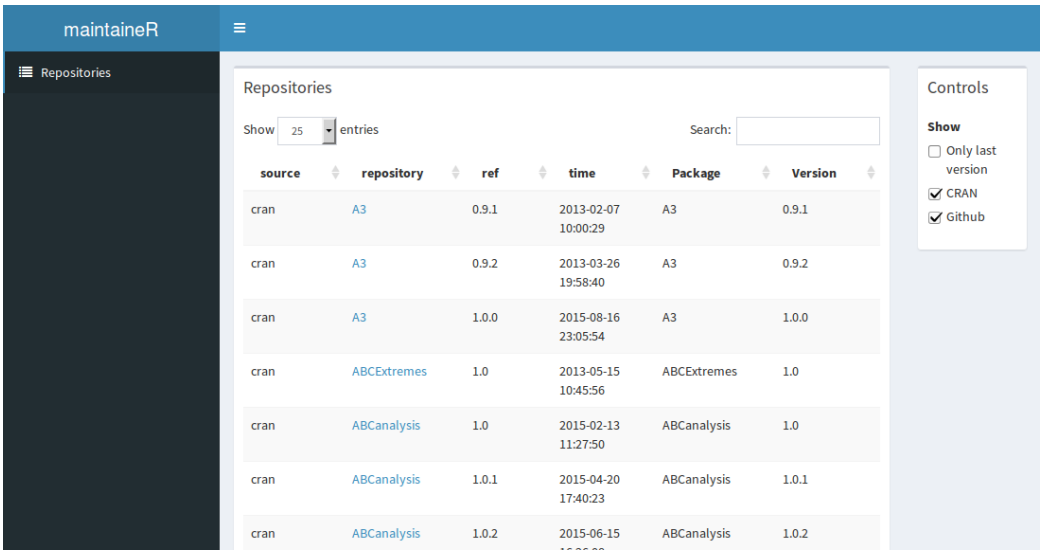


Figure 9.2: Screenshot of the package selection view.

Summary shows a table containing the content of the package's *DESCRIPTION* file.

History shows the release history of the package, its dependencies and/or its reverse dependencies.

[3] `https://github.com/ecos-umons/maintaineR`
[4] `http://shiny.rstudio.com`
[5] `https://rstudio.github.io/shinydashboard/`

Dependencies shows the list or the graph of all dependencies and reverse dependencies of a package.

Namespace shows the content of the namespace of the package.

Clones displays all function clones that are present in other R packages.

9.3.1 Historical view

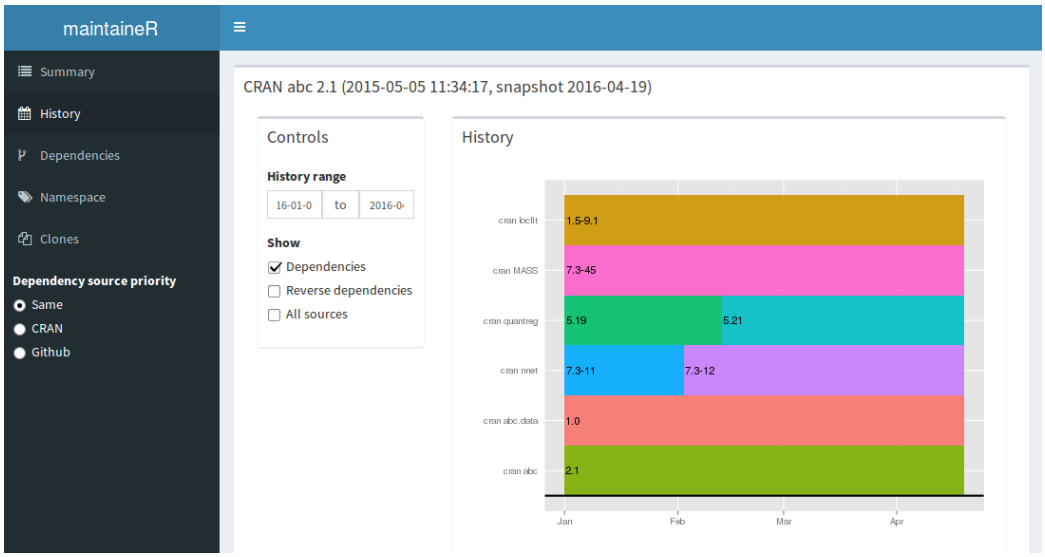


Figure 9.3: Screenshot of the **History** view for release of package `abc` (version 2.1) and its dependencies on *CRAN*.

An example of the historical view is given in Figure 9.3. By default, it shows the release dates of the package on a timeline. As shown in Figure 9.3 it is possible to restrain the timeline to a shorter period and to add dependencies to and reverse dependencies from other packages. When a dependency is available both on *CRAN* and *GitHub*, the dashboard prioritizes by default the one available in the same source as the current package. It is however possible to specify which package source to prioritize ("dependency source priority" in Figure 9.3) or show packages from all sources ("all sources" in Figure 9.3). The ability to visualize globally the history of release dependencies is useful for a developer in order to spot recent changes when the package encounters an error.

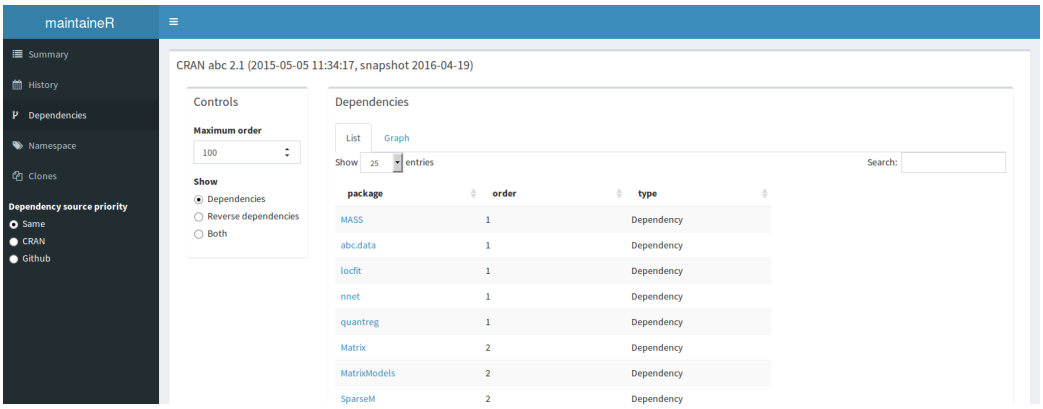


Figure 9.4: **Dependency list** view of dependencies of package *abc* (2.1)

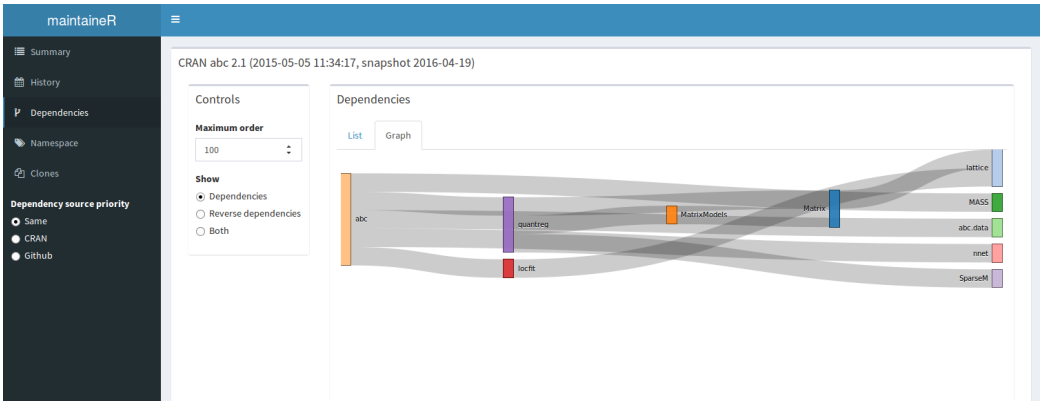


Figure 9.5: **Dependency graph** view of the dependencies of package `abc` (version 2.1), visualized as a Sankey diagram.

