An Empirical Comparison of Dependency Issues in OSS Packaging Ecosystems

Alexandre Decan, Tom Mens and Maëlick Claes
COMPLEXYS Research Institute
University of Mons, Belgium
Email: { first . last } @ umons.ac.be

***Abstract*—Nearly every popular programming language comes with one or more open source software packaging ecosystem(s), containing a large collection of interdependent software packages developed in that programming language. Such packaging ecosystems are extremely useful for their respective software development community. We present an empirical analysis of how the dependency graphs of three large packaging ecosystems (npm, CRAN and RubyGems) evolve over time. We study how the existing package dependencies impact the resilience of the three ecosystems over time and to which extent these ecosystems suffer from issues related to package dependency updates. We analyse specific solutions that each ecosystem has put into place and argue that none of these solutions is perfect, motivating the need for better tools to deal with package dependency update problems.**

***Index Terms*—software repository mining; software ecosystem; package dependency management; software evolution; software distribution**

I. Introduction

Traditionally, software engineering research has focused on understanding and improving the development and evolution of individual software projects. The widespread use of collaborative open source software development tools (such as Git and GitHub) has lead to an increased popularity of so-called *software ecosystems*, and research focus is shifting to understanding and improving their dynamics.

Software ecosystems are large collections of interdependent software components that are maintained by large and geographically distributed communities of collaborating contributors. Typical examples of open source software ecosystems are distributions for Linux operating systems and *packaging ecosystems* for specific programming languages. Such packaging ecosystems tend to be very large, containing from tens to hundreds of thousands of packages, with even an order of magnitude more dependencies between them.

Very few empirical studies have quantitatively analysed the similarities and differences between packaging ecosystems. Such comparative studies are urgently needed, in order to understand how techniques and tools that are specific to an ecosystem affect the evolution of its package dependency structure. Empirical results observed for one ecosystem may not necessarily generalise to another one, implying that ecosystem-specific solutions may be needed, for example to improve how package dependencies are managed in presence of problematic package updates that may affect (transitively) dependent packages.

This article studies issues related to the evolution of package dependencies in three seemingly similar packaging ecosystems, namely the npm, CRAN and RubyGems package distributions for the JavaScript, R and Ruby programming language, respectively. We observe that their dependency structure is different, and evolves differently over time. Despite these differences, the dependency graphs also reveal some interesting commonalities. We also study to which extent the versioning mechanism and the recommended use of version constraints affects the evolution of each packaging ecosystem. Our empirical findings are accompanied by quotes from different ecosystem contributors that highlight the reasons for, and potential impact of, the observed differences.

The remainder of this article is structured as follows. Section II discusses related work. Section III motivates the selected packaging ecosystems, presents the data extraction process, and introduces the empirical research questions. Sections IV to VIII each target a specific research question. Section IX presents the threats to validity of our study. Section X concludes by summarising the main research findings and outlining future work.

II. Related work

Bogart et al. [1] carried out a qualitative comparison of three different ecosystems (npm, CRAN, Eclipse) in order to understand the impact of community values, tools and policies on breaking changes. Their analysis relied on interviews with developers, and provided useful insights. It is complementary to the quantitative analysis presented here, which is based on an empirical historical comparison of the package dependency structure of the packaging ecosystems npm, CRAN and RubyGems.

Our current paper extends the results of [2] in which we compared the dependency graphs of three programming language packaging ecosystems without taking the temporal dimension into account. In the current paper, we study and compare structural issues related to transitive package dependencies and how these issues evolve over time.

Many researchers have studied package dependencies issues in different programming language packaging ecosystems. Wittern et al. [3] studied the evolution of JavaScript packages in npm. The CRAN packaging ecosystem has been studied previously [4], [5], and dependencies have been shown as an important cause of errors in R packages both on CRAN and

GitHub [6]. Blincoe et al. [7] looked at Ruby as part of a larger GitHub study on the emergence of software ecosystems. Bavota et al. [8] studied the evolution of dependencies in the Apache ecosystem and highlighted that dependencies have an exponential growth and must be taken care of by developers. When a change occurs in a package, this change may break its dependent packages, the dependent packages of their dependent packages, and so on. They found that developers were reluctant to upgrade the version of the software they depend upon. Robbes et al. [9] studied the ripple effect of API method deprecation in the Smalltalk ecosystem and revealed that API changes can have a large impact on the system and remain undetected for a long time after the initial change.

The problems of co-installability in package-based distributions such as Debian [10] has been extensively studied, and efficient tools have been proposed to identify and solve these problems [11]. Historical information allows to make these tools even more precise [12]. Beyond co-installability issues, Di Cosmo et al. [13] claims that problems related to package upgrades are equally important, and that more automated solutions to address these problems are required. This paper empirically validates these claims, by studying problems related to package updates in presence of dependent packages and by analysing how large popular packaging ecosystems currently (fail to) cope with these problems.

III. Methodology

A. Selected Packaging Ecosystems

In order to empirically study how the dependency graph of a packaging ecosystem evolves over time, it is important to choose packaging ecosystems that have a sufficiently long lifetime. We imposed a minimal lifetime of five years for candidate packaging ecosystems.

We decided to focus on programming language packaging ecosystems, i.e., ecosystems revolving around package distributions for specific programming languages. The reason for doing so is that such packaging ecosystems tend to have a very active community of contributors, making the ecosystems very large, and causing difficulties in the evolution of these packaging ecosystems.

TABLE I
Characteristics of npm, CRAN and RubyGems.

Characteristic	npm	CRAN	RubyGems
URL	npmjs.com	cran.r-project.org	rubygems.org
Language	JavaScript	R	Ruby
snapshot date	2016-06-28	2016-04-26	2016-09-07
packages	317,159	9,568	122,791
package releases	1,927,750	57,530	685,591
dependencies	7,644,295	128,113	1,674,823
oldest package	2010-11-09	1997-10-08	2009-07-25
new pkgs. in 2015	113,613	1,660	18,639
updates in 2015	711,317	8,140	121,394

For the current study, we selected three popular programming languages, namely JavaScript, R and Ruby. These three dynamic programming languages belong to the top 20 of most popular programming languages (according to the PYPL[1] and Tiobe[2] programming language popularity indexes). For each of these languages, we decided to study their official software package distribution as packaging ecosystem. For JavaScript, this is the npm package manager. For R, CRAN is the official distribution of R packages. For Ruby, RubyGems is the most used package provider. Each of them is very big, as summarised in Table I.

B. Data Extraction

We used different approaches for extracting the historical package metadata for each packaging ecosystem, due to the different tools used by these ecosystems. All the data and scripts are available through a replication package at https://github.com/ecos-umons/saner2017-ecos.

