The Effect of Compatible Version Ranges on Maven Dependency Resolution Stability and Flexibility

Cathrine Paulsen
Delft University of Technology
Delft, The Netherlands

Sebastian Proksch
Delft University of Technology
Delft, The Netherlands

***Abstract*—Dependencies in the JAVA/MAVEN ecosystem are predominantly declared using version pins, which restrict updates and create conflicts when multiple versions of the same dependency are required. Conflicts are mediated automatically using a nearest-first heuristic, providing no compatibility guarantees, or manually by developers, requiring continuous maintenance effort. From an analysis of 266 MAVEN projects on GITHUB, we find that one in four conflicts resolve to versions that may expose clients to future breaking changes, and two in three projects manually mediate conflicts, highlighting the need for *compatibility-aware* conflict mediation. Compatible version ranges could enable such mediation, but their effectiveness depends on how compatibility is determined. We investigate how replacing version pins with ranges derived from client-agnostic compatibility detection affects resolution outcomes. We compare ranges derived from *semantic versioning* (SEMVER), inferring compatibility from version numbers, with analysis-based ranges, validating API and behavioral compatibility through static and dynamic analysis. We find that SEMVER-based ranges are flexible but often introduce breaking changes, whereas the analysis-based ranges provide stronger stability but limited flexibility. These findings reveal an inherent stability-flexibility trade-off in client-agnostic compatible range generation, motivating hybrid approaches leveraging client context to achieve a better balance.**

***Index Terms*—dependency management, compatibility detection, compatible version range generation**

I. INTRODUCTION

The reuse of open-source software components as dependencies is widespread in modern software development [1]. Keeping these dependencies up-to-date is crucial, as new releases may include important bug fixes and security patches. To facilitate updates, a dependency manager should ideally resolve dependencies in a way that is *stable* to avoid introducing breaking changes [2], *flexible* to reduce the risk of version conflicts and resolution failure [3, 4], and *transparent* by providing full and accurate dependency trees so that dependency-related issues can be traced [5]. A *reliable dependency resolution* satisfies these properties, but current dependency management practices often fail to provide all three.

The reliability of the resolution process largely depends on how dependencies are declared. Restrictive *version pinning* prevents updates [2, 6] and reduces the risk of breaking changes, but leads to outdated and vulnerable dependencies downstream [3, 4]. In contrast, open *version ranges* are more permissive, increase resolution flexibility through automatic updates [3], but may introduce breaking changes. Developers struggle to balance the stability of pinning and the flexibility of ranges [7], but generally favor stability [6, 8].

Semantic Versioning (SEMVER) [9] communicates compatibility through version numbers (`MAJOR.MINOR.PATCH`) where updates within the same `MAJOR` version should be backwards compatible. SEMVER implies *compatible version ranges* and tries to foster automated and compatible updates, with the goal to improve resolution flexibility without compromising stability. Unfortunately, MAVEN packages often break this convention [10, 11]. 99% of MAVEN dependency declarations use version pins [4], perhaps due to the absence of a practical way to derive accurate compatible version ranges.

MAVEN-specific challenges add to the general problem of dependency management. First, the widespread *pinning* frequently causes conflicting version declarations for the same dependency [12]. MAVEN mediates these conflicts heuristically by resolving the nearest version declaration, which may introduce breaking changes only detectable during runtime [13]. Second, MAVEN allows the undeclared use of transitive dependencies, which complicates the tracing of dependency-related issues and prevents timely vulnerability discovery [4, 5]. Together, these issues reduce the stability, flexibility, and transparency needed for reliable dependency resolution.

Existing work addresses dependency issues primarily *after* resolution, for example, by detecting whether a dependency update is compatible. These compatibility detectors include dynamic analyses using client tests to identify behavioral incompatibilities [14–17], and static analyses using bytecode, AST, or call-graph differencing to identify API-breaking changes [2, 11, 18]. Many of these approaches are *client-specific*, analyzing compatibility from a single client's perspective and therefore require recomputation for every client-dependency pair. In contrast, recent research took a *client-agnostic* perspective to generating compatible version ranges [19]. The resulting MARCO tool combines static bytecode differencing and dynamic cross-version testing, using the dependency's tests instead of the client's, to replace version pins with compatible version ranges.

In this paper, we investigate how compatible version ranges affect the stability and flexibility of dependency resolution in MAVEN projects, where version pinning is the predominant dependency declaration strategy. Version pins can lead to conflicts that MAVEN resolves using its nearest-first heuristic, which provides no compatibility guarantees, or manual

overrides, which improve stability but require continuous developer effort to maintain. Replacing pins with compatible version ranges enables automatic compatibility-aware conflict mediation, but the effectiveness of this approach depends on how compatibility is determined. To understand how conflicts are currently mediated and evaluate compatible version ranges derived from client-agnostic compatibility detection as a scalable alternative, we address the following research questions:

RQ_1: How are version conflicts mediated in Maven, and how stable are the resulting resolutions under SemVer*?*

RQ_2: How effective is client-agnostic compatibility detection at detecting breaking changes?