9.3.2 Package dependency

The package dependency list view (seen in Figure 9.4) and package dependency graph view (seen in Figure 9.5) allows the visualization of package dependencies and reverse dependencies. Reverse dependencies show the packages that depend on a given package and allow a package maintainer to know who depends upon her package, and eventually help him minimize the ripple effect of any changes he desires to make.

Ideally, a package update should not require updates or changes to packages that depend on it. Showing to the maintainer the dependencies and reverse dependencies of a given package, may help her find the causes of any errors introduced by package dependencies, as well as warn her about potential errors or conflicts introduced by this package in its reverse dependencies.

Similar to the historical view, the dependency view prioritizes dependency based on the "dependency source priority" option.

9.3.3 Namespaces

R resolves function and variable names using environment objects, which are hash tables associating names to objects exported by the package namespace. When a variable or a function is referenced, the interpreter cycles through a list of environments. When two packages define the same function name, the last imported function will mask the first imported one. Although it is still possible to call the first function by specifying the package name with a special notation, this can lead to conflicts. For example, suppose that package A depends on packages B and C and uses a function f defined in B. If a new version of package C introduces a new function with the same name f, there is a chance that this creates a conflict.

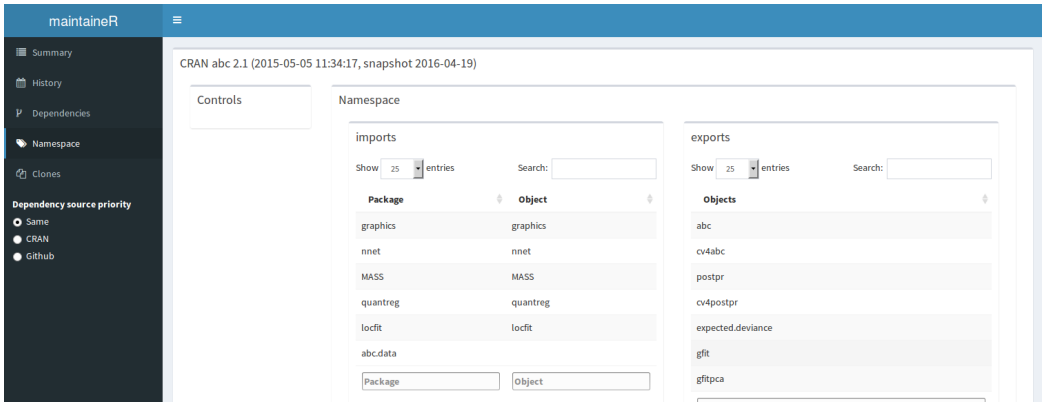


Figure 9.6: **Namespace** view for the *CRAN* package `abc` (version 2.1).

Name	Type	Conflicts
abc	function	gvcm.cat_1.6 forams_2.0-4 pomp_0.49-2
cv4abc	function	None
cv4postpr	function	None
expected.deviance	function	None
postpr	function	None

Figure 9.7: **Conflict** view for the *CRAN* package `abc` (version 2.1).

Figure 9.6 shows the **Namespace** view for a package, displaying objects, such as functions or methods, imported and exported by the package. Additionally it displays the potential conflicts introduced with (particular versions of) other packages.

9.3.4 Function clones

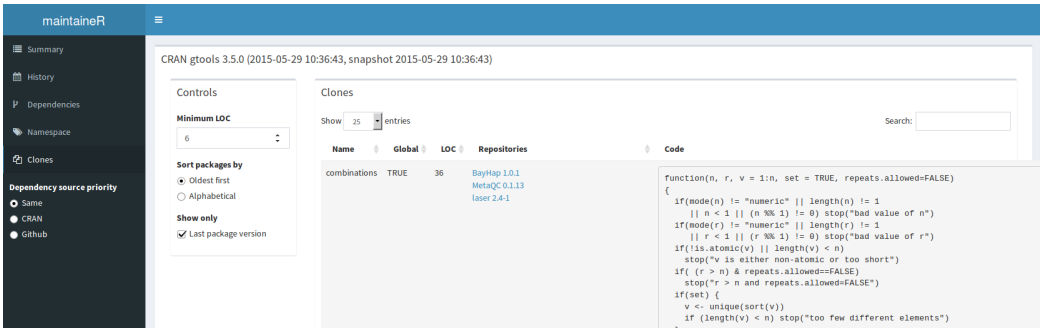


Figure 9.8: **Clones** view for the *CRAN* package `gtools` (version 3.5.0), filtered to show only clones of at least 6 lines and the last version of each package that were available when *gtools* 3.5.0 was released (2015-05-29).

The **Clones** view, displayed in Figure 9.8, determines which global or local functions defined in the package are perfect duplicates (so-called "Type 1" clones) of a function defined in another package. The user interface allows to display only clones that correspond to the last available version of packages.

9.4 Conclusion

We presented *maintaineR*, a web-based dashboard for analyzing maintainability of R packages, by offering visualization and the results of analyses of the package release history, package dependencies, package namespace, potentially conflicting function names across packages, and identical function clones.

CHAPTER 10

Conclusion

In this chapter we summarize the contributions presented in this dissertation. We present the characteristics an ecosystem needs to have in order to replicate each studies presented in Chapters 5, 6, 7 and 8, and compare the result with ours. We also sum up the other limitations of our approach for designing our conceptual framework from Chapter 4, and for conducting empirical analyses in Chapters 5, 6, 7 and 8. Finally we present future research opportunities opened by the contributions of this dissertation.

10.1 Contributions

In Chapter 1 we presented the research context and the research goal of this dissertation. In that chapter we defined the following research goals:

1. To define a conceptual framework in order to be able to extract data from different component repositories, analyze it to study maintainability issues, and interpret the results.

2. To understand dependency relationships between components and how to better deal with them.

3. To develop tools to support developer and user communities of component-based software ecosystems.

After exploring the state of the art in Chapter 2, we defined in Chapter 4 a terminology to facilitate the study of the two package-based ecosystems, the R and Debian ecosystems, used as case studies and presented in Chapter 3. We also presented in Chapter 4 the general methodology to extract data from package repositories, analyze it and produce results.