The npm package manager resolves packages by name and version through a registry website implementing the *CommonJS Package Registry* specification. Using this official public registry we retrieved the list of all packages available in June 2016. For each of these 317,159 packages, we collected all the available metadata of all associated releases, corresponding to nearly 2 million releases.

We extracted all the metadata of CRAN packages using extractoR, a publicly available[3] R package that we developed specifically for this purpose. It downloads the CRAN package sources, extracts their contents and stores the file metadata. We collected the metadata of 9,568 packages with their associated versions and dependencies, accounting for 57,530 releases.

For RubyGems, we directly parsed the list of 122,791 available packages in September 2016 from the RubyGems website. For each of these packages, we queried the JSON API of RubyGems to list all available releases and their associated metadata. This allowed us to obtain 719,853 package releases. For 7,256 packages we observed that many releases had a release date set to a single same day (either 25 July 2009 or 10 August 2011). As their chronological order did not correspond to the order induced by their versioning scheme, we assume that this phenomenon resulted from an automatic migration from other packaging ecosystems. We therefore ignored, for all concerned packages, all package releases corresponding to these two days, except for the latest available package release based on the versioning scheme. This represented 34,262 dropped package releases, leaving us 685,591 package releases available for our analysis.

For each package release, we extracted from its metadata the release date, the release name (i.e., version number) and the list of packages on which it depends (from field `dependencies` for npm, `Depends` and `Imports` for CRAN and `runtime` for RubyGems). Based on all the data, we computed the monthly package dependency graph of each ecosystem from 2007 to 2016.

Figure 1 shows the evolution of the number of available packages and, in dashed lines, the evolution of the number

[1] http://pypl.github.io/PYPL.html (December 2016)
[2] http://www.tiobe.com/tiobe-index/ (December 2016)
[3] https://github.com/ecos-umons/extractoR

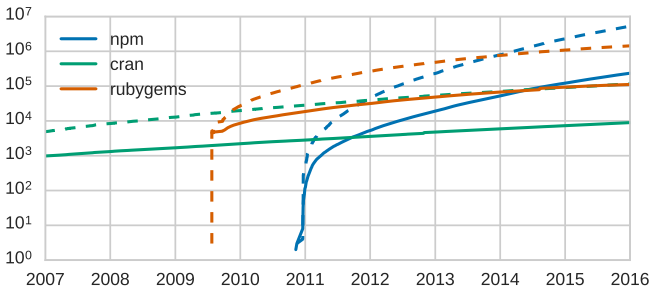

Fig. 1. Evolution of the number of packages (straight lines) and package dependencies (dashed lines) for each ecosystem.

of dependencies. It reveals that more and more packages and dependencies are added to each ecosystem, but with a different rate of growth. To verify this, we performed a linear regression using an ordinary least square method, and a linear regression on the logarithmic values (i.e., an exponential regression) on the number of packages and on the number of dependencies in each ecosystem. The obtained R^2-values, shown in Table II, suggest an *exponential increase* in the number of packages and in the number of dependencies for npm and CRAN, and a *linear increase* in both the number of packages and the number of dependencies for RubyGems.

TABLE II
R^2-VALUES FOR REGRESSIONS ON THE NUMBER OF PACKAGES AND ON THE NUMBER OF DEPENDENCIES.

# packages	npm	CRAN	RubyGems
linear	0.84	0.87	**0.99**
exponential	**0.89**	**0.97**	0.82

# dependencies	npm	CRAN	RubyGems
linear	0.77	0.79	**0.96**
exponential	**0.89**	**0.97**	0.83

C. Research Questions

Given the package growth, compounded by an even larger number of dependency relationships between these packages, we were curious about the impact of package updates and package removals on the other packages and on the ecosystem. To explore this impact, we will study the following research questions in the remainder of this paper:

Section IV: *To which extent do packages depend on other packages?* With this question we aim to study if the proportion of packages with dependencies increases over time, to which extent there are direct and transitive dependencies, and how the dependency structure differs across the three ecosystems.

Section V: *How resilient is an ecosystem to failures in dependent packages?* With this question we explore how the presence of transitive dependencies may lead to the propagation of package failures throughout the ecosystem.

Section VI: *How frequent are package updates?* Given that changes in packages may be problematic for dependent packages, we study how frequently package updates occur.

Section VII: *To which extent do package maintainers make use of dependency constraints?* To reduce the risk of the package dependency update problems, some package maintainers rely on specifying minimal and/or maximal constraints on the allowed versions of a dependency. We explore the extent and effect of this practice, especially in combination with a semantic versioning policy that is recommended by some ecosystems.

Section VIII: *Why should package maintainers be careful with dependency constraints?* In this question we argue that relying on dependency constraints may lead to other problems, like an increased risk of package co-installability problems, as well as an important delay before packages benefit from important updates in packages they depend upon.

IV. To which extent do packages depend on other packages?

Our first research question pertains to the omnipresence of package dependencies in the considered ecosystems. One of the main reasons why dependencies between packages emerge in an ecosystem is because of software reuse, a basic principle of software engineering [14]. Software packages often rely on (i.e., reuse) the functionality offered by other packages (e.g., libraries), rather than reimplementing the same functionality. Packaging ecosystems make it easier for developers to rely on functionality provided by other packages, by offering automated tools to install and manage multiple packages.

While dependencies tend to reduce development effort for individual packages, they increase the complexity of the packaging ecosystem as a whole through the need to manage these dependencies in presence of multiple package releases. This complexity can be the cause of many maintainability issues and failures [1], [8], and understanding this complexity is one of the goals of this paper.

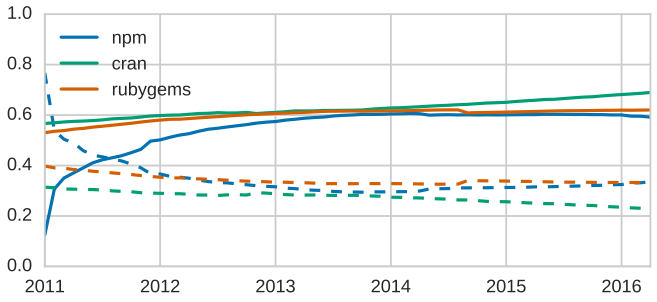

Fig. 2. Evolution of the proportion of packages with dependencies (straight lines) and of isolated packages (dashed lines).

