Empir Software Eng (2019) 24:381–416
https://doi.org/10.1007/s10664-017-9589-y

An empirical comparison of dependency network evolution in seven software packaging ecosystems

Alexandre Decan[1] · **Tom Mens**[1] · **Philippe Grosjean**[1]

Published online: 10 February 2018

Abstract Nearly every popular programming language comes with one or more package managers. The software packages distributed by such package managers form large software ecosystems. These packaging ecosystems contain a large number of package releases that are updated regularly and that have many dependencies to other package releases. While packaging ecosystems are extremely useful for their respective communities of developers, they face challenges related to their scale, complexity, and rate of evolution. Typical problems are backward incompatible package updates, and the risk of (transitively) depending on packages that have become obsolete or inactive. This manuscript uses the libraries.io dataset to carry out a quantitative empirical analysis of the similarities and differences between the evolution of package dependency networks for seven packaging ecosystems of varying sizes and ages: Cargo for Rust, CPAN for Perl, CRAN for R, npm for JavaScript, NuGet for the .NET platform, Packagist for PHP, and RubyGems for Ruby. We propose novel metrics to capture the growth, changeability, reusability and fragility of these dependency networks, and use these metrics to analyze and compare their evolution. We observe that the dependency networks tend to grow over time, both in size and in number of package updates, while a minority of packages are responsible for most of the package updates. The majority of packages depend on other packages, but only a small proportion of packages accounts for most of the reverse dependencies. We observe a high proportion of "fragile" packages due

Communicated by: Gabriele Bavota and Andrian Marcus

✉ Alexandre Decan
Alexandre.Decan@umons.ac.be

Tom Mens
Tom.Mens@umons.ac.be

Philippe Grosjean
Philippe.Grosjean@umons.ac.be

[1] COMPLEXYS Research Institute, University of Mons, Mons, Belgium

to a high and increasing number of transitive dependencies. These findings are instrumental for assessing the quality of a package dependency network, and improving it through dependency management tools and imposed policies.

Keywords Software repository mining · Software ecosystem · Package manager · Dependency network · Software evolution

1 Introduction

Traditionally, software engineering research has focused on understanding and improving the development and evolution of individual software systems. The widespread use of online collaborative development solutions surrounding distributed version control tools (such as Git and GitHub) has lead to an increased popularity of so-called *software ecosystems*, 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.

Software 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. Complicated and changing dependencies are a burden for many developers and are often referred to as the "dependency hell" (Artho et al. 2012; Bogart et al. 2016). If not properly maintained, the presence of such dependencies may become detrimental to the ecosystem quality. Indeed, developers are reluctant to upgrade their dependencies (Bavota et al. 2015), while outdated dependencies have been shown to be more vulnerable to security issues (Cox et al. 2015). Researchers have therefore been actively studying the evolution dynamics of packaging dependency networks in order to support the many problems induced by their macro-level evolution (Decan et al. 2016; González-Barahona et al. 2009). A famous example of such a problem was the left-pad incident for the npm package manager. Despite its small size (just a few lines of source code), the sudden and unexpected removal of the left-pad package caused thousands of direct and indirect dependent projects to break, including very popular ones such as Atom and Babel (Schlueter 2016; Haney 2016).

Comparative studies between package dependency networks of different ecosystems are urgently needed, to understand their similarities and differences and how these evolve over time. Such studies may help to improve software analysis tools by taking into account specific ecosystem characteristics to better manage and control the intrinsic fragility and complexity of evolving software ecosystems.

The current paper builds further upon our previous work. In Decan et al. (2016) we empirically compared the package dependency network of three popular packaging ecosystems: CRAN for R, PyPI for Python and npm for JavaScript. These analyzes focused on the structural complexity of these dependency networks. We found important differences between the considered ecosystems, which can partly be explained by the functionality offered by the standard library of the ecosystem's underlying programming language, as well as by other ecosystem specificities. This implies that the findings for one ecosystem cannot necessarily be generalized to another. We therefore suggested to extend the analyses by considering more ecosystems, and by taking into account the evolution of the dependency networks. In Decan et al. (2017), we started to carry out an historical analysis of package dependency network evolution for CRAN, npm, and RubyGems. We studied to which extent packages rely on other packages, as well as to which extent packages updates are

problematic in the presence of (transitive) package dependencies. We observed that, because of the presence of many transitive dependencies, a package failure may potentially affect many other packages. While each ecosystem provides specific and different ways to reduce the impact of problematic package updates, none of these solutions are perfect and package maintainers remain faced with occasional update problems.

The current paper extends the results of Decan et al. (2017) by considering seven different packaging ecosystems for as many different programming languages: Cargo for Rust, CPAN for Perl, CRAN for R, npm for JavaScript, NuGet for the .NET development platform, Packagist for PHP, and RubyGems for Ruby. As far as we know, this is the first work to compare that many different ecosystems, and to use the recent libraries.io dataset for that purpose. Another novelty is that we introduce three new metrics to facilitate ecosystem comparison despite the diversity of the considered ecosystems is terms of age and size: the *Changeability Index* captures an ecosystem's propensity to change over time; the *Reusability Index* captures the ecosystem's amplitude and extent of reuse; and the *P-Impact Index* assesses the fragility of an ecosystem.

The remainder of this article is structured as follows. Section 2 discusses related work. Section 3 presents the used terminology, motivates the selected packaging ecosystems and explains the data extraction process. Sections 4 to 7 each address a specific research question.

Section 4 studies our first research question: "*How do package dependency networks grow over time?*" We observe a continuing growth of the number of packages and their dependency relationships. Given that we observed in Decan et al. (2017) that package dependencies may be problematic in case of package updates, Section 5 studies a second research question: "*How frequently are packages updated?*" Because package dependencies lead to an increased fragility of the ecosystem, Section 6 studies a third research question "*To which extent do packages depend on other packages?*" Section 7 studies the fourth research question: "*How prevalent are transitive dependencies?*" Indeed, due to the prevalence of transitive dependencies in the package dependency network, package failures may propagate through the network and may impact large parts of the ecosystem.

Section 8 puts our research findings into perspective, by discussing how an ecosystem's policy influences the observed results, what are the limitations of existing techniques to deal with package dependencies and package updates, and how our results could form the basis of ecosystem-level health analysis tools. Section 9 presents the threats to validity of our study. Section 10 outlines future work, by providing initial evidence for laws of software ecosystem evolution, and suggesting to explore software ecosystem evolution from a complex network or socio-technical network point of view. Finally, Section 11 concludes.

2 Related work

The research domain of software ecosystems is huge. We refer the reader to some recent key references for further reading (Manikas and Hansen 2013; Jansen et al. 2013; Serebrenik and Mens 2015). Given that the current article specifically focuses on *packaging* ecosystems, and more in particular on technical dependencies in package *dependency networks*, this section reports mainly on the related work in those areas. Although very interesting in their own right, social dependency networks are out of scope for the current work, and work related to such networks will therefore not be discussed here.

Many researchers have studied (and compared) technical dependency networks at the level of components contained within individual software projects (e.g., studying the

modularity of the dependency network between classes in a Java project (Dietrich et al. 2008; Zanetti and Schweitzer 2012)). A detailed account of such works is outside the scope of the current article, since our focus is at the ecosystem level, i.e., we consider dependencies across different projects (as opposed to within individual projects).

Many researchers have studied package dependencies issues in a variety of programming language packaging ecosystems. Most studies, however, were limited to a single ecosystem. Wittern et al. (2016) studied the evolution of a subset of JavaScript packages in npm, analyzing characteristics such as their dependencies, update frequency, popularity, version numbering and so on. Abdalkareem et al. (2017) also carried out an empirical case study of npm, focusing on what they refer to as "trivial" packages, and the risk of depending on such packages. The results were inconclusive, in the sense that depending on trivial packages can be useful and unrisky, provided that they are well implemented and tested. The CRAN packaging ecosystem has been previously studied (Hornik 2012; Germán et al. 2013; Decan et al. 2015), and dependencies have been shown to be an important cause of errors in R packages both on CRAN and GitHub (Decan et al. 2016). Blincoe et al. (2015) looked at Ruby as part of a larger GitHub study on the emergence of software ecosystems, and observed that most ecosystems are centered around one project and are interconnected with other ecosystems. Bavota et al. (2015) 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. Considering that changes of a package might break its dependent packages, Bavota et al. found that developers were reluctant to upgrade the packages they depend on. Robbes et al. (2012) 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.

Santana and Werner (2013) focused on the visualization aspects of software ecosystem analysis, and proposed a social visualization of the interaction between contributors of the community on the one hand, and a technical visualization of the ecosystem's project dependencies on the other hand. They did not focus, however, on how to compute or visualize metrics about the ecosystem.

The dependence on packages with security vulnerabilities has been studied in industrial software projects (Cadariu et al. 2015). Cox et al. revealed that systems using outdated dependencies are four times more likely to have security issues as opposed to systems that are up-to-date (Cox et al. 2015).

Very little research results are available that actually compare dependency and maintainability issues across different packaging ecosystems. Bogart et al. (2016) compared three ecosystems (npm, CRAN and Eclipse) in order to understand the impact of community values, tools and policies on breaking changes. They carried out a qualitative analysis by relying on interviews with developers of the studied ecosystems. Specifically related to package dependencies, they identified two main types of mitigation strategies adopted by package developers to reduce their exposure to changes in other packages: limiting the number of dependencies; and selecting only dependencies to packages that they trust. Bogart's work complements the current paper, which is based on a quantitative empirical comparison of the dependency networks of packaging ecosystems.

Inspired by our own previous work (Decan et al. 2016, 2017), Kikas et al. (Kikas et al. 2017) carried out an empirical comparison of the dependency networks of three ecosystems (npm, RubyGems and Rust), confirming our own findings related to the ecosystems' fragility and vulnerability to transitive dependencies.

3 Methodology

3.1 Preliminaries

All empirical analysis presented in the current article is supported by a replication package (Decan and Mens 2017). Table 1 informally defines all terms used in this article. The parts of the term indicated between parentheses in the first column of the table will be implicitly assumed if they are clear from the context.

Table 1 Informal definition of terms used in this article

Term	Informal definition
(Packaging) Ecosystem	The collection and history of all tools, software artifacts and community members surrounding a particular *package manager*.
Package Manager	A coherent collection of software tools that automates the process of installing, configuring, upgrading or removing software *packages* on a computer's operating system in a consistent manner.
Package	A computer program providing specific functionalities. A package usually exists in many versions which are called *release*s. By abuse of language, a *package* at time t denotes its latest available *release* at time t.
(Package) Release	A specific version of a *package* that can be accessed and installed through the *package manager*. It usually comes in the form of an archive file containing what is needed to build, configure and deploy the package version, and includes a *manifest* containing important metadata such as its owner, name, description, timestamp, and a list of direct *dependencies* to other *packages* that are required for its proper functioning.
(Package) Update	A new *release* of a package, provided by the *package manager*, that succeeds (i.e., corresponds to a higher version number or timestamp) a previous release of the same package.
(Package) Dependency Network (at time t)	A graph structure in which the nodes represent all the *packages* made available by the *package manager* at time t, and the directed edges represent *direct dependencies* between the latest available *releases* at time t.
Dependency	An explicitly documented reference (in the manifest of a *release*) to another *package* that is required for its proper functioning. A dependency can specify constraints to restrict the supported *releases* of the target package. Dependencies that are explicitly documented in the release manifest (i.e., edges in the *dependency network*) are called **direct dependencies**. Those that are part of the transitive closure of the *dependency network* are called **transitive dependencies**. Transitive dependencies that are not direct are called **indirect dependencies**.
Reverse Dependency	Reverse dependencies are obtained by following the edges of the *dependency network* in the opposite direction. As for normal dependencies, they can be **direct**, **transitive** or **indirect**.
Required package	A *package* that is the target of at least one *dependency* from another *package*. In a similar vein, we define **transitively required**.
Dependent (package)	A *package* that is the target of at least one *reverse dependency* from another *package*. In a similar vein, we define **transitively dependent**.
Connected package	A package that is either a *required* or a *dependent* package.
Top-level package	A *dependent package* that is not a *required* package.

3.2 Statistical analysis techniques

One of the statistical techniques that will be used in this article is *survival analysis* (a.k.a. event history analysis) (Aalen et al. 2008). It is a technique that models "time to event" data with the aim to estimate the survival rate of a given population, i.e., the expected time duration until a specific "event" happens (such as death of a biological organism, failure of a mechanical component, recovery of a disease). A common non-parametric statistic used to estimate survival functions is the Kaplan-Meier estimator (Kaplan and Meier 2012).

Survival analysis models take into account the fact that some observed subjects may be "censored", either because they leave the study during the observation period, or because the event of interest was not observed on them during the observation period. In empirical software engineering, survival analysis has been used to estimate the survival of open source projects over time (Samoladas et al. 2010), to analyze the use and removal of functions in PHP code (Kyriakakis and Chatzigeorgiou 2014), to analyze dead Java code (Scanniello 2011), to analyze the survival of database access libraries in Java code (Goeminne and Mens 2015; Decan et al. 2017), and to analyze survival of developers in open source projects (Lin et al. 2017). Inspired by this research, in this paper we will use the technique to analyze the survival of package releases in packaging ecosystems.