Subsequent chapters focused on the second research goal. We carried out one empirical analysis on the Debian ecosystem in Chapter 5 and three on the R ecosystem in Chapters 6, 7 and 8.

We empirically analyzed the evolution of *strong conflicts* among Debian packages for 10 years of available history for the i386 architecture. We showed that Debian maintainers make a specific effort to reduce *strong conflicts* as much as possible, which must be accepted only when they describe component incompatibilities that cannot be otherwise eliminated.

Using the statistical technique of survival analysis, we found that

- in general the presence of *strong conflicts* does not impact the lifetime of a package ...

- ... but packages that are always in *strong conflict* have a smaller survival probability than those who are not;

- the longer a package has survived without *strong conflicts*, the less likely it is that a *strong conflict* will appear;

- *strong conflicts* that are already present upon package introduction tend to stay present much longer than *strong conflicts* that appear later;

- half of the *strong conflicts* that appear after package introduction stay a short amount of time (< 1 month).

Using metrics related to the presence, amount and duration of *strong conflicts*, we could identify several packages that have been reported as problematic by the Debian community in the past. We have shown how various of these issues would have been prevented by using recently developed tools, but several issues spotted by our metrics are not captured by any existing tool. This is a strong motivation for introducing these metrics in the future into the repository quality assurance process. Moreover, the simplicity of our metrics makes them easily transposable to other package repositories for which direct conflict relationships are defined.

We studied the general structure of the R ecosystem by considering its major package repositories: *CRAN*, *Bioconductor*, *R-Forge* and *GitHub*. In total, we analyzed the origin and the dependencies of more than 12,000 packages that were available in March 2015.

We observed that *CRAN*, the official R package distribution, is the center of the ecosystem and that other package repositories mainly require *CRAN* packages. On the other hand we observed that *GitHub* is becoming increasingly used as a collaborative development platform for R packages, both for packages already distributed on *CRAN* and *BioConductor*, as well as for new packages that do not have any counterpart in the considered distributions or forges. This increased use of *GitHub* does not seems to have any positive or negative effect on the growth of *CRAN* or *BioConductor*.

While *GitHub* packages distributed on *CRAN* tend to be older than those that are not, their age cannot fully explain whether or not they are distributed through *CRAN*. Additionally, many R package developers make use of *GitHub* as a distribution platform. Their packages contain instructions to be installed from *GitHub*, and are often exclusively distributed through *GitHub*.

With the aim to assess the maintainability of R packages we collected daily information from the CRAN website of the automated check of each package for each flavor. We observed that some packages are more error prone than others on specific operating systems such as MacOS X and Solaris. These systems have a high number of packages with errors that are introduced by the developers of the package itself when they release it. For the Windows and Linux operating systems, an important fraction of errors were unrelated to any change in the code of the package itself or any package dependency.

The majority of errors are resolved within a few days without any developer intervention. The errors that need to be fixed by the package developers are primarily errors introduced when a package they rely upon was modified. This means that package updates often cause backward incompatible change in dependent packages and that maintenance effort hence needs to be focused on fixing errors caused by others. While we couldn't conduct the same analysis on *GitHub* packages because of the lack of results for a similar checking

process, we hypothesize this problem is potentially worse for *GitHub* packages because *CRAN* packages that are required by *GitHub* packages are more prone to be updated than other *CRAN* packages.

Finally we studied the problem of inter-project software clones from an ecosystemic point of view in *CRAN* packages. While identical cloned functions of at least 6 lines of code appear to be present in a rather small portion of the code of all packages, they represent hundred of thousands of lines of code. Moreover, they are present in one out of four packages that together make up half of all *CRAN* code.

While the majority of inter-package function clones couldn't have been avoided by relying on a simple dependency relationships, there are still tens of thousands lines of code that could be removed using dependencies.

In summary we gained the following general insights on maintainability issues in component-based software ecosystems:

- The presence of *strong conflicts* has a higher chance of impacting a package lifetime if it appeared when the package was introduced in the ecosystem and when this conflict cannot be fixed.
- Conflicts take more time to be fixed when they are introduced at the same time as the package.
- Using simple metrics, a historical analysis of strong conflicts between packages reveal problems not detected by existing tools.
- Backward incompatible updates are responsible for the majority of errors that require an update of the package to be fixed.
- Inter-project clones are sometimes a necessity and cannot always be removed by relying on dependencies.

Finally we presented in Chapter 9 *maintaineR*, a web-based dashboard for supporting R package maintainer, by offering visualizations and the results of analyses of the package release history, package dependencies, package namespace, potentially conflicting function names across packages, and identical function clones. Contrary to existing tools available to the R community, this dashboard allows to do these visualization both on *CRAN* and *GitHub* packages.

10.2 Generalizability

One of the major limitation of this thesis is the potential lack of generalization of the results. Indeed, for each of the empirical studies, we restricted ourselves

to a single case study: Debian for Chapter 5 and R for Chapters 6, 7 and 8. Therefore these studies needs to be reproduced on other component-based software ecosystems to assess whether our findings can be generalized [50]. A possible future topic of research would be trying to generalize the results to other component-based software ecosystems.

While the conceptual framework presented in Chapter 4 aims at facilitating reproduction of our experiments on other ecosystems, constraints that could prevent such replication study. For example dependency relationships are defined as conjunction of packages but don't currently allow using disjunctions of packages. The framework was sufficient for the study of Debian because we relied on an external tool to our framework to compute *strong conflicts* between packages.

It would be interesting to study other ecosystems, and compare the differences and similarities with our results. We present here the different characteristics that a component-based software ecosystem needs to have in order to replicate our studies and compare results with the ones we obtained.

For example, the study conducted in Chapter 5 should be replicated on other Linux distributions such as Fedora and Ubuntu. One good candidate to replicate the studies on R would be the *Perl* modules available on *CPAN*[1] and *GitHub*. Like *CRAN*, *CPAN* is a long-lived repository containing thousands of software components. While it is older and contains more modules than *CRAN* contain packages, it does not have a policy as strong as *CRAN*. Replicating our studies on this ecosystem and comparing results could give insights on how a strong policy, like *CRAN*'s, influence maintainability issues.

Replicating the historical study of co-installability issues The replication on another ecosystem of the study of Chapter 5 requires to have access to daily snapshots of available components and *strong conflicts* between them. In order to compare the results, each snapshot must contain only one version of each package. Moreover, the information on *strong conflicts* should ideally be computed using the coinst tool. This is possible if components of the target ecosystem have the following characteristics:

- A **name** to uniquely identify the component;
- **Version** tags to uniquely identify the different versions of the component;
- The **period of availability** of each version in order to compute the daily snapshots;

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

- A list of **dependencies**, defined as a conjunction of disjunction of component names, with optional version constraints;
- A list of **conflicts**, defined as a conjunction of component names, with optional version constraints.