Figure 2 shows the evolution over time of the proportion of packages that have at least one dependency, as well as of the proportion of isolated packages, i.e., packages that have no dependency nor reverse dependency. We observe that a majority of packages declare dependencies, and that the proportion of such packages increases over time. We also observe that since around 2013 the proportion of isolated packages remains stable for npm and RubyGems, but continues to decrease (slowly)

over time for CRAN. In April 2016, 60% of all packages on npm and RubyGems have dependencies, while this is around 70% for CRAN packages. On the other hand, around 33% of all npm and RubyGems packages are isolated, while this is only 22% for CRAN.

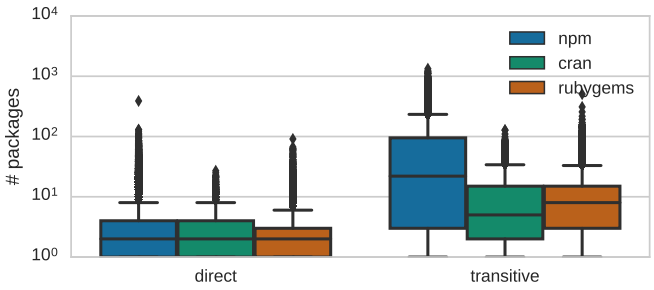

Fig. 3. Distribution of the number of direct dependencies and transitive dependencies by package, in April 2016.

When considering only those packages with at least one dependency, the boxplots in Figure 3 show the distribution of the number of direct and transitive dependencies by package in the three ecosystems, in April 2016. We observe that, while a majority of the packages with dependencies have very few *direct* dependencies, they have a much higher number of *transitive* dependencies. For CRAN and RubyGems, half of the packages with dependencies have respectively at most 5 and 8 transitive dependencies. For npm, this value is much higher, with half of the packages with dependencies having at least 22 transitive dependencies, and a quarter having at least 95 transitive dependencies.

Among all packages having at least one dependency, we statistically compared for each ecosystem the number of direct and transitive dependencies using a one-sided non-parametric Mann-Whitney's U test. The null hypothesis states that the distribution of both populations is equal. The alternative hypothesis states that the distribution of one population is greater than the distribution of the other population. The null hypothesis was significantly rejected, with p-values reported in Table III. For the distribution of the populations of *direct* dependencies, we found that RubyGems $<$ CRAN $<$ npm. For the distribution of *transitive* dependencies, we found that CRAN $<$ RubyGems $<$ npm.

Because of the huge difference in the number of packages in the considered ecosystems (163,658 in npm versus 6,500 in CRAN and 72,094 in RubyGems), we report in Table III the effect size of Mann-Whitney's U tests using Cliff's δ [15]. We used the abbreviations N, S, and M to stand for, respectively, a *negligible*, *small* and *medium* effect size, as usually interpreted. The reported effect sizes coincide with the differences we visually observed in Figure 3. Those differences do not only concern the absolute number of direct or transitive dependencies, but also the relative number of transitive dependencies relatively to the number of direct ones. We computed that, on average, a package has 22.1 times more transitive dependencies than direct ones in npm, which is very high, and also much higher than in CRAN or RubyGems (respectively 3.64 and 6.35).

TABLE III
STATISTICAL COMPARISON OF THE DISTRIBUTIONS OF THE NUMBER (≥ 1) OF (A) DIRECT AND (B) TRANSITIVE DEPENDENCIES.

alt. hypothesis	p-value	Cliff's δ
(a) CRAN $<$ npm	$< 10^{-5}$	0.03 (N)
(a) RubyGems $<$ npm	$< 10^{-6}$	0.23 (S)
(a) RubyGems $<$ CRAN	$< 10^{-6}$	0.21 (S)
(b) CRAN $<$ npm	$< 10^{-6}$	0.36 (M)
(b) RubyGems $<$ npm	$< 10^{-6}$	0.31 (S)
(b) CRAN $<$ RubyGems	$< 10^{-6}$	0.09 (N)

So why do npm packages tend to rely more on other packages than in CRAN and RubyGems? The completeness of the standard library of Ruby and R may explain this phenomenon. Ruby's standard library typically contains a wide variety of packages that may be used to address many common tasks and problems. Similarly, while R might lack more generic programming tools that are provided by packages such as XML, jsonlite, stringr, Rcpp or RCurl, it still contains a quite extensive library to manipulate data and conduct statistical analyses. In contrast, the standard library for JavaScript is kept intentionally small according to JavaScript creator Brendan Eich [16]: "*The real standard library people want is more like what you find in Python or Ruby, and it's more batteries included, feature complete, and that is not in JavaScript. That's in the npm world or the larger world.*" This may explain why JavaScript developers tend to rely more often on third-party packages.

Summary:

- A majority of packages in npm, CRAN and RubyGems rely on other packages.
- There are proportionally more packages with dependencies in CRAN than in npm or RubyGems, and this proportion increases over time.
- While packages tend to have few direct dependencies, they have a much higher number of transitive ones.
- npm and CRAN packages have more direct dependencies than RubyGems packages.
- npm packages have many more transitive dependencies than CRAN or RubyGems packages.

V. How resilient is an ecosystem to failures in dependent packages?

Because the three ecosystems heavily use dependencies, they face the risk of having important points of failure. These failures can be caused by different reasons: a package may get removed entirely from the ecosystem, a package may become archived because it no longer passes the quality checks or because its developer is no longer available, a package may be updated in backward incompatible ways, and so on.

Because of the presence of many transitive dependencies, a package failure may potentially affect many other packages. An example of this was experienced in npm in March 2016.

The sudden and unexpected removal of a package called left-pad had a large impact on the ecosystem, breaking thousands of dependent packages, including those that were not even aware they were (indirectly) using it: "*This impacted many thousands of projects. [...] We began observing hundreds of failures per minute, as dependent projects – and their dependents, and their dependents... – all failed when requesting the now-unpublished package.*" [17]

Maintainers from CRAN also share this concern: "*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 [...]*" [18]

In order to assess the resilience of a packaging ecosystem E to the failure of a package p, we define the following package impact metrics for a given ecosystem snapshot:

impact $\mathbf{I}(p, E)$ = the *total number* of packages in E that depend directly or transitively on p, i.e., all packages that could potentially be impacted by a failure of p.

relative impact $\mathbf{RI}(p, E) = \frac{\mathbf{I}(p,E)}{|E|}$, i.e., the *proportion* of all packages in E that depend directly or transitively on p.