RQ_3: How do client-agnostic compatible version ranges affect Maven*'s dependency resolution?*

We find a quarter of version conflicts in Maven projects are resolved to versions that are incompatible with the other declared versions under SemVer assumptions, indicating that current mediation strategies do not reliably produce compatibility-preserving resolutions and can therefore limit resolution stability. Both client-agnostic strategies (SemVer, MaRCo) identify breaking changes in clients with a *comparable* F1 score of 0.70, but with *stark differences* in recall and precision: SemVer-based ranges are wide but often include breaking changes, whereas MaRCo-based ranges avoid breaking changes at the cost of being much narrower. We conclude that automated detection of compatible version ranges can address resolution reliability, but a perfect balance between stability and flexibility is not possible with client-agnostic detection alone. Instead, it offers a scalable foundation for hybrid compatibility approaches combining client-agnostic and client-specific analysis to widen ranges without sacrificing stability and scalability.

Overall, this paper presents the following main contributions:

- We show that conflicts are commonly manually mediated and that, even under ideal SemVer assumptions, a quarter of conflict resolutions are unstable.
- We show that client-agnostic compatible version ranges can improve resolution flexibility or stability, but not both.
- We discuss challenges and future directions for hybrid compatibility detection that is scalable *and* precise.

All data and scripts used in this paper are available in our online replication package [20].

II. Related Work

Verifying compatibility between two versions requires checking binary, source, and behavioral compatibility [21]. Binary and source compatibility, known as static, syntactic, or API compatibility, is detectable at compile time by static analysis tools like japicmp [22] and revapi [23] which use bytecode differencing. In contrast, behavioral, dynamic, or semantic incompatibilities occur at runtime and are difficult to detect statically [18]. Since formal behavioral specifications are rare [24], tests are often used to approximate behavioral compatibility.

CompCheck [17] builds a knowledge base of client-dependency incompatibilities using client tests and control flow graphs to check whether a client is affected by the incompatibilities, showcasing two common techniques in compatibility checkers. *Cross-client testing* uses multiple clients' tests motivated by the often low dependency coverage of single-client testing. *Reachability analysis* uses control flow or call graphs to determine whether a breaking change is reachable by a client to ensure that the compatibility decision is client-specific. Mujahid et al. [15] and DeBBI [14] identify breaking changes using cross-client testing, prioritizing test suites with high dependency usage to reduce computational cost. Dependabot estimates compatibility from the fraction of clients with passing builds after an update, although developers found high-quality test suites more helpful [16]. Ranger [4] replaces version pins by safe, compatible ranges to address vulnerability propagation in Maven, detecting breaking changes using bytecode differencing, Sembid, client tests, and client-specific reachability analysis. Sembid [18] detects client-agnostic behavioral breaking changes statically by call graph differencing and heuristic pattern matching, but cannot detect all breaking changes detected by client tests. Hejderup et al. [2] found that client tests typically have low dependency coverage and introduced Uppdatera which uses AST differencing and reachability analysis to detect client-specific behavioral breaking changes statically. Maracas [11] extends japicmp by using annotations to exclude internal parts of the API to reduce false positives.

In addition to compatibility types, there are different compatibility perspectives: *client-specific* and *client-agnostic*. The client-specific perspective looks at the question of whether two dependency versions are compatible by analyzing the client(s) using the dependency. There may be incompatibilities that the client never calls, in which case the versions are considered compatible. Client-agnostic approaches, in contrast, assess compatibility independently of the client code. Maracas and Sembid are client-agnostic, the former considering only static compatibility and the latter approximating behavioral compatibility via static analysis. Uppdatera and Ranger combine client-specific and -agnostic approaches, and approximate dynamic compatibility using static analysis. *Cross-version testing* using the dependency's tests instead of the client's is a client-agnostic approach to behavioral compatibility. Mostafa et al. [25] used this method to collect incompatibilities, suggesting it may be suitable for detection, and Ochoa Venegas [26, p. 208] speculates it may address scalability issues in client-specific approaches. MaRCo provides a client-agnostic detector using japicmp to statically detect API incompatibilities and cross-version testing to dynamically detect behavioral incompatibilities by fetching dependency tests from their GitHub repositories, but its effectiveness is unclear.

In summary, existing work provides methods for detecting incompatible dependency updates, primarily focusing on client-specific analyses that determine compatibility from an individual client's perspective. While such analyses can

achieve high precision, they suffer from limited scalability and recall due to incomplete coverage as many clients do not test their dependencies [2, 17]. Client-agnostic approaches address scalability but have seen little exploration beyond static analysis or lack large-scale empirical validation.