As several research questions require to measure statistical dispersion, we borrowed ideas from econometrics and used the Lorenz curve (Lorenz 1905) and the related Gini index (Gini 1912). Those two techniques are usually applied to assess the inequality of the wealth distribution among people, regions, countries, and so on. The Lorenz curve is typically used to compare graphically the cumulative proportion of income versus the cumulative proportion of individuals, illustrating the inequality of a wealth distribution. The Gini coefficient (or Gini index) is a widely used social and economic indicator to cope with unevenly distributed data. Its value is comprised between 0 and $1 - \frac{1}{n}$, where n is the size of the considered population. A value of 0 expresses perfect equality and a value of $1 - \frac{1}{n}$ expresses maximal inequality among individuals, where one individual possesses all of the wealth of the given population.

Gini index has been previously used in empirical software engineering. Considering software metrics data as wealth distributions, Vasa et al. (2009) showed that many software metrics not only display high Gini values, but that these values are remarkably consistent over time. Giger et al. (2011) used the index to investigate how changes made to source code are distributed in the Eclipse project. Goeminne and Mens (2011) measured the inequality of different kinds of activity in open source software projects using different econometrics, including Gini, and found empirical evidence of highly skewed distributions in the activity of developers involved in open source software projects.

3.3 Selected packaging ecosystems

This article focuses on programming language ecosystems, and more specifically packaging ecosystems revolving around package managers for specific programming languages. Such ecosystems tend to have a very active community of contributors, making their dependency networks very large, and causing difficulties in managing and analyzing the evolution of these networks. Given that these ecosystems serve a similar goal, namely to serve the developer community surrounding a particular programming language, it makes sense to empirically compare them.

The seven ecosystems we selected form a representative collection of package managers, covering different programming languages, dependency network sizes and ages, as

summarized in Table 2. On 1 April 2017, these ecosystems hosted together 5,812 k releases for more than 830 k packages. Among those package releases, we identified 20,425 k dependency relationships. A brief description of each considered package manager is presented below:

- Cargo is the official package manager for Rust, a compiled programming language released in 2012 by Mozilla. Since 2014, its official package registry is crates.io, usually referred to as Cargo. It is the youngest and smallest of the selected ecosystems.
- CPAN (cpan.org) stands for Comprehensive Perl Archive Network and is the oldest considered ecosystem. It was introduced in 1995 as a large collection of Perl software, an interpreted programming language developed in 1987.
- CRAN (cran.r-project.org), the Comprehensive R Archive Network, is the second oldest ecosystem we consider. It constitutes the official repository of the statistical computing environment R. It has the particularity of following a "rolling release" policy, meaning that only the latest release of a package can be automatically installed from CRAN. As a consequence, packages must always be compatible with the latest release of each of their dependencies, as well as with the latest version of the R language.
- npm (npmjs.com), started in 2010, is the official package registry for the JavaScript runtime environment Node.js. It is the largest considered ecosystem with nearly half a million packages.
- NuGet (nuget.org), formerly known as NuPack, is the official package manager developed by Microsoft for the .NET development platform. By extension, NuGet also designates NuGet Gallery, the central package repository for NuGet.
- Packagist (packagist.org) is the default package repository for Composer, the de-facto standard package manager for the interpreted, web-oriented programming language PHP. Although Packagist was started in 2012, it also hosts packages that were developed prior to its release, in the early days of PHP (1994).
- RubyGems (rubygems.org) is the largest collection of packages for Ruby, an interpreted object-oriented programming language. RubyGems was started on *Pi Day* 2004 and, like Packagist, also hosts packages that were developed prior to its release.

Unless explicitly mentioned, all conducted statistical analyses considered the whole lifetime of each ecosystem up to 1 January 2017. In the accompanying figures we decided to display only the period starting from 1 January 2012 to 1 January 2017 for the sake of clarity.

Table 2 Characteristics of the selected packaging ecosystems on 1 April 2017: creation year, language, number of packages, number of releases, number of dependencies across all releases, and release date of the oldest package

Manager	Creation	Lang.	Pkg.	Rel.	Deps.	Oldest pkg.
Cargo	2014	Rust	9 k	48 k	150 k	2014-11
CPAN	1995	Perl	34 k	259 k	1,078 k	1995-08
CRAN	1997	R	12 k	67 k	164 k	1997-10
npm	2010	JavaScript	462 k	3,038 k	13,611 k	2010-11
NuGet	2010	.NET	84 k	936 k	1,665 k	2011-01
Packagist	2012	PHP	97 k	669 k	1,863 k	1994-08
RubyGems	2004	Ruby	132 k	795 k	1,894 k	1999-12

3.4 Data extraction process

For our empirical study, we relied on information about package releases and dependencies collected by the open source discovery service libraries.io (Nesbitt and Nickolls 2017). The extracted data falls under the CC-BY-SA 4.0 license.[1] libraries.io extracted all the metadata from the manifest of each package, based on the list of packages provided by the official registry of the packaging manager.

At the time of carrying out our experiment, libraries.io provided package release data for 33 popular package managers in total. We excluded those that we considered too small (less than 5,000 packages). From the remaining 17 packaging ecosystems, we selected seven for which it was possible to obtain all the necessary metadata from the package manifests statically: Cargo, CPAN, CRAN, npm, NuGet, Packagist, and RubyGems.

We excluded the other packaging ecosystems from our study for a variety of reasons: because they were too domain-specific, targeting a specific software framework (e.g., Meteor) or software component (e.g., WordPress and Atom); because they host a subset of packages available through another already considered ecosystem (e.g., Bower manages a subset of npm); or because the developers of libraries.io informed us that important dependency information was incomplete or missing (e.g., GO, PyPi, Maven, CocoaPods, Clojars or Hackage). The latter case applied to two very popular and important packaging ecosystems, namely Maven for Java and PyPi for Python. For these package managers, (the list of) package dependencies can be dynamically defined and may depend on the environment that interprets the manifest at installation time, and hence are not available statically. An interesting topic of future work would therefore constitute the automated analysis and extraction of such dynamic dependencies.

To ascertain the correctness of the data provided by libraries.io, we manually cross-checked its retrieved dependency metadata with our own (less recent) datasets for CRAN, npm and RubyGems that we had used in our previous work (Decan et al. 2017). The metadata matched for the considered period, convincing us of its correctness.

For CRAN, we completed the metadata extracted from libraries.io with data about archived package releases (i.e., releases that used to be distributed on CRAN but are no longer available through the package manager). To achieve this, we relied on extractoR, a publicly available[2] R package that was developed specifically for the purpose of mining and analyzing CRAN packages (Claes et al. 2014). With the help of extractoR, we retrieved the metadata of 1,078 additional packages and 5,182 additional package releases.

For npm, we observed that the left-pad incident (Schlueter 2016; Haney 2016) seems to have lead some developers to design packages whose sole purpose is to depend on as many other packages as possible. For instance, the npm-gen-all package is defined as a package that "*will create a multitude of npm projects that will depend on every npm package published*". We identified around 250 of such packages, and explicitly ignored them for our analyzes since they only introduce noise and do not serve any useful purpose.

For each package release of each considered packaging ecosystem, we considered the list of packages on which it depends. We restricted the dependencies to those required to install and execute the package. Dependencies that are only required to develop or test a package were excluded from our analyzes because not all ecosystems make use of them.

[1] Creative Commons Attribution-ShareAlike 4.0 International, see https://creativecommons.org/licenses/by-sa/4.0/.

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

Even for ecosystems that support them, not every package declares a complete and reliable list of development or test dependencies. Depending on the ecosystem, this means that we restricted ourselves to dependencies of type "runtime", "imports", "depends" and "normal", while omitting dependencies of type "development", "optional", "enhances", "suggests", "build", "configure", "test", "develop" or "dev". We also excluded dependencies that target packages that were not available through the package manager (e.g., packages that are hosted directly on the web or on Git repositories). This represents less than 2.5% of all dependencies in Cargo, CRAN, npm, NuGet and RubyGems, around 9.35% of the dependencies in Packagist and 30.11% of those in CPAN. A possible explanation for this higher proportion of unavailable dependencies in CPAN relates to the presence of very old packages that are not maintained anymore and still depend on packages that are no longer available on CPAN.

4 RQ_1: How do package dependency networks grow over time?

As a first research question, we study how fast each packaging ecosystem and package dependency network is growing over time. Being aware of this speed of growth is important, since it may become increasingly difficult to manage the ecosystem without putting in place proper policies, processes, quality standards and tool support capable of managing this growth (Hornik 2012). Our hypothesis is that the number of new packages must continuously increase in order to offer new functionality to the ecosystem users. On the other hand, the increase should not be too fast, since a higher number of dependencies between packages makes the ecosystem more interconnected and therefore more complex.

We computed the growth of each package dependency network by counting its number of nodes (packages) and edges (package dependencies). The evolution of the number of packages is presented in Fig. 1, using a logarithmic scale for the y-axis. Figure 2 presents the evolution of the number of dependencies for monthly snapshots of the dependency networks, and looks quite similar to the previous one. We conclude that **all considered ecosystems continue to grow over time.**

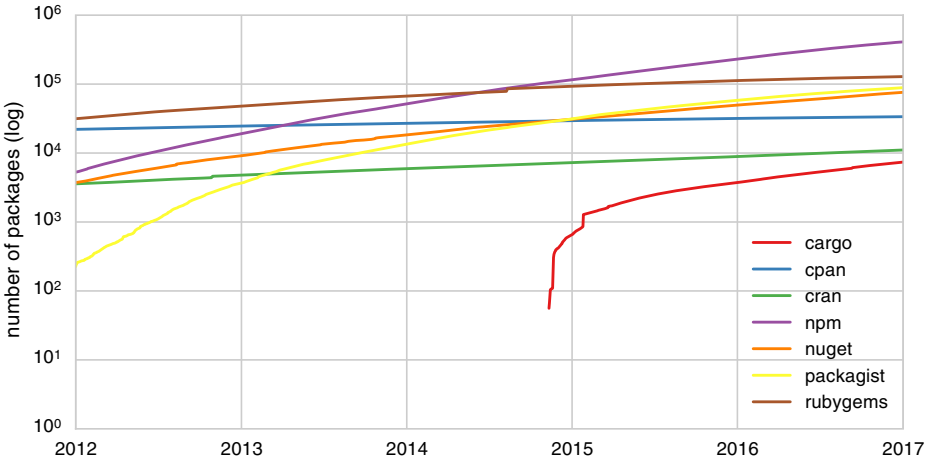

Fig. 1 Evolution of the number of packages

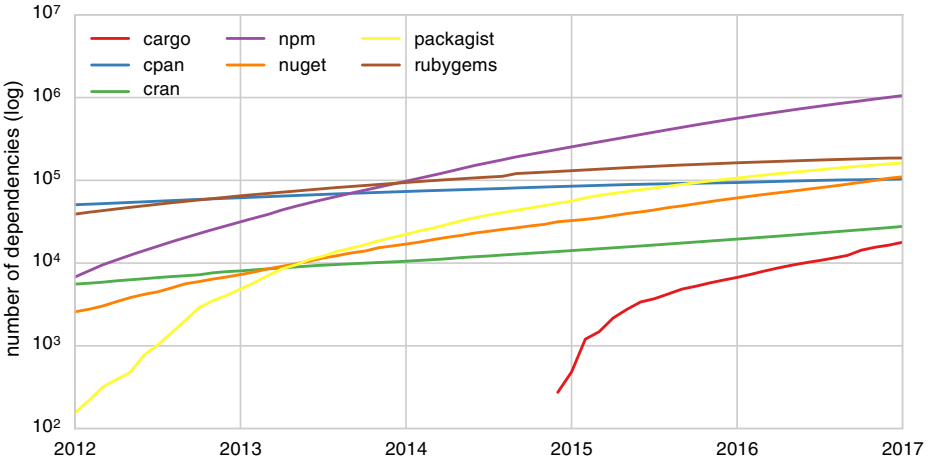

Fig. 2 Evolution of the number of dependencies (considering for each point in time the latest available release of each package)

To determine whether the dependency networks have a different speed of growth according to both size metrics, we carried out a regression analysis using different parametric growth models. The R^2 values reflecting the "goodness of fit" of the models[3] are summarized in Table 3. Only the linear and exponential models are presented as these invariably have the highest R^2 values of all considered growth models. We observe that Cargo and CPAN reveal a linear growth for both size metrics (with $R^2 \geq 0.97$ in all cases). CRAN and npm are on the other side of the spectrum, with an observed exponential growth for both size metrics (with $R^2 \geq 0.92$ in all cases). NuGet falls somewhere in between, growing exponentially in number of dependencies, but linearly in number of packages. Packagist and RubyGems have the opposite behavior, growing linearly in number of dependencies but exponentially in number of packages.

To find out if the number of dependencies is growing faster than the number of packages, we computed the ratio of the number of dependencies over the number of packages. While this ratio remains stable for CPAN, Packagist and RubyGems, it increases for Cargo, CRAN, npm and NuGet, suggesting an increasing complexity relative to the number of packages.

