An Empirical Study of Version Conflicts in Maven-Based Java Projects

Author: Valentin-Vlad Mihăilă[1] (V.V.Mihaila-1@student.tudelft.nl)
Supervisors: Sebastian Proksch[1] Cathrine Paulsen[1]
[1]*EEMCS Faculty, Delft University of Technology, The Netherlands*

Introduction

- Software projects commonly use third-party libraries to increase software quality, developer productivity and reduce development costs [1].
- However, intensive use of libraries may cause what developers call *dependency hell*. One such manifestation are **version conflicts** which occur when a project depends on multiple versions of the same library. In Java projects, only one library version is loaded at runtime, which may lead to **errors or unexpected program behaviour at runtime** when the conflicting versions are incompatible.
- Prior work by Wang et al. [2] studied dependency conflicts in open-source software, but identified only 39 cases of version conflicts targeted at one well-maintained software ecosystem. This narrow scope limits the generalizability of their findings.
- The study fills gaps in existing research by investigating **124 GitHub pull requests** that addressed version conflicts in **85 open-source Java projects** built with Maven.

Research Questions

- **RQ1**: How can we quantitatively measure developer effort spent resolving version conflicts?
- **RQ2**: To what extent does adherence to Semantic Versioning mitigate runtime errors caused by version conflicts?
- **RQ3**: What resolution strategies do developers use to fix version conflicts?

Background

Maven Dependency Resolution Model

In Maven, developers declare third-party libraries as *direct dependencies* in a `pom.xml` file, which often have their own dependencies (called *transitive dependencies*). By default, Maven dependencies are specified using fixed version numbers. This can easily cause version conflicts, especially when dependencies transitively rely on distinct versions of shared third-party libraries (Figure 1).

To mediate version conflicts, Maven applies a "nearest wins" strategy, selecting the dependency that is closest to the root of the dependency tree. However, if the selected dependency lacks functionality required by components that rely on different, incompatible versions of the same library, runtime errors such as `NoSuchMethodError` may be triggered.

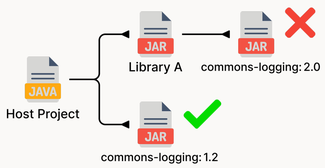


Figure 1. Example of a version conflict caused by distinct versions of a shared third-party library.

The Maven Dependency Plugin [3] offers the `dependency:tree` goal to help developers detect version conflicts. Nevertheless, the output can become very long and the tool does not indicate which conflicts might be harmful.

Semantic Versioning (SemVer)

Proposed as a partial solution to the dependency hell problem, SemVer defines a three-digit versioning scheme meant to encode compatibility promises (Figure 2). According to this scheme, conflicting versions that differ in the `PATCH` or `MINOR` parts should be backward-compatible.

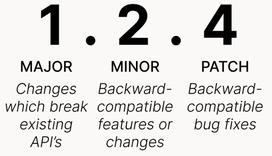


Figure 2. SemVer scheme.

Despite the benefits of SemVer, Raemaekers et al. [4] found that while many Maven Central libraries appear to adhere to SemVer guidelines, breaking changes are sometimes introduced even in minor version updates.

Data Collection

The three-step data collection process (Figure 3) replicates the methodology of Wang et al. [2] to identify documented occurrences of version conflicts in open-source projects, extending it to a larger and more diverse sample.

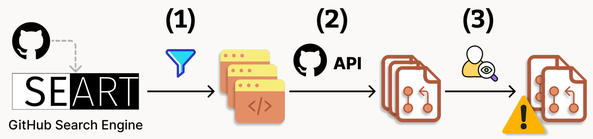


Figure 3. Overview of the three-step process to collect version conflicts.

(1) **5,919 Java projects** created in the last 10 years were selected using the SEART GitHub Search Engine [5], based on popularity and maintainability criteria (≥ 50 stars, ≥ 50 total issues);

(2) **196 merged pull requests (PRs)** were identified using the GitHub REST API [6] based on relevant keywords (e.g., "`NoSuchMethodError`", "version conflict*");

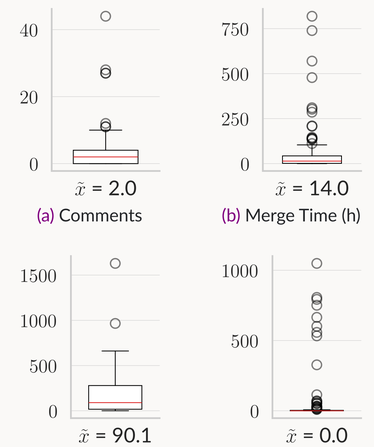
(3) A final dataset of **124 PRs in 85 Java projects** was compiled after manual inspection, excluding false positives and duplicates.

RQ1: Developer Effort

To address understanding gaps about the level of developer effort involved in resolving version conflicts, we investigated four metrics derived from GitHub activity data (Figure 4):

- **Comments**: Most PRs (84.7%) had between 0 and 5 comments. However, 15.4% were identified as bot-generated or automation commands;
- **Merge Time**: The median time to merge was 14 hours, moderately correlated with the number of comments ($\rho = 0.38$, $n = 124$);
- **Detection to Resolution Time**: For 29 PRs linked to issues, the median time from detection to resolution was 90.1 hours, moderately correlated with the merge time ($\rho = 0.52$, $n = 29$);
- **Java Line Changes**: The majority of PRs (71.8%) made no changes to Java source code.

Using z-score normalization, we found that **86.3% PRs were merged more quickly** than other PRs from the same repository. In a subset of 85 PRs, **74.1% PRs also had fewer comments**.