In this paper, we complement existing work by evaluating the potential of client-agnostic compatibility detection for improving dependency resolution reliability. Unlike prior studies, we are not focusing on detecting breaking changes after resolution. We want to integrate compatibility information directly into the resolution by considering compatible version ranges. We compare two client-agnostic strategies: SEMVER which infers compatibility from version numbers, and MARCO which infers compatibility through static and dynamic analysis, and assess their trade-offs. We did not consider SEMBID as it is not publicly available, nor MARACAS since it uses JAPICMP which is also the basis of MARCO's static compatibility analysis.

III. STUDY OVERVIEW

This paper combines an empirical study of conflict mediation in the MAVEN ecosystem with an evaluation of how compatible version ranges derived from two client-agnostic compatibility detectors affect dependency resolution in real-world MAVEN projects. Figure 1 provides a high-level overview of the study. We collected JAVA projects built with MAVEN from GITHUB created between January 1, 2023 and May 1, 2024 with $\geq$10 stars and $\geq$50 commits, resulting in 735 projects ①. The timeframe ensures the projects had time to establish dependency management practices and reflect contemporary practices. The star and commit thresholds follow related work to exclude trivial projects while maintaining diversity [4, 27].

In Step ②, we analyze how conflicts are mediated in practice and assess the stability of the resulting resolutions under SEMVER assumptions (RQ_1, Section IV). For this analysis, projects must compile, leaving 446 eligible projects. Of these, 266 have dependencies and form the dataset for RQ_1. To explore the effect of replacing pins with compatible version ranges, we leverage the MARCO toolkit [19] which we adapt for large-scale range replacement and extend with an additional SEMVER-based compatibility detector as a baseline (Section V). In Step ③, we evaluate the MARCO and SEMVER detectors on existing datasets of (non-)breaking dependency updates to assess how likely they are to produce ranges that exclude incompatible and include compatible versions (RQ_2, Section VI). Finally, Step ④ replaces version pins in the projects with the detected compatible version ranges and compares the resolution outcomes before and after replacement to determine how the different ranges affect resolution stability and flexibility (RQ_3, Section VII). Stability regressions are detected by executing project test suites before and after replacement. Of the 266 compiling projects with dependencies, 105 have runnable test suites and form the RQ_3 dataset.

The paper concludes with a discussion of our findings, the potential and limitations of client-agnostic compatible version ranges, and directions for hybrid compatibility approaches to improve reliable dependency resolution (Section VIII).

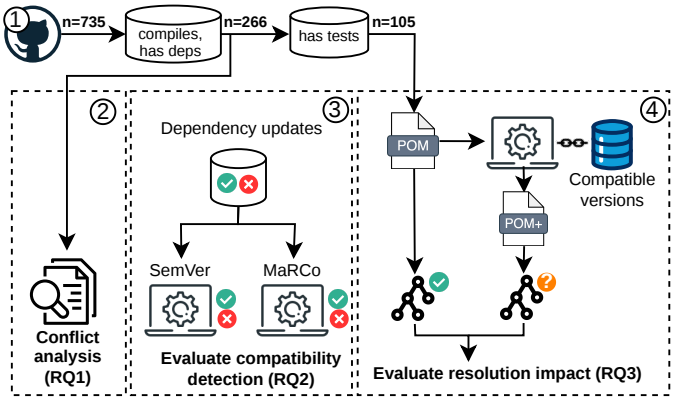


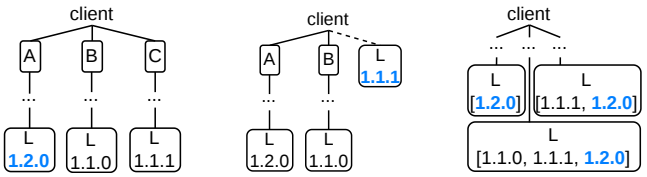
Fig. 1. Study overview.

IV. CONFLICT MEDIATION IN MAVEN (RQ_1)

Version conflicts are a common consequence of dependency reuse, as a project's dependencies may require different versions of the same library. MAVEN must mediate these conflicts by resolving a single version. Figure 2 illustrates the different conflict mediation modes in MAVEN, which depend on which dependency declaration strategies are involved. A dependency is identified by its *groupId* and *artifactId* (GA), while a specific release of a library is identified by *groupId*, *artifactId*, and *version* (GAV). We use the terms *dependency* and *library* interchangeably to refer to the GA.

A *version conflict* occurs when there are at least two different version pins (GAVs) declared for the same dependency (GA) in a project's dependency tree. These conflicts are *unmanaged* by default, and Figure 2(a) shows how MAVEN applies its default mediation strategy by selecting the version declared closest to the root of the dependency tree in breadth-first order [13]. This resolution mode is structurally fragile, as changes to unrelated dependencies may alter the dependency tree and cause a different version to be resolved. A conflict becomes *managed*, see Figure 2(b), when developers override MAVEN's default mediation outcome in two ways: (i) A `dependencyManagement` section directly overrides the resolved version or (ii) an unused conflicting dependency is declared as a direct dependency to exploit the nearest-first mediation. Managed conflicts are not sensitive to dependency tree changes, improving resolution stability, but require continuous maintenance. Finally, although version ranges are rarely used in MAVEN, they provide an alternative mode to avoid version conflicts. When different version ranges are declared for the same library, MAVEN resolves the latest version in the intersection of all ranges, see Figure 2(c). If no intersection exists, resolution fails. Replacing version pins with version ranges containing compatible versions could therefore enable compatibility-aware conflict mediation by resolving the latest mutually compatible version automatically.