We conclude that **the increase in size and complexity varies across ecosystems**. The observed differences do not seem to depend on the ecosystem size or age. For example, CRAN is one of the smallest and oldest ecosystems and npm the largest and much more recent, but they both exhibit an exponential growth rate according to both size metrics. We assume that external factors, such as the popularity of the ecosystem or the activity of its contributor community, play a role in this growth rate. Determining these external factors and how they influence the ecosystem growth remains a topic of future work.

[3] $R^2 \in [0, 1]$ and the closer to 1 the better the model fits the data.

Table 3 R^2-values of regression analysis on the evolution of the size metrics

# packages	Cargo	CPAN	CRAN	npm	NuGet	Packagist	RubyGems
linear	**0.99**	**0.97**	0.84	0.83	**0.92**	0.77	0.88
exponential	0.82	0.87	**0.97**	**0.92**	0.91	**0.89**	**0.93**
# dependencies							
linear	**0.97**	**1.00**	0.98	0.85	0.89	**0.93**	**1.00**
exponential	0.85	0.98	**0.99**	**0.98**	**0.99**	0.88	0.97

Bold items correspond to the highest values

Summary. To answer RQ_1 we studied the growth of package dependency networks over time, based on the number of packages and their dependencies. We observed that the dependency networks of all studied ecosystems tend to grow over time, though the speed of growth may differ. We also analysed the ratio of dependencies over packages as a simple measure of the network's complexity, and observed that this complexity remains stable for some ecosystems, while it tends to increase for others.

5 RQ_2: How frequently are packages updated?

Updating a package to a new release, regardless of whether it contains new features, bug fixes or API changes, is a common and natural process for a maintainer. However, such package updates can often be quite challenging in presence of package dependencies (Morris 2016): "*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.*" This threat is confirmed by previous empirical research observing that package updates may cause many maintainability issues or even failures in dependent packages (Di Cosmo et al. 2008; Bavota et al. 2015; Bogart et al. 2016; Decan et al. 2017).

To provide an upper bound estimate of how often such issues may arise, we compare across the considered ecosystems how frequently packages are being updated. Figure 3 shows the evolution of the monthly number of package updates for each ecosystem. We observe that, depending on the ecosystem, **the number of package updates either remains stable or tends to grow over time.**

For the smallest ecosystem Cargo and the two oldest ecosystems CPAN and CRAN, the number of updates remains more or less stable. For RubyGems we observe a slight increase in the number of updates. For npm, NuGet, and Packagist the observed growth is considerably larger. We hypothesize that the frequency of package updates is related not only to the size of the ecosystem but also to the popularity of the ecosystem and its associated programming language.

The notable exception is CRAN which, despite being linked to the popular R language, exhibits a relatively low monthly number of updates. A plausible explanation is that CRAN package maintainers are encouraged to limit the frequency of package updates because of CRAN's "rolling release" policy that imposes packages to be up-to-date with their dependencies (CRAN Repository Maintainers 2016): "*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."

While Fig. 3 provides a global view on the package update frequency, let us narrow this further down by distinguishing between *required* and *dependent* packages. Both types of packages face opposing forces influencing their update frequency. On the one hand, required packages need to be updated regularly to take into account changes requested by the developers of their dependents. On the other hand, dependent packages prefer to have limited updates of their dependencies as this may introduce backward incompatibilities. This is indeed a complaint of many package maintainers (Mens 2015): "*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 professional projects I was part of.*"

We therefore carry out a cross-ecosystem comparison of the time between successive releases of a package, by performing a survival analysis over the population of all package releases of each ecosystem. We distinguish between packages that are required and those that are not. For each release we consider the time required for a more recent release of the same package to become available in the package manager. Figure 4 presents the Kaplan-Meier survival curves estimating the survival function of the probability that a release is not yet updated at time t.

Disregarding CRAN, we observe a high similarity across ecosystems, with **a probability higher than 50% for a package release to be updated within two months**, regardless of whether the package is required or not. The higher resilience of CRAN packages to new updates can again be explained by CRAN's policy, which is more demanding with respect to package updates.

We also observe that the population of required packages (Fig. 4 right) receives updates considerably more frequently than those that are not required (Fig. 4 left). Indeed, the survival curves for updates of required packages are invariably lower. We statistically tested this observation by performing a log-rank test to compare the survival curves of non required packages to those of required packages. The test confirms with significance level $\alpha = 0.01$

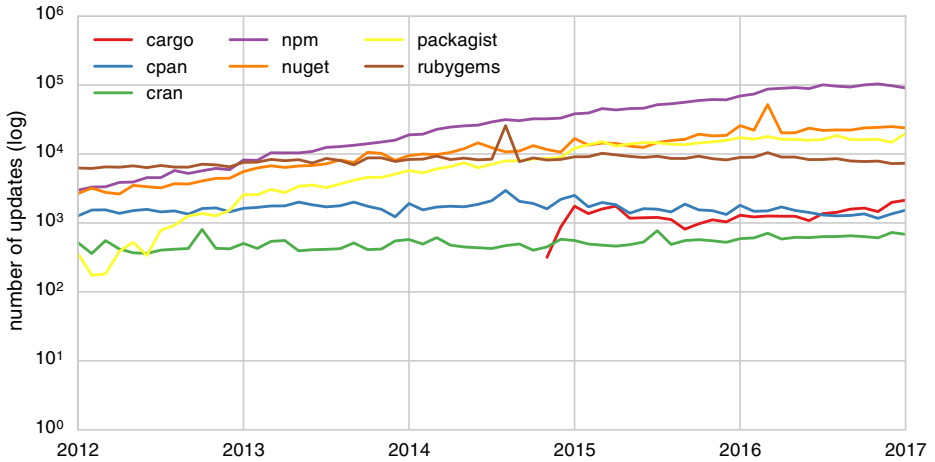

Fig. 3 Evolution of number of package updates by month (using a logarithmic y-axis)

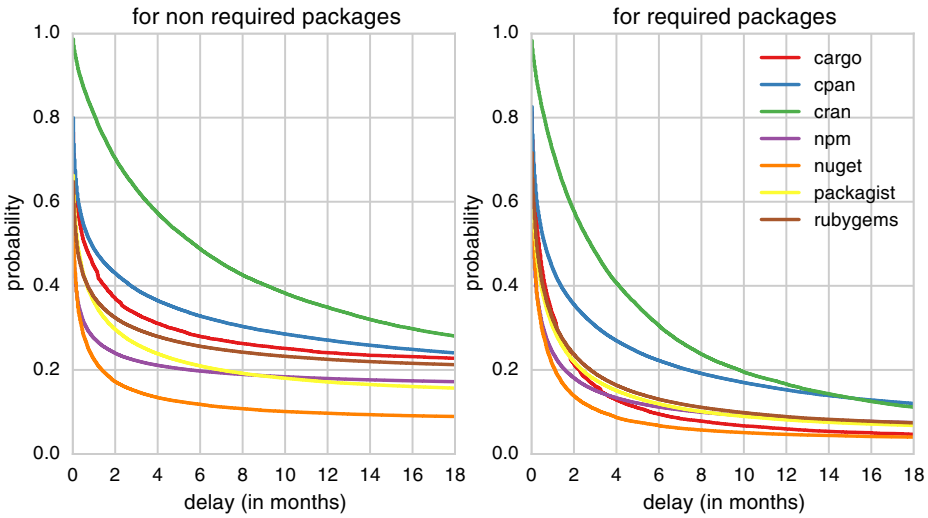

Fig. 4 Survival probability of a package release (i.e., time until a more recent release becomes available). Left plot shows packages that are not required while right plot shows those that are required

that **required packages are updated significantly more often than packages that are not required**.

While the update frequency is rather similar for all ecosystems, let us drill even further down and consider the distribution of this frequency over individual packages. Figure 5 compares the proportion of packages of each ecosystem having a given number of updates. To facilitate visual comparison, we created three distinct bags corresponding to a more or less equal proportion (about one third each) of the total number of packages of the ecosystem.

With the exception of CPAN, between 26% and 33% of all packages were never updated, between 35% and 45% of all packages were updated between 1 and 4 times, and between 27% and 36% of all packages were updated at least 5 times. The higher proportion

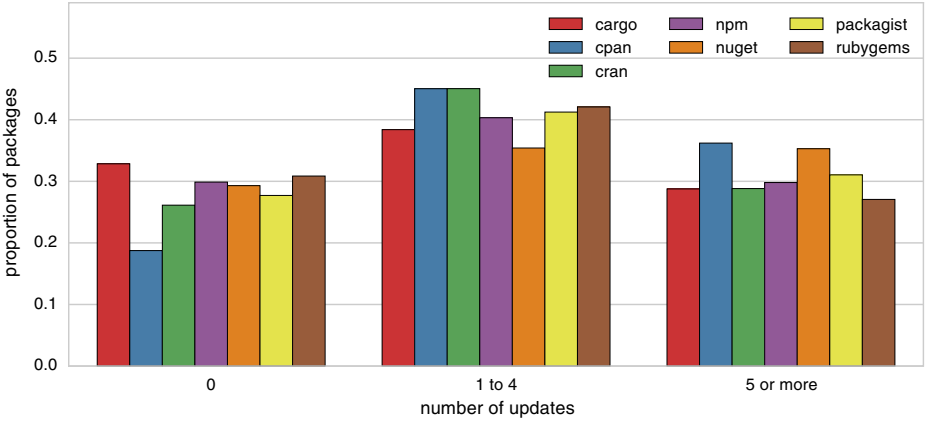

Fig. 5 Proportion of packages having a given number of updates

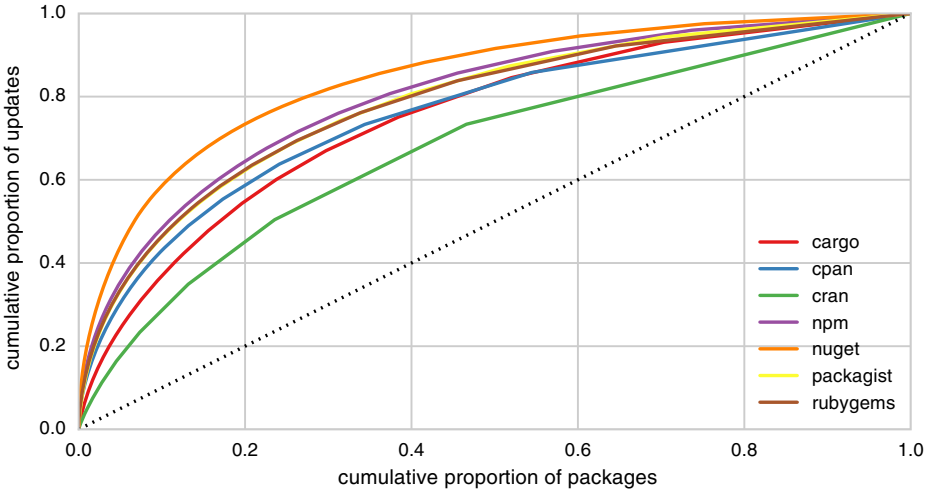

Fig. 6 Inverted Lorenz curves for the number of package updates in 2016. Only packages that are at least updated once in 2016 are considered

of updated packages for CPAN is arguably due to is age:[4] most of its packages were already available for years and, compared to the packages in the other ecosystems, had a significantly longer time to receive updates.

Figure 5 suggests that **the number of updates is not evenly distributed across packages**: regardless of the considered ecosystem, close to one third of all packages receive 5 or more updates, and close to one third of all packages receive no update at all. Figure 6 presents an (inverted) Lorenz curve that sheds more light on the extent of the inequality in the distribution of the number of updates across packages. It shows the cumulative proportion of updated packages responsible for the cumulative proportion of updates.

To limit the statistical bias induced by packages that are not updated anymore, we only considered "active" packages that were updated at least once in 2016. These active packages represent between 15.9% (for the oldest ecosystem CPAN) to 53.1% (for the newest ecosystem Cargo) of the packages. Note that, although CRAN is the second oldest ecosystem in our list, its percentage of active packages is fairly high (34.4%). This should not come as a surprise, since CRAN's rolling release policy more or less forces packages to update regularly, to avoid them becoming archived.

Based on this figure, we do observe a difference in the distribution inequality for the different ecosystems. For all ecosystems except CRAN, a minority of packages (from 27% for NuGet to 45% for Cargo) is responsible for more than 80% of all package updates. In contrast, CRAN has a more equal distribution: 60% of all packages are required to reach 80% of all package updates.

The inequality of these distributions can only partly be explained by the fact that required packages are updated more frequently (cf. Fig. 4). We therefore hypothesize that the package age (i.e., the time elapsed since its first release was introduced) also plays an important role. The intuition is that younger (and hence, less mature) packages are more subject to changes than older (and hence, more stable) ones.

[4] CPAN is twice as old as the other considered ecosystems except for CRAN.