(a) Comments (b) Merge Time (h) (c) Detection to Resolution Time (h) (d) Java Line Changes

Figure 4. Distributions of developer effort metrics derived from GitHub PR activity.

RQ1 Findings: *The four proposed PR activity metrics are not fully reliable, as they may be **affected by unrelated changes, external delays or automation commands**. These findings suggest that purely quantitative measures may be insufficient on their own, highlighting the need for **qualitative validation**.*

RQ2: SemVer Adherence

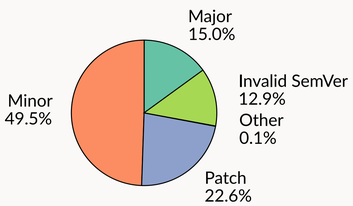


Figure 5. Distribution of semantic differences in 52,417 version conflicts.

To address literature gaps about whether SemVer actually mitigates runtime failures caused by version conflicts, we identified **52,417 version conflicts** across 70 of the 85 Java projects. Each pair of conflicting versions was assigned a **semantic difference**: *Major*, *Minor*, *Patch* or *Other*. Overall, **87.1% of the conflicts adhered to the SemVer syntax** (Figure 5).

A manual inspection of **35 PRs with runtime errors due to version conflicts** revealed 7 cases caused by backward incompatibility between library versions. Notably, three of these instances involved minor or patch differences, indicating **violations of SemVer's backward-compatibility guarantees**.

RQ2 Findings: *Despite widespread use of SemVer syntax, **80% of observed runtime errors resulted from forward incompatibilities** which are not covered by SemVer. Additionally, as highlighted in the three concrete cases, **even SemVer-compliant libraries may break backward-compatibility**.*

RQ3: Resolution Strategies

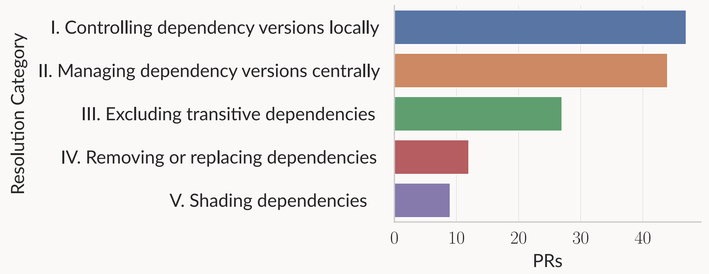


Figure 6. Distribution of resolution strategies used by developers to address version conflicts in 124 PRs from 85 Maven-based Java projects.

To extend existing research into how developers resolve version conflicts [2], we manually reviewed the sample of 124 PRs and identified **five categories of common resolution strategies**, covering **95.2% of the sample** (Figure 6).

RQ3 Findings: ***Library harmonization** (i.e., aligning) of library versions was the most frequently applied strategy (**Categories I and II**), resolving conflicts in **67.7% of the sample**. In addition, we observed a smaller but noteworthy use of **conflict prevention strategies** (such as the Maven Enforcer Plugin [7]) that offer promising opportunities to detect and avoid conflicts earlier in the development process.*

Conclusions and Future Work

- **RQ1**: Quantitative activity metrics alone offer limited insight into developer effort, as they are **often noisy and influenced by confounding factors**. Future work should incorporate **qualitative methods** to validate such effort estimations (e.g., developer surveys [8] or interviews [9]).
- **RQ2**: Our findings suggest that **SemVer's effectiveness is limited in practice** and **relying solely on version numbers is often insufficient to ensure compatibility**. To mitigate such risks, developers should validate selected versions through **compatibility testing**.
- **RQ3**: Our study revealed **library harmonization as the dominant resolution strategy**, supporting the development of version harmonization tools such as `LibHarmo` [8]. Although less frequent in our dataset, conflict prevention strategies show potential for **solving version conflicts before they manifest at runtime**, motivating further research into their adoption and long-term effectiveness.

Ultimately, our study provides real-world insights into version conflicts as a common manifestation of dependency hell and aims to inform the development of more effective dependency management tools and practices. By helping developers better manage project dependencies, we hope to support more reliable and maintainable software.

References

[1] P. Mohagheghi and R. Conradi, "Quality, productivity and economic benefits of software reuse: A review of industrial studies," *Empirical Software Engineering*, vol. 12, no. 5, pp. 471–516, 2007.

[2] Y. Wang *et al.*, "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*, Lake Buena Vista FL USA: ACM, 2018, pp. 319–330.

[3] *Apache Maven Dependency Plugin*, [Online]. Available: `https://maven.apache.org/plugins/maven-dependency-plugin/`.

[4] S. Raemaekers *et al.*, "Semantic versioning and impact of breaking changes in the Maven repository," *Journal of Systems and Software*, vol. 129, pp. 140–158, 2017.

[5] O. Dabic *et al.*, "Sampling projects in github for MSR studies," in *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*, IEEE, 2021, pp. 560–564.

[6] *GitHub REST API documentation*, [Online]. Available: `https://docs-internal.github.com/en/rest?apiVersion=2022-11-28`.

[7] *Apache Maven Enforcer Plugin*, [Online]. Available: `https://maven.apache.org/enforcer/maven-enforcer-plugin/`.

[8] K. Huang *et al.*, "Interactive, effort-aware library version harmonization," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020, New York, NY, USA: Association for Computing Machinery, 2020, pp. 518–529.

[9] I. Pashchenko *et al.*, "A Qualitative Study of Dependency Management and Its Security Implications," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20, New York, NY, USA: Association for Computing Machinery, 2020, pp. 1513–1531.