What Do Package Dependencies Tell Us About Semantic Versioning?

Alexandre Decan, *Member, IEEE,* Tom Mens, *Senior Member, IEEE*

Abstract—The semantic versioning (semver) policy is commonly accepted by open source package management systems to inform whether new releases of software packages introduce possibly backward incompatible changes. Maintainers depending on such packages can use this information to avoid or reduce the risk of breaking changes in their own packages by specifying version constraints on their dependencies. Depending on the amount of control a package maintainer desires to have over her package dependencies, these constraints can range from very permissive to very restrictive. This article empirically compares semver compliance of four software packaging ecosystems (Cargo, npm, Packagist and Rubygems), and studies how this compliance evolves over time. We explore to what extent ecosystem-specific characteristics or policies influence the degree of compliance. We also propose an evaluation based on the "wisdom of the crowds" principle to help package maintainers decide which type of version constraints they should impose on their dependencies.

Index Terms—D.2 Software Engineering; D.2.7 Distribution, Maintenance, and Enhancement; D.2.7.g Maintainability; D.2.7.n Version control; D.2.8 Metrics/Measurement; D.2.13 Reusable Software; D.2.13.b Reusable libraries; D.2.16 Configuration Management; D.2.16.f Software release management and delivery

1 Introduction

Contemporary software development is increasingly relying on reusable software packages, stored in open source package distributions for a wide variety of programming languages (e.g., Maven for Java, npm for JavaScript, Rubygems for Ruby, Packagist for PHP, Cargo for Rust). These package distributions can be considered as software ecosystems, consisting of large collections of interdependent software projects developed and maintained in the same environment [1]. In the remainder of this paper we will refer to the package dependency networks of such package distributions as *packaging ecosystems*.

These packaging ecosystems can suffer from a wide variety of health problems, that can be of social or business-related nature, or related to technical issues [2], [3]. This paper focuses on the technical health issues of package dependency networks. Typical examples are problems related to software bugs, security vulnerabilities, outdated and unmaintained packages, and source code that is of low quality, untested or duplicated. Such technical problems can become quite difficult to manage because of the sheer size and complexity of package dependency networks and because of the speed of growth and change of such networks [4], [5].

Semantic versioning, hereafter abbreviated as semver, has been proposed as a solution to the so-called "dependency hell" to which maintainers of software packages in package ecosystems are often confronted. Maintainers of packages that depend on reusable libraries need to cope with a delicate balance. They need to keep their package dependencies up to date to be able to benefit from bug and security fixes and new functionalities, but it may require significant effort to upgrade the package dependencies, especially if changes are backward incompatible [4]. Maintainers of reusable packages are confronted with the same challenge: they need to regularly update their packages with new functionalities and bug or security fixes to keep their "clients" satisfied; but they should avoid introducing breaking changes too frequently as this imposes an extra burden on the maintainers of dependent packages.

semver provides a partial solution by introducing a set of simple rules that suggest how to assign version numbers to inform developers about potentially breaking changes. Based on this, packages can specify dependency constraints that allow automatic patch updates or minor updates for "trusted" dependencies. However, since semver is just a policy, it cannot really be imposed, only embraced by a software community as an acceptable way to express whether package releases introduce breaking changes. Not respecting the policy can cause major problems due to unexpected breaking changes in dependent packages.

An anecdotal example in the npm packaging ecosystem was the backward incompatible minor release 1.7.0 of package underscore that led many maintainers of dependent packages to complain about not respecting the semver policy.[1] Another example, in the Rubygems packaging ecosystem, was the backward incompatible minor release 0.5.0 of package i18n that notably broke the popular activerecord package, on which relied over 900 packages (accounting for more than 5% of the entire packaging ecosystem at the time).

The goal of this paper is to assess to what extent maintainers in different packaging ecosystems rely on the semver policy to define the dependency constraints for the packages they maintain, and to what extent semver can be assumed to be followed by required packages. Although the

• A. Decan and T. Mens are with the Software Engineering Lab, University of Mons, Avenue Maistriau 15, B-7000 Mons, Belgium.
E-mail: alexandre.decan@umons.ac.be and tom.mens@umons.ac.be

Manuscript received October 19, 2018; revised April 4, 2019.

1. github.com/jashkenas/underscore/issues/1805

rules of semver are quite simple, the problem of checking whether a version is compatible is undecidable in general. Even in practice, it is extremely difficult to assess that a version does not contain backward incompatible changes. Only a meticulous analysis of the semantics of the code can allow to assess its compatibility. Because it is impractical to do so at ecosystem scale, we use package dependency constraints as a proxy to assess if a maintainer trusts a required package w.r.t. backward compatibility. Dependency constraints can be either compliant to, more restrictive than, or more permissive than what is suggested by the semver policy. Analysing the evolution of this semver-compliance at the packaging ecosystem level provides insights in the extent to which an ecosystem embraces semver. Inspired by Bogart et al. [4], who observed that different ecosystem communities have different habits and values, we expect to find important variations across these ecosystems.

The empirical findings could be used as a basis for ecosystem contributors. For example, a package maintainer could be informed about how safe it is to use a semver-compliant package dependency constraint based on the wisdom of the crowds principle. If other dependents of the required package use semver-compliant constraints, it is likely that that package is respecting the semver policy.

The remainder of this paper is structured as follows. Section 2 summarises the main principles of semantic versioning. Section 3 discusses related work. Section 4 introduces the necessary terminology and presents the research methodology. Section 5 to Section 9 each address the research questions and present the main research findings. Section 10 discusses the actionability of the results and outlines future research avenues. Section 11 presents the threats to validity of the research and Section 12 concludes.

2 Semantic versioning

Tom Preston-Werner, co-founder of GitHub, introduced the semver specification in September 2011.[2] It can be considered as a *de facto* standard that is commonly used by many software package management systems and other open source or even commercial software projects. In the remainder of this manuscript we will rely on version 2.0.0 (June 2013) of the specification [6].

The main purpose of semver is to allow providers of software packages to specify to what extent changes in newer package releases are backward compatible. Developers that write software depending on such packages can use this information to decide how "permissive" they can be in automatically accepting new releases of such packages. Being more restrictive allows developers to keep full control over package dependencies, at the risk of packages becoming outdated and not being able to benefit from backward compatible updates. Being more permissive allows one to automatically benefit from patch updates containing bug or security fixes [7].

While the idea of semver is very promising, it is not always followed by package maintainers. In his npm blog, Bharathvaj Ganesan [8] states that "SemVer is very popular and the most misused software versioning scheme in the JavaScript universe" and "many contributors of popular libraries usually do not care or break the rules of the semantic versioning". This is confirmed by a recent survey of more than 2,000 developers from different software ecosystems [9] where, even if 92% of the respondents (for npm) claim they always increment the leftmost digit (semantic versioning) if a change might break downstream code, still 70% of the respondents declare they find out that a dependency changed because something breaks when they try to build their own package. These observations motivated us to empirically study to what extent different ecosystems comply to semver.

semver proposes a multi-component version scheme major.minor.patch[-tag] to specify the type of changes that have been made in new package releases. Backward incompatible changes require an update of the major version component, important backward compatible changes (e.g., adding a new functionality that does not affect the existing ones) require an update of the minor version component, and backward compatible bug fixes require an update of the patch component. The optional tag component allows to specify pre-releases (e.g., 2.1.3-alpha, 0.5.0-beta, 3.0.0-rc). Pre-releases indicate unstable versions that might not satisfy the intended compatibility requirements as denoted by their associated normal version.

The combined use of semver and a *dependency constraint* mechanism allows package maintainers to decide how flexible their packages can be in accepting future releases of their dependencies. semver has been introduced and used in different software packaging ecosystems, such as Cargo (for Rust), npm (for JavaScript), Packagist (for PHP) or Rubygems (for Ruby). The degree to which semver is respected may vary from one ecosystem to another, and depends on ecosystem-specific characteristics, such as the way in which dependency constraints are expressed and enforced. The types of constraints used express the extent to which a package dependency is compliant with semver (e.g., npm uses ^ to accept all minor and patch updates), or on the contrary more restrictive (e.g., npm uses ∼ to allow patch updates only) or more permissive (e.g., npm uses latest to accept any future release).

3 Related Work

Managing package dependencies and their associated dependency constraints requires a delicate trade-off between the optimistic approach of adopting new versions of required packages but having to face potential breaking changes, and the conservative approach of relying on stable but outdated versions of required packages with the risk of missing potentially important updates. Commercial software development may favor the conservative approach, because the risk, cost and effort involved with updating dependencies may be too high for something that is already working. We hypothesise that open source software development is more likely to favor the optimistic approach.

Many researchers have studied dependencies in packaging ecosystems. Bavota et al. [10] studied the evolution of dependencies in the Apache ecosystem and highlighted that the number of dependencies is growing exponentially and must be taken care of by developers. They found that developers were reluctant to upgrade the packages they

2. semver.org

depend upon because of breaking changes. Developers are more likely to adopt a new version only when it includes major improvements.

Decan et al. [11] studied the CRAN ecosystem and observed that more than 40% of the failures observed in CRAN packages are caused by incompatible changes in their dependencies. Decan et al. [5] compared the evolution of the package dependency networks of Cargo, CPAN, CRAN, npm, NuGet, Packagist and Rubygems. They proposed novel metrics to capture their growth, changeability, reusability and fragility. They observed that an increasing majority of the packages depend on other packages, and there is a high proportion of "fragile" packages due to a high and increasing number of (transitive) dependencies.