As illustrated in Figure 2, conflicts may be mediated automatically using nearest-first resolution or manually through explicit overrides. Nearest-first mediation is structurally fragile and may resolve versions incompatible with other version pins. Manual overrides improve stability by enforcing a fixed



(a) Nearest-first (b) Manual override (c) Ranges

Fig. 2. Available resolution modes in Maven. Each diagram shows a dependency tree of a client with transitive dependencies on library L. Boxes denote dependency declarations and the resolved version is highlighted in blue. A dotted edge indicates a manual override by the client to enforce a specific version. Version ranges are represented by brackets.

version but require continuous effort to maintain. Compatible version ranges provide a promising alternative by allowing MAVEN to automatically resolve mutually compatible versions without manual intervention; however, their effectiveness depends on whether compatible ranges exist that can reconcile conflicting declarations. To understand how these trade-offs between stability, flexibility, and developer effort affect dependency resolution in practice, this section analyzes version conflicts caused by version pins in real-world MAVEN projects, how often they are manually mediated, and whether the resulting resolutions are stable under SEMVER assumptions. While SEMVER assumptions often do not hold in practice [10], they provide a theoretical best-case estimate of the stability of Maven's conflict mediation outcomes.

We used the *Maven Dependency Plugin* (MDP) on the 266 projects that compiled successfully and declared at least one dependency. These projects had, on average, 52.9 resolved dependencies and 61.4 unique version pins.

A. Prevalence and Mediation of Conflicts

To understand how version conflicts are handled in practice, we first analyze their prevalence and how they are mediated in real-world MAVEN projects. In particular, we distinguish between conflicts resolved by MAVEN'S nearest-first strategy and those manually mediated by developers. The prevalence of conflicts and the extent of manual mediation provide insight into the limitations of MAVEN'S default conflict mediation.

Methodology. We use MDP to extract information about the dependency declarations. The `tree` goal [28] outputs the resolved dependency tree and flags certain dependency declarations with additional information. We are interested in the flag *omitted by conflict*, which indicates a concrete dependency declaration was omitted due to a conflict with a previous declaration. This flag will only appear for version pins (i.e., *soft constraints*), as MAVEN will fail resolution when version ranges (i.e., *hard constraints*) cannot be fulfilled.

To determine whether conflicts have been manually mediated, we look for evidence of the two manual mediation strategies previously mentioned. First, we check the output of the `tree` goal: entries marked as *managed from* indicate that the `dependencyManagement` section has been used to override the resolved version. Second, developers may declare an otherwise unused direct dependency to exploit the nearest-first mediation. We detect these cases with the `analyze` goal by identifying the *unused declared* dependencies that also appear among the detected conflicts.

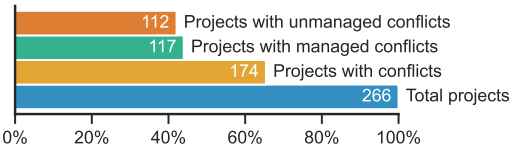


Fig. 3. Prevalence of projects with managed and/or unmanaged conflicts.

Results. Figure 3 shows how many of the 266 projects have at least one managed and/or unmanaged conflict. 65% of the projects had conflicts, with on average 13.0 conflicting version pins (GAVs) over 11.4 dependencies (GAs). We also see that projects tend to have many instances of conflicting version pins (13.0 per project), but each conflict is rather small in scope (1.1 conflicting pins per GA). 67% of projects with conflicts perform manual mediation of at least one conflict. On average, projects using manual mediation have more conflicting pins (12.2) than projects with unmanaged conflicts (7.4). These results suggest that MAVEN's nearest-first mediation becomes less effective as dependency complexity increases, leading to more frequent manual intervention to maintain stable resolutions.

B. Conflict Resolution Stability under SemVer

Conflict mediation may not result in a resolved version that is compatible with all conflicting pins. These incompatibilities can introduce latent breaking changes that may surface as the project evolves, requiring developer effort to debug and resolve. To assess this risk, we analyze conflict mediation outcomes under SEMVER compatibility assumptions. Specifically, we evaluate (i) whether the resolved version is *SemVer-stable* with respect to all conflicting pins, and (ii) whether the conflict is *SemVer-solvable*, meaning that a *SemVer-stable* resolution exists. We use SEMVER as a theoretical best-case baseline of the stability of the current resolution outcome and the ability of SEMVER to provide stable resolutions. In practice, the actual resolution stability and solvability are likely worse.