Replicating the study of the topology of the R ecosystem While many results presented in Chapter 6 are very specific to the R ecosystem, the study could be replicated on any component-based ecosystem with different repositories or distributions. The replication would be possible if components of the target ecosystem have the following characteristics:

- A **name** to uniquely identify the component;
- **Version** tags to uniquely identify the different versions of the component;
- The **period of availability** of each version;
- A list of **dependencies**, defined as a conjunction of component names.

Replicating the study of the maintainability issues The replication on another ecosystem of Chapter 7, requires the target ecosystem to have a daily checking process of all packages similar to *R CMD check*. More specifically, each component of the target ecosystem needs to have the following characteristics:

- A **name** to uniquely identify the component;
- **Version** tags to uniquely identify the different versions of the component;
- The **period of availability** of each version in order to compute daily snapshots;
- A **period of error state** of each version, in order to identify when a version is broken;
- A list of **dependencies**, defined as a conjunction of component names.

Replicating the study of inter-package clones The replication on another ecosystem of the study of Chapter 8, requires the target ecosystem to have component with the following characteristics:

- A **name** to uniquely identify the component;
- **Version** tags to uniquely identify the different versions of the component;
- The **period of availability** of each version;
- A list of **dependencies**, defined as a conjunction of component names.

Moreover, components need to contain code defined in a single programming language with **reusable code fragments**. A code fragment is reusable if the programming language provides a way to easily call it. While most programming languages have such code fragments, the paradigm of the programming language could create some differences in the results or their implications. For example it might be more difficult to reuse a method in another context than a function.

In addition, it should be possible to efficiently identify all **identical code fragments** across all components of the ecosystem. In order to compare the results with those of Chapter Chapter 8, the programming language of these code fragments should also provide some **modularity mechanism** in order to restrict the scope of the reusable code fragments, such as private and public functions or methods.

10.3 Limitations

In Chapters 5, 6, 7 and 8, we listed the different threats to validity of the results of the empirical study conducted. In this section we present what we believe are the major limitations of the approach followed in this dissertation to study maintainability issues in component-based software ecosystems.

10.3.1 Package extraction

For all empirical studies presented in this dissertation, we relied on package data or meta-data available from online repositories or archives. The first major limitation of our approach is the reliability of these repositories and archives.

For the Debian case study we used daily snapshots of package metadata that are available in Debian archives[2]. While the availability of these snapshots gives a reliable source of information to know which packages were available or not at a given point in time, these snapshots are not available before March 2006. However we believe that the last 10 years of history are sufficient to obtain significant results.

While *CRAN* packages are supposed to never be removed and only archived, we have noticed a few packages or package versions that have disappeared over time since we started extracting *CRAN* packages in September 2013. Even though we have extracted daily snapshots of non-archived *CRAN* packages, they are limited to a period beginning in September 2013. Moreover there is no public archive of such snapshots.

GitHub packages suffer from even more issues regarding package extraction. There is no list of all publicly available R packages hosted on *GitHub*. We tried to build such a list using both *GithubArchive*[3] and the official *GitHub* API[4] to first get the list of all *GitHub* repositories.

Despite *GithubArchive* containing a list of the daily events of all *GitHub* repositories, it is not exempt from missing data. Although using *GitHub* API might be more reliable to list all currently available *GitHub* repositories, it can't be used to get the ones that have been removed. In both cases, these approaches only give a list of *GitHub* repositories tagged with the R language. Repositories tagged with no or another language might also contain R packages. We considered as a R package the repositories that contain a *DESCRIPTION* file at the root of the latest version of their master branch.

This approach only allows to accurately measure the number of R packages available on *GitHub* for the date of extraction. Computing the number of packages available for previous points in time might result in an under-approximation of the actual value.

We know that there are packages that are completely ignored using this methodology because they are contained in a sub-directories of their *GitHub* repository. This is the case for *extractoR*, the package we developed for extracting data from R packages. This is also the case for package *feather*[5] that is contained in the "R" sub-directory of its *GitHub* repository. Moreover its repository is tagged with the *C++* language as it also contains Python and C++ code.

It is important to note that there are two major reasons for not listing packages in sub-directory. First there were more than 140,000 non-forked

[2] `http://snapshot.debian.org/archive/debian`
[3] `https://www.githubarchive.org/`
[4] `https://developer.github.com/v3/`
[5] `https://github.com/wesm/feather`

GitHub repositories tagged in April 2016. It is impractical to check for sub-directories either by directly making requests to *GitHub*, or by cloning all repositories and searching for *DESCRIPTION* files on disk. Secondly there are many R repositories that contain scripts and copies of the packages required by those scripts. Thus listing all R packages included in sub-directories would highly over-estimate the actual number of R packages hosted on *GitHub*.

10.3.2 Identifying distributed *GitHub* R packages

For all identified R packages we checked whether they were ready for release by verifying the presence of a README file with specific installation instructions. Although a manual verification did not reveal any false positives, there may have been false negatives that we have not considered.

More importantly we identified packages that contained installation instructions, such as calls to the *install_github* function from package *devtools*. Because *devtools* is a development tool, it doesn't mean that a package README providing such installation instructions is ready for production. Identifying packages that are considered ready for production by their developers is something not obvious, even for a human being. One solution to check for this would be asking to the package maintainer.

10.3.3 Identifying errors in R packages

In Chapter 7 we identified errors in packages relying on snapshots of results of the official *R CMD check* tool used on *CRAN*. We know that many errors appear or disappear without any change in the package or one of its strong dependencies. This could influence the results for packages with many dependencies. Indeed such an error could be introduced or fixed at the same time as an update of the package or one of its transitive dependencies.

10.4 Future Work

Our work can be extended in multiple ways but can also lead to further research topics. In this section we first present how the empirical analysis conducted in previous chapters could be extended. Then we give directions on how the tools we developed in the context of this thesis could be extended. Finally we give possible research directions based on the results from the empirical studies of this thesis.

10.4.1 Empirical study extension

We confirmed our findings on the problems of maintainability of R packages by conducting e-mail interviews [114] with a few R package developers who maintain packages on *GitHub*. However these results are not necessarily representative of the entire community. It is not clear how these problems are currently perceived and addressed by developers and maintainers of R packages. One possible extension of the study from Chapter 7 is a more extensive survey to get better insight in the use of *GitHub* as a package distribution platform and its impact on the R ecosystem.

Similarly, in Chapter 8, we studied the presence of Type-1 function clones across CRAN packages in an objective way. Our findings would benefit from performing a subjective survey with actual R package maintainers. In particular, we wish to know to which extent they are aware of cloning behavior and purposefully resort to the practice of code cloning, and if they perceive the presence of clones as something good or bad.

Related to Chapter 8, we only considered Type-1 clones. A direct extension would be to consider Type-2 clones as well. For example, it would be interesting to find out whether Type-1 clones become Type-2 clones, and how long it takes. The same could be done with Type-3 clones.

10.4.2 Tooling

While in Chapter 5 we studied the evolution of *strong conflicts* in Debian and proposed a simple way of detecting problems related to them using a historical analysis, we did not provide an actionable tool to support the Debian community.