Figure 7 presents the proportion of package updates in 2016 in terms of the age of the packages being updated. The results reflect our intuition. With the exception of CPAN and CRAN, **the majority of the updates involve packages that are up to 12 months old.** For Cargo and Packagist, respectively 56% and 50% of the updates involve packages of less than 6 months. The inequality is even more pronounced for npm, where more than 62% of the updates are done for packages of less than 3 months old.

CPAN and CRAN do not follow this rule. Indeed, the majority of package updates for them (respectively 58% and 59%) involve packages that are older than 2 years. We believe that this different change behavior is due to the fact that these two ecosystems are much older than the other ones.

Summary. We made the following observations in response to *RQ2: How frequently are packages updated?*

- The number of package updates in an ecosystem remains stable or tends to grow over time.
- Most package releases are quickly updated within few months.
- The number of package updates is distributed unequally: a minority of active packages is responsible for most of the package updates.
- Young or required packages receive package updates more often.
- Some of the observed behaviour appear to depend on the age of the ecosystem.

Given that we have observed many similarities in the change dynamics of the considered ecosystems, but also some notable differences that appear to depend on the ecosystem's age, we wish to capture in a single time-dependent metric the specific characteristics that reflect the propensity for an ecosystem to change. Such a metric must be comparable across ecosystems and must reflect both the amplitude and the importance of the considered ecosystem characteristics. Inspired by the famous Hirsch index (Hirsch 2005), which comprises in a single indicator a measure of both quantity and impact of the scientific output of a researcher (Costas and Bordons 2007), we therefore propose the following ecosystem *Changeability Index*:

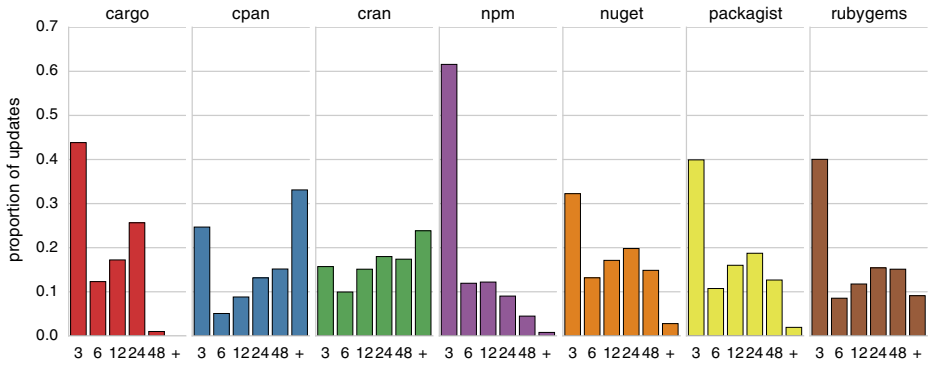

Fig. 7 Proportion of updates in 2016 by package age (in months)

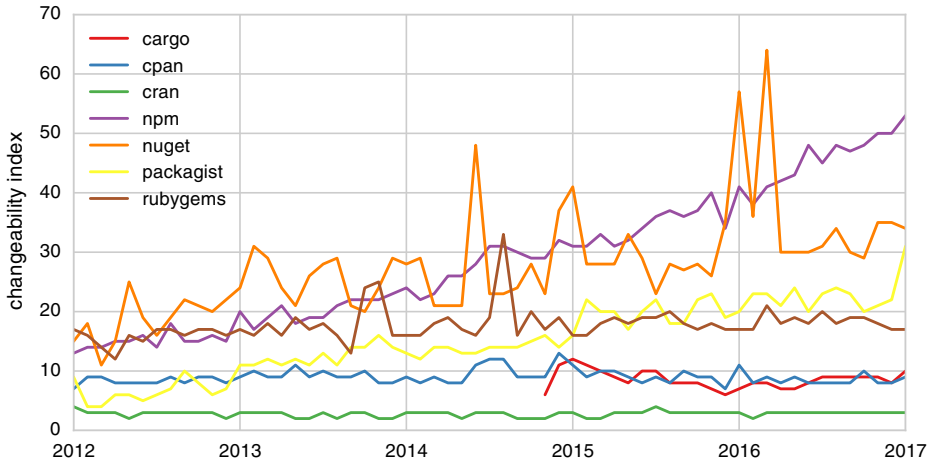

Fig. 8 Cross-ecosystem comparison of the evolution of the Changeability Index

Definition 1 The **Changeability Index** of an ecosystem E at time t is the maximal value n such that there exist n packages in E at time t having been updated at least n times during the last month.[5]

By considering the n most updated packages, this index takes into account the highly skewed distribution and the dispersion of updates we observed in Figs. 6 and 7. It therefore appears to be an appropriate measure of both the amplitude (number of packages) and the importance (number of package updates) of the propensity for an ecosystem to change. An important feature of this index is that it is largely independent of the ecosystem's size (expressed in number of packages). This makes it easy to compare the evolution of the index between ecosystems of varying sizes (cf. Table 2).

Figure 8 shows the evolution of the Changeability Index. Unsurprisingly, we find a low and constant value for CRAN and CPAN, by far the two oldest ecosystems. Cargo, CPAN and RubyGems also seem to have a more or less constant Changeability Index over time. npm appears to be the most "volatile" ecosystem, reflected by the highest and fastest growing Changeability Index. NuGet also features a high and increasing Changeability Index. For NuGet, we also observe some important peaks in June 2014 and early 2016, corresponding to a significant number of small, automatic and synchronized updates in a large number of packages related to the TypeScript.DefinitelyTyped project. These updates coincide with important releases of the TypeScript language of the Microsoft .NET platform.

6 *RQ*$_3$: To which extent do packages depend on other packages?

One of the main reasons why packages depend on others is to enable software reuse, a basic principle of software engineering (Sametinger 1997). Dependencies allow packages to use

[5]Because the choice of one month period may seem arbitrary, we also computed this index for several other periods, and did not observe different behaviors.

the functionality offered by other packages (e.g., libraries), avoiding the need to reimplement the same functionality. Packaging ecosystems make it easier for developers to reuse code from other packages, by offering automated tools to manage multiple packages and their dependencies. On the other hand, dependencies increase the risk of having important maintainability issues and failures (Bavota et al. 2015; Bogart et al. 2016). These failures can be caused by different events: 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.

Package maintainers share this concern. An Eclipse developer mentioned "*I only depend on things that are really worthwhile. Because basically everything that you depend on is going to give you pain every so often. And that's inevitable*" (Bogart et al. 2016). A CRAN developer stated "*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 [...]*" (Mens 2015). In earlier work, we observed that more than 40% of the failures observed in CRAN packages were caused by incompatible changes in required packages (Decan et al. 2016).

Not all packages make use of dependencies in a similar way. Figure 9 shows the evolution over time of the proportion of connected packages, i.e., packages that are either dependent or required. Regardless of the ecosystem, we observe that **a large majority of the packages are connected** (from 62% for NuGet to 79% for CRAN in January 2017).

Interestingly, smaller ecosystems (Cargo, CPAN, CRAN and Packagist) exhibit a behavior that is different from the larger ecosystems (npm, NuGet and RubyGems). **Smaller ecosystems tend to have a higher proportion of connected packages, with an increasing trend over time.**

To verify if the connectedness of packages is spread over the entire ecosystem, we computed the largest *weakly connected component* for the latest snapshot of each dependency network. A weakly connected component of a directed graph is a subgraph in which each

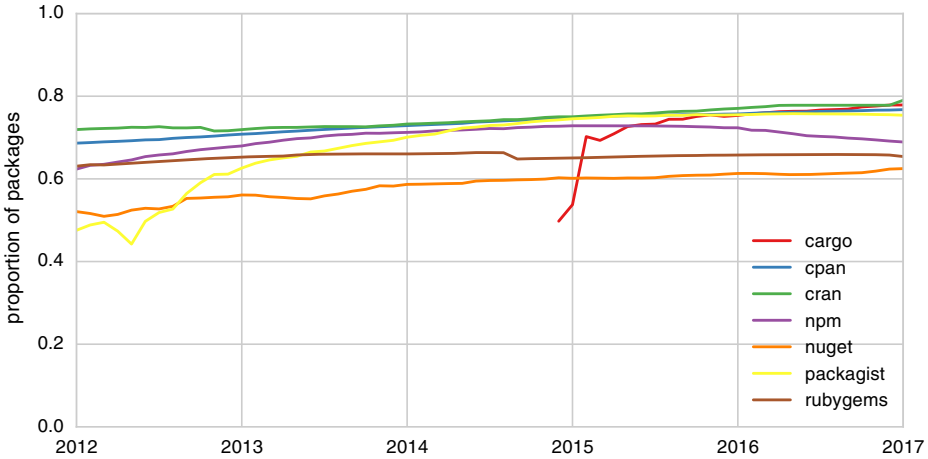

Fig. 9 Proportion of connected packages

vertex is connected to every other vertex by an undirected edge path. We found that the overwhelming majority of connected packages (from 89% for NuGet to 99% for CPAN) are part of this component.

Given that a package can be connected either because it has dependents or because it requires packages (or both), we computed the proportion of dependent and required packages for each ecosystem. Their evolution is presented in Fig. 10, and reveals that most packages are connected because they depend on other packages. We observe that **a majority of packages depends on a small minority of other packages** and that **the proportion of dependent packages increases over time while the proportion of required packages remains quite stable**.

The fact that the behavior of Cargo deviates from the other ecosystems in the beginning of Cargo's lifetime (end of 2014 – early 2015) is very likely due to the fact that a larger proportion of packages was created to form the foundations or "building blocks" of the ecosystem on which future packages can rely. To a lesser extent, a similar behavior can be observed for the Packagist ecosystem, that was created in 2012.

Not all required packages are equally required in terms of number of dependents. Figure 11 shows an (inverted) Lorenz curve that represents the inequality among required packages, i.e., the cumulative proportion of reverse dependencies in function of the cumulative proportion of required packages. We observe that **a very small proportion of required packages concentrates a very high proportion of reverse dependencies**. For instance, from only 5% (for npm) to 17% (for NuGet) of required packages concentrate more than 80% of all reverse dependencies.

To study how this unequal distribution changes over time, we computed the corresponding Gini inequality index for each month during the last five years. As the considered population of packages differ in size, and to allow comparisons across ecosystems, we normalized the Gini index by dividing it by $1 - \frac{1}{n}$. The results are shown in Fig. 12. We observe that the inequality index is similar and continuously increases for all ecosystems. On 1 January 2017, it ranges from 0.77 (for NuGet) to 0.87 (for npm), indicating a very unequal distribution of the number of dependent packages in all ecosystems.

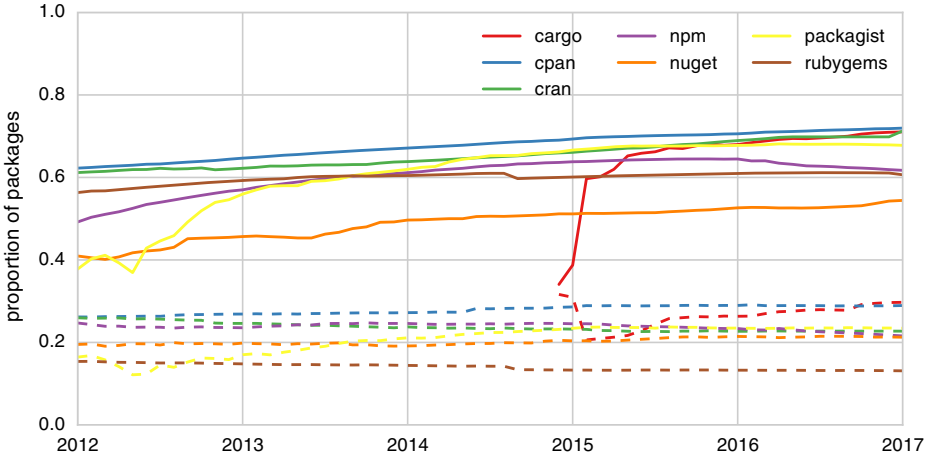

Fig. 10 Proportion of dependent (straight lines) and required (dashed lines) packages

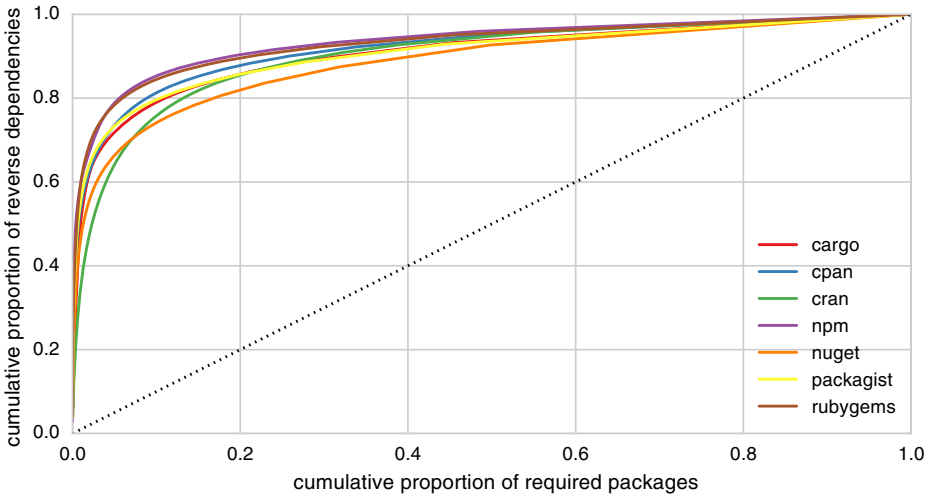

Fig. 11 Inverted Lorenz curves for the number of reverse dependencies for required packages on 1 January 2017

Summary. We made the following observations in response to RQ_3: *To which extent do packages depend on other packages?*

- Dependencies are abundant in all packaging ecosystems.
- Most packages are connected, mainly because they depend on other packages, and the proportion of connected packages increases over time.
- Dependencies are not evenly spread across packages: less than 30% of the packages are required by other packages, and less than 17% of all required packages concentrate more than 80% of all reverse dependencies. This unequal distribution of dependent packages increases over time.