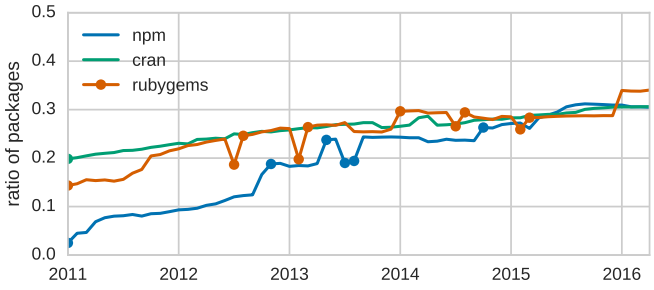

Fig. 4. Evolution of the relative impact of each ecosystem's most required package.

Figure 4 shows the evolution over time of $\max_{p \in E} \mathbf{RI}(p, E)$ for $E \in \{\text{npm}, \text{CRAN}, \text{RubyGems}\}$. It represents the relative impact of each ecosystem's most required package. Notice that the most required package of an ecosystem may change over time. Such changes are illustrated by the markers on the figure, and involve underscore, mime, async and inherits for npm, lattice for CRAN, and multijson, activesupport and json for RubyGems. Such packages that are (transitively) required by many other packages can have a huge impact on the ecosystem, as a failure in those packages can affect an important number of other packages. We observe that this proportion tends to increase over time, and exceeds 30% of all packages for each ecosystem in 2016. For instance, on April 2016, inherits was npm's most required package, with 84,324 dependent packages. CRAN's most required package lattice had 2,886 dependent packages, and RubyGems's most required package json had 39,570 dependent packages.

Figure 5 shows the evolution of the number of packages having a relative impact greater than or equal to 2%, i.e., the evolution of the number of packages p in ecosystem $E \in \{\text{npm}, \text{CRAN}, \text{RubyGems}\}$ for which $\mathbf{RI}(p, E) \geq 0.02$. The choice of 2% seems to be a reasonable value because $\mathbf{RI}(\text{npm}, \text{left-pad}) = 2.05\%$ before its unexpected removal from the npm ecosystem, causing major problems. We also computed the results for higher relative impacts (5% and 10%) and the observed differences between the three ecosystems were similar to the ones obtained for the 2% threshold.

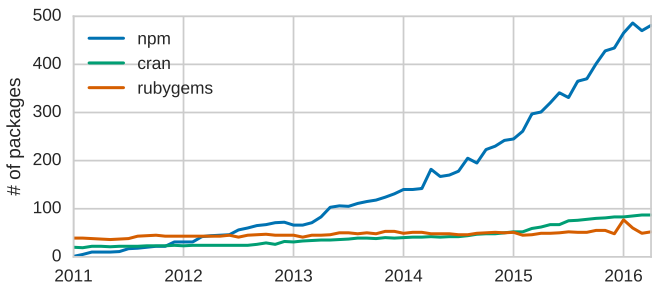

Fig. 5. Evolution of the number of packages having a relative impact greater than 2%.

For CRAN and RubyGems, we observe a slight increase of the number of packages having a relative impact greater than or equal to 2% over time. For npm, the increase is considerably more important. While in 2012, it accounted for 44 packages, there were 481 such packages in April 2016, corresponding to one package out of 600 that has a high failure impact. While we did expect npm to have more packages with a high failure impact, we were surprised to see such a big difference with CRAN and RubyGems.

Such packages represent a potential Achilles' heel for the ecosystem: removing only one of them can impact a large proportion of the other packages in the ecosystem, as was the case with left-pad in npm. Although this incident lead npm managers to prevent authors from removing their packages from the ecosystem, in October 2016 RubyGems still allowed authors to easily remove their packages, without any consideration on the dependent packages. CRAN does not allow to remove packages, but they may become archived, implying that they cannot be automatically installed, and thus preventing the installation of dependent packages as well.

To prevent or reduce the risk of such cascading failures, it is important for package maintainers to be aware of the packages they depend upon. While maintainers are usually aware of the direct dependencies of their packages because they explicitly declare them, we believe that few maintainers have a clear idea on which packages they depend indirectly, especially since, as seen in Figure 3, these may be very numerous. For example, a package such as the popular react in npm has only 2 direct dependencies, but transitively depends on 44 additional packages. As a consequence, each of the 3,121 packages that directly depends on this popular package implicitly requires at least 46 additional packages.

Summary:

- Packages with many transitive dependencies have a negative impact on the ecosystem's resilience.
- Some packages potentially impact $> 30\%$ of all packages in each ecosystem, and this proportion is increasing over time.
- Each ecosystem has an increasing number of packages whose failure can impact an important number of (transitively) dependent packages. Such packages are particularly abundant in npm.

VI. How frequent are package updates?

Package dependencies not only increase the overall complexity of an ecosystem, but also may be problematic in case of package updates, since new package releases (i.e., package updates) may have undesired consequences on dependent packages. For CRAN in particular, we observed that 41% of the package errors were caused by backward incompatible changes [6]. We showed that, on average, developers introduce one backward incompatible change every 20 releases.

While the policy of some packaging ecosystems may require a package to be in a stable state at the moment of its introduction, this does not mean that the package will never get updated. Publishing a new release of a package, regardless of whether it contains new features, bug fixes or API changes, is a common and natural process for a maintainer: "*Change in an API is inevitable as your knowledge and experience of a system improves. Managing the impact of this change can be quite a challenge when it threatens to break existing client integrations.*" [19]

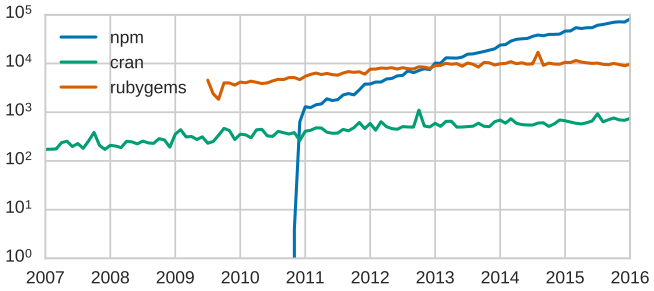

Fig. 6. Number of new package releases by month.

Figure 6 shows the monthly number of new releases for the three packaging ecosystems. As expected, all three are very active in terms of package updates. We computed that 60% of npm packages, 48% of CRAN packages and 27% of RubyGems packages were at least updated once in 2015.

To understand the effect of the presence of package dependencies on the frequency of package updates, we used the statistical technique of *survival analysis* [20]. Survival analysis models [21] estimate the survival rate of a population over time, considering the fact that some elements of the population may leave the study, and that for some other elements the event of interest does not occur during the observation period. In our case, the observed event of interest is the delay between two consecutive package updates.