Mostafa et al. [12] detected backward compatibility problems in Java libraries by performing regression tests on version pairs, and by inspecting bug reports related to version upgrades. Xavier et al. [13] extracted breaking changes from APIs through a diff tool that collected such changes between two versions of a Java library. In a large-scale empirical study on API breaking changes, they found that Java libraries are often backward incompatible and the rate of breaking changes increases over time.

Bogart et al. [4] conducted a qualitative comparison of npm, CRAN and Eclipse, to understand the impact of community values, tools and policies on breaking changes. They found that there are two main types of mitigation strategies to reduce the exposure to changes in dependencies: limiting the number of dependencies, and depending only on "trusted packages". In a follow-up work, they interviewed more than 2,000 developers about values and practices in 18 ecosystems [9]. Among other findings, they observed that package maintainers are frequently exposed to breaking changes, and mainly discover them at build time.

The semantic versioning policy was introduced to help developers identify incompatible updates in required packages, but it is not always well-respected by developer communities. Wittern et al. [14] studied the evolution of a subset of JavaScript packages in npm, analysing characteristics such as their dependencies, update frequency, popularity, version numbering and so on. They observed that the versioning conventions that maintainers use for their packages are not always compatible with semver.

Raemaekers et al. investigated the use of semver in 22K Java libraries in Maven over a seven-year time period. They found that breaking changes appear in one third of all releases, including minor releases and patches [15], implying that semver is not a common practice in Maven. Because of this, many packages use strict dependency constraints and package maintainers avoid upgrading to newer versions of dependent packages. Their approach relies on clirr, a tool that detects breaking API changes through static analysis of Java code. However, the tool does not detect the presence of breaking changes due to logical changes in the code. While a similar tool could be developed for other languages as well, it requires a clear separation between the public and private API, and such a distinction does not explicitly exist in many dynamic languages such as JavaScript or Python, making the accurate detection of breaking changes much harder.

While dependency constraints can be helpful to prevent dependent packages from breaking after the introduction of a backward incompatible change, such constraints can lead to dependency conflicts when multiple versions of a same library are required by different packages. Wang et al. [16] studied such dependency conflicts in Java projects. In the presence of multiple installed versions of a same package, because the JVM loads one version and shadows the others, issues may occur when the loaded version fails to provide a required feature. They conducted an empirical study of those issues in 71 Apache projects, and showed that dependency conflicts are very common in practice. They also provided a tool to detect such issues.

Dependency conflicts can lead to co-installability issues when multiple versions of a same library are not allowed to be installed at the same time. The problem of identifying a set of versions that satisfy all dependency constraints is known to be NP-complete, and was notably addressed by Di Cosmo et al. [17], [18], [19].

Several researchers have studied the problem of outdated package dependencies. Kula et al. [20] analyzed over 6K Java libraries in Maven to investigate the latency in adopting the latest release of dependency targets. They also investigated over 4.6K GitHub projects with dependencies on 2.7K distinct Maven packages and found that more than 80% of the studied systems have outdated dependencies [21]. Decan et al. [22] studied the use of package dependency constraints in npm, CRAN and Rubygems. They observed that, while strict dependency constraints prevent backward incompatibility issues, they also increase the risk of having dependency conflicts, outdated dependencies and missing important updates.

Cox et al. [23] revealed that systems with outdated dependencies are four times more likely to have security issues than systems that are up-to-date. Derr et al. [24] conducted a survey with more than 200 mobile app developers in the Android ecosystem to investigate the use of outdated libraries, and reported that almost 98% of 17K actively used library versions with a known security vulnerability could be easily fixed by updating the library. Decan et al. [25] empirically studied nearly 400 security reports for 269 npm packages, and found that more than 40% of the releases depending on a vulnerable package do not automatically benefit from the security fixes because of too restrictive dependency constraints.

Cox et al. [23] also proposed different metrics to quantify a software system's *dependency freshness*. They assessed the usefulness of these metrics through interviews with technical consultants and showed that their objective metrics are in agreement with the subjective perception of dependency freshness. Gonzalez-Barahona et al. [26] introduced the concept of technical lag to measure how outdated a system is with respect to its dependencies. They defined technical lag in terms of a lag function and lag aggregation function for packages. Decan et al. [27] measured such technical lag for npm packages and observed that a large number of packages have outdated dependencies, and a large number of dependencies have a technical lag of several months. As a follow-up work, Zerouali et al. [28] proposed and validated a formal framework for technical lag measurement.

In order to help developers decide when to use which version of a software library, Mileva et et al. [29] proposed an approach and associated tool based on the "wisdom of the crowds". If a library version is used by more developers, it is more likely to be recommended. The tool was validated on hundreds of Java libraries in the Maven repository. They acknowledge that other context-specific factors need to be taken into account to recommend the most appropriate version of a library, and that a cost-benefit analysis is required to decide whether or not to switch to a new library version.

Most of the related work about breaking changes relies on static code analysis of the component being updated and on the computation of a call graph of the dependent component to assess the compatibility of the component being updated and whether this update leads to breaking changes. Such fine-grained analyses are very valuable, but can be time-consuming and remain restricted to a particular programming language. They do not scale up to the ecosystem-level, because of the massive amount of packages and package dependencies, and because of the large amount of packages that get updated every day. Therefore, we propose a complementary approach that relies on package dependency metadata only. Such a coarse-grained analysis is language-independent and scalable to the size of contemporary packaging ecosystems. On the downside, it is less precise, because it only assesses the *perceived* backward compatibility on the basis of the semantic versioning policy.

4 Methodology and Terminology

4.1 Terminology

This section presents the terminology that will be used in this paper. All main terms are highlighted in **boldface**.

Software **packaging ecosystems** are collections of software **packages**, each of which can have one or more **package releases** that are determined by a unique **version number**. We assume all version numbers to be of the form $x.y.z$ (where all three version components are positive integers). Package releases with a version number in the $[1.0.0, +\infty[$ version range are called **production releases**[3] while package releases with a version number in the $[0.0.0, 1.0.0[$ version range are called **initial development releases**.

Package releases can express **dependencies** on other packages. If package release R depends on package P, R is called a **dependent**, while P is called a **required package**. P is said to have a **reverse dependency** to R. A dependency can specify which releases of a required package are allowed to be selected for installation by means of a **dependency constraint**. Such constraints express a **version range** of version numbers allowed by the dependency. For example, constraint $< 2.0.0$ defines the version range $[0.0.0, 2.0.0[$, signifying that *any* package release below version 2.0.0 is allowed. Dependency constraints that target only initial development releases will be called **pre-1.0.0 constraints**. The complement (i.e., the right boundary of the version interval is 1.0.0 or above) will be called **post-1.0.0 constraints**.

Dependency constraints allow maintainers of dependent packages to benefit from the semver standard (detailed in Section 2) as they allow to restrict the range of releases of a required package to the ones that are expected to be backward compatible. If a dependency constraint adheres to semver, we will call it **compliant**. Dependency constraints may also be more **permissive** or more **restrictive** than what semver dictates. How to interpret this however depends on the version range expressed by the dependency constraint.

A post-1.0.0 constraint will be considered **compliant** if the version range only includes all minor releases and patches above the left boundary. It will be **permissive** if the version range includes *more* versions than the ones considered backward compatible by semver. It will be **restrictive** if the version range includes *fewer* versions than the ones considered backward compatible by semver. For example, version range $[1.1.0, 2.0.0[$ is compliant; version range $[1.1.0, 3.0.0[$ is permissive since it encompasses releases from multiple major versions (all releases with major version number 2 would be accepted as well) ; and version range $[1.1.0, 1.2.0[$ is restrictive since only patches to 1.1.0 will be included, but no higher minor versions.

For pre-1.0.0 constraints, the official semver standard only considers strict constraints targeting a single version (i.e., a singleton version range) to be compliant.[4] In practice, as we will see later, this is often considered as too restrictive.

semver considers **pre-release versions** (not to be confused with initial development releases) corresponding to alpha-releases, beta-releases or release candidates (e.g., 2.1.3-alpha, 0.5.0-beta or 3.0.0-rc) to be "unstable and might not satisfy the intended compatibility requirements as denoted by its associated normal version." Because of this, and because of the fact that developers are generally aware of the fact pre-releases are meant to be unstable and are expected to have breaking changes, [5] we will not consider pre-releases in our analyses. This is in line with the way in which the npm semantic versioner deals with pre-releases: "[...] pre-release versions frequently are updated very quickly, and contain many breaking changes that are (by the author's design) not yet fit for public consumption. Therefore, by default, they are excluded from range matching semantics."[6]

4.2 Selected packaging ecosystems

Open source packaging ecosystems exist for most of the major contemporary programming languages. In this article we focus on packaging ecosystems that are known to recommend the use of semver, and that allow developers to specify version constraints over their package dependencies.