Similarly to how we characterized an ecosystem's propensity to change by means of a Changeability Index, we define an ecosystem's *Reusability Index*. It comprises in a single indicator a measure of both the amplitude (number of required packages) and the extent (their number of dependent packages) of reuse. By considering the n most required packages, this index takes into account the important inequality we observed in Fig. 12.

Definition 2 The **Reusability Index** of an ecosystem E at time t is the maximal value n such that there exist n required packages in E at time t having at least n dependent packages.

Figure 13 shows the evolution of the Reusability Index over time. We observe that it is increasing over time for all ecosystems, but at a different rate. We confirmed this through a regression analysis using different growth models. Both npm and NuGet exhibit an exponential increase. The other ecosystems exhibit a linear increase, with a higher regression coefficient for Packagist, Cargo and RubyGems (respectively 0.08, 0.05 and 0.05) than for the older ecosystems CPAN and CRAN (0.02 for both). The obtained R^2 values are summarized in Table 4.

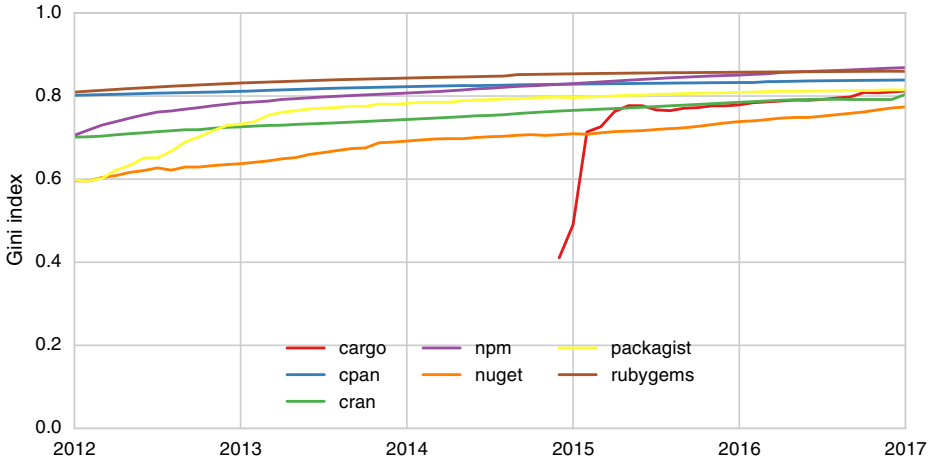

Fig. 12 Evolution of the normalized Gini index reflecting the inequality of the number of dependent packages per required package

The higher values and growth rate for npm could be explained by the relative poorness of JavaScript's standard library. Unlike the standard libraries of most other languages, the one of JavaScript is kept intentionally small for reasons explained by its creator Brendan Eich (Hemel 2010): "*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.*" The result of this is that npm contains a large and increasing number of packages that provide basic functionality on which many other packages depend.

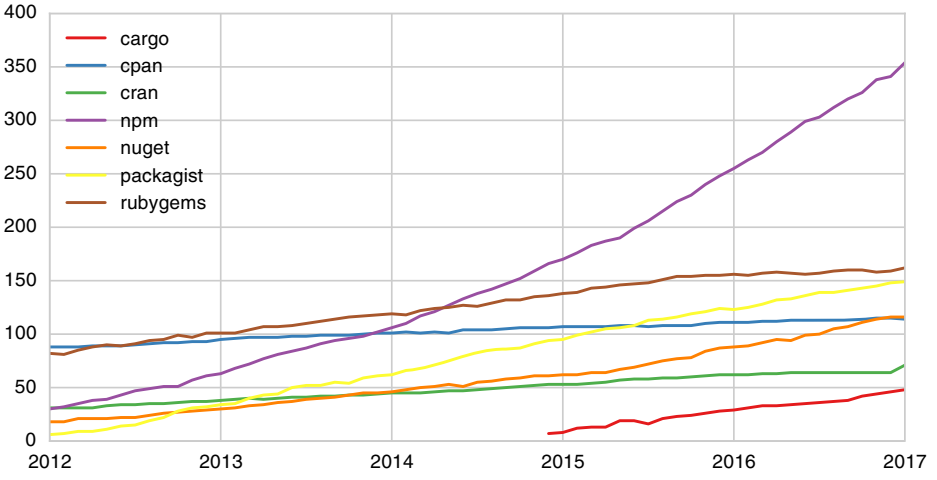

Fig. 13 Cross-ecosystem comparison of the evolution of the Reusability Index

Table 4 R^2-values of regression analysis on the evolution of the Reusability Index

	Cargo	CPAN	CRAN	npm	NuGet	Packagist	RubyGems
linear	**0.98**	**0.98**	**0.99**	0.97	0.97	**1.00**	**0.98**
exponential	0.90	0.97	0.98	**0.98**	**0.99**	0.85	0.97

Bold items correspond to the highest values

7 RQ_4: How prevalent are transitive dependencies?

While RQ_3 focused on the presence of *direct* dependencies between packages, RQ_4 focuses on the additional "*hidden*" reuse induced by *transitive* dependencies. Transitive dependencies may cause package failures to potentially affect many other packages. Such highly transitively required packages represent a potential Achilles' heel in an ecosystem: breaking or removing only one of them can impact a large proportion of the other packages in the ecosystem.

A striking 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 over five thousand transitive dependents (> 2% of all npm packages at that time), including packages whose maintainers were not even aware they depend on 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.*" (Schlueter 2016)

As another example, in November 2010, release 0.5.0 of i18n in RubyGems notably broke the popular ActiveRecord package, on which relied over 900 packages (> 5% of all packages).

To reveal the prevalence of transitive dependencies in the studied ecosystems, the boxplots in Fig. 14 show the distribution of the number of direct and transitive dependencies

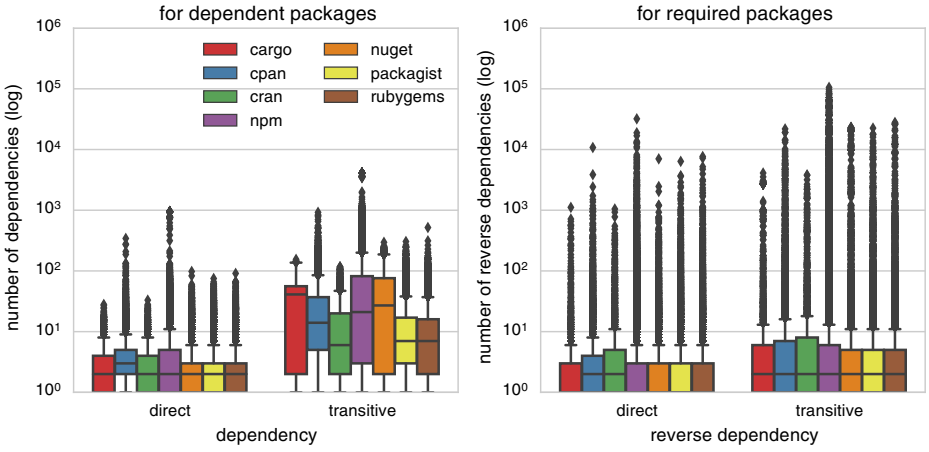

Fig. 14 Distribution of the number of dependencies for dependent packages (left) and of reverse dependencies for required packages (right), in January 2017

for dependent packages (left), and reverse direct and reverse transitive dependencies for required packages (right) for comparison. We observe that, **while a majority of the dependent packages have few direct dependencies, they have a much higher number of transitive dependencies.** For instance, half of the dependent packages in Cargo, npm and NuGet have at least 41, 21 and 27 transitive dependencies, respectively, where their median number of direct dependencies is only 2.

Figure 15 shows the evolution of the ratio between the total number of transitive dependencies and the total number of direct dependencies. For CPAN, CRAN, Packagist and RubyGems this ratio is stable over time, while it is increasing for the three other ecosystems. In January 2017, it is even from 2 to 3 times higher for Cargo, npm and NuGet than for the other ecosystems. The observed peak for CPAN in July/August 2015 is the result of a temporary change in the list of dependencies of package ExtUtils-MakeMaker. During those two months, this highly required package (with more than 16k transitive dependents) transitively relied on 11 additional packages, leading each of those transitive dependents to have 11 additional transitive dependencies.

The observed significant variations starting from early 2016 can be explained by local phenomena. For npm, the decrease of the ratio is most likely a reaction to the aforementioned left-pad incident. For Cargo, the observed increase was caused by the appearance of around 500 additional dependents for a set of strongly connected packages with many dependencies, including among others the popular clippy, quickcheck, regex, simd and serde packages. For NuGet, we identified that Newtonsoft.Json, a package with 30 transitive dependencies, gained in few months more than 1,700 (resp. 2,100) additional dependents (resp. indirect dependents).

While maintainers are usually aware of the direct dependencies of their packages because they explicitly declare them, they typically have a much less clear idea on which packages they depend indirectly, because most tools that help developers in managing dependencies do not take transitive dependencies into account, even though such transitive dependencies can be very numerous.

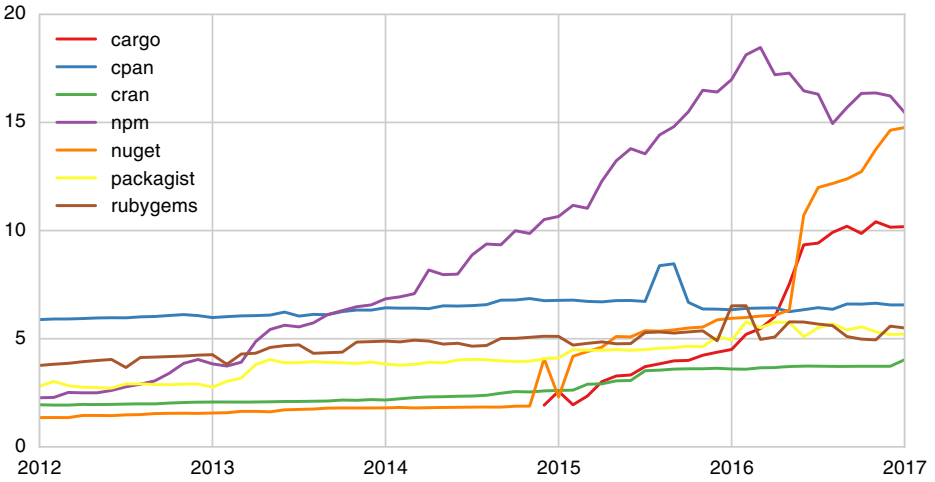

Fig. 15 Evolution of the ratio between the number of transitive dependencies and the number of direct dependencies

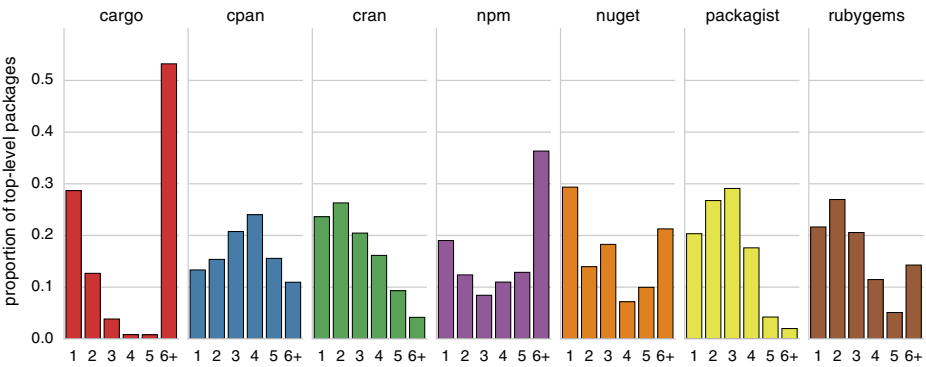

Fig. 16 Proportion of top-level packages by depth of their dependency tree, in January 2017

For example, on 1 January 2017, a package such as the popular react in npm has only 3 direct dependencies, but transitively depends on 12 additional packages. As a consequence, each of the 7,296 packages that directly depends on react implicitly requires 15 additional packages.