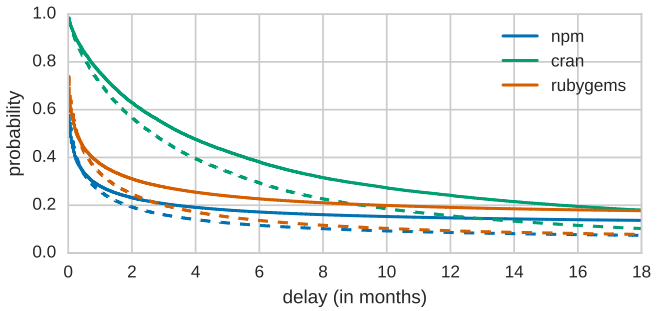

Fig. 7. Survival curves estimating the probability of **not** updating a package. Dotted lines show the probability for packages with reverse dependencies.

The survival curve in Figure 7 uses a Kaplan-Meier estimator to estimate the probability of not updating a package for a certain amount of time. We observe that, regardless of the ecosystem, a majority of the updates occur in a very short delay since the previous update. In addition to this, the probability for a package to be updated within 18 months after the last release exceeds 80%.[4] If we restrict the analysis to packages with at least one reverse dependency, i.e., packages required by at least one other package, we obtain the dotted survival curves in Figure 7. As they seem quite close to the unrestricted survival curves, we performed a log-rank test to compare the survival curves. It turns out that packages with reverse dependencies are updated significantly more often than packages without ($\alpha = 0.99$).

While these results are valid regardless of the considered packaging ecosystem, we do observe an important difference between npm and RubyGems on one side and CRAN on the other side: new package releases are much more spread over time on CRAN than on npm or RubyGems. This difference can probably be explained by CRAN's policy, which is more demanding with respect to package update, package quality and package obsolescence [6]. In particular, maintainers are asked to somewhat limit frequent package updates: "*Submitting updates should be done responsibly and with respect for the volunteers' time. Once a package is established (which may take several rounds), 'no more than every 1–2 months' seems appropriate.*" [22]

Maintainers can be helped in the management of dependency updates by tools that monitor dependencies and notify the maintainers when a new release of a package dependency is available, or when an important update needs to be deployed. For instance, web-based dashboards like gemnasium.com, requires.io or dependencyci.com provide these features as a continuous integration process, and are even free for open source projects. However, at the time of writing this paper, these tools monitored direct dependencies only and, therefore, did not detect update problems beyond the first level of the

[4]Since the probability of **not** updating a package after 18 months is less then 0.2 in Figure 7.

dependency hierarchy. This is quite surprising given how relatively easy it should be to extend these tools to monitor transitive dependencies.

Summary:

- Most packages tend to be updated shortly after a previous update.
- Within 18 months, over 80% of all packages have received an update.
- Packages required by other packages are updated more frequently than packages that are not.
- CRAN packages are updated less often than npm and RubyGems packages.

VII. To which extent do package maintainers make use of dependency constraints?

In order to avoid packages becoming broken due to a dependency update, most ecosystems allow package maintainers to specify dependency constraints on the versions of the packages they depend upon. Such constraints typically allow maintainers to explicitly select the desirable or allowed releases of a dependency, and to explicitly exclude the undesirable ones, e.g., those that can contain backward incompatible changes.

Dependency constraints are typically used to specify a *minimal* version (e.g., $>=$ 1.2.3) or a *maximal* version (e.g., $<$ 1.3.0) of a dependency. Often, combination of such constraints can be expressed using a specific notation (e.g., $\sim$ 1.2.3 on npm is equivalent to $>=$ 1.2.3 and $<$ 1.3.0). It is also possible to impose *strict* dependency constraints, which specify that the dependent package should match exactly one version. Such *strict* constraints should be used with caution as they can be a source of problems. For instance, shortly after the removal of left-pad from npm, a functionally identical version of left-pad was published as version 1.0.0, but "*we continued to observe many errors. This happened because a number of dependency chains [...] explicitly requested 0.0.3.*" [17]

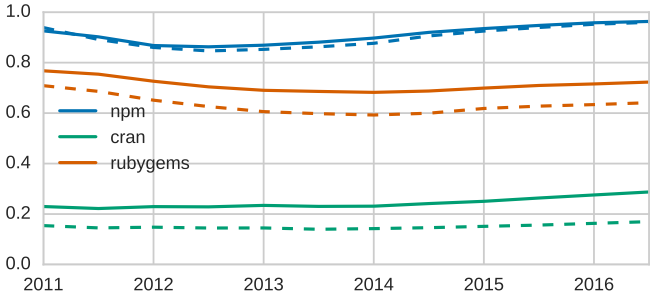

Fig. 8. Proportion of packages (straight lines) and proportion of dependencies (dotted lines) that make use of a dependency constraint.

Figure 8 shows the number of packages and dependencies that make use of a dependency constraint, proportionnaly to the number of packages with dependencies, and to the total number of dependencies. We observe that dependency constraints are very common for npm and RubyGems packages. In early 2016, more than 95% (resp. 63%) of all dependencies in npm (resp. RubyGems) imposed a dependency constraint. This represents more than 95% (resp. 71%) of all packages that have a dependency in npm (resp. RubyGems).

In contrast, less than 30% of the packages with dependencies in CRAN impose a constraint on a dependency. The case of CRAN is quite specific: by default, only the latest available release of a package can be automatically installed from CRAN. This "rolling release" policy implies that a package must always be up-to-date with its dependencies. Any dependency constraint that is not satisfied by the latest available release of a dependency makes the package uninstallable. A maintainer is thus forced to systematically adapt its package whenever a backward incompatible update occurs in one of its dependencies. It follows that R maintainers rarely specify constraints on the required release of dependent packages, except to specify a minimal required version. The solution imposed by CRAN to deal with package dependency updates is a continuous integration process based on the R CMD check tool that is run on a daily basis. If a package's tests fail because of a dependency, its maintainer is asked to resolve the problem before the next major R release.

CRAN's policy tries to encourage package maintainers to limit the impact of an update [22]: "*Changes to CRAN packages causing significant disruption to other packages must be agreed with the CRAN maintainers well in advance of any publicity. [...] If an update will change the package's API and hence affect packages depending on it, it is expected that you will contact the maintainers of affected packages and suggest changes, and give them time to prepare updates before submitting your updated package.*" Unfortunately, this policy is not always sufficient [23]: "*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.*"

Minimal and *maximal* dependency constraints are especially useful in combination with the use of *semantic versioning*[5]. Semantic versioning proposes a simple set of rules and requirements that dictate how version numbers are assigned and incremented based on the three-number version format Major.Minor.Patch. Package updates that correspond to bug fixes that do not affect the API should only increment the Patch version number, backward compatible updates should increment the Minor version number, and backward incompatible updates have to increment the Major version number.