To analyze these ecosystems, we rely on version 1.2.0 of the libraries.io Open Source Repository and Dependency Metadata dataset [30]. While libraries.io contains historical data of packages and their dependencies for many packaging ecosystems, we excluded those that: do not host a sufficiently large number of packages (e.g., Julia has fewer than 3K packages); have an incomplete or inaccurate list of dependencies (e.g., Maven or PyPI); are a subset of another ecosystem (e.g., Bower is a subset of npm); target a specific tool or framework (e.g., Meteor); have their own specific versioning policy or do not support nor recommend semver

3. From semver FAQ, "If your software is being used in production, it should probably already be 1.0.0." [6]

4. From semver specification, "Major version zero (0.y.z) is for initial development. Anything may change at any time. The public API should not be considered stable." [6]

5. blog.npmjs.org/post/115305091285/introducing-the-npm-semantic-version-calculator

6. docs.npmjs.com/misc/semver

(e.g., Go, Hackage, CPAN or CRAN). Based on these exclusion criteria, we retained four packaging ecosystems: Cargo for the Rust programming language, npm for JavaScript, Packagist for PHP and Rubygems for Ruby.

Table 1 presents some characteristics of these packaging ecosystems. We observe that some ecosystems are much older than others. For all empirical analyses, packages and dependencies from the entire history of each ecosystem were considered. However, to facilitate cross-ecosystem comparison, the figures in the paper present the observations for a 5-year time interval only, from 2013-01-01 to 2017-12-31.

TABLE 1
Characteristics of the curated dataset (January 1, 2018).

Pkg. Eco.	Creation	Language	#pkg	#rel	#dep
Cargo	2014	Rust	13K	73K	257K
npm	2010	JavaScript	630K	4,181K	19,030K
Packagist	2012	PHP	121K	798K	2,168K
Rubygems	2004	Ruby	141K	809K	1,923K

The four considered ecosystems do not necessarily agree on the way they adhere to semver. A main variation point is how they interpret semver-compliance for initial development releases. Most package management tools implement a variant of semver that is more permissive for initial development releases. For example, Cargo, npm and Packagist assume that patches for initial development releases will not introduce breaking changes. Rubygems is even more permissive and treats initial development releases in the same way as production releases.

4.3 Data curation

For each selected packaging ecosystem, we consider all its packages and all releases for these packages. For each package release, we consider its list of dependencies. When declaring dependencies, a package maintainer can specify the purpose of the dependency (e.g., it is needed to execute, develop or test the package). We excluded the dependencies that are only needed to test or develop a package because not all considered ecosystems make use of them, and not every package declares a complete and reliable list of such dependencies. We therefore consider only those dependencies that are required to install and execute the package, and hence more accurately reflect what is needed to actually use the package. In the ecosystems we analyzed, these dependencies are either labeled "runtime" or "normal".

We only consider "internal" dependencies between packages belonging to the ecosystem, i.e., we exclude all dependencies targeting external sources that are/were not available through the packaging ecosystem (e.g., packages that are hosted directly on the web or on Git repositories).

As explained in Section 4.1 we exclude all pre-release versions (e.g., 2.1.3-alpha, 0.5.0-beta or 3.0.0-rc) since they are known to be unstable and not necessarily representative of their associated normal version. They are of no interest for our analysis, and including them would only bias the results. Specifically for the npm ecosystem, we also excluded around 52K "spam" packages. These packages were automatically created by tools and developers abusing the npm API to publish new packages. They are either "funny" packages depending on a very large number of other packages[7] or replications/variations of existing packages.[8] We did not find such spam packages in the other three ecosystems.

Table 1 summarises the curated dataset, by reporting the number of packages (#pkg), package releases (#rel), and dependencies (#dep) that will be considered for our analysis. A replication package of our analysis is available on doi.org/10.5281/zenodo.2563141.

4.4 Constraint normalisation

In addition to the fact that different packaging ecosystems implement different variants of semver, they tend to use different syntactic notations for specifying dependency constraints, or may interpret the same notation in a different way. For example, in Packagist and Rubygems the dependency constraint 1.0 means that 1.0.0 is the only allowed release; while for npm this constraint also allows all patch releases (e.g. 1.0.1, 1.0.2, and so on); and Cargo is even more tolerant since this constraint even accepts all minor releases. Needless to say, this can lead to confusion for developers that are involved in multiple packaging ecosystems.

TABLE 2
Examples of version constraints and their corresponding version range.

Constr.	Cargo	npm	Packagist	Rubygems
=1.0.0	[1.0.0]	[1.0.0]	[1.0.0]	[1.0.0]
1.0.0	[1.0.0, 2.0.0[	[1.0.0]	[1.0.0]	[1.0.0]
1.0	[1.0.0, 2.0.0[	[1.0.0, 1.1.0[	[1.0.0]	[1.0.0]
1	[1.0.0, 2.0.0[	[1.0.0, 2.0.0[	[1.0.0]	[1.0.0]
∼1.2.3	[1.2.3, 1.3.0[	[1.2.3, 1.3.0[	[1.2.3, 1.3.0[	[1.2.3, 1.3.0[
∼1.2	[1.2.0, 1.3.0[	[1.2.0, 1.3.0[	[1.2.0, 2.0.0[	[1.2.0, 2.0.0[
∼1	[1.0.0, 2.0.0[	[1.0.0, 2.0.0[	[1.0.0, 2.0.0[	N/A
^1.2.3	[1.2.3, 2.0.0[	[1.2.3, 2.0.0[	[1.2.3, 2.0.0[	N/A
>1.2.3	]1.2.3, +∞[	]1.2.3, +∞[	]1.2.3, +∞[	]1.2.3, +∞[
∼0.1.2	[0.1.2, 0.2.0[	[0.1.2, 0.2.0[	[0.1.2, 0.2.0[	[0.1.2, 0.2.0[
^0.1.2	[0.1.2, 0.2.0[	[0.1.2, 0.2.0[	[0.1.2, 0.2.0[	N/A

Table 2 shows some typical examples of version ranges corresponding to how specific version constraint notations are interpreted in each of the considered packaging ecosystems.[9] The cell color indicates whether the version constraint is compliant (white cells), permissive (green cells) or restrictive (red cells) w.r.t. the semver-standard. We observe for example that the ^ notation is semver-compliant for production releases in all ecosystems (except Rubygems where this notation does not exist). Similarly, the $=x.y.z$ and $\sim x.y.z$ notations are more restrictive for production releases in all ecosystems, and the $>$ notation is more permissive. For initial development releases, the $\sim$ and ^ notations are more permissive than semver.

Given the observed notational differences of dependency constraints, we will work directly with version ranges instead. The advantage of this approach is that the subsequent analysis in this paper will be agnostic of the specific notation used by the different packaging ecosystems. We

7. e.g., npm-gen-all whose purpose is to "create a multitude of npm projects that will depend on every npm package published".

8. e.g., npmdoc-*, npmtest-*, *-cdn, etc. Most of them are no longer available through npm.

9. Rubygems' pessimistic operator $\sim>$ is represented as $\sim$ in this table.

only require a parser to convert dependency constraints into version ranges. Based on the available documentation, we wrote a parser for each of the four considered ecosystems. These parsers were able to cope with the large majority of dependency constraints. The average monthly proportion of parsable constraints was 99.9% for Cargo, 97.8% for npm, 96.7% for Rubygems, and 91.0% for Packagist.

The lower value for Packagist is mostly due to constraints defined before 2015, where "only" 86.8% of them could be parsed, while 95.5% of the constraints defined since 2015 could be parsed. We cannot explain this difference, but believe it is related to the development of its package manager, Composer, whose first alpha was released on 2013-07-03, and whose first final version was released on 2016-04-05.

5 How frequent are pre-1.0.0 constraints?

semver considers initial development releases as unstable. In addition to this, the considered ecosystems differ in the way they interpret compliance for pre-1.0.0 constraints targeting these releases. To determine whether we should distinguish pre-1.0.0 constraints from post-1.0.0 constraints in our analysis, we computed, for each month, the proportion of pre-1.0.0 constraints in all releases newly distributed during that month. The results are presented in Figure 1.

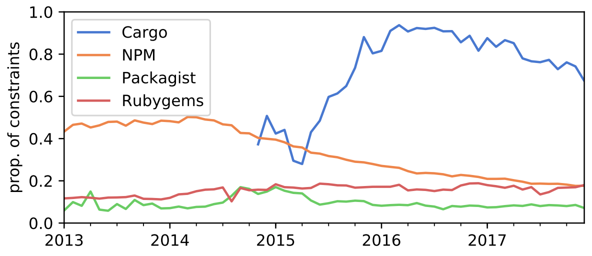

Fig. 1. Monthly proportion of pre-1.0.0 constraints in newly distributed releases.

This monthly proportion is very high for Cargo (72.7% on average), and non-negligible for npm (33.8% on average), Rubygems (15.5% on average) and Packagist (9.5% on average). The high proportion of pre-1.0.0 constraints in Cargo can be explained by the higher proportion of initial development releases in this (very) young package manager: on January 2018, only 1,240 packages out of 13K have reached a production release. We also observe a decreasing proportion of pre-1.0.0 constraints in npm from April 2014 onwards. This is a consequence of new npm policies to reduce the use of initial development releases.[10]

We did not expect to see such a high proportion of pre-1.0.0 constraints, as it implies that many package releases depend on packages that are still in the initial development phase. Because, according to semver, such packages are supposed to be unstable, there is an increased risk of depending on them.

10. See for instance github.com/npm/init-package-json/commit/363a17bc31bf653bb9575105eea62fb4664ad04b or github.com/npm/node-semver/issues/79.

To assess how many packages are affected by this phenomenon, we computed the monthly proportion of packages required by a pre-1.0.0 constraint. The proportions are even higher than what we observed in Figure 1, confirming that many packages in initial development phase are already used by other packages. On average, the proportion of required packages targeted by pre-1.0.0 is 80.8% for Cargo, 51.6% for npm, 30.9% for Rubygems and 18.3% for Packagist.