Not only does the number of indirect dependencies contribute to the difficulty of identifying required packages, but also because these dependencies can be deeply nested in the dependency tree. Consider co, one of the most required packages in npm. This package has 2,507 direct dependents and 51,419 indirect dependents. More than 50% of its indirect dependents require co at a depth $\geq$ 5, i.e., it is a dependency of a dependency of a dependency of an indirect dependency.

To illustrate that co is not an isolated case, we computed the depth at which transitively required packages can be found. For this purpose, we consider *top-level packages*, i.e., packages that depend on other packages but that are not required themselves. Such top-level packages hence constitute the periphery of the dependency network, and their transitive closure will cover all dependencies of all packages. Top-level packages represent between 41% and 56% of all the packages available in January 2017.

Figure 16 shows the proportion of top-level packages having a dependency tree of given depth in January 2017. **Regardless of the ecosystem, the majority of top-level packages have a deep dependency tree.** More than half of the top-level packages have a dependency tree depth of at least 3.

Some ecosystems have an even deeper nesting of dependencies. For npm, more than 50% of its top-level packages have a dependency tree depth of at least 5. We hypothesize that this is a combination of the recent surge in popularity of the ecosystem, combined with the lack of an extensive standard library, leading developers to rely on other packages even for basic features.

Similarly, more than 50% of the top-level Cargo packages have a dependency tree depth of at least 6, and 25% of the top-level Cargo packages have a dependency tree depth of at least 10. We assume that this is mainly related to the very young age of the Rust language and its Cargo package manager, leading developers to first try to develop smaller building bricks that are only a thin layer over previous ones and that can then be used by other packages to provide more "high-level" libraries such as those available in other languages.

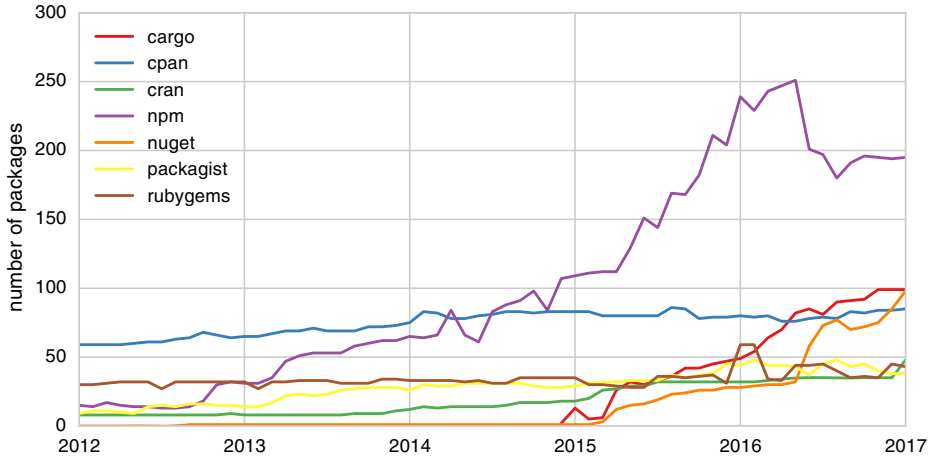

Fig. 17 Number of packages that are transitively required by at least 5% of all packages

Summary. We made the following observations in response to RQ_4: *How prevalent are transitive dependencies?*

- For each ecosystem, the majority of dependent packages have few direct dependencies but a high number of transitive dependencies.
- More than half of the top-level packages have a dependency tree of depth 3 or higher.

Given that dependencies cause package failures to propagate to its dependents, and given the prevalence of transitive dependencies in each ecosystem, we are interested in a metric that reflects the fragility of an ecosystem because of the presence of highly required packages that may impact large parts of the ecosystem. We propose the parametric *P-Impact Index*, defined as follows:

Definition 3 The **P-Impact Index** of an ecosystem E at time t is the number of packages in E at time t that are transitively required by at least P% of all the packages in E.

The P-Impact Index allows to quantify the number of packages that could have a high impact (at least P%) in the ecosystem because of their numerous transitive dependents. Figure 17 shows the evolution of the 5-Impact Index. The choice of 5% was motivated by the example of the ActiveRecord package in RubyGems on which relied $>$ 5% of all RubyGems packages at the time of the reported problem. We also computed the P-Impact Index for other values of P (e.g., for 2% corresponding to the impact of left-pad in npm) and obtained similar results.

While the 5-Impact Index of Packagist and RubyGems is nearly stable over time, it continuously increases for the other ecosystems. This increase is particularly prominent for Cargo, npm and NuGet, which also exhibit the highest values in January 2017. For npm, such a high Impact Index was expected, due to its large number of packages and the higher

depth of their dependency tree. As for Fig. 15, the variations observed from early 2016 onward are most likely a reaction to the left-pad incident.

The high impact index of Cargo is somewhat surprising given its smaller size. While this ecosystem had only 7,421 packages in January 2017, its 5-Impact Index was already of 99, which represents more than 1.3% of all its packages.

Based on the results of this impact analysis, we observe that 4 out of 7 ecosystems were able to restrain the fragility induced by a growing number of packages and their increasing reuse. The highest impact and growth was observed for npm, NuGet and Cargo, suggesting that these ecosystems should make an effort to reduce their complexity and hence their fragility.

8 Discussion

In Sections 4 to 7 we addressed the four research questions empirically, through historical analysis of the dependency networks of seven packaging ecosystems. This section complements this empirical analysis with additional information that may partly explain some of the observed differences.

In particular, Section 8.1 discusses the effect of ecosystem-specific policies on our findings, while Section 8.2 compares the automated support for package dependency updates that has been put in place by the different ecosystems, and discusses the limitations of such support. Finally, Section 8.3 discusses the usefulness of integrating some of our proposed dependency network metrics into software ecosystem health analysis dashboards.

8.1 Why policies matter

While our empirical comparison revealed many similarities across packaging ecosystems, we also observed some important differences. For instance, CRAN features the lowest Changeability Index, one of the lowest Reusability Indices, a lower ratio of transitive to direct dependencies, and one of the lowest observed dependency depths for its top-level packages. This is very likely due to the fact that CRAN imposes a stricter policy on its package maintainers than the other considered package managers.

CRAN follows a "rolling release" policy that imposes packages to be up-to-date with their dependencies (CRAN Repository Maintainers 2016). An automated continuous integration process based on the R CMD check tool verifies interpackage compatibility on a daily basis. Maintainers of packages that fail the check are asked to resolve the problem before the next major R release, and their packages get archived if they do not do so. CRAN also appears to have different evolution dynamics in many respects: despite its exponential growth, it has a lower number of monthly package updates and a corresponding higher probability of survival of package releases. CRAN also witnesses a lower inequality in the distribution of package updates, resulting in a significantly lower Changeability Index. A plausible explanation is that package maintainers are encouraged to limit the frequency of package updates to once every one or two months, in order to keep this rolling release policy manageable for package maintainers (CRAN Repository Maintainers 2016).

We did not find any evidence of the existence of such policies related to package updates or package dependencies for the other ecosystems we studied. We also do not believe that those ecosystems are willing to adopt a similar process. Indeed, it would require package maintainers to quickly react to backward incompatible changes in package dependencies, which represents a frequent and potentially heavy additional workload. This concern is

shared by CRAN package maintainers who consequently try to minimize or avoid dependencies on other packages, or even consider alternatives to CRAN for the distribution of their packages because of this (Decan et al. 2016; Bogart et al. 2015; Mens 2015).

For the other ecosystems, the main guidelines we found do not seem to relate to package updates nor package dependencies. They rather have to do with recommendations to use semantic versioning or respecting the "default semantics" of the package manager. For instance, contrary to one's intuition, NuGet automatically selects the oldest available release that satisfies the package dependency constraints.

Another policy that may play an important role is the package removal policy. Indeed, if authors are allowed to remove their packages from the ecosystem, this increases the risk of breaking (transitive) dependents upon package removal. Even if an ecosystem prevents packages from being removed, authors can still decide to update their packages to an empty release, leading to a potentially similar outcome. Some ecosystems such as Cargo or NuGet explicitly prevent packages from being removed from the ecosystem. The same is now also true for npm, who introduced this policy as a consequence of the left-pad incident. However, in May 2017 RubyGems still allowed authors to easily remove their packages. In a similar vein, in May 2017 CRAN still archived packages, implying that they cannot be automatically installed anymore, and thus preventing the installation of dependent packages as well. Hence, removal of packages can still have a high negative impact in those ecosystems.

8.2 Limitations of existing support for package dependency updates

Di Cosmo et al. (2008) claims that problems related to package updates are important, and that more automated solutions to address these problems are required. This paper empirically validated these claims, by studying problems related to package updates in presence of dependent packages and by analyzing how large popular packaging ecosystems currently (fail to) cope with these problems. While we have discussed in Section 8.1 how some packaging ecosystems rely on ecosystem-specific policies, let us now focus on technical solutions provided by each ecosystem to cope with package updates in presence of dependencies.

To avoid packages from breaking due to a dependency update, most ecosystems allow package maintainers to specify *dependency constraints* on the versions of the packages they depend upon. An empirical analysis of the use of dependency constraints in CRAN, npm and RubyGems was presented in Decan et al. (2017). Dependency 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. While the use of dependency constraints can be beneficial to prevent backward incompatibility issues, it may as a side-effect prevent packages to benefit from updates that are released in a dependency. This can be problematic, especially if the updates contain bug fixes (Cox et al. 2015) or address security vulnerabilities. For this reason, since October 2017 GitHub monitors the dependencies of Ruby and JavaScript projects and triggers alerts when vulnerabilities are detected.

Combining the use of dependency constraints with *semantic versioning* can enable packages to benefit from compatible updates while preventing backward incompatible ones. 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 corresponding 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, while it is easy to impose a semantic versioning syntax (as is the case for Cargo, npm and Packagist), package maintainers can always decide, voluntarily or not, to break the associated versioning semantics (Raemaekers et al. 2014).

Package maintainers can be assisted in managing their dependency updates by automated 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 free to use 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 dependency hierarchy.

While it would be very desirable for these tools to take into account transitive dependencies as well, implementing such support is potentially very computationally expensive, especially in the presence of dependency constraints. Indeed, it is not unusual that several distinct releases of a same package satisfy the dependency constraints imposed by a dependent package. These releases can potentially have different lists of required packages or dependency constraints which, in turn, can potentially be satisfied by different releases, and so on, leading to an increasingly large number of potential dependency trees. Moreover, because of transitive dependencies, a same package can be the target of different sets of dependency constraints. Identifying all the releases that satisfy these sets of constraints is a complex problem. Additionally, all considered package management systems implicitly define a conflict between any two distinct releases of a same package. This means that one cannot install two different releases of a same package, or in some cases (CPAN, CRAN, NuGet and RubyGems), even two packages that transitively depend on two distinct releases of a same package. While solutions to this problem were developed specifically for some ecosystems (e.g., Debian or RPM, see Di Cosmo and Vouillon (2011) and Vouillon and Di Cosmo (2013)), they are usually based on a SAT-solver, are not easy to implement and are potentially computationally expensive to use.

To summarize, many techniques have been proposed and are being used in different combinations in each ecosystem to facilitate the work of package maintainers in presence of dependency updates. Given the fact that each technique has specific drawbacks, a perfect solution does not exist. Moreover, in addition to a good package management policy and a proper combination of the aforementioned techniques, it is essential for all package maintainers to be disciplined and act responsibly. They should limit updates to their packages, communicate with maintainers of dependent packages, limit the number of dependencies to other packages, advertise backward incompatible changes and deprecation warnings, respect the imposed policies and versioning schemes, use appropriate continuous integration and monitoring tools, and deploy bug and security fixes not only in the latest release but also in older branches.

8.3 Toward ecosystem-level health analysis dashboards

Several dashboards for open source software development analytics are emerging. One of those is GrimoireLab,[6] an open source software analytics engine commercialized by the

[6] http://grimoirelab.github.io

Spanish company Bitergia.[7] Through private e-mail communication we discussed with two of Bitergia's team members about the usefulness and relevance to extend their tool suite with ecosystem-wide analyzes such as those proposed in this article. They confirmed that there is indeed a need for analytics at the ecosystem level: "*[...] what we were producing was initially focused on a project and now we need to understand and provide insights about a huge amount of projects that in the end are part of an ecosystem.*" More in particular, they agreed that there is a need for metrics that measure the health of the ecosystem, such as the ones we proposed based on the technical dependencies between software packages: "*There is a lot of interest by companies in learning about the 'health' of FOSS components, and that implies learning about the components of their dependencies, and of their 'siblings'. In other words, they know that the health of a single component depends on the health of the ecosystem in which it is produced and used. From the point of view of people producing software, they want to track everything around them. Just as a single example, they need to know if modules on which their software depend are healthy or not. From the point of view of users, that happens as well. For example, to understand the security problems of a product, they need to understand the security problems of all its dependencies, and in many cases, of their siblings developed by the same community.*"

We also discussed over e-mail with the developers of dependencyci.com, a continuous integration tool for monitoring package dependencies. In particular, we asked them to share their view on the importance of package dependency networks and the potential problems caused by transitive dependencies, the focal point of our empirical analysis. They agreed that dependency-related problems tend to propagate over the dependency network:

- "*Whenever a measure has an upward or downward impact on its own dependencies an update in one project can cause a network-update effect that can make the whole network very noisy until it settles. Interestingly there is a direct correlation with the dependency update problem in open source that follows the same pattern.*"
- "*[...] there may be a force multiplier/dampening effect up and down the tree. Relying on a project that only has one contributor, but that project is very simple and has few dependencies itself, might be acceptable. But depending upon a project that has hundreds of dependencies or security vulnerabilities and only one contributor is most likely going to cause trouble.*"

They also stressed the importance of transitive dependencies for two problems that were not specifically part of our empirical analysis, namely license compatibility and security breaches: "*transitive dependencies are incredibly useful when looking at things like license incompatibilities. Especially when a project's more permissive license impacts upon any of the software built upon it. Which can have direct impact up the tree. It's also useful for security notifications, some bugs will have impact on all users, regardless of where in the dependency tree the problem is.*"

The above discussions comfort our conviction that it is useful and relevant to integrate ecosystem-level measurements of dependency network evolution (inspired by those presented in the current article) into existing software health analysis dashboards. However, as will be discussed in Section 10.3, the technical aspects of package dependencies and updates should be complemented with social aspects of developer interaction in order to come to a holistic socio-technical health analysis.