The different techniques employed in this study may be aggregated into a metrics-based dashboard targeted to Debian package maintainers and users. It could also be replicated for all supported architectures, besides the i386 we studied here, and integrated into a platform such as *Debsources*, that has been specifically created to analyze and reason about the evolution of the Debian distribution [28].

Similarly, *maintaineR* does not integrate some of the results presented in this dissertation. For instance it does not report anything about the *R CMD check* result evolution. It could also be extended with information from other package repositories such as *Bioconductor*.

Finally our work on clone detection in R packages could lead to a general clone detection and refactoring tool for R. *sourceR*, the R package we developed to identify inter-package Type-1 function clones is currently a first step in that direction. Indeed it is able to detect such clones between two packages,

inside a single package or inside any piece of R code such as a script. *sourceR* could be extended in three different directions. First it could easily propose a way to automatically remove the clones found by adding the appropriate dependencies between packages. Secondly it could be extended to detect Type-1 clones that are not functions. Finally it could also be extended to detect Type-2 and Type-3 clones.

10.4.3 Future topics of research

The interviews with R developers revealed that the lack of dependency constraints on R packages can be problematic. While this problem can't be solved without implication from the official maintainers of R itself, one possible direction could be automatically generating such constraints based on static code analysis.

Being able to generate such constraints could be used as the basis for future research studies on the R ecosystem. It could be used as a proxy to measure the quality of a package and assess how reliable the package is, i.e., how easily it can be installed or reused depending on the distribution it comes from.

R package meta-data does not allow to declare and take into account conflict relationships, *strong conflicts* could appear because of dependency constraints. Moreover incompatibilities between packages could also appear at run-time. Extending our conceptual framework to take into account those dependencies and constraints would allow to study *strong conflicts* in the R ecosystem.

Another possible research topic is the study of archived packages on *CRAN*. Using the daily snapshots of available packages on *CRAN*, it should be possible to see how often archived packages are taken over by a new maintainer, whether their development continues actively on another development forge like *GitHub*, or whether they simply stay inactive.

In this thesis we focused on technical aspects of maintainability issues in component-based software ecosystems. Another future research track concerns a *socio-technical analysis* of these maintainability issues. Previous research have conducted in this topic and in particular in the Ruby ecosystem [145]. Package author and maintainer information could be used to carry out a socio-technical analysis of the ecosystem. Similarly, other data sources pertaining to package development could be used, such as mailing lists, issue trackers, activity on Q&A websites such as Stack Overflow, download statistics, and many more.

These additional data sources would allow us to answer a whole range of new questions such as the followings. How does social interaction influence

strong conflicts between components? Are backward incompatibilities more likely to appear between packages maintained by different people? Do we observe similarities in terms of cloning behavior inside packages maintained by the same person? Is cloning more frequent between packages developed by the same persons? How much overlap exists between communities such as *GitHub* and *CRAN*?

Finally, we wish to study the impact of maintainability issues on reproducible research. Because it is first a language for statistical analysis and data manipulation, R is often used by researchers for data analysis. There exist repositories containing reproducible research results containing R code such as the *R-Journal*[6]. We could use the tools we developed and the insight we gained on maintainability issues to check whether or not the code included in the *R-Journal* is really reproducible and does not encounter errors over time. Moreover we could asses whether these problems can be avoided by ongoing initiatives to facilitate R package management and producing reproducible results. Examples are the *Drat R Archive Template*[7], the *Managed R Archive Network* [8], the *Reproducible R Toolkit*[9], that provides an R function checkpoint, which ensures that all of the necessary R packages are installed with the correct version; and *packrat*, a portable and reproducible dependency management system for R projects.

[6] `https://journal.r-project.org/`
[7] `https://github.com/eddelbuettel/drat`
[8] `http://mran.revolutionanalytics.com/packages`
[9] `http://projects.revolutionanalytics.com/rrt`

List of Figures

List of Tables

Bibliography

[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, Ralf Treinen, and Stefano Zacchiroli. A modular package manager architecture. *Information and Software Technology*, 55(2):459–474, 2013.

[4] 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.

[5] 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.

[6] 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.

[7] Pietro Abate and Ralf Treinen. The dose-debcheck primer. `https://gforge.inria.fr/docman/view.php/4395/8241/debcheck-primer.html`, October 2012. retrieved November 2012.

[8] Roberto Abreu and Rahul Premraj. How developer communication frequency relates to bug introducing changes. In *Joint Int'l Workshop*

on Principles of software evolution (IWPSE) and ERCIM software evolution workshop, pages 153–158. ACM, 2009.

[9] Anonymous R package maintainer. Private e-mail communication, November 2015.

[10] 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.

[11] Brenda S. Baker. On finding duplication and near-duplication in large software systems. In *Working Conf. Reverse Engineering (WCRE)*, pages 86–95, 1995.

[12] 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.

[13] Pamela Battacharya, Liudmila Ulanova, Iulian Neamtiu, and Sai Charan Koduru. An empirical analysis of bug reports and bug fixing in open source Android apps. In *European Conf. Software Maintenance and Reengineering (CSMR)*, pages 133–143. IEEE Computer Society, 2013.

[14] 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.

[15] 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.

[16] Karl Beecher, Andrea Capiluppi, and Cornelia Boldyreff. Identifying exogenous drivers and evolutionary stages in FLOSS projects. *Journal of Systems and Software*, 82(5):739–750, 2009.

[17] A. Begel, R. DeLine, and T. Zimmermann. Social media for software engineering. In *Workshop on Future of Software Engineering Research*, 2010.

[18] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore M. Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. Soft. Eng.*, pages 577–591, September 2007.

[19] Christian Bird, Alex Gourley, Prem Devanbu, Michael Gertz, and Anand Swaminathan. Mining email social networks. In *Int'l Conf. Mining Software Repositories (MSR)*, pages 137–143. ACM Press, 2006.

[20] Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. Germán, and Premkumar T. Devanbu. The promises and perils of mining Git. In *Int'l Conf. Mining Software Repositories (MSR)*, pages 1–10. IEEE Computer Society, 2009.

[21] Kelly Blincoe, Francis Harrison, and Daniela Damian. Ecosystems in github and a method for ecosystem identification using reference coupling. In Massimiliano Di Penta, Martin Pinzger, and Romain Robbes, editors, *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015*, pages 202–211. IEEE, 2015.

[22] Jaap Boender and Sara Fernandes. Small world characteristics of FLOSS distributions. In *Software Engineering and Formal Methods Collocated Workshops – Revised Selected Papers*, pages 417–429, 2013.

[23] Jan Bosch and P. Bosch-Sijtsema. From integration to composition: on the impact of software product lines, global development and ecosystems. In *Int'l Conf. Software Product Lines*. Springer, 2009.

[24] Karl Broman. R package primer – a minimal tutorial. `http://kbroman.org/pkg_primer/`, 2015.

[25] John Businge, Alexander Serebrenik, and Mark G. J. van den Brand. An empirical study of the evolution of Eclipse third-party plug-ins. In *EVOL/IWPSE*, pages 63–72, 2010.