Methodology. We represent a version as $v = (M, m, p)$, denoting its major, minor, and patch components, respectively. For each dependency, we form pairs of the *resolved version* $v_r = (M_r, m_r, p_r)$ and its *conflicting version pins* $v_c = (M_c, m_c, p_c)$ and measure whether the current resolution outcome is stable under SEMVER assumptions, and whether a stable resolution outcome even exists.

A conflict resolution is SEMVER-*stable* if the resolved version v_r is within the same non-zero major version as all conflicting pins v_c and is not a minor downgrade: $\forall v_c$: $M_r = M_c \land M_r > 0 \land m_r \geq m_c$. This definition follows the official SEMVER definition, where minor upgrades should preserve compatibility while downgrades may remove functionality [9]. Similarly, a conflict is SEMVER-*solvable* if all the conflicting pins v_c and the resolved version v_r are within the same non-zero major version: $\forall v_c$: $M_r = M_c \land M_r > 0$.

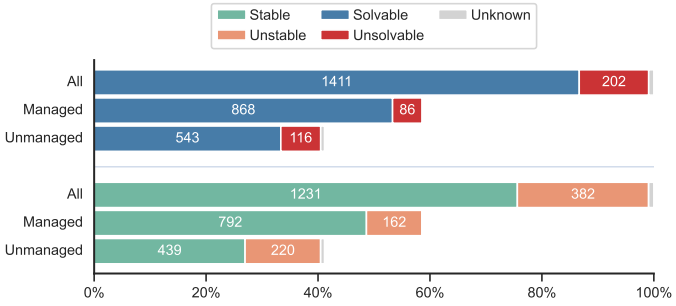


Fig. 4. Distribution of SEMVER solvability (top) and stability (bottom) of the different conflict types.

If this is the case, a SEMVER-stable resolution exists, i.e., the latest version in the common major. Under ideal SEMVER compliance, SEMVER-based ranges that include upgrades within a major version would allow MAVEN to automatically resolve the stable outcome.

Results. Figure 4 shows how many of the 670 unmanaged and 958 managed conflicts are SEMVER-solvable and resolved in a SEMVER-stable way. Out of the total 1628 conflicts, 382 (23%) resolved to versions that are unstable even under ideal SEMVER assumptions. These unstable resolutions represent potential compatibility risks, as the resolved version is not guaranteed to be compatible with all conflicting pins. Future modifications to the dependency tree or project code may therefore expose currently latent breaking changes that require manual conflict mediation or refactoring efforts to fix.

Furthermore, unmanaged conflicts are more likely to result in unstable resolutions than managed conflicts (33% vs. 17%). This suggests that MAVEN 's default nearest-first mediation frequently selects versions that are less compatible under SEMVER than those chosen through manual mediation, highlighting the limitations of automatic conflict resolution. Notably, 12% of conflicts involve version pins across different major versions, making them inherently unsolvable using SEMVER-based version ranges, even under perfect SEMVER compliance.

Overall, these findings show that a significant fraction of conflict mediation outcomes introduce potential compatibility risks, and that while manual mediation improves stability, it requires ongoing developer effort. This highlights both the limitations of conflict mediation in MAVEN and the need for automated approaches capable of deriving compatible version ranges beyond SEMVER.

> **Answer RQ$_1$:** Conflicting version pins are common in MAVEN, affecting 65% of projects, and one in four conflicts resolve to versions that may expose clients to future breaking changes under SEMVER assumptions. This indicates that most projects have some degree of reduced resolution reliability and risk future dependency-related issues.

V. EXPERIMENTAL SETUP

RQ$_1$ shows that reliable dependency resolution in the MAVEN ecosystem is often undermined by conflicting version pins. These conflicts are mediated using compatibility-agnostic heuristics or manual overrides, which can introduce instability or require continuous maintenance effort, while limiting flexibility by preventing updates. Replacing *version pins* with *version ranges* changes MAVEN's resolution behavior. While MAVEN resolves pins using a heuristic nearest-first strategy, it resolves ranges by selecting the latest version in the intersection of all version ranges. This behavior makes *compatible* range replacement a promising approach for achieving stable and flexible dependency resolution, provided that compatibility can be determined effectively. To explore the impact of compatible version ranges on resolution, we utilize the MARCO toolkit [19] as our experimental framework, which replaces version pins with compatible version ranges.

A. MaRCo as Experimental Framework

Internally, MARCO has two main components: the REPLACER and the GENERATOR. The REPLACER replaces a project's POM file, where its dependencies are declared, with a modified POM containing the replaced dependency declarations. To perform the replacement, the REPLACER requires a mapping from dependency versions to compatible ranges. The GENERATOR computes and stores these mappings. The components can be used independently or together, allowing us to study both compatibility detection (RQ$_2$) and range replacement (RQ$_3$) with the same framework. Figure 5 shows how we apply the MARCO components in our experiments.