Findings. A majority of required packages in Cargo and npm are still in an initial development phase. For these two ecosystems, more than one third of the dependency constraints are pre-1.0.0 constraints. For Rubygems and Packagist, even if this proportion is lower, it still represent on average 15.5% and 9.5% of all constraints. It is therefore important to distinguish between pre-1.0.0 and post-1.0.0 constraints to analyze semver-compliance.

6 Are pre-1.0.0 constraints semver-compliant?

Dependency constraints represent to some extent the confidence that the maintainer of a package has in its required packages. By definition, a semver-compliant pre-1.0.0 constraint targets exactly one release, i.e., the version range only captures a single version of the required package. Since it is not possible to be more restrictive, a pre-1.0.0 constraint can only be *compliant* or *permissive*.

A permissive constraint might indicate that minor or patch releases of a required package are not expected to be backward incompatible or it could also reflect that the maintainer of the dependent package prefers not to specify an upper bound on the set of allowed releases and to manually adapt its package for each backward incompatible release of the required package. This is confirmed by anecdotal qualitative evidence [4]: "most interviewed developers adopted a reactive strategy for most of their dependencies. They wait to hear about problems from others (in advance, or after things have broken): upstream developers contacting them about breaking changes, failing tests after dependency updates, or platform maintainers warning of changes that would affect them".

To study the degree to which pre-1.0.0 constraints adhere to semver, we computed the monthly proportion of those pre-1.0.0 dependency constraints that are either *compliant* or *permissive*. These proportions are shown in Figure 2 relative to the total number of pre-1.0.0 constraints defined in newly distributed releases for each month.

We observe that most constraints are permissive, regardless of the ecosystem, and this proportion increases over time. For December 2017, the proportion of permissive constraints is of 96.5%, 74.3%, 93.4% and 91.3% respectively for Cargo, npm, Packagist and Rubygems.

Such high proportions are likely to be a consequence of the specific interpretation of semver by these ecosystems for initial development releases. Indeed, semver assumes that patch releases should not be considered as backward compatible, while the four ecosystems are more permissive and consider these releases as being compatible, as explained in their respective documentation. It is therefore not surprising

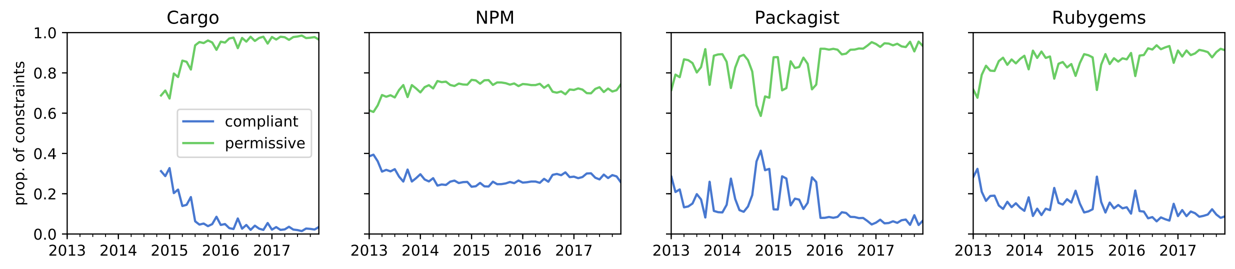

Fig. 2. Monthly proportion of **pre-1.0.0** constraints that are compliant or permissive.

that a majority of constraints are more permissive than semver.

The interpretation of semver for Rubygems is even more permissive as it considers minor releases as being backward compatible as well. Consequently, the only way to release breaking changes in initial development releases is to pass the 1.0.0 version barrier. Maintainers in Rubygems seem to follow this more permissive policy: we identified that 51% of the constraints allow minor releases as well, while this proportion is less than 0.01% for Cargo and npm, and of 13.8% for Packagist.

Findings. A large majority of the pre-1.0.0 constraints are not compliant with semver. They tend to follow a more permissive ecosystem-specific interpretation of semver for initial development releases.

7 Are post-1.0.0 constraints semver-compliant?

Having focused on pre-1.0.0 constraints in the previous research question, we will now study semver-compliance of post-1.0.0 constraints. By using a semver-compliant post-1.0.0 constraint, the package maintainer trusts that the required package does not introduce backward incompatible changes in non-major releases. A *permissive* constraint could reflect that even major releases of a required package are not expected to be backward incompatible. Using a *restrictive* constraint might indicate that the package maintainer does not trust the changes in its dependency or prefers to keep full control over how to deal with these changes.

To study the degree of semver-compliance of post-1.0.0 constraints, we computed the monthly proportion of those dependency constraints that are either *compliant*, *permissive* or *restrictive*. Figure 3 illustrates that all considered ecosystems become more compliant over time, and both the proportions of permissive and restrictive constraints decrease over time. However, the degree of compliance is quite different for each ecosystem. Considering the last month of the observation period (December 2017), most constraints are compliant in Cargo (96.1%), npm (80.6%) and Packagist (73.7%). For Rubygems, this proportion is of 40.1%, slightly beyond the proportion of permissive constraints (43.1%).

The non-negligible proportion of restrictive constraints in npm (17.9%), Packagist (18.6%) and Rubygems (16.7%) may indicate that many package maintainers do not trust the changes in their dependencies as these restrictive constraints prevent the automatic adoption of minor releases. We looked more specifically at these constraints, and found that 82.7% of them in npm, 32.3% in Packagist and 51.7% in Rubygems prevent the automatic adoption of patch releases as well.

We observe that Cargo has an evolutionary behavior that is quite different from the other packaging ecosystems, with a very sudden change from almost 100% of permissive constraints to almost 100% of compliant ones in early 2016. This change can be explained by a new policy restricting the use of (permissive) wildcard constraints. Since January 2016, Cargo prohibits the use of these constraints because "a version requirement of * says 'This will work with every version ever,' which is never going to be true. Libraries should always specify the range that they do work with, even if it's something as general as 'every 1.x.y version"'.[11]

Figure 3 also reveals that the degree of compliance for npm starts to increase rapidly since early 2014. This trend break coincides with the introduction by npm of the ^ notation for expressing dependency constraints that are compliant with semver, and the use of this notation as the default one by npm tools. Similarly for Packagist, we observe a migration from restrictive to compliant constraints starting from mid 2014. This phenomenon could correspond to several changes in Composer, the default package manager of Packagist, that were intended to address co-installability issues and to facilitate and greatly speed up the installation of packages in the presence of non-strict constraints.

Rubygems differs from the other ecosystems in its higher proportion of permissive constraints. This is due to the presence of many "optimistic" constraints[12] that only specify a lower bound on the versions that are required during installation. Finally, the peak that can be observed in Figure 3 for Rubygems in August 2014 is the consequence of a massive import of more than 25K package releases in Rubygems, resulting in invalid release dates for these releases.

Findings. A large majority of post-1.0.0 constraints in Cargo, npm and Packagist are compliant with semver, and all considered ecosystems become more compliant over time. The proportion of permissive and restrictive constraints decreases over time. On December 2017, there are still more than 16% of the constraints in npm, Packagist and Rubygems that are restrictive and prevent minor releases, and in many cases patch releases as well, from being automatically adopted.

11. doc.rust-lang.org/cargo/faq.html
12. guides.rubygems.org/patterns/#pessimistic-version-constraint

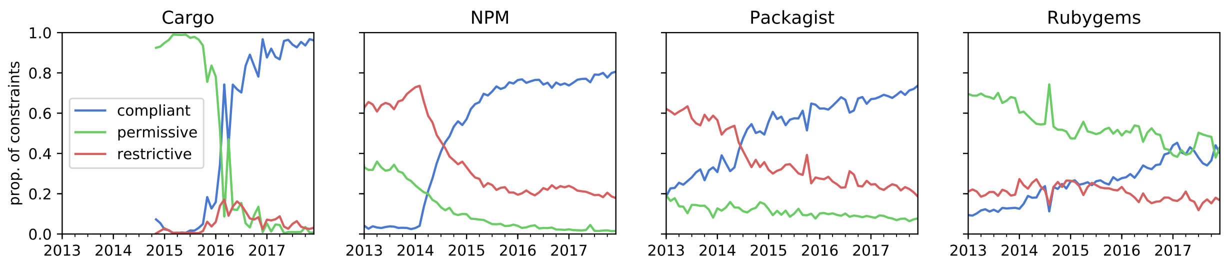

Fig. 3. Monthly proportion of **post-1.0.0** constraints that are compliant, permissive or restrictive.

8 When are dependency constraints changed?

Dependency constraints provide a way to allow the *automatic* adoption of new *compatible* releases of required packages, while rejecting those that are likely to be incompatible. However, some package maintainers may prefer to keep full control, and decide to adopt new releases of required packages by *manually* updating the dependency constraint so that it accepts these new releases. To quantify this phenomenon, we measured the frequency of updating dependency constraints. We found that, on average, constraints are updated every 6.9 releases in npm, 5.7 releases in Packagist, 5.3 releases in Rubygems, and every 3.4 releases in Cargo.