[26] John Businge, Alexander Serebrenik, and Mark G. J. van den Brand. Survival of Eclipse third-party plug-ins. In *Int'l Conf. Software Maintenance (ICSM)*, pages 368–377, 2012.

[27] John Businge, Alexander Serebrenik, and Mark G. J. van den Brand. Analyzing the Eclipse API usage: Putting the developer in the loop. In *European Conf. Software Maintenance and Reengineering (CSMR)*, pages 37–46. IEEE Computer Society, 2013.

[28] Matthieu Caneill and Stefano Zacchiroli. Debsources: Live and historical views on macro-level software evolution. In *Int'l Symp. Empirical Software Engineering and Measurement*, page 28, 2014.

[29] A. Capiluppi and K. Beecher. Structural complexity and decay in FLOSS systems: An inter-repository study. In *European Conf. Software Maintenance and Reengineering (CSMR)*, pages 169–178, March.

[30] Andrea Capiluppi, Patricia Lago, and Maurizio Morisio. Characteristics of open source projects. In *European Conf. Software Maintenance and Reengineering (CSMR)*, pages 317–327. IEEE Computer Society, 2003.

[31] Andrea Capiluppi, Maurizio Morisio, and Juan Fernandez Ramil. The evolution of source folder structure in actively evolved open source systems. pages 2–13. IEEE Computer Society, 2004.

[32] Andrea Capiluppi, Maurizio Morisio, and Juan Fernandez Ramil. Structural evolution of an open source system: A case study. In *Int'l Workshop Program Comprehension*, pages 172–182, 2004.

[33] Andrea Capiluppi and Juan F. Ramil. Studying the evolution of open source systems at different levels of granularity: Two case studies. In *Int'l Workshop on Principles of Software Evolution*, pages 113–118. IEEE Computer Society, 2004.

[34] Maëlick Claes, Decan Alexandre, and Tom Mens. Inter-component dependency issues in software ecosystems. In *IEEE Book to be published in 2016*. 2016.

[35] 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.

[36] 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.

[37] Maëlick Claes, Tom Mens, and Philippe Grosjean. On the maintainability of cran packages. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 308–312. IEEE, 2014.

[38] 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.

[39] Eric Clayberg and Dan Rubel. *Eclipse: Building Commercial-Quality Plug-ins*. Addison-Wesley Professional, 2 edition, April 2006.

[40] 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.

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

[42] Laura A. Dabbish, H. Colleen Stuart, Jason Tsay, and James D. Herbsleb. Social coding in github: transparency and collaboration in an open software repository. In *Int'l Conf. Computer Supported Cooperative Work*, pages 1277–1286, 2012.

[43] 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.

[44] Alexandre Decan, Tom Mens, Maelick Claes, and Philippe Grosjean. When github meets CRAN: An analysis of inter-repository package dependency problems. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016.

[45] Tom DeMarco and Timothy Lister. *Peopleware: Productive Projects and Teams*. Dorset House, 2nd edition, February 1999.

[46] 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.

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

[48] 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.

[49] Trung T. Dinh-Trong and James M. Bieman. The FreeBSD project: A replication case study of open source development. *IEEE Trans. Soft. Eng.*, 31(6):481–494, 2005.

[50] Chris Drummond. Replicability is not reproducibility: Nor is it good science.

[51] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Int'l Conf. Software Maintenance (ICSM)*, pages 109–118, September 1999.

[52] N. Ducheneaut. Socialization in an open source software community: A socio-technical analysis. *Computer Supported Cooperative Work (CSCW)*, 14(4):323–368, August 2005.

[53] P. Meier E.L. Kaplan. Nonparametric estimation for incomplete observations. *J. American Statistical Association*, 53(282):457–481, 1958.

[54] Brian Fitzgerald. The transformation of open source software. *MIS Quarterly*, 30(3), 2006.

[55] R. Tynan G. Madey, V. Freeh. The open source software development phenomenon: An analysis based on social network theory. In *Americas Conf. on Information Systems*, pages 1806–1813, 2002.

[56] Santiago Gala-Pérez, Gregorio Robles, Jesús M. González-Barahona, and Israel Herraiz. Intensive metrics for the study of the evolution of open source projects: case studies from apache software foundation projects. In Thomas Zimmermann, Massimiliano Di Penta, and Sunghun Kim, editors, *Int'l Conf. Mining Software Repositories (MSR)*, pages 159–168. IEEE / ACM, 2013.

[57] José A Galindo, David Benavides, and Sergio Segura. Debian packages repositories as software product line models. towards automated analysis. In *Int'l Workshop on Automated Configuration and Tailoring of Applications (ACoTA)*, pages 29–34, 2010.

[58] Harald Gall, Mehdi Jazayeri, René Klösch, and Georg Trausmuth. Software evolution observations based on product release history. In *Int'l Conf. Software Maintenance (ICSM)*, pages 160–166. IEEE Computer Society, September 1997.

[59] Daniel M. Germán. The GNOME project: a case study of open source, global software development. *Software Process: Improvement and Practice*, 8(4):201–215, 2003.

[60] Daniel M. Germán, Bram Adams, and Ahmed E. Hassan. The evolution of the R software ecosystem. In *European Conf. Software Maintenance and Reengineering (CSMR)*, pages 243–252, 2013.

[61] 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.

[62] GitHub ANTsR issue 8. ANTsR: towards CRAN & standardization of development. `https://github.com/stnava/ANTsR/issues/8`, 2015.

[63] Michael W. Godfrey and Qiang Tu. Evolution in open source software: A case study. In *Int'l Conf. Software Maintenance (ICSM)*, pages 131–142. IEEE Computer Society, 2000.

[64] Mathieu Goeminne and Tom Mens. A framework for analysing and visualising open source software ecosystems. In *Int'l Workshop on Principles of Software Evolution*, pages 42–47, 2010.

[65] Mathieu Goeminne and Tom Mens. Evidence for the Pareto principle in open source software activity. In Magiel Bruntink and Kostas Kontogiannis, editors, *Workshop on Software Quality and Maintainability (SQM)*, volume 701, pages 74–82. CEUR-WS.org, 2011.

[66] Mathieu Goeminne and Tom Mens. Analyzing ecosystems for open source software developer communities. In Slinger Jansen, Michael Cusumano, and Sjaak Brinkkemper, editors, *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry*. Edward Elgar, 2013.

[67] Jesus M. Gonzalez-Barahona, Gregorio Robles, and Daniel Izquierdo-Cortazar. The metricsgrimoire database collection. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 478–481, Piscataway, NJ, USA, 2015. IEEE Press.

[68] Jesús M. González-Barahona, Gregorio Robles, Martin Michlmayr, Juan José Amor, and Daniel M. Germán. Macro-level software evolution: a case study of a large software compilation. *Empirical Software Engineering*, 14(3):262–285, 2009.

[69] Jesus M. Gonzalez-Barahona, Gregorio Robles, Miguel Ortuño-Pérez, Luis Rodero-Merino, José Centeno-González, Vicente Matellán-Olivera, Eva M. Castro, and Pedro de las Heras Quirós. *Analyzing the anatomy of GNU/Linux distributions: methodology and case studies (Red Hat and Debian)*, chapter 2, pages 27–58. Idea Group Publishing, Hershey, Pennsylvania, USA, 2005.