By default, the mappings are generated using a client-agnostic compatibility detection strategy, which uses JAPICMP to ensure API compatibility and cross-version testing to verify behavioral compatibility. For RQ$_2$, we extend the GENERATOR with a SEMVER-based strategy and apply MARCO with both strategies on a dataset of (non-)breaking dependency updates to evaluate how well they identify breaking changes. For RQ$_3$, we combine the GENERATOR and the REPLACER to perform full range replacement in real-world MAVEN projects, once with each detector, to assess how the MARCO- and SEMVER-based compatible version ranges affect the resolution process.

B. Extending the Generator with SemVer

To assess the viability of client-agnostic compatibility detectors to derive compatible version ranges, we extend the GENERATOR with a SEMVER-based compatibility detection strategy. The SEMVER-based detector uses the SEMVER library [29], closely following the official SEMVER definition [9]. While the GENERATOR can be extended with any compatibility detector, we do not consider client-specific ones as they require recomputation for every client–dependency pair. Client-agnostic detectors like MARCO and SEMVER instead enable the precomputation of reusable compatibility mappings, which is essential for the large-scale range replacement in RQ$_3$.

C. Enabling Complete Range Replacement

To assess how compatible version ranges influence the resolution process, we need to replace all version pins declared in each project's dependency tree. This requires modifying both direct and transitive dependency declarations, which MARCO

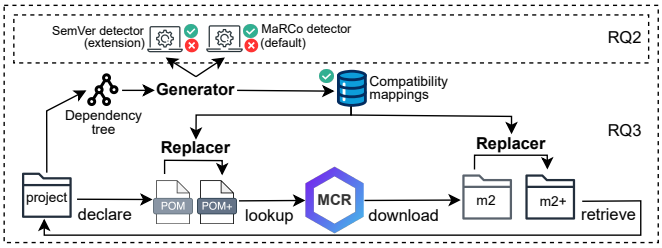


Fig. 5. Application of MARCO in RQ_2 and RQ_3

does not support out-of-the-box. We therefore need to extend our experimental setup to apply MARCO at every step of the resolution process.

A project declares its direct dependencies in its POM file. During resolution, MAVEN retrieves the POMs of all referenced dependencies, direct and transitive, from the MAVEN Central Repository (MCR) and caches them in the local `m2` folder. To perform our large-scale range replacement study, we leverage this behavior when applying the REPLACER to replace: i) the direct dependency declarations in a project's POM, and ii) the transitive dependency declarations in the POMs in the `m2` folder. The REPLACER requires compatibility mappings generated by the GENERATOR, which we apply separately with both detectors to each referenced dependency in each project's (verbose) dependency tree. This setup enables us to directly compare the compatibility detectors on a large scale to assess how client-agnostic range replacement affects resolution.

VI. BREAKING CHANGE DETECTION (RQ_2)

We use the MARCO and SEMVER strategies to compute compatibility mappings to derive compatible version ranges. To assess how likely the ranges are to exclude incompatible and include compatible versions, we evaluate how well the detectors identify breaking changes. We apply the detectors to library version pairs labeled compatible or incompatible, reporting recall, precision, and F1-score. High recall indicates the generated ranges correctly exclude most incompatible versions, whereas high precision indicates the ranges correctly include most compatible versions.

A. Datasets

We evaluate the detector's performance on two datasets containing compatible and incompatible library version pairs: a small manually verified dataset and a larger quantitative dataset. We treat incompatible pairs as the positive class. The labels reflect client-specific compatibility, meaning an update that is incompatible for one client may be compatible for another, and vice versa. Because the detectors are client-agnostic, we therefore do not expect perfect detection accuracy. Instead, we want to quantify the trade-off between stability (recall) and flexibility (precision) that each detector offers to understand how conservative or permissive the generated ranges will be.

Manually verified dataset. The UPPDATERA [2] dataset contains 19 library version pairs, including a manually verified ground truth, project test results, and evaluations of the static client-specific UPPDATERA detector. Although it enables comparison with high-quality, manually verified ground truth and to client-specific detection, the small sample size limits the generalizability of the performance metrics.

Quantitative dataset. To the best of our knowledge, there is no single dataset that includes both compatible and incompatible library version pairs and covers both API and behavioral compatibility. We therefore construct a HYBRID dataset by combining the BUMP [27] and DEPENDABOT [19] datasets.

The BUMP dataset contains 571 incompatible library version pairs from Dependabot pull requests (PRs) where the update caused build failures. Datapoints are labeled with a type and failure category. We discard datapoints with the *plugin* type because MARCO does not replace plugin declarations, and *POM* type because these may update multiple dependencies, making the cause of failure ambiguous. After filtering, 372 incompatible version pairs of 112 libraries from 114 projects remain. The DEPENDABOT dataset contains 481 compatible version pairs of 167 libraries from accepted Dependabot PRs that were merged without changes in 43 projects. These datasets form the HYBRID dataset, enabling a more comprehensive evaluation of breaking change detection performance.