We expect more restrictive constraints to be updated more often, reflecting the manual adoption of new compatible releases of a required package. To verify this hypothesis, we rely on the statistical technique of survival analysis (a.k.a. event history analysis) [31]. This technique models "time to event" data with the aim to estimate the survival rate of a given population, i.e., the expected time duration until the event of interest occurs (e.g., death of a biological organism, failure of a mechanical component, recovery of a disease). 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 for them during the observation period. A common non-parametric statistic used to estimate survival function is the Kaplan-Meier estimator [32].

The event being considered for this analysis is "the dependency constraint is changed". The survival analysis reveals if the type of constraint being used (compliant, permissive or restrictive) affects the time it takes (since the introduction of the constraint) until the dependency constraint is changed. Figure 4 shows Kaplan-Meier survival curves for the considered event. For the reasons explained in Section 5, we distinguish between pre-1.0.0 and post-1.0.0 constraints.

With the notable exception of post-1.0.0 constraints in Cargo, we observe that the more permissive are constraints, the less frequently they are updated. This is valid for pre-1.0.0 as well as post-1.0.0 constraints. For post-1.0.0 constraints in Cargo, however, permissive constraints are updated more quickly than compliant ones. This is very likely to be the consequence of the policy change in 2016 prohibiting the use of wildcard constraints, leading all dependency constraints that were relying on this permissive operator to be changed in a short period of time.

We carried out log-rank tests to compare whether statistically significant differences could be found between the survival curves for the three types of constraints within each ecosystem. We performed pairwise comparisons between compliant, permissive and restrictive constraints, first for pre-1.0.0 and then for post-1.0.0 constraints. The differences were statistically confirmed at $\alpha = 0.05$, i.e., the null hypotheses H_0 assuming that the survival curves are the same were rejected with p-values < 0.05 (adjusted following Bonferroni-Holm method to control family-wise error rate [33]). This confirms a statistically significant difference in the time required to change a constraint. Only for Cargo, we were not able to reject H_0 when comparing permissive and restrictive post-1.0.0 constraints (p-value $= 0.64$).

With again the exception of permissive constraints in Cargo, we also observe from Figure 4 that pre-1.0.0 constraints are more often updated than their post-1.0.0 counterparts. A possible explanation is that packages in their initial development phase are updated more often, causing in turn their dependent packages to update more frequently. We also believe that packages in the initial development phase are less concerned with API stability than packages in production. We performed a survival analysis of the event "release is updated" for both initial development and production releases, and confirmed through log-rank tests that initial development releases are indeed updated more often than production releases for npm, Packagist and Rubygems ($\alpha = 0.05$).

Findings. Dependency constraints remain unchanged during 3.4 to 6.9 releases on average, depending on the ecosystem. The more permissive a constraint is, the less often it is updated. Pre-1.0.0 constraints are more often updated than post-1.0.0 constraints.

9 To what extent do packages follow semantic versioning?

If a package is known to respect the semver policy, then we expect most of its dependents to use compliant constraints, since the maintainers of such dependent packages face little or no risk of breaking changes. If a package is known to disobey the semver policy, then we expect dependent packages to use either permissive or restrictive constraints depending on whether their maintainer follows an optimistic or pessimistic versioning strategy, and depending on how frequently the required package is updated. The

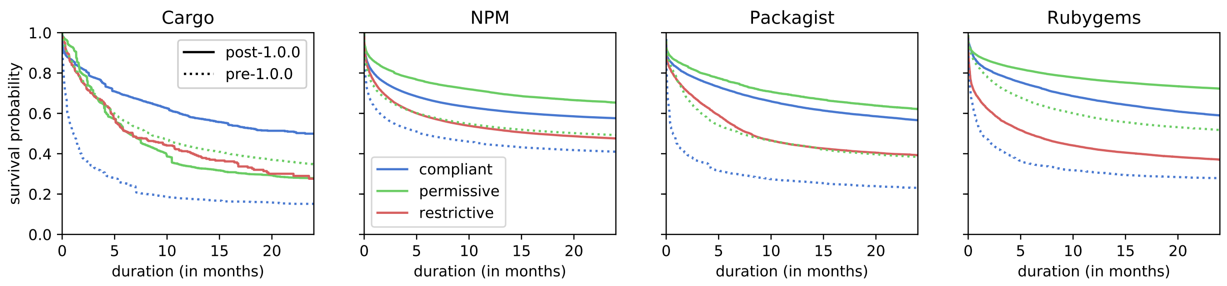

Fig. 4. Survival analysis for event "the dependency constraint is changed", grouped by type of constraint (compliant, permissive or restrictive). The survival probability for pre-1.0.0 constraints is shown with dotted lines, and with straight lines for post-1.0.0 constraints.

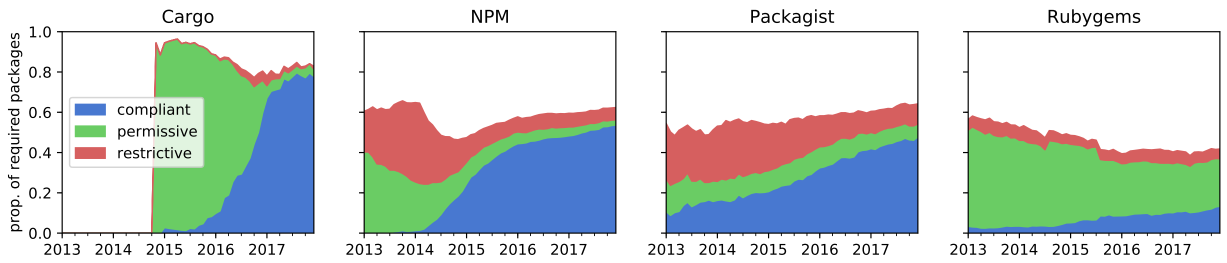

Fig. 5. Proportion of required packages "specialized" towards a specific constraint type (compliant, permissive or restrictive) for its reverse dependencies, based on monthly snapshots of the package dependency network.

TABLE 3
Concrete examples of specialization analysis for some frequently used packages, and suggested recommendation for package maintainers desiring to depend on these packages.

package	ecosystem	dependents	compliant	permissive	restrictive	decision
serde	Cargo	592	**575 (97.1%)**	0	17	adhere to semantic versioning
mage2pro/core	Packagist	51	1	**50 (98.0%)**	0	set permissive dependency constraint
react-scripts	npm	57	1	0	**56 (98.2%)**	impose restrictive dependency constraint
rails	Rubygems	997	288	506	203	undecided
rest-client	Rubygems	387	173	137	77	undecided

pessimistic strategy avoids breaking changes by preventing newer minor releases or patches from being used. It should be followed if the required package regularly introduces breaking changes in such minor or patch releases. The optimistic strategy corresponds to a reactive approach where the package maintainer allows for new releases of dependent packages, even if they may cause breaking changes. If this happens, the maintainer of the dependent package will have to address the problem. Anecdotal evidence, gathered from a survey over 2,000 developers in 18 ecosystems [9] appears to confirm the use of the optimistic strategy: developers report that breaking changes are fairly infrequent, and that developers only take action when they notice that something breaks because of a change in a dependency.

In what follows, we adopt the assumption that package maintainers generally know what they are doing when defining dependency constraints (even though there may be exceptions), hence it is unlikely that the majority of them would be doing it wrong at the same time. Based on the above reasoning, we hypothesise that, if many dependents of the same required package use compliant constraints, it is likely that this package respects the semver policy. A similar reasoning could be made for permissive and restrictive constraints.

As anecdotal evidence of this hypothesis, let us consider the cases of underscore and lodash. Both packages are distributed on npm and provide similar features. While lodash claims to follow semver, underscore is known not to have respected semver in the past,[13] notably in versions 1.7.0 and 1.8.0, respectively released in August 2014 and February 2015. An analysis of dependency constraints in the last snapshot of npm indicates that while 80.4% of constraints targeting lodash are compliant with semver, only 46.2% of those targeting underscore are. The latter proportion is barely higher than the proportion of restrictive constraints (41.7%). As such, based on the "wisdom of the crowds" principle, new packages desiring to depend on underscore can assume that it is more safe to use restrictive constraints, while for lodash they can use compliant constraints.

Relying on a commonly used threshold in statistical analysis, we consider the reverse dependencies of a required package to be "in agreement" if "at least 95%" of them use the same type of constraint (i.e., compliant, permissive or restrictive). In those cases, we will call the required package "specialized".

We computed monthly snapshots of each package dependency network and, for each snapshot, we computed the proportion of such "specialized" required packages by

13. See, e.g., github.com/jashkenas/underscore/issues/1684

taking into account the dependencies specified in the latest available release of each package. Knowing that the interpretation of semver for initial development releases differs from one ecosystem to another (see Section 4.2), we consider post-1.0.0 constraints only for this analysis. To prevent the analysis from being biased by packages that are no longer maintained, we removed from each snapshot packages that were not updated during the last 365 days. We also excluded required packages with less than two reverse dependencies as it makes little sense to consider a "specialization" on the basis of one constraint only.

The results are visualised in Figure 5 using a "stacked" line plot: the proportion of required packages that are specialized towards compliant, permissive or restrictive constraints (shown in different colors) are added up so that the total proportion of specialized required packages becomes immediately apparent. We observe that most required packages in Cargo, npm and Packagist are specialized, and that their proportion increases over time. On December 2017, we found 82.9% of such specialized packages in Cargo, 62.3% in npm and 64.1% in Packagist. The majority of these packages are specialized towards being compliant (94% of the specialized packages in Cargo, 86% in npm and 74.5% in Packagist).