Ideally, the combination of dependency constraints with semantic versioning should make it easier for package maintainers to manage dependency updates. Unfortunately, although semantic versioning is strongly recommended for npm or RubyGems, it cannot be enforced. Even if maintainers are required to adopt the Major.Minor.Patch notation, they can always decide, voluntarily or not, to break the associated versioning semantics. As an example, a convention that seems to have emerged in RubyGems is to allow the introduction of backward incompatible changes in minor version bumps

[5]http://semver.org/

for *pre-release* packages. For instance, in 2010, release 0.5.0 of i18n notably broke the popular ActiveRecord gem, on which relied 874 packages (5.2% of RubyGems packages at this time). Similar problems were observed by Raemaekers et al. [24] for the Maven ecosystem, in which many Java projects do not respect the recommended semantic versioning scheme.

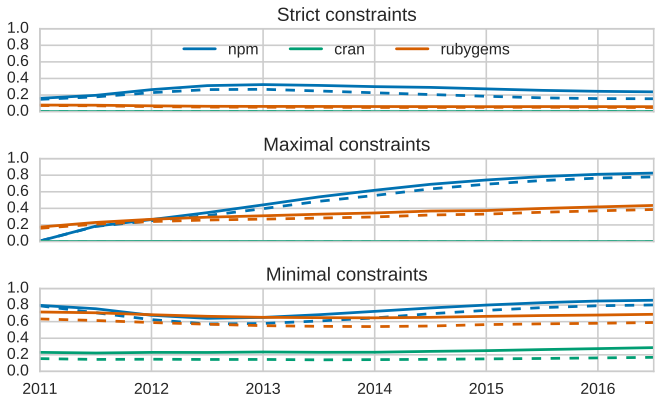

Fig. 9. Proportion of packages with dependencies (straight lines) and dependencies (dotted lines) that specify a *strict*, *minimal* or *maximal* dependency constraint.

Figure 9 illustrates the evolution of the use of *strict*, *minimal* and *maximal* dependency constraints in the three considered packaging ecosystems.[6] We observe that packages on CRAN do not tend to rely on *strict* or *maximal* constraints. This is a direct consequence of the aforementioned "rolling release" policy of CRAN.

For npm and RubyGems, both the proportion of packages and the proportion of dependencies that declare a *minimal* dependency constraint are relatively stable through time, but the proportions of packages or dependencies that declare *maximal* constraints increase over time. The latter suggests that more and more packages rely on *maximal* constraints to prevent, limit or control dependency updates. We also observe a high proportion of *strict* dependency constraints for npm and RubyGems. In the latest considered snapshot, this accounts for around 15% of all the dependencies in npm and 5% in RubyGems. We did not expect such a high proportion of *strict* constraints, given the problems that are associated with their use.

Summary:

- In combination with semantic versioning, dependency constraints can prevent packages to break due to dependency updates.
- A high proportion of packages on RubyGems and npm use dependency constraints, and the proportion of *maximal* constraints is increasing over time. An important number of their packages uses *strict* dependency constraints, but this proportion remains stable over time.
- Because of CRAN's versioning policy, few of its packages rely on dependency constraints. If they do, they tend to use *minimal* constraints.

VIII. Why should package maintainers be careful with dependency constraints?

While *strict* and *maximal* dependency constraints can be helpful to prevent packages to break after the introduction of a backward incompatible change in a dependency, such constraints can lead to co-installability issues [11]. This is especially true for *strict* dependency constraints that, by definition, target a single unique package version.

Tools such as gem or bundler for Ruby install packages at a system-wide level, and implicitly define a conflict between any two distinct releases of a same package. This means that one cannot install two releases of a same package, or two packages that depend (directly or transitively) on two distinct releases of a same package. There are tools to create isolated (or virtual) environments, e.g., RVM or isolate. Such tools allow to install different releases of the same package in separate, isolated locations, and prevent co-installability issues between different projects. However, they do not solve the co-installability issues that arise when two or more packages require different releases of the same package to be installed in a same environment.

npm's package manager offers a different technical approach to manage the co-installation of different releases of a dependency: when a package is installed, each dependency is installed in a subdirectory of its dependent package or, in other words, these dependencies are *vendored* at installation time. This allows different releases of a same package to be installed without co-installability issues[7], and may explain why we observed a higher proportion of *strict* dependency constraints for npm in Figure 9.

Thanks to its "rolling release" policy, co-installability issues do not arise for CRAN packages. Indeed, CRAN only allows the automatic installation of the latest available release of a package, and thus no two distinct releases of a same package can be installed. As a package must always compatible with the latest available release of each of its dependencies, any package update benefits to all dependent packages.

While techniques to address co-installability issues have already been developed and implemented for Debian and RPM [11], we are not aware of their use in any of the considered packaging ecosystems.

Co-installability is not the only issue to consider when specifying constraints on a dependency. Given that packages are quite frequently updated (cf. Figure 7), a package requiring a specific release of a dependency prevents it to automatically benefit from updates. This can be problematic, especially if the updates contain security or bug fixes. When using *strict* or *maximal* dependency constraints, maintainers therefore need to check regularly whether new important versions of their

[6]Notice that as a dependency constraint can be both *minimal* and *maximal*, the sum of the ratios can exceed 1.

[7]Problems may still occur at runtime, however, if distinct versions of a same package need to share objects.

dependencies are released, and weaken the dependency constraint in order to be able to benefit from this update. Tools that monitor package dependencies for updates can be very helpful in such a situation.

To quantify this phenomenon, we again used the technique of *survival analysis* to estimate the probability for a dependency constraint to remain unmodified for a certain amount of time. We carried out such an analysis both for *strict* and non *strict* (i.e., *minimal* or *maximal*) dependency constraints. The results are shown in Figure 10. Given the low proportion of *strict* dependency constraints in CRAN, we excluded this ecosystem from the analysis.

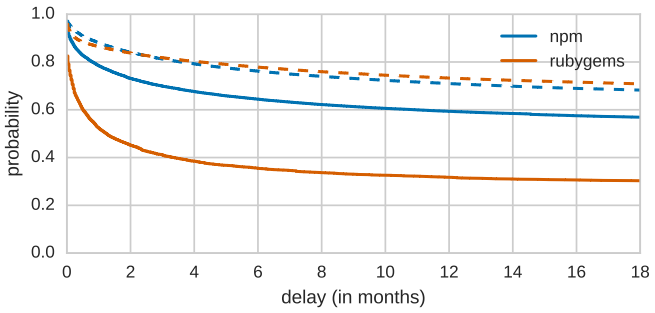

Fig. 10. Survival curve of probability that a dependency constraint remains unmodified. *Strict* dependency constraints are represented by straight lines; *non-strict* dependency constraints are represented by dotted lines.