[70] Georgios Gousios and Diomidis Spinellis. Ghtorrent: Github's data from a firehose. In Michele Lanza, Massimiliano Di Penta, and Tao Xie, editors, *9th IEEE Working Conference of Mining Software Repositories, MSR 2012, June 2-3, 2012, Zurich, Switzerland*, pages 12–21. IEEE Computer Society, 2012.

[71] Georgios Gousios, Bogdan Vasilescu, Alexander Serebrenik, and Andy Zaidman. Lean ghtorrent: Github data on demand. In Premkumar T. Devanbu, Sung Kim, and Martin Pinzger, editors, *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, pages 384–387. ACM, 2014.

[72] D. P. Harrington and T. R. Fleming. A class of rank test procedures for censored survival data. *Biometrika*, 69:553–566, 1982.

[73] I. Herraiz, G. Robles, J. M. Gonzalez-Barahona, A. Capiluppi, and J. F. Ramil. Comparison between SLOCs and number of files as size metrics for software evolution analysis. In *European Conf. Software Maintenance and Reengineering (CSMR)*, Bari, Italy, March 2006.

[74] Yoshiki Higo, Shinji Kusumoto, and Katsuro Inoue. A metric-based approach to identifying refactoring opportunities for merging code clones in a Java software system. *J. Software Maintenance and Evolution: Research and Practice*, 20(6):435–461, 2008.

[75] 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.

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

[77] Ayelet Israeli and Dror G. Feitelson. The Linux kernel as a case study in software evolution. *J. Systems and Software*, 83(3):485–501, 2010.

[78] S. Jansen, A. Finkelstein, and S. Brinkkemper. A sense of community: A research agenda for software ecosystems. In *Int'l Conf. Software Engineering (ICSE)*, pages 187–190, May 2009.

[79] Slinger Jansen, Sjaak Brinkkemper, and Anthony Finkelstein. Business Network Management as a Survival Strategy: A Tale of Two Software

Ecosystems. In *Int'l Workshop on Software Ecosystems*, pages 34–48, Sun Site, 2009. CEUR.

[80] Slinger Jansen, Michael Cusumano, and Sjaak Brinkkemper, editors. *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry.* Edward Elgar, 2013.

[81] Corey Jergensen, Anita Sarma, and Patrick Wagstrom. The onion patch: migration in open source ecosystems. In *ESEC/FSE*, pages 70–80, New York, NY, USA, 2011. ACM.

[82] Y. Jia, D. Binkley, M. Harman, J. Krinke, and M. Matsushita. KClone: A proposed approach to fast precise code clone detection. In *Int'l Workshop on Software Clones*, 2009.

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

[84] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. Germán, and Daniela Damian. The promises and perils of mining GitHub. In *Int'l Conf. Mining Software Repositories (MSR)*, pages 92–101, 2014.

[85] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. Soft. Eng.*, 28(7):654–670, 2002.

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

[87] 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.

[88] John P. Klein and Melvin L. Moeschberger. *Survival Analysis: Techniques for Censored and Truncated Data.* 2013.

[89] David Kleinbaum. *Survival Analysis a Self Learning Text.* Springer, second edition, 2005.

[90] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Int'l Symp. Static Analysis*, pages 40–56, July 2001.

[91] Raghavan Komondoor and Susan Horwitz. Eliminating duplication in source code via procedure extraction. Technical report 1461, UW-Madison Dept. of Computer Sciences, December 2002.

[92] K. Kontogiannis, R. De Mori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *J. Automated Software Engineering*, 3(1/2):79–108, June 1996.

[93] Rainer Koschke. *Software Evolution*, chapter Identifying and Removing Software Clones, pages 15–36. Springer, 2008.

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

[95] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *Working Conf. Reverse Engineering (WCRE)*, pages 253–262, 2006.

[96] C.J. Krebs. *Ecology: The experimental analysis of distribution and abundance*. Harper and Row, 1972.

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

[98] F. Lanubile and T. Mallardo. Finding function clones in web applications. In *European Conf. Software Maintenance and Reengineering (CSMR)*, pages 379–386, 2003.

[99] Jeff Leek. How i decide when to trust an R package. `http://simplystatistics.org/?p=4409`, November 2015.

[100] Meir Manny Lehman. Programs, life cycles, and laws of software evolution. *Proc. IEEE*, 68(9):1060–1076, September 1980.

[101] Meir Manny Lehman, Dewayne E. Perry, and Juan Fernandez Ramil. Implications of evolution metrics on software maintenance. 1998.

[102] J. Li and M. D. Ernst. CBCD: Cloned buggy code detector. In *Int'l Conf. Software Engineering (ICSE)*, pages 310–320, 2012.

[103] E. Linstead and P. Baldi. Mining the coherence of GNOME bug reports with statistical topic models. In *Int'l Conf. Mining Software Repositories (MSR)*, pages 99–102, 2009.

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

[105] Mircea Lungu. *Reverse Engineering Software Ecosystems*. PhD thesis, University of Lugano, 2009.

[106] 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.

[107] Fabio Mancinelli, Jaap Boender, Roberto Di Cosmo, Jerome Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. Managing the complexity of large free and open source package-based software distributions. In *Int'l Conf. Automated Software Engineering (ASE)*, pages 199–208, 2006.

[108] Konstantinos Manikas. Revisiting software ecosystems research: A longitudinal literature study. *Journal of Systems and Software*, 117:84–103, 2016.

[109] Konstantinos Manikas and Klaus Marius Hansen. Software ecosystems: A systematic literature review. *J. Systems and Software*, 2012.

[110] N. Mantel. Evaluation of survival data and two new rank order statistics arising in its consideration. *Cancer Chemother Rep.*, 50(3):163–170, 1966.

[111] Juan Martinez-Romo, Gregorio Robles, Jesús M. González-Barahona, and Miguel Ortuño-Perez. Using social network analysis techniques to study collaboration between a FLOSS community and a company. In *Open-Source Software*, volume 275, pages 171–186. Springer, 2008.

[112] 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)*, 2013.

[113] 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.

[114] Tom Mens. Anonymized e-mail interviews with R package maintainers active on CRAN and GitHub. Technical report, University of Mons, 2016.

[115] Tom Mens, Maëlick Claes, Philippe Grosjean, and Alexander Serebrenik. Studying evolving software ecosystems based on ecological models. In Tom Mens, Alexander Serebrenik, and Anthony Cleve, editors, *Evolving Software Systems*, pages 297–326. Springer, 2014.

[116] Tom Mens and Mathieu Goeminne. Analysing the evolution of social aspects of open source software ecosystems. In *Int'l Workshop on Software Ecosystems*, pages 1–14. CEUR Workshop Proccedings, June 2011.

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

[118] Stanley Milgram. The small world problem. *Psychology Today*, 67(1):61–67, 1967.

[119] Audris Mockus, R.T. Fielding, and J.D. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Trans. Software Engineering and Methodology*, 11(3):309–346, 2002.

[120] 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.