B. Methodology

Each datapoint represents an update of a specific dependency (GA), from an old version v_{old} to a new version v_{new}, forming the tuple (GA, v_{old}, v_{new}). We provide the tuple as input to the GENERATOR, once with the default MARCO detector and once with the SEMVER detector. The output may be compatible, incompatible, or inconclusive. Because the MARCO detector requires library JARs to verify API compatibility, and runnable library tests to verify behavioral compatibility, it may be unable to evaluate some datapoints. To ensure fair comparison, we compute precision, recall, and F1-score using only the datapoints with conclusive results for both detectors: 349 (94%) of the BUMP datapoints, 321 (67%) of the DEPENDABOT datapoints, and 9 (47%) of the UPPDATERA datapoints. We oversample the 321 conclusive DEPENDABOT datapoints by 8.7% to ensure equal representation of compatible and incompatible updates in the HYBRID dataset.

C. Results

Figure 6 summarizes the performance of the MARCO and SEMVER detectors in terms of recall, precision, and F1-score. For the UPPDATERA dataset, we also include the performance of the UPPDATERA tool and the project tests for comparison. Figure 7 shows the reasons for inconclusive results. On the UPPDATERA dataset, half of these were caused by failing to locate a MAVEN-based GITHUB repository for the dependency, and half by compilation failures preventing the behavioral compatibility check. On the HYBRID dataset, 38% of inconclusive results were due to failing to locate repositories, 35% due to repositories not using MAVEN as their build system, and 26% due to no (runnable) tests preventing the behavioral compatibility check.

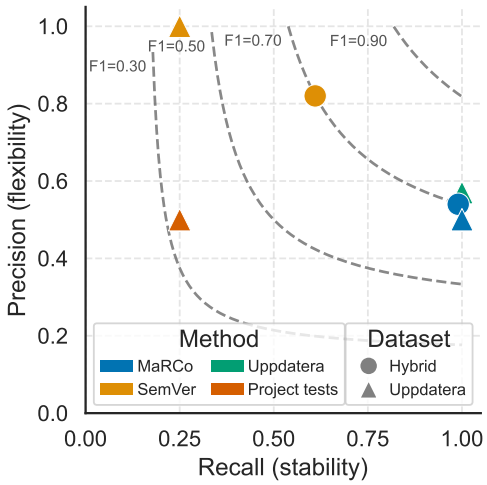


Fig. 6. Recall-precision trade-off for each detector.

Hybrid dataset. The SEMVER detector labeled 62% of incompatible and 87% of compatible updates correctly, resulting in 0.62 recall, 0.81 precision, and 0.70 F1-score. The default MARCO detector labeled 99% of incompatible and 16% of compatible updates correctly, resulting in 0.99 recall, 0.54 precision, and 0.70 F1-score.

The two incompatible updates labeled incorrectly by MARCO as compatible were the update of `org.codehaus.plexus:plexus-io` from 3.2.0 to 3.3.0 and `net.minidev:json-smart` from 2.4.8 to 2.4.9. Manual inspection revealed these were examples of dependency maintainers introducing (un)intentional changes not covered by tests and caused runtime failures in the dependents. Specifically, `plexus-io:3.3.0` introduced a breaking bug [30], which was fixed in 3.3.1 and a regression test was added [31]. The `json-smart` update showcases a similar problem: 2.4.9 introduced a depth limit which was not covered in 2.4.8's tests [32], which was surpassed in one of the clients [33]. This supports Hyrum's law [34], illustrating that bug fixes can break clients relying on the buggy behavior.

Uppdatera dataset. The SEMVER detector identified 1/4 incompatible and 5/5 compatible updates correctly, achieving 0.25 recall, 1.00 precision, and 0.40 F1-score. The default MARCO detector indentified 4/4 incompatible and 1/5 compatible updates correctly, achieving 1.00 recall, 0.50 precision, and 0.67 F1-score. Comparatively, UPPDATERA identified 4/4 incompatible and 2/5 compatible updates correctly, achieving 1.00 recall, 0.57 precision, and 0.73 F1-score. Project tests identified 1/4 incompatible and 4/5 compatible updates correctly, achieving 0.25 recall, 0.50 precision, and 0.33 F1-score. Being a client-specific compatibility approach, it is unsurprising that UPPDATERA performs the best as it is able to achieve a higher precision than MARCO by considering client reachability of the detected incompatibilities. MARCO performs similar to UPPDATERA, however, despite being client-agnostic, suggesting that adding client-specific reachability could further improve precision.

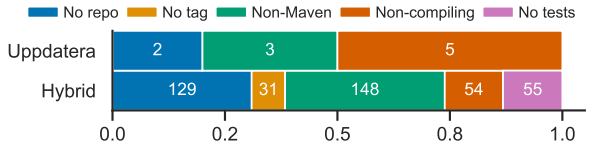


Fig. 7. Causes of inconclusive MARCO evaluations.