Although the analysis does not measure *why* dependency constraints have been modified, it suggests that *strict* dependency constraints need to be modified much more frequently than non-strict ones. For instance, the probability for a *strict* dependency constraint in npm (resp. RubyGems) to be modified in its first six months is 0.36 (resp. 0.65), while this probability is only 0.24 (resp. 0.22) for *non-strict* constraints.

This confirms our intuition that *strict* constraints need be adjusted to benefit from more recent dependency releases, while this is not systematically the case for the other types of constraints. For instance, *minimal* dependency constraints are not affected by such an issue since they will be satisfied by any upcoming package update. The case of *maximal* dependency constraints is trickier. Depending on the constraint and the versioning policy of the targeted package, such a constraint can be satisfied either by many new dependency releases or by none of them. In the first case, the dependent package immediately benefits from these new releases, while the second case is comparable to the use of a *strict* dependency constraint.

Following the same reasoning, although *strict* constraints are adjusted more frequently than *minimal* or *maximal* constraints, we hypothesise that it also takes longer for *strict* constraints to be adjusted to benefit from a new version of a dependency. We verified this hypothesis by computing the **adjustment delay** for each *strict* dependency constraint as the time between the release of a new version of the dependency, and the time when the dependency constraint was effectively adjusted. The analysis involves only *strict* dependency constraints that (a) target an existing release of a package, and that (b) were adjusted consequently to (c) a new release of this package. This respectively corresponds to 69,823 (32%) of all *strict* dependency constraints in npm, and 21,124 (58%) in RubyGems.

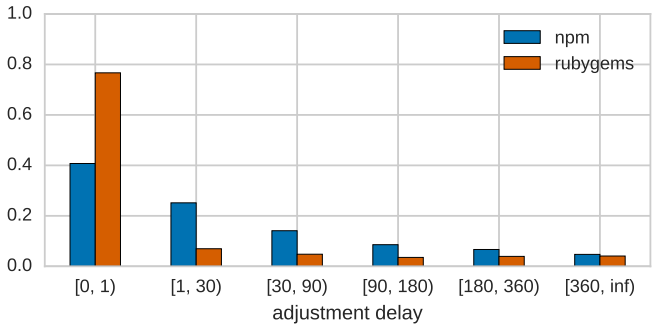

Fig. 11. Proportion of adjusted *strict* dependency constraints for each class of adjustment delay.

We classified the adjustment delay in different intervals: within the first day (i.e., it took no more than 24 hours to potentially benefit from the new release of the dependency), between one day and a month, between one and three months, between three and six months, between six months and a year, and more than a year.

Figure 11 shows the number of **adjusted** *strict* dependency constraints for each considered delay interval, proportionally to the total number of *strict* dependency constraints. We observe that around 41% of all considered *strict* constraints in npm, and 77% in RubyGems, are adjusted within the first day. Around 20% of the considered *strict* constraints in npm, and 12% in RubyGems, remain unmodified for at least 90 days. More importantly, around 5% of them in npm, and 4% in RubyGems, are not adjusted within a year. These are rather long adjustment delays during which the package will not benefit from the changes of a dependency.

Summary:

- Dependency constraints can lead to co-installability issues, and can prevent packages to benefit from potentially important dependency updates.
- Co-installability issues are addressed differently by the three ecosystems: npm relies on a technical solution, while CRAN addresses it in its policy. No official solution is provided for RubyGems.
- At least one out of ten dependencies with *strict* constraints take more than three months to be adjusted if the dependent package gets updated, and often much longer.
- While packages on npm or RubyGems are impacted by such a delay in quite similar proportions, packages on CRAN immediately benefit from dependency updates because of its “rolling release” policy.

IX. Threats to Validity

With the exception of CRAN where we extracted the data directly from package metadata, our analyses rely on

data that were somehow automatically gathered by npm and RubyGems. This data sometimes differs from the one that is contained in the package metadata, notably on RubyGems where we identified issues with the release date of some package releases.

The dependency graphs we constructed for Section VII rely on the collected metadata, meaning that implicit (dynamic or static) dependencies were not taken into account. While we explicitly considered versions and dependency constraints in Section VII, we relied on the chronological order of package releases instead to build the graphs of Sections IV and V. This chronological order should match the logical order induced by the versioning scheme, except for packages for which multiple branches are maintained in parallel. This is the case, for example, for some highly required packages in npm and RubyGems, explaining the irregularities observed in Figure 4. Given the proportion of affected package releases (3.2% on npm, 3.3% on RubyGems) and given that our observations are based on global trends, it is unlikely that this phenomenon affects our findings.

While Section VIII identified potentially important issues with the use of dependency constraints and the delay before a package can benefit from a dependency update, we did not consider *why* packages are updated, and made no distinction between updates containing security or bug fixes and those containing only API changes. This distinction is important, because typically security updates do not follow a specific development cycle, and are less prone to break dependent packages.

X. Conclusion and Future Work

This article presented an empirical inter-ecosystem comparison of package dependency issues. Our quantitative analysis considered the historical evolution of three packaging ecosystems npm, CRAN and RubyGems for the JavaScript, R and Ruby programming languages, respectively. We studied to which extent software packages rely on other packages, as well as to which extent packages updates are problematic in presence of (transitive) package dependencies. We quantified whether existing solutions such as semantic versioning and dependency constraints solve this problem.

Regardless of the considered ecosystem, package maintainers somehow face problems related to dependency updates in their packages. The problems and their solutions vary from one ecosystem to another, and depend both on the policies and the technical aspects of each ecosystem. Packages dependency updates have non-negligible maintenance costs because their effect can propagate transitively. While each ecosystem provides specific and different solutions to reduce problematic package updates, none of these are perfect. Update problems can still occur, implying the need for better tools and policies to reduce, prevent and correct them before they actually become problematic. There is generally no perfect solution to manage dependency updates. If the use of dependency constraints prevents, to some extent, backward incompatible updates issues, the frequent use of *strict* constraints increases the risk to miss important updates, and the risk of facing co-installability issues.