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

[122] Kumiyo Nakakoji, Yasuhiro Yamamoto, Yoshiyuki Nishinaka, Kouichi Kishida, and Yunwen Ye. Evolution patterns of open-source software systems and communities. In *Int'l Workshop on Principles of Software Evolution*, pages 76–85, New York, NY, USA, 2002. ACM.

[123] S. Neu, M. Lanza, L. Hattori, and M. D'Ambros. Telling stories about GNOME with Complicity. In *Working Conf. Software Visualisation (VISSOFT)*, pages 1–8. IEEE, 2011.

[124] Raymond Nguyen and Richard C. Holt. Life and death of software packages: an evolutionary study of debian. In *Center for Advanced Studies on Collaborative Research (CASCON)*, pages 192–204, 2012.

[125] Lucas Nussbaum and Stefano Zacchiroli. The ultimate Debian database: Consolidating bazaar metadata for quality assurance and data mining. In *Working Conf. Mining Software Repositories (MSR)*, pages 52–61, 2010.

[126] Jeroen Ooms. Possible directions for improving dependency versioning in R. *R Journal*, 5(1):197–206, June 2013.

[127] OSI. Open source definition. 2007.

[128] Javier Perez, Romuald Deshayes, Mathieu Goeminne, and Tom Mens. SECONDA: Software ecosystem analysis dashboard. In Tom Mens, Anthony Cleve, and Rudolf Ferenc, editors, *European Conf. Software Maintenance and Reengineering (CSMR)*, pages 527–530, 2012.

[129] Richard Peto and Julian Peto. Asymptotically efficient rank invariant test procedures. *Journal of the Royal Statistical Society*, 135(2):185–207, 1972.

[130] Wouter Poncin, Alexander Serebrenik, and Mark G. J. van den Brand. Process mining software repositories. In *European Conf. Software Maintenance and Reengineering (CSMR)*, pages 5–14, 2011.

[131] The Debian Project. Debian policy manual, section 5. `https://www.debian.org/doc/debian-policy/ch-controlfields.html`, March 2016. retrieved March 2016.

[132] The Debian Project. Debian policy manual, section 7. `https://www.debian.org/doc/debian-policy/ch-relationships.html`, March 2016. retrieved March 2016.

[133] R devel mailing list. R package dependency issues when namespace is not attached. `http://r.789695.n4.nabble.com/R-tt4629828.html`, 2015.

[134] Eric S. Raymond. *The Cathedral and the Bazaar*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 1999.

[135] 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.

[136] G. Robles, J. J. Amor, J. M. Gonzalez-Barahona, and I. Herraiz. Evolution and growth in large libre software projects. In *Int'l Workshop on Principles of Software Evolution*, pages 165 – 174, Lisbon, September 2005. IEEE Computer Society.

[137] Gregorio Robles, Jesus M. Gonzalez-Barahona, and Israel Herraiz. Evolution of the core team of developers in libre software projects. In *Int'l Conf. Mining Software Repositories (MSR)*, pages 167–170. IEEE Computer Society, 2009.

[138] C. K Roy, J. R Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[139] 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.

[140] Ioannis Samoladas, Lefteris Angelis, and Ioannis Stamelos. Survival analysis on the duration of open source projects. *Information & Software Technology*, 52(9):902–922, 2010.

[141] StackOverflow. Writing robust R code: namespaces, masking and using the '::' operator. `http://stackoverflow.com/questions/10947159/writing-robust-r-code-namespaces-masking-and-using-the-operator`, 2015.

[142] Richard Stallman. Why free software is better than open source, 1998.

[143] Richard M. Stallman. The GNU Manifesto. In Joshua Gay, editor, *Free Software, Free Society: Selected Essays of Richard M. Stallman*, pages 33–41. GNU Press, Boston, 2002.

[144] 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.

[145] M. M. Mahbubul Syeed, Klaus Marius Hansen, Imed Hammouda, and Konstantinos Manikas. Socio-technical congruence in the ruby ecosystem. In *Proceedings of The International Symposium on Open Collaboration*, OpenSym '14, pages 2:1–2:9, New York, NY, USA, 2014. ACM.

[146] A. Terceiro, L.R. Rios, and C. Chavez. An empirical study on the structural complexity introduced by core and peripheral developers in free software projects. In *Brazilian Symp. Software Engineering (SBES)*, pages 21 –29, 27 2010-oct. 1 2010.

[147] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.2*, 2008.

[148] Ferdian Thung, Tegawendé F. Bissyandé, David Lo, and Lingxiao Jiang. Network structure of social coding in GitHub. In *CSMR*, pages 323–326, 2013.

[149] Ralf Treinen and Stefano Zacchiroli. Common upgradeability description format (cudf) 2.0. *The Mancoosi project (FP7)*, 3, 2009.

[150] Matthew Van Antwerp and Gregory R. Madey. The importance of social network structure in the open source software developer community. In *Hawaii Int'l Conf. System Sciences (HICSS)*, pages 1–10. IEEE Computer Society, 2010.

[151] Bogdan Vasilescu, Vladimir Filkov, and Alexander Serebrenik. StackOverflow and GitHub: Associations between software development and crowdsourced knowledge. In *SocialCom/PASSAT*, page to appear. IEEE, 2013.

[152] Bogdan Vasilescu, Alexander Serebrenik, Mathieu Goeminne, and Tom Mens. On the variation and specialisation of workload - A case study of the gnome ecosystem community. *Empirical Software Engineering*, 19(4):955–1008, 2014.

[153] 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.

[154] Martti Viljainen and Marjo Kauppinen. Framing management practices for keystones in platform ecosystems. In Slinger Jansen, Michael Cusumano, and Sjaak Brinkkemper, editors, *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry*. Edward Elgar, 2013.

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

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

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

[158] Michael Weiss, Gabriella Moroiu, and Ping Zhao. Evolution of open source communities. In Ernesto Damiani, Brian Fitzgerald, Walt Scacchi, Marco Scotto, and Giancarlo Succi, editors, *OSS*, volume 203 of *IFIP*, pages 21–32. Springer, 2006.

[159] 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.

[160] Michel Wermelinger, Yijun Yu, and Angela Lozano. Design principles in architectural evolution: a case study. In *Int'l Conf. Software Maintenance (ICSM)*, 2008.

[161] A. J. Willis. The ecosystem: an evolving concept viewed historically. *Functional Ecology*, 11:268–271, 1997.

[162] Liguo Yu and Srini Ramaswamy. Mining CVS repositories to understand open-source project developer roles. In *Int'l Conf. Mining Software Repositories (MSR)*. IEEE Computer Society, 2007.

[163] Yue Yu, Gang Yin, Huaimin Wang, and Tao Wang. Exploring the patterns of social behavior in GitHub. In *Int'l Workshop on Crowd-based Software Development Methods and Technologies*, pages 31–36, 2014.

[164] Alexey Zagalsky, Carlos Gómez Teshima, Daniel M German, Margaret-Anne Storey, and Germán Poo-Caamaño. How the r community creates and curates knowledge: A comparative study of stack overflow and mailing lists.