What Do Package Dependencies Tell Us About Semantic Versioning?

Alexandre Decan
Software Engineering Lab
University of Mons (UMONS)
Mons, Belgium
alexandre.decan@umons.ac.be

Tom Mens
Software Engineering Lab
University of Mons (UMONS)
Mons, Belgium
tom.mens@umons.ac.be

***Abstract*—This presentation abstract reports on the research results published in 2019 in *IEEE Transactions on Software Engineering*.[1] The semantic versioning 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. The article empirically compared semantic versioning compliance of four software packaging ecosystems, and studied how this compliance evolves over time. We explored to what extent ecosystem-specific characteristics or policies influence the degree of compliance. We also proposed 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.**

I. Introduction

Contemporary software development increasingly relies on reusable software packages, stored in open source package registries such as npm, Rubygems, Packagist and Cargo. The dependency networks formed by the packages contained in these registries form so-called *packaging ecosystems*.

Semantic versioning (semver) has been proposed as a solution to the so-called *dependency hell* to which software maintainers in such ecosystems are often confronted. Maintainers of software that depends on reusable packages need to keep their 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 these dependencies, especially if changes are backward incompatible. Providers of reusable packages need to regularly provide new releases with extra functionalities, bug fixes and security fixes to keep their "consumers" satisfied; while they should avoid introducing breaking changes as this imposes a burden on those consumers.

semver partially addresses this challenge by introducing a set of simple rules 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 be imposed, only embraced 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.

The goal of the paper was to assess to what extent maintainers in four packaging ecosystems (Cargo, npm, Packagist and Rubygems) 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. We analysed the dependency constraints in the package dependency networks of the four ecosystems over a five-year time period. Dependency constraints can be either compliant to, more restrictive than, or more permissive than what is suggested by the semver policy. 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.

We observed that constraints in Rubygems are more permissive than for the other ecosystems, suggesting that Rubygems does not adhere to the semver specification. We also observed that ecosystems tend to be more permissive than semver for packages during initial development (i.e., releases with version $0.y.z$), which assume patch updates to be compliant, whereas the semver specification does not. This is especially relevant for Cargo packages that 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.

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 fi- nanced by FWO-Vlaanderen and F.R.S.-FNRS.

[1]https://doi.org/10.1109/TSE.2019.2918315