Solutions exist to (partially) address these problems, but do not tend to take into account the presence of transitive dependencies. The use of semantic versioning, in combination with *upper bound* constraints, allows to some extent to benefit from some updates without being affected by backward incompatible ones. The use of isolated environments can reduce the risks of co-installability issues. A more radical solution to deal with those two problems, as proposed by CRAN, is to require packages to always be up-to-date with the latest available release of its dependencies. While this completely avoids co-installability issues, and makes sure that a package benefits from dependency updates, it also means that a maintainer is forced to systematically adapt its package whenever a backward incompatible update occurs in one of its dependency packages. This policy fully exposes packages to a perpetual maintenance process, and puts an important burden on the maintainers.

In future work, following the spirit of mixed method research [25], we aim to complement our current *quantitative* analysis and comparison of packaging ecosystems with a *qualitative* analysis. This will be achieved by carrying out surveys with ecosystem developers, similar to the one conducted by Bogart et al. [1] but specifically focused on software dependencies issues. Among others, this will increase our understanding of *why* package are updated (e.g., bug or security fixes, API changes, etc.), and *how* developers of dependent packages are affected and react to updates.

We aim to extend our empirical analysis to other packaging ecosystems, by including ecosystems for other programming languages, as well as by considering package distributions for other domains (e.g., Linux distributions, plugin-based development environments like Eclipse and NetBeans, ...). This will allow us to assess whether the considered domain influences the ecosystem's structure and its evolution over time. We also aim to study *mixed source* ecosystems (containing a combination of both open and closed software, potentially using a wide variety of software licences) in order to explore how this affects the problems, challenges and solutions related to dependency issues.

Finally, we aim to carry out *socio-technical* comparisons of the considered ecosystems, in order to understand how the collaboration structure of the community of ecosystem contributors affects the technical package dependency structure and conversely [26].

Acknowledgments

This research was carried out in the context of ARC research project AUWB-12/17-UMONS-3 "Ecological Studies of Open Source Software Ecosystems". We express our gratitude to Philippe Grosjean and the anonymous reviewers for the very useful feedback on an earlier version of this article.

References

[1] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "How to break an API: Cost negotiation and community values in three software ecosystems," in *Int'l Symp. Foundations of Software Engineering*, 2016.

[2] A. Decan, T. Mens, and M. Claes, "On the topology of package dependency networks — a comparison of three programming language ecosystems," in *European Conference on Software Architecture Workshops*. ACM , Nov. 2016.

[3] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the JavaScript package ecosystem," in *Int'l Conf. Mining Software Repositories*. ACM, 2016, pp. 351–361.

[4] A. Decan, T. Mens, M. Claes, and P. Grosjean, "On the development and distribution of R packages: An empirical analysis of the R ecosystem," in *European Conference on Software Architecture Workshops*, 2015, pp. 41:1–41:6.

[5] K. Hornik, "Are there too many R packages?" *Austrian Journal of Statistics*, vol. 41, no. 1, pp. 59–66, 2012.

[6] A. Decan, T. Mens, M. Claes, and P. Grosjean, "When GitHub meets CRAN: An analysis of inter-repository package dependency problems," in *Int'l Conf. Software Analysis, Evolution, and Reengineering*. IEEE, Mar. 2016, pp. 493–504.

[7] K. Blincoe, F. Harrison, and D. Damian, "Ecosystems in GitHub and a method for ecosystem identification using reference coupling," in *Int'l Conf. Mining Software Repositories*. IEEE, 2015, pp. 202–211.

[8] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "How the Apache community upgrades dependencies: an evolutionary study," *Empirical Software Engineering*, vol. 20, no. 5, pp. 1275–1317, 2015.

[9] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to API deprecation? the case of a Smalltalk ecosystem," in *Int'l Symp. Foundations of Software Engineering*. ACM , 2012.

[10] P. Abate, R. Di Cosmo, L. Gesbert, F. L. Fessant, R. Treinen, and S. Zacchiroli, "Mining component repositories for installability issues," in *Int'l Conf. Mining Software Repositories*, 2015, pp. 24–33.

[11] R. Di Cosmo and J. Vouillon, "On software component co-installability," in *Joint European Conf. Software Engineering / Foundations of Software Engineering*. ACM, 2011, pp. 256–266.

[12] M. Claes, T. Mens, R. D. Cosmo, and J. Vouillon, "A historical analysis of Debian package incompatibilities," in *Int'l Conf. Mining Software Repositories*, 2015, pp. 212–223.

[13] R. Di Cosmo, S. Zacchiroli, and P. Trezentos, "Package upgrades in foss distributions: Details and challenges," in *1st Int'l Workshop on Hot Topics in Software Upgrades*. New York, NY, USA: ACM, 2008.

[14] J. Sametinger, *Software Engineering with Reusable Components*. Springer, 1997.

[15] N. Cliff, *Ordinal methods for behavioral data analysis*. Psychology Press, Sep. 1996. [Online]. Available: http://www.worldcat.org/isbn/0805813330

[16] Z. Hemel, "Javascript: A language in search of a standard library and module system," http://zef.me/blog/2856/javascript-a-language-in-search-of-a-standard-library-and-module-system, February 2010.

[17] I. Z. Schlueter, "The npm blog: kik, left-pad, and npm," http://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm, March 2016.

[18] T. Mens, "Anonymized e-mail interviews with R package maintainers active on CRAN and GitHub," University of Mons, Tech. Rep., 2015. [Online]. Available: http://arxiv.org/abs/1606.05431

[19] B. Morris, "Rest apis don't need a versioning strategy, they need a change strategy," http://www.ben-morris.com/rest-apis-dont-need-a-versioning-strategy-they-need-a-change-strategy/, October 2016.

[20] D. G. Kleinbaum and M. Klein, *Survival Analysis: A Self-Learning Text*, 3rd ed. Springer, 2012.

[21] I. Samoladas, L. Angelis, and I. Stamelos, "Survival analysis on the duration of open source projects," *Information & Software Technology*, vol. 52, no. 9, pp. 902–922, 2010.

[22] CRAN Repository Maintainers, "CRAN repository policy," https://cran.r-project.org/web/packages/policies.html, September 2016.

[23] J. Ooms, "Possible directions for improving dependency versioning in R," *R Journal*, vol. 5, no. 1, pp. 197–206, Jun. 2013.

[24] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning versus breaking changes: A study of the Maven repository," in *Working Conf. Source Code Analysis and Manipulation*, Sept 2014, pp. 215–224.

[25] R. B. Johnson and A. J. Onwuegbuzie, "Mixed methods research: A research paradigm whose time has come," *Educational Researcher*, vol. 33, no. 7, pp. 14–26, Oct. 2004.

[26] T. Mens, "An ecosystemic and socio-technical view on software maintenance and evolution," in *Int'l Conf. Software Maintenance and Evolution*. IEEE, 2016.