[7] https://bitergia.com

9 Threats to validity

The metadata for all studied ecosystems was automatically gathered from libraries.io, with the exception of CRAN where we extracted the data directly from package metadata using the extractoR tool.[8] Because there is no full guarantee that these tools produce correct results, we manually verified the correctness of the gathered data, and we cross-checked with other available metadata based on our previous research (Decan et al. 2017), thereby reducing this threat to a minimum.

The package (release) data we used was up-to-date up to April 2017. Depending on the ecosystem's package removal policy, packages that were removed from the ecosystem before that date may have been absent from our analysis if no historical data was preserved by the ecosystem after package removal.

We constructed the dependency networks by relying on the list of dependencies explicitly provided in each package manifest. As a consequence, vendored dependencies and dynamically defined dependencies were not considered in our analyses. Since our collected dependencies underestimate actual reuse, we believe that this threat does not affect our results.

Some of our analyses are based on monthly snapshots of dependency networks, and we relied on the chronological order of package releases to build them. While this chronological order should match the logical order induced by the versioning scheme in most cases, this is not the case for instance for packages having multiple branches that are maintained in parallel. This is, however, unlikely to affect our findings given the scale of our analyzes.

Some analyses may be affected by local phenomena (see Fig. 8 or Fig. 15 for instance). As far as possible, we tried to pinpoint and interpret these phenomena. While some of these phenomena are perfectly legitimate, others are explained by a quality problem in the extracted data. For instance, the peak in the number of updates in August 2014 for RubyGems (Fig. 3) corresponds to the massive import of 25 K package releases in RubyGems, resulting in an incorrect creation date for those package releases, which does not reflect the real date of their availability to the Ruby world.

We do not make any claims that our results can be generalized beyond package dependency networks similar to those that we have analyzed, i.e., the main package managers for specific programming languages. While the analyses that we have carried out can easily be applied to any other type of package dependency network such as WordPress, Eclipse or Atom, we do not expect to obtain similar results, because their packages tend to be more high-level, intended to be installed by end-users rather than be reused (through dependencies) by other packages.

10 Future work

Based on the empirical analysis that we carried out and its ensuing discussion, this section presents a number of interesting avenues of future work. Section 10.1 postulates some laws of software ecosystem evolution that could be derived from our analysis. Section 10.2 proposes to study software ecosystems and their evolution from a complex networks perspective. Finally, Section 10.3 considers to extend the technical dependency analysis with a social counterpart, by also studying the ecosystem's community of contributors.

[8] https://github.com/ecos-umons/extractoR

10.1 Laws of software ecosystem evolution

Lehman's famous laws of software evolution reflect established empirical observations of how individual software systems tend to evolve over time (Lehman et al. 1997). Based on the empirical findings in this article, we hypothesize that similar laws govern the ecosystem-level evolution of package dependency networks. Arguably the most popular laws of software evolution are the ones of *Continuing Growth*, *Continuing Change* and *Increasing Complexity*.

If we agree to measure the size of an ecosystem's dependency network in terms of number of packages or number of dependencies, then we can claim to have found initial empirical evidence for the law of *Continuing Growth* at ecosystem level, as a side-effect of answering RQ_1.

We also found partial empirical evidence for the law of *Continuing Change* at ecosystem level, as a side-effect of our results for RQ_2 where we studied the frequency of package updates, and found that the number of package updates remains stable or tends to grow over time for the studied ecosystems. Similarly, our proposed Changeability Index was increasing over time for most of the considered ecosystems.

We also found partial support for the law of *Increasing Complexity*, if we assume that the ratio of the number of dependencies over the number of packages is an indicator of a dependency network's complexity. Another possible indicator of complexity is the ratio of transitive over direct dependencies, which was found to grow over time for all studied ecosystems (cf. Section 7). The P-Impact Index also provided evidence of an increasing fragility of the considered ecosystems.

These three laws focus on structural and technical aspects of software. Lehman has postulated other laws as well, primarily concerning the organizational and social aspects of evolving software. Since these aspects have not been part of our current empirical study, we cannot provide any initial evidence for them. It therefore remains an open topic of future work to study to which extent Lehman's laws also hold at the level of packaging ecosystems, and whether other laws may also hold at this level.

10.2 Complex networks

Complex networks are networks or graphs that contain emerging structural properties that typically do not occur in simple network structures such as lattices or random graphs (Barabási 2016). The networks of many real-world systems (e.g. the brain, social networks and computer networks) have been shown to reveal complex network properties, such as scale-freeness, the small world phenomenon, and power law behavior. Earlier work has revealed such complex network characteristics for class dependency graphs of individual open source software systems (e.g., Myers (2003) and Zheng et al. (2008)). Inspired by Cataldo et al. (2014), we hypothesize that package dependency networks of open source packaging ecosystems also reveal such complex network behavior.

For example, we found a very *unequal distribution of connectivity* for each ecosystem, characteristic of power law or Pareto law behavior (Goeminne and Mens 2011). First of all, the proportion of required packages (Fig. 10) was invariably low for each ecosystem (ranging between 20% and 30%, and even lower for RubyGems). Secondly, a very low proportion of these required packages concentrated a very high proportion of reverse dependencies (Figs. 11 and 12). At the other side of the spectrum we found a fairly high proportion (ranging between 40% and 60%) of top-level packages (i.e., connected packages that are not required by other packages) in all ecosystems. Moreover, the majority of these top-level

packages had dependency trees of depth three or higher. We also observed a rather *unequal distribution of package updates* for each ecosystem, since a major proportion of package updates was concentrated in a minority of packages (Fig. 6).

These initial findings make us confident that it would be worthwhile to study, compare and exploit the complex network properties of ecosystem package dependency networks as part of future work.

10.3 Socio-technical ecosystem and community health analysis

In the current article we have only focused on technical dependencies between packages belonging to the same ecosystem. As explained in Mens (2016), it would be very useful to study the ecosystem dynamics from a socio-technical point of view, combining information from the package dependency network with information from the social network of ecosystem contributors.

Socio-technical networks have been used frequently at the level of individual software projects, for example to predict software failures (Bird et al. 2009; Posnett et al. 2013), to predict project or contributor abandonment (Constantinou and Mens 2017), to measure successful builds (Kwan et al. 2011) and many more. We are not aware, however, of any attempt to study, exploit and compare the evolution of socio-technical networks across multiple software ecosystems.

An interesting way to turn such socio-technical analysis into actionable results consists in focusing on software ecosystem and software community health aspects, by analyzing and predicting social or technical events that may be detrimental to the health (e.g. quality, survival, sustainability, diversity) of the package dependency network or the social network of package contributors. Indeed, there appears to be a general drive in the open source community to measure the health of open source communities and the software ecosystems they maintain. As an illustration of this, in September 2017, the Linux Foundation officially announced the CHAOSS project for Community Health Analytics of Open Source Software.[9] As part of this larger initiative, our own interuniversity SECOHealth project[10] will focus on understanding and assisting the health dynamics of software ecosystems.

11 Conclusion

We carried out an empirical comparison of the package dependency networks of seven packaging ecosystems, each associated to a different programming language, namely Cargo, CPAN, CRAN, npm, NuGet, Packagist and RubyGems. The range of considered ecosystems varied in size and age. Some ecosystems were very large (e.g., npm has over 3 million package releases), while others were very old (e.g. CPAN has a release history of more than 20 years).

The presented research is the first to compare that many different ecosystems. Previous research was limited to individual ecosystems, or at best only two or three ecosystems were compared. In addition, the presented research is the first to use the libraries.io dataset containing metadata of software package dependencies of several millions of open source libraries stored in dozens of different package managers.

[9] https://chaoss.community

[10] https://www.secohealth.org (October 2017 - September 2019

Our research questions focused on the growth, changeability, reusability and fragility of the considered package dependency networks. By studying the size of package dependency networks over time in terms of their number of packages and dependencies, we observed that these networks tend to grow over time, though the speed of growth may differ. Taking the ratio of dependencies over packages as a simple measure of the network's complexity, we observed that this complexity either remains stable or increases over time.

Considering the fact that package dependencies frequently cause problems when required packages are updated (Decan et al. 2017), we studied the changeability and reuse of packages over time, based on the number of package updates and package dependencies, respectively. In general, package releases are updated within few months. While most packages receive updates, only a small proportion of packages concentrates most of the updates. Reverse dependencies exhibit a similar unequal distribution: while dependencies are abundant and while most packages are connected, only a small proportion of packages accounts for a large majority of all reverse dependencies. Moreover, these inequalities tend to increase over time. We also proposed novel metrics to facilitate cross-ecosystem comparison. These indicators, inspired by the Hirsch index, combine the amplitude and the importance of change and reuse in an ecosystem.

Direct reverse dependencies represent only the tip of the iceberg. Many packages in the analyzed package dependency networks were found to have a high number of transitive reverse dependencies, implying that package failures can affect a large number of other packages in the ecosystem. We therefore defined an indicator quantifying the fragility of an ecosystem based on the number of packages having a high potential transitive impact, and observed an increase for most of the considered ecosystems.

The proposed indicators provide initial evidence for some of the laws of software evolution at the ecosystem level: continuing growth, continuing change and increasing complexity. The indicators also reveal differences across the considered ecosystems. These differences highlight ecosystem particularities such as specific policies (e.g., the "rolling release" policy of CRAN induces a lower Changeability Index), the age of the ecosystem (e.g., lower Changeability Index for the old CPAN ecosystem and higher Impact Index for the young Cargo ecosystem), or the incompleteness of the standard library of the programming language (leading to a higher Reusability Index for npm).

Our research findings are instrumental for assessing the quality of package dependency networks, and supporting it through proper dependency management tools, better policies, and ecosystem health analysis dashboards.

Acknowledgements This research was carried out in the context of FRQ-FNRS collaborative research project R.60.04.18.F "SECOHealth", ARC research project AUWB-12/17-UMONS-3 "Ecological Studies of Open Source Software Ecosystems", and FNRS Research Credit J.0023.16 "Analysis of Software Project Survival". We express our gratitude to Andrew Nesbitt and Ben Nickolls, both from libaries.io and dependencyci.com, for making the package manager dependency data available, and for the very useful email discussions. We thank Jesus Gonzalez-Barahona and Daniel Izquierdo from Bitergia for their relevant feedback. We thank Eleni Constantinou, Alexander Serebrenik and Damian Tamburri for proofreading this work.