The situation is quite different for Rubygems where the proportion of specialized required packages decreases over time, and is much lower than in the three other ecosystems (41.8% in December 2017). Moreover, the specialized packages in Rubygems are mainly specialized towards being permissive (56.5% of the specialized packages), and only to a much lesser extent towards being compliant (32%). These observed lower proportions for Rubygems are a direct consequence of the lower proportion of compliant constraints and the higher proportion of permissive ones that were reported in Section 7.

Specialized packages can be used to recommend dependency constraints. Knowing whether or not a required package is specialized, and towards which constraint type it is specialized, can be useful to package maintainers desiring to depend on it. Based on the wisdom of the crowds, they can decide whether or not they should specify dependency constraints in a similar way as other dependents of the required package. We illustrate this on the basis of some concrete examples of popular packages in different ecosystems, listed in Table 3. For some packages, the large majority of their dependents agree on the type of dependency constraint, implying that it is probably safe to set the same type of constraint in one's own package. This is the case for the first three listed packages.

In some cases it is more difficult to make a decision, because no majority consensus is reached. Hence it is up to each package maintainer to decide what is most appropriate to do. For the Rubygems rails case, the indecisiveness appears to be related to the fact that rails relies on a variant of the semver policy that allows minor releases to deprecate and remove existing features.[14]

Verifying that a package follows semver requires checking the compatibility between consecutive releases of this package, a problem that is known to be undecidable in general. Indeed, "any nontrivial property that involves what the program does (rather than a lexical or syntactic property of the program itself) must be undecidable" [34, p. 403]. Even in practice, it is extremely difficult to assess that a (declared compatible) version does not contain backward incompatible changes, because the changes can be numerous and intricate, and because it does not suffice to check the compatibility of the API only. Only a complete semantic analysis of the underlying code can ascertain its compatibility. This implies that the accuracy of our approach cannot be assessed based on "ground truth". Instead, we rely on a split validation process, a sampling approach to assess a model's performance, notably used in offline validations of recommender systems [35].

14. see github.com/rails/rails/issues/9979

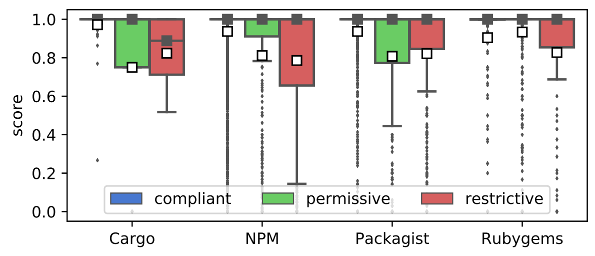

Fig. 6. Results for the split validation process: boxplots showing the distributions of the accuracy scores computed against recommended constraint types for packages in the training set. Median values are represented with a black square and mean values with a white square.

The split validation process consists of dividing the dataset into two parts: a training set consisting of observations up to a given date, and a test set consisting of subsequent observations that are validated against the model. For the training set we selected all specialized packages from the 2017-01-01 snapshot of each ecosystem. Each package in this training set defines a recommended constraint type, according to its specialization. The test set contains all new dependency constraints defined towards these specialized packages during 2017. The type of these constraints is compared to the type of the recommended constraint. The result of this comparison is used to compute the accuracy score of the specialized package, as the proportion of those new constraints whose type corresponds to the one that was recommended. The scores are reported in Figure 6.

We observe that the accuracy scores are high, especially for compliant constraints. The mean value (represented with a white square) ranges between 0.75 and 0.97. The global average score is 0.91 (respectively 0.94, 0.86 and 0.80 for compliant, permissive and restrictive constraints). This indicates that the "wisdom of the crowds" principle holds when specialized packages are used to recommended dependency constraints.

Findings. A majority of the required packages in Cargo, npm and Packagist are specialized, and mostly towards receiving semver-compliant dependencies. semver is less followed in Rubygems, where specialized packages have a tendency to receive more permissive dependencies. It makes sense to rely on the wisdom of the crowds to recommend dependency constraints.

10 Discussion and Future Work

10.1 Dependency hell and co-installability issues

Regardless of whether a required package respects semver or not, co-installability is an important factor to consider when choosing an appropriate dependency management strategy. Co-installability issues may arise when two different packages depend on different versions of another package [19], a consequence of dependency constraints being too restrictive. Packaging ecosystems differ in how their package manager allows to install packages, depending on the implemented package installation strategy (e.g., global vs. local environment, flat or vendored vs. nested installation, etc.). Cargo, npm and Packagist do not seem to suffer (at least not directly) from such co-installability issues. For example, for npm, if two different versions of a package are needed by different dependencies, both versions will be installed in their own folder and each package can depend on the specific version it needs. However, the objects that are created within these two different versions are not necessarily compatible or interchangeable and could lead to other issues (e.g., dependency conflicts [16]). Rubygems packages, on the other hand, are exposed to co-installability issues because gem, the default package manager, installs packages at a system-wide level, and implicitly defines a conflict between any two distinct releases of a same package. This could lead maintainers to avoid the use of tight upper-bounded dependency constraints, thereby providing a possible explanation of the observed higher proportion of permissive constraints in Rubygems (cf. Figure 5).

semver presents itself as a way to alleviate the dependency hell: it provides a way for maintainers to specify looser constraints while still limiting their exposure to breaking changes. These loose constraints potentially accept more releases of a required package, therefore limiting the risk of conflicts. Being able to suggest to maintainers of dependent packages when it is safe to rely on semver-compliant constraints as opposed to more restrictive constraints is, indirectly, a (partial) solution to the dependency hell.

10.2 Differences between packaging ecosystems

The wide range of syntactic and semantic differences in how to specify dependency constraints in different ecosystems (cf. Table 2) can be quite confusing to package maintainers, especially if they contribute to multiple ecosystems. For this reason, we recommend to introduce a common ecosystem-independent notational convention for specifying dependency constraints, and integrate this in (a new version of) the semantic versioning specification.

Our empirical analysis revealed that the extent to which semver is respected strongly depends on the considered ecosystem. In addition to this, ecosystem-specific characteristics and policy changes tend to have an important impact on semver compliance. We list these for the four ecosystems we analyzed.

Cargo, the youngest ecosystem, seems to have learned lessons from the "best practices" and negative experiences of other ecosystems. While Cargo has recommended the use of semver since its very beginning, in the early days its package maintainers continued to use permissive dependency constraints. The decision of Cargo to remove wildcard constraints in January 2016 led to an important increase in semver-compliance.

In npm, the largest ecosystem we analyzed, package maintainers initially tended to use the restrictive $\sim$ constraint which is not semver-compliant. Two important changes in the ecosystem policy led to increased semver-compliance. In early 2014, the $^\wedge$ constraint was introduced and replaced $\sim$ as the default constraint. In April 2014, the use of 1.0.0 as default package version was encouraged, leading to a slight but continuous increase of the proportion of production releases.

Packagist was found to have an evolutionary behavior similar to npm, with a fairly high proportion of compliant dependencies near the end of the observation period. The progressive introduction of the Composer package manager (from July 2013 to April 2016) lead to an increase of "correctly specified" (i.e., parseable) constraints, which probably allowed maintainers to automatically identify issues with their package dependencies. In mid 2014, several changes were introduced in Composer to address co-installability issues and to better support non-strict constraints and speed up the package installation in presence of such constraints. This coincided with an observed migration from restrictive to compliant constraints.

Rubygems, the oldest considered ecosystems, was created before the introduction of the semver specification. About 39% of all Rubygems packages were created before semver 2.0.0 (compared to only 5.4% in npm, 4.4% in Packagist, and 0% in Cargo). This perhaps explains why Rubygems (in contrast to the other studied ecosystems) lacks an explicit semver constraint operator (cf. Table 2), why package maintainers may not consider semver as the *de facto* versioning scheme for Rubygems, and why we found such a low proportion of compliant constraints.

The same analysis could be replicated on other packaging ecosystems that recommend semver, and the results may differ to a small or big extent from the ecosystems above. For example, the study by Raemakers et al. [15] of the Maven packaging ecosystem revealed that many required packages do not adhere to semver, in that they introduce breaking changes during minor and patch releases. As a consequence, many dependent packages prefer to specify restrictive constraints. This is an important difference with the four ecosystems we studied.

Given the important differences between ecosystems, it would be useful for each ecosystem community to have a very clear and documented policy of how to respect semantic versioning, and of how this differs from other ecosystems. In addition, migration guidelines should be provided to help newcomers get accustomed to these policies, especially if they come from another ecosystem following other rules and strategies.

10.3 The magic zero problem

When analysing semver-compliance for initial development releases (i.e., 0.y.z), we found that packaging ecosystems are generally more tolerant than what the semver specification suggests. Indeed, most ecosystems assume that patches of initial development releases remain backward compatible. This explains why we found a high proportion of pre-1.0.0 dependency constraints that are more permissive than semver (cf. Figure 2). Specifically for Cargo, we observed that the majority of its packages are still in their "initial development" phase, explaining the different behavior w.r.t. the other analyzed ecosystems. How the semantic versioning specification deals with initial development releases has been an active point of discussion, referred to as the *magic zero problem*.[15] Suggestions have been proposed to make the semantic versioning less confusing w.r.t. 0.y.z versions. Unfortunately, these suggestions have not been adopted yet.