References

Aalen O, Borgan O, Gjessing H (2008) Survival and event history analysis: a process point of view springer. https://doi.org/10.1007/978-0-387-68560-1

Abdalkareem R, Nourry O, Wehaibi S, Mujahid S, Shihab E (2017) Why do developers use trivial packages? an empirical case study on npm. In: Joint Meeting on Foundations of Software Engineering (ESEC/FSE), pp 385–395. https://doi.org/10.1145/3106237.3106267

Artho C, Suzaki K, Di Cosmo K, Treinen R, Zacchiroli RS (2012) Why do software packages conflict? In: Int'l conference mining software repositories, pp 141–150. https://doi.org/10.1109/MSR.2012.6224274

Barabási AL (2016) Network science. Cambridge University Press, Cambridge

Bavota G, Canfora G, Di Penta M, Oliveto R, Panichella S (2015) How the apache community upgrades dependencies: an evolutionary study. Empir Softw Eng 20(5):1275–1317. https://doi.org/10.1007/s10664-014-9325-9

Bird C, Nagappan N, Gall H, Murphy B, Devanbu P (2009) Putting it all together: using socio-technical networks to predict failures. In: Int'l symposium software reliability engineering. IEEE Computer Society, pp 109–119. https://doi.org/10.1109/ISSRE.2009.17

Blincoe K, Harrison F, Damian D (2015) Ecosystems in GitHub and a method for ecosystem identification using reference coupling. In: Int'l conference mining software repositories. IEEE, pp 202–211. https://doi.org/10.1109/MSR.2015.26

Bogart C, Kästner C, Herbsleb J (2015) When it breaks, it breaks: how ecosystem developers reason about the stability of dependencies. In: Automated software engineering workshop, pp 86–89. https://doi.org/10.1109/ASEW.2015.21

Bogart C, Kästner C, Herbsleb J, Thung F (2016) How to break an API: cost negotiation and community values in three software ecosystems. In: Int'l symposium foundations of software engineering. https://doi.org/10.1145/2950290.2950325

Cadariu M, Bouwers E, Visser J, van Deursen A (2015) Tracking known security vulnerabilities in proprietary software systems. In: Int'l conference software analysis, evolution, and reengineering, pp 516–519. https://doi.org/10.1109/SANER.2015.7081868

Cataldo M, Scholtes I, Valetto G (2014) A complex networks perspective on collaborative software engineering. Advances in Complex Systems 17(7-8). https://doi.org/10.1142/S0219525914300011

Claes M, Mens T, Grosjean P (2014) On the maintainability of CRAN packages. In: Int'l conference software maintenance, reengineering, and reverse engineering. IEEE, pp 308–312. https://doi.org/10.1109/CSMR-WCRE.2014.6747183

Constantinou E, Mens T (2017) Socio-technical evolution of the Ruby ecosystem in GitHub. In: Int'l Conference Software Analysis, Evolution and Reengineering (SANER), pp 34–44. https://doi.org/10.1109/SANER.2017.7884607

Costas R, Bordons M (2007) The h-index: advantages, limitations and its relation with other bibliometric indicators at the micro level. J Informetrics 1(3):193–203. https://doi.org/10.1016/j.joi.2007.02.001

Cox J, Bouwers E, van Eekelen M, Visser J (2015) Measuring dependency freshness in software systems. In: Int'l conference software engineering. IEEE Press, pp 109–118

CRAN Repository Maintainers (2016) CRAN repository policy. https://cran.r-project.org/web/packages/policies.html

Decan A, Mens T (2017) Replication package for An Empirical Comparison of Dependency Network Evolution in Seven Software Packaging Ecosystems. https://doi.org/10.5281/zenodo.1109685

Decan A, Mens T, Claes M, Grosjean P (2015) On the development and distribution of R packages: an empirical analysis of the R ecosystem. In: European conference software architecture workshops, pp 41:1–41:6. https://doi.org/10.1145/2797433.2797476

Decan A, Mens T, Claes M (2016) On the topology of package dependency networks — a comparison of three programming language ecosystems. In: European conference software architecture workshops. ACM. https://doi.org/10.1145/2993412.3003382

Decan A, Mens T, Claes M, Grosjean P (2016) When GitHub meets CRAN: an analysis of inter-repository package dependency problems. In: Int'l conference software analysis, evolution, and reengineering. IEEE, pp 493–504. https://doi.org/10.1109/SANER.2016.12

Decan A, Goeminne M, Mens T (2017) On the interaction of relational database access technologies in open source java projects. In: Bagge A, Mens T, Osman H (eds) Post-proceedings of the 8th Seminar on Advanced Techniques and Tools for Software Evolution, vol 1820, pp 26–35. CEUR-WS.org

Decan A, Mens T, Claes M (2017) An empirical comparison of dependency issues in OSS packaging ecosystems. In: Int'l conference software analysis, evolution, and reengineering, pp 2–12. https://doi.org/10.1109/SANER.2017.7884604

Di Cosmo R, Vouillon J (2011) On software component co-installability. In: Joint european conference software engineering / foundations of software engineering. ACM, pp 256–266. https://doi.org/10.1145/2025113.2025149

Di Cosmo R, Zacchiroli S, Trezentos P (2008) Package upgrades in FOSS distributions: Details and challenges. In: 1St int'l workshop on hot topics in software upgrades. ACM, New York. https://doi.org/10.1145/1490283.1490292

Dietrich J, Yakovlev V, McCartin C, Jenson G, Duchrow M (2008) Cluster analysis of Java dependency graphs. In: Symposium software visualization. ACM, pp 91–94. https://doi.org/10.1145/1409720.1409735

Germán DM, Adams B, Hassan AE (2013) The evolution of the R software ecosystem. In: European conference software maintenance and reengineering, pp 243–252. https://doi.org/10.1109/CSMR.2013.33

Giger E, Pinzger M, Gall H (2011) Using the Gini coefficient for bug prediction in eclipse. In: Int'l workshop on principles of software evolution. ACM, pp 51–55. https://doi.org/10.1145/2024445.2024455

Gini C (1912) Variabilità e mutabilità. Memorie di metodologica statistica

Goeminne M, Mens T (2011) Evidence for the Pareto principle in open source software activity. In: Workshop on Software Quality and Maintainability (SQM), CEUR workshop proceedings, vol 701, pp 74–82. CEUR-WS.org

Goeminne M, Mens T (2015) Towards a survival analysis of database framework usage in Java projects. In: Int'l conference software maintenance and evolution. https://doi.org/10.1109/ICSM.2015.7332512

González-Barahona JM, Robles G, Michlmayr M, Amor JJ, Germán DM (2009) Macro-level software evolution: a case study of a large software compilation. Empir Softw Eng 14(3):262–285. https://doi.org/10.1007/s10664-008-9100-x

Haney D (2016) NPM & left-pad: Have we forgotten how to program? http://www.haneycodes.net/npm-left-pad-have-we-forgotten-how-to-program/

Hemel Z (2010) 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

Hirsch JE (2005) An index to quantify an individual's scientific research output. Proc Natl Acad Sci USA 102(46):16,569–16,572. http://www.jstor.org/stable/4152261

Hornik K (2012) Are there too many R packages? Austrian J Stat 41(1):59–66

Jansen S, Cusumano M, Brinkkemper S (eds.) (2013) Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry. Edward Elgar

Kaplan EL, Meier P (2012) Nonparametric estimation from incomplete observations. J Am Stat Assoc 53(282):457–481

Kikas R, Gousios G, Dumas M, Pfahl D (2017) Structure and evolution of package dependency networks. In: Int'l Conference Mining Software Repositories (MSR), pp 102–112. https://doi.org/10.1109/MSR.2017.55

Kwan I, Schroter A, Damian D (2011) Does socio-technical congruence have an effect on software build success? a study of coordination in a software project. IEEE Trans Soft Eng 37(3):307–324. https://doi.org/10.1109/TSE.2011.29

Kyriakakis P, Chatzigeorgiou A (2014) Maintenance patterns of large-scale PHP web applications. In: Int'l conference software maintenance and evolution, pp 381–390. https://doi.org/10.1109/ICSME.2014.60

Lehman MM, Fernandez Ramil J, Wernick PD, Perry DE, Turski WM (1997) Metrics and laws of software evolution – the nineties view. In: Int'l symposium software metrics. IEEE Computer Society, pp 20–32. https://doi.org/10.1109/METRIC.1997.637156

Lin B, Robles G, Serebrenik A (2017) Developer turnover in global, industrial open source projects: insights from applying survival analysis. In: Int'l Conference Global Software Engineering (ICGSE). https://doi.org/10.1109/ICGSE.2017.11

Lorenz MO (1905) Methods of measuring the concentration of wealth. Publ Am Stat Assoc 9(70):209–219. https://doi.org/10.1080/15225437.1905.10503443

Manikas K, Hansen KM (2013) Software ecosystems: a systematic literature review. J Syst Softw 86(5):1294–1306. https://doi.org/10.1016/j.jss.2012.12.026

Mens T (2015) Anonymized e-mail interviews with R package maintainers active on CRAN and GitHub. Tech. rep., University of Mons. arXiv:1606.05431

Mens T (2016) An ecosystemic and socio-technical view on software maintenance and evolution. In: Int'l conference software maintenance and evolution. IEEE. https://doi.org/10.1109/ICSME.2016.19

Morris B (2016) 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/

Myers CR (2003) Software systems as complex networks: structure, function, and evolvability of software collaboration graphs. Phys Rev E 68:046,116

Nesbitt A, Nickolls B (2017) Libraries.io open source repository and dependency metadata. https://doi.org/10.5281/zenodo.808273

Posnett D, D'Souza R, Devanbu P, Filkov V (2013) Dual ecological measures of focus in software development. In: Int'l conference software engineering. IEEE, pp 452–461. https://doi.org/10.1109/ICSE.2013.6606591
Raemaekers S, van Deursen A, Visser J (2014) Semantic versioning versus breaking changes: a study of the Maven repository. In: Working conference source code analysis and manipulation, pp 215–224. https://doi.org/10.1109/SCAM.2014.30
Robbes R, Lungu M, Röthlisberger D. (2012) How do developers react to API deprecation? the case of a Smalltalk ecosystem. In: Int'l symposium foundations of software engineering. ACM. https://doi.org/10.1145/2393596.2393662
Sametinger J (1997) Software engineering with reusable components. Springer, Berlin
Samoladas I, Angelis L, Stamelos I (2010) Survival analysis on the duration of open source projects. Inf Softw Technol 52(9):902–922. https://doi.org/10.1016/j.infsof.2010.05.001
Santana F, Werner CML (2013) Towards the analysis of software projects dependencies: an exploratory visual study of software ecosystems. In: Int'l Workshop on Software Ecosystems (IWSECO), CEUR workshop proceedings, vol 987, pp 7–18. CEUR-WS.org
Scanniello G (2011) Source code survival with the Kaplan Meier estimator. In: Int'l conference software maintenance, pp 524–527. https://doi.org/10.1109/ICSM.2011.6080823
Schlueter IZ (2016) The npm blog: kik, left-pad, and npm. http://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm
Serebrenik A, Mens T (2015) Challenges in software ecosystems research. In: European conference software architecture workshops, pp 40:1–40:6. https://doi.org/10.1145/2797433.2797475
Vasa R, Lumpe M, Branch P, Nierstrasz O (2009) Comparative analysis of evolving software systems using the Gini coefficient. In: Int'l conference software maintenance, pp 179–188. https://doi.org/10.1109/ICSM.2009.5306322
Vouillon J, Di Cosmo R (2013) Broken sets in software repository evolution. In: Int'l Conference Software Engineering (ICSE). IEEE Press, pp 412–421. https://doi.org/10.1109/ICSE.2013.6606587
Wittern E, Suter P, Rajagopalan S (2016) A look at the dynamics of the JavaScript package ecosystem. In: Int'l conference mining software repositories. ACM, pp 351–361. https://doi.org/10.1145/2901739.2901743
Zanetti MS, Schweitzer F (2012) A network perspective on software modularity. In: ARCS Workshops, pp 1–8
Zheng X, Zeng D, Li H, Wang F (2008) Analyzing open-source software systems as complex networks. Physica A 387(24):6190–6200. https://doi.org/10.1016/j.physa.2008.06.050

Alexandre Decan After several years of doctoral studies at the University of Mons on the subject of data quality in relational databases, he obtained in 2013 a PhD thesis entitled "Certain Query Answering in First-Order Languages".

Since then, he has been a postdoctoral researcher at the University of Mons (Belgium) within the Software Engineering Lab.

Author of many publications related to the maintenance and evolution of software ecosystems, he has been involved in several research projects such as the Action de Recherche Concertée ECOS (Ecological Studies of Open Source Software Ecosystems), FEDER-IDEES (European Regional Development Fund), and the joint FNRS-FRQ Québec-Wallonie collaborative research project SECOHealth (Socio-Technical Methodology and Analysis of Software Ecosystem Health).

Tom Mens is full professor at the Department of Computer Science of the University of Mons (Belgium), where he is directing the Software Engineering Lab since October 2003. He is vice-president of the UMONS research institute INFORTECH and active member of research institutes COMPLEXYS and NUMEDIART. His main research interests are formal foundations, tool support and empirical analysis of software evolution, software quality, software modelling and software ecosystems. He published countless scientific articles on the aforementioned research topics in peer-reviewed international journals, conferences and workshops. He was program chair of ICSM 2013, CSMR 2012 and CSMR 2011. He was keynote speaker for ICSME 2016. He co-organised numerous scientific workshops. He was program committee member and reviewer of numerous international journals, conferences and workshops related to software engineering, software modelling and software evolution. He co-edited two Springer books: "Software Evolution" with S. Demeyer in 2008, and "Evolving Software Systems" with A. Serebrenik and A. Cleve in 2014. He is former director of the ERCIM Working Group on Software Evolution and the ESF Research Network RELEASE. He is or has been involved in several interuniversity research projects and networks, including the Belgian Excellence of Science project SECO-ASSIST, the joint FNRS-FRQ Québec-Wallonie collaborative research project SECOHealth, and the Action de Recherche Concertée ECOS "Ecological Studies of Open Source Software Ecosystems".

Philippe Grosjean is a professor at the COMPLEXYS Research Institute at the University of Mons in Belgium. He is the head of the Numerical Ecology of Aquatic Systems Unit. He is a Bioengineer with a Ph.D. in Marine Biology from the Free University of Brussels. His research targets how complex ecosystems evolve with external pressure, being marine plankton, corals or open source software.