10.4 Tool support

Tool support cannot be underemphasized for dependency management. There are already quite a lot of dependency monitoring tools available (e.g., GitLab's integration of Gemnasium [36], GitHub's dependency analysis [37], Tidelift [38] or Greenkeeper [39]). Some of these are, or can be, integrated as part of continuous integration tools. These monitoring tools allow package maintainers to be notified when new releases of their dependencies become available. Some tools help in automatically upgrading to such releases, or to check that the new release does not introduce breaking changes, by automatically executing test suites. For such tools to be effective, package maintainers should provide rigorous and complete test suites. Provided that complete and reliable information is available about bugs and security vulnerabilities in packages, tools could also suggest to adopt specific releases that fix relevant bugs or security issues, even when the imposed dependency constraint does not allow it.

With respect to dependency constraints and semantic versioning, however, more and better tool support is needed. We are not aware of any tool that supports maintainers in choosing appropriate dependency constraints. While there are tools that can be used to detect breaking changes in reusable libraries (e.g., Clirr [40] and Revapi [41] for Java, NDepend [42] for .NET), they primarily focus on changes affecting their (public) API, and therefore cannot report backward incompatibilities due to semantic changes in the library source code. To the best of our knowledge, most tools supporting semantic versioning are either tools that automatically increment version components based on a manually specified list of tagged changes (e.g., semantic-release [43] or GitVersion [44]) or tools that compute the set of versions that are accepted or rejected by a given dependency constraint (e.g., npm semver calculator [45] or Packagist Semver Checker [46]). The only known exception is Dependabot [47], a service that provides a semver "compatibility score" for pairs of consecutive releases. The score is based on the proportion of (a subset of) dependent projects whose test suite did not fail when executed with the new version of the dependency. The tool's accuracy mainly depends on the set of selected dependents and on the quality of their test suites.

The wisdom of the crowds principle presented in Section 9 could be used to suggest to package maintainers which type of dependency constraint to use for depending on required packages. As a proof-of-concept, we implemented an ecosystem-independent command-line prototype tool for such support, relying on the libraries.io API for gathering historical package dependency information. This prototype, available as part of our replication package, accepts the name of a package and displays the proportion of reverse dependency constraints that are compliant, permissive or restrictive on a monthly basis.

While functional, the very slow response time of the API combined with the large size of the dependency graphs makes the prototype tool too slow to be of practical use. The observed performance problems could be overcome by integrating ecosystem-specific support in existing dependency monitoring tools that have fast access to the dependencies and their dependency constraints for all package releases in the ecosystem.

Continuous monitoring of changes in dependency constraints would also help detect when a new release of a required package unexpectedly becomes backward incompatible. This would be the case if many dependents of this required package have decided to update their dependency constraints to exclude the new release or to select it as a new minimal allowed version after having adopted it. Package maintainers could also be informed about which release of a required package is safest to use, by analyzing the proportion of other packages that depend on each release, and how this proportion evolves over time.

While most dependency monitoring tools focus on the dependent package, there are also opportunities for supporting maintainers of required packages. For example, a historical dependency (constraint) analysis may inform them how long it takes for specific releases of their packages to be adopted by dependent packages, and to what extent older releases of a package are still being used and why. Such information will allow them to improve the adoption rate of new package releases, and to decide whether to backport bug and security fixes to previous major releases.

10.5 Future work

The presented empirical study followed a coarse-grained analysis of semver-compliance in package dependency networks. Such a study could be complemented by finer-grained in-depth analyses, that study the specific characteristics and internal details of individual package releases, and how they relate to semver-compliance. Examples of potentially relevant package characteristics would be its number of contributors, its code quality, the presence of bugs, security issues and failed builds. In particular, it would be very interesting to study to what extent semver-compliance relates to software quality and technical debt. Are packages with permissive dependency constraints more subject to such quality issues than those with compliant or restrictive constraints? How does the ecosystem play a role in this?

A finer-grained analysis is also required to understand why and how breaking changes manifest themselves, in order to estimate the effort required to address them, for individual packages as well as for entire package dependency graphs. The main challenge is to come up with

15. github.com/semver/semver/issues/221

solutions that can scale to huge dependency networks such as the one of npm, containing millions of package releases and dependencies, without even considering the transitive dependency relationships. Another challenge is to find efficient analysis algorithms that can be applied over a wide range of highly dynamic programming languages (such as JavaScript, PHP, and Ruby, the main languages for three of the four considered ecosystems).

While the current study focused only on internal dependencies between packages contained *within* the ecosystem, it may be interesting to expand the analysis *beyond* the ecosystem boundaries, similar in spirit to Decan et al. [11]. On the one hand, we aim to study the types of constraints that are used for *outgoing* dependencies from ecosystem packages to external software that is not distributed via the package manager. On the other hand, we would like to study the constraints for *incoming* dependencies from external software to ecosystem packages. Such dependencies are likely to specify other types of constraints than the internal packages, since the packaging ecosystem has little or no control over the release policy of packages it does not host.

11 Threats to validity

We discuss the threats that may affect the validity of our findings, following the structure recommended by Wohlin et al. [48].

Threats to *construct validity* concern the relation between the theory behind the experiment and the observed findings. Our analyses were based on a preparatory parsing step to convert ecosystem-specific constraint notations to a generic version range notation. Since the large majority of constraints could be parsed, this is unlikely to affect our results. On the contrary, the extra parsing phase makes our approach more generic, and hence easy to generalise to other ecosystems. The accuracy of our findings assumes that the package dependency metadata extracted from libraries.io is correct. We manually checked this assumption in previous work [5] relying on the same dataset. Our findings also depend on the "noise" that may be present in the original data provided by the packaging ecosystems. As explained in Section 4.3 for npm, we removed such noise by excluding 52K "spam" packages that do not correspond to real development. For Rubygems, the noise generated by the massive import of over 25K package releases in August 2014 is unlikely to impact our findings as its effect affected only one month of the considered 5-year observation period.

Threats to *internal validity* concern factors internal to the study that could influence the results. The principal threat is our assumption that dependency constraints can be used as a proxy to assess if a maintainer trusts a required package w.r.t. backward compatibility. We believe this assumption to be correct, as dependency constraints are the common (and only) way for maintainers of dependent packages to control and limit their exposure to breaking changes in required packages. Dependency constraints are especially useful to control this exposure when semantic versioning (or another versioning policy that dictates how version numbers should be incremented w.r.t. backward compatibility) is expected to be followed.

Threats to *conclusion validity* concern the relation between the treatment used in the experiments and the actual observed outcomes. Given that our empirical analyses are based on historical observations, they are not affected by such threats.

The threats to *external validity* concern whether the results can be generalized outside the scope of the present study. The proposed approach to assess the semver-compliance of packages is certainly generalizable to other packaging ecosystems since it is mainly observational and based on the wisdom of the crowds principle. The use of a generic version range notation as opposed to relying on ecosystem-specific version contraint notations also makes the approach applicable to other packaging ecosystems. The observed findings themselves, however, are ecosystem-specific, since they are highly dependent on the ecosystem's policies and practices. We already found important differences w.r.t. semver-compliance among the four packaging ecosystems we analyzed, and we expect to see more such differences in other ecosystems. Indeed, not every package ecosystem uses dependency constraints in the same way. For example, the "rolling release" policy of CRAN imposes that a package must always be up-to-date with its dependencies. Any dependency constraint that is not satisfied by the latest available release of a dependency makes the package not installable, even if the new release is compatible. As a consequence, nearly all dependency constraints in CRAN do not specify an upper bound [22] and hence, are not meaningful to assess backward compatibility.

12 Conclusion

Maintainers of software packages in large package dependency networks are frequently confronted with the "dependency hell" of breaking changes because of backward incompatible updates of (transitively) required packages. To cope with this, they need to resort to tools and policies to reduce the exposure to breaking changes while continuing to be able to benefit from bug and security fixes. Semantic versioning (semver) has been proposed as one of the ways to do so.

We empirically studied the degree of semver-compliance in four large packaging ecosystems (Cargo, npm, Packagist and Rubygems), by analysing the dependency constraints in their package dependency network over a five-year time period, considering runtime dependencies only. Dependency constraints were classified in three categories: those that are compliant with the semver specification, those that are more permissive and those that are more restrictive. We generally observed that the proportion of compliant constraints increases over time for all ecosystems, while ecosystem-specific notations, characteristics, maturity and policy changes play an important role in the degree of such compliance. This aligns with the findings of Bogart et al. [4], [9], who observed that different ecosystem communities have different habits and values. For example, constraints in Rubygems are more permissive than for the other ecosystems, indicating that Rubygems does not adhere to the semver specification.

In a similar vein we observed that ecosystems tend to be more permissive than semver for packages during initial development (i.e., releases $0.y.z$), since they assume patch updates to be compliant, whereas the semver specification does not. This is especially relevant for Cargo packages that, because of Cargo's young age, still rely a lot on such initial development releases. For production packages (i.e., releases 1.0.0 or above), the proportion of compliant constraints is high (except for Rubygems) and increasing for all ecosystems. Still, a significant proportion of dependency constraints are too restrictive, preventing the automatic adoption of minor releases and patches.

We assessed and confirmed that the "wisdom of the crowds" principle can be used to allow to decide which type of constraint to use for new dependencies to existing required packages. If the large majority of dependencies to a given required package "agree" on the constraint type they use, this constraint type can be recommended for other packages desiring to depend on the same required package.

These and related results can form the basis for a next generation of semver-aware dependency management tools that can be integrated into existing continuous integration processes. As such, the difficult task for package maintainers to keep their packages up to date will be alleviated.

Acknowledgments

This research was carried out in the context of FRQ-FNRS collaborative research project R.60.04.18.F *SECOHealth* and Excellence of Science project 30446992 *SECO-ASSIST* financed by FWO-Vlaanderen and F.R.S.-FNRS.

References

[1] M. Lungu and M. Lanza, "The small project observatory: a tool for reverse engineering software ecosystems," in *Int'l Conf. Software Engineering*, 2010, pp. 289–292.

[2] S. Jansen, "Measuring the health of open source software ecosystems: Beyond the scope of project health," *Information and Software Technology*, vol. 56, no. 11, pp. 1508–1519, November 2014.

[3] S. Hyrynsalmi, M. Seppänen, T. Nokkala, A. Suominen, and A. Järvi, "Wealthy, healthy and/or happy — what does 'ecosystem health' stand for?" in *Int'l Conf. Software Business*. Springer, 2015, pp. 272–287.

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

[5] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, Feb 2018.

[6] T. Preston-Werner, "Semantic versioning 2.0.0," https://semver.org, June 2013.

[7] C. Bogart, C. Kästner, and J. Herbsleb, "When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies," in *Automated Software Engineering Workshop*, Nov. 2015, pp. 86–89.

[8] B. Ganesan. NPM dependency errors? then you're doing it wrong. [Online]. Available: https://medium.com/netscape/npm-dependency-errors-then-youre-doing-it-wrong-635160a89150

[9] C. Bogart, A. Filippova, C. Kästner, J. Herbsleb, and F. Thung. (2017) Values and practices in 18 software ecosystems. [Online]. Available: http://breakingapis.org/survey/

[10] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "The evolution of project inter-dependencies in a software ecosystem: the case of Apache," in *Int'l Conf. Software Maintenance*, 2013.

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

[12] S. Mostafa, R. Rodriguez, and X. Wang, "Experience paper: A study on behavioral backward incompatibilities of Java software libraries," in *Int'l Symp. Software Testing and Analysis*. ACM, 2017, pp. 215–225.

[13] L. Xavier, A. Brito, A. Hora, and M. T. Valente, "Historical and impact analysis of API breaking changes: A large-scale study," in *Int'l Conf. Software Analysis, Evolution and Reengineering (SANER)*, Feb 2017, pp. 138–147.

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

[15] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic versioning and impact of breaking changes in the Maven repository," *Journal of Systems and Software*, vol. 129, pp. 140 – 158, 2017.

[16] Y. Wang, M. Wen, Z. Liu, R. Wu, R. Wang, B. Yang, H. Yu, Z. Zhu, and S.-C. Cheung, "Do the dependency conflicts in my project matter?" in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 319–330.

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

[18] J. Vouillon and R. Di Cosmo, "Broken sets in software repository evolution," in *Int'l Conf. Software Engineering (ICSE)*. IEEE Press, 2013, pp. 412–421.

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

[20] R. G. Kula, D. M. German, T. Ishio, and K. Inoue, "Trusting a library: A study of the latency to adopt the latest Maven release," in *Int'l Conf. on Software Analysis, Evolution, and Reengineering*, March 2015, pp. 520–524.

[21] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies? an empirical study on the impact of security advisories on library migration," *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, Feb. 2018.

[22] A. Decan, T. Mens, and M. Claes, "An empirical comparison of dependency issues in OSS packaging ecosystems," in *Int'l Conf. Software Analysis, Evolution, and Reengineering*, 2017, pp. 2–12.

[23] J. Cox, E. Bouwers, M. van Eekelen, and J. Visser, "Measuring dependency freshness in software systems," in *Int'l Conf. Software Engineering*. IEEE Press, 2015, pp. 109–118.

[24] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes, "Keep me updated: An empirical study of third-party library updatability on Android," in *ACM Conf. on Computer and Communications Security*, October 2017.

[25] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *International Conference on Mining Software Repositories*, May 2018.

[26] J. M. Gonzalez-Barahona, P. Sherwood, G. Robles, and D. Izquierdo, "Technical lag in software compilations: Measuring how outdated a software deployment is," in *IFIP International Conf. on Open Source Systems*, 2017, pp. 182–192.

[27] A. Decan, T. Mens, and E. Constantinou, "On the evolution of technical lag in the npm package dependency network," in *IEEE International Conference on Software Maintenance and Evolution*, September 2018.

[28] A. Zerouali, J. M. González-Barahona, T. Mens, A. Decan, E. Constantinou, and G. Robles, "A formal framework for measuring technical lag in component repositories – and its application to npm," *Journal of Software: Evolution and Process*, 2019.

[29] Y. M. Mileva, V. Dallmeier, M. Burger, and A. Zeller, "Mining trends of library usage," in *Proceedings of (IWPSE) and (Evol) workshops*. ACM, 2009, pp. 57–62.

[30] A. Nesbitt and B. Nickolls, "Libraries.io open source repository and dependency metadata," 2018. [Online]. Available: https://doi.org/10.5281/zenodo.1196312

[31] O. Aalen, O. Borgan, and H. Gjessing, *Survival and Event History Analysis: A Process Point of View*. Springer, 2008.

[32] E. L. Kaplan and P. Meier, "Nonparametric estimation from incomplete observations," *Journal of the American Statistical Association*, vol. 53, no. 282, pp. pp. 457–481, 1958. [Online]. Available: http://www.jstor.org/stable/2281868

[33] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandinavian Journal of Statistics*, vol. 6, no. 2, pp. 65–70, 1979. [Online]. Available: http://www.jstor.org/stable/4615733

[34] J. E. Hopcroft, R. Motwani, and J. D. Ullman, “Introduction to automata theory, languages, and computation, 2nd edition,” *SIGACT News*, vol. 32, no. 1, pp. 60–65, Mar. 2001. [Online]. Available: http://doi.acm.org/10.1145/568438.568455
[35] F. Ricci, L. Rokach, B. Shapira, and P. B. Kantor, *Recommender systems handbook*. New York; London: Springer, 2011.
[36] (2019) Gemnasium on gitlab. [Online]. Available: https://docs.gitlab.com/ee/user/project/import/gemnasium.html
[37] (2019) Github security alerts for vulnerable dependencies. [Online]. Available: https://help.github.com/en/articles/about-security-alerts-for-vulnerable-dependencies
[38] (2019) Tidelift. [Online]. Available: https://tidelift.com
[39] (2019) Greenkeeper. [Online]. Available: https://greenkeeper.io
[40] (2019) Clirr. [Online]. Available: http://clirr.sourceforge.net
[41] (2019) Revapi. [Online]. Available: https://revapi.org
[42] (2019) NDepend. [Online]. Available: https://www.ndepend.com
[43] (2019) semantic-release. [Online]. Available: https://semantic-release.gitbook.io/semantic-release
[44] (2019) Gitversion. [Online]. Available: https://github.com/GitTools/GitVersion
[45] (2019) Npm semver calculator. [Online]. Available: https://semver.npmjs.com
[46] (2019) Packagist Semver Checker. [Online]. Available: https://semver.mwl.be
[47] (2019) Dependabot SemVer stability score. [Online]. Available: https://dependabot.com/compatibility-score
[48] C. Wohlin, M. C. Ohlsson, P. Runeson, B. Regnell, M. Höst, and A. Wesslén, *Experimentation in Software Engineering*. Springer, 2000.

Alexandre Decan Alexandre Decan conducted his doctoral studies at the University of Mons on the subject of data quality in relational databases. He obtained a PhD in Science in 2013 for the thesis entitled “Certain Query Answering in First-Order Languages”. He is post-doctoral researcher at the Software Engineering Lab of the University of Mons (Belgium), where he has authored many well-cited publications related to the maintenance and evolution of software ecosystems. He has been actively involved in several research projects such as the Action de Recherche Concertée “Ecological Studies of Open Source Software Ecosystems”, the Walloon IDEES project portfolio of the European Regional Development Fund, the joint FNRS-FRQ Québec-Wallonie collaborative research project “Socio-Technical Methodology and Analysis of Software Ecosystem Health” and the joint Belgian FNRS-FWO Excellence of Science project “Automated Assistance for Developing Software in Ecosystems of the Future”. For more information, visit decan.lexpage.net.

Tom Mens Prof. Dr. Tom Mens obtained a PhD in Science in 1999 at the Vrije Universiteit Brussel. He became a lecturer at the University of Mons (UMONS) in October 2003, where he is currently full professor, director of the Software Engineering Lab, and vice-president of the INFORTECH Research Institute. His current research interests include software evolution, quality and health management of software ecosystems, and software modeling. He co-organises the ICSE Software Health (SoHeal) workshops, and co-edited two Springer books “Software Evolution” and “Evolving Software Systems”. He published numerous highly-cited scientific articles in peer-reviewed international software engineering conferences and journals. He was keynote speaker for ICSME 2016 and program chair of ICSM 2013, CSMR 2012 and CSMR 2011. He is project leader of the FNRS-FRQ Québec-Wallonie collaborative research project “Socio-Technical Methodology and Analysis of Software Ecosystem Health” and the joint Belgian FNRS-FWO Excellence of Science project “Automated Assistance for Developing Software in Ecosystems of the Future”. He is a senior member of the IEEE, IEEE Computer Society member, and member of the ACM. For more information visit staff.umons.ac.be/tom.mens.