D. Conclusions

Across datasets, MARCO consistently achieved high recall and low precision, while SEMVER exhibits the opposite trend with low recall and high precision. MARCO-generated ranges are therefore unlikely to contain breaking changes but may exclude some compatible versions, resulting in ranges that are stable but inflexible. Conversely, SEMVER-generated ranges are likely to include more compatible versions but may also include breaking changes, resulting in ranges that are unstable but flexible. The SEMVER results further confirm previous findings [10, 11] that although updates across major versions are typically breaking, many breaking changes still occur within the same major version. Overall, both client-agnostic detectors are just as effective at breaking change detection in terms of F1-score, but differ significantly in their stability-flexibility trade-off. We will further investigate the impact this has on dependency resolution in RQ_3.

> **Answer RQ_2:** Both client-agnostic detectors achieve similar overall effectiveness with an F1-score of 0.70, but display opposite trade-offs in terms of recall and precision: MARCO showed consistently high recall and low precision, while SEMVER showed high precision and low recall.

VII. Impact on Maven's Resolution (RQ3)

This section investigates how client-agnostic compatible version range replacement affects MAVEN's dependency resolution by comparing the dependency trees of 105 open-source GITHUB projects before and after replacing their version pins with ranges derived from the MARCO- and SEMVER-based detectors. As shown in RQ_2, both detectors have the same F1-score for detecting breaking changes, but represent opposite ends of the precision-recall spectrum. We expect the generated ranges to exhibit a similar trade-off between resolution stability and flexibility.

Methodology. We apply the MARCO framework to the 105 projects with dependencies that compile and have tests, allowing us to verify whether API or behavioral breaking changes are introduced, as follows.

1) Generate each project's dependency tree, which downloads all required dependencies into the `m2` folder.
2) Generate compatibility mappings for each dependency declaration in the dependency trees using the GENERATOR.
3) Replace project and dependency POMs with the REPLACER using the previously generated mappings.
4) Re-compile the projects, re-run the tests, and re-generate the dependency trees. The trees, compilation, and test results are then used for evaluation.

Because dependency trees may change after replacement, pulling in new (versions of) dependencies, steps 1-3 are repeated until the resolution stabilizes. The entire process is performed twice: once with the GENERATOR configured to use the MARCO detector, and once with the SEMVER-based detector. To ensure there is no cross-contamination between experiments, each run starts with an empty `m2` folder.

It is not guaranteed that the MARCO framework replaces every declaration. The MARCO detector can only replace dependencies for which both API and behavioral compatibility can be verified, and the SEMVER-based detector requires version strings to follow the SEMVER format.

Unfortunately, MAVEN tools also fetch dependencies of their build plugins from the same `m2` folder. We therefore prevent replacements for the `plexus-utils`, `commons-collections`, and `velocity` packages, and for any POMs in `org.apache.maven` to prevent breaking the MAVEN tooling.

Evaluation metrics. To quantify how range replacement affects dependency resolution, we define the following metrics.

Success rate. Fraction of projects with no regressions after replacement, i.e. the projects still resolve, compile, and pass their tests.

Replacement rate. Fraction of resolved dependencies that originate from a replaced dependency declaration.

Downgrades/Upgrades. The number of dependencies for which a lower/higher version is resolved.

Downgrade/upgrade steps. The number of versions the newly resolved version is behind/ahead of the previously resolved version.

Change rate. The fraction of dependencies that resolve to a different version after replacement.

Change magnitude. The sum of upgrade and downgrade steps.

Range size. The number of versions in a generated range.

Change rate, *change magnitude*, and *range size* are indicators of flexibility, describing how much and how freely the resolved dependencies can vary. *Success rate* reflects both stability and flexibility, indicating the resolution is less stable if compilation or test failures increase, or less flexible if resolution failures increase. Finally, the *replacement rate* measures impact by how broadly the range replacement was applied.

Results. Across all 105 projects, range replacement required computing compatibility decisions over 141K and 100K (GA, v_{old}, v_{new}) tuples using the MARCO- and SEMVER-based detectors, respectively. This shows the scalability of client-agnostic over client-specific methods for large-scale compatibility analyses that involve thousands of dependencies. Figure 8 shows that the framework managed to replace at least one dependency declaration in 88 (84%) of projects with the MARCO detector, and in 104 (99%) of projects with the SEMVER-based detector. It further confirms our expectation that the MARCO-based replacements preserve compatibility, improving resolution stability: 81 projects showed no compile or runtime regressions, resulting in a success rate of 92%. On the other hand, the SEMVER-based replacements preserved compatibility poorly: only 54 projects (52%) compiled and 19 (35%) of the compiling projects passed their tests. With an overall success rate of only 18%, SEMVER frequently introduces both API and behavioral breaking changes, resulting in compile and runtime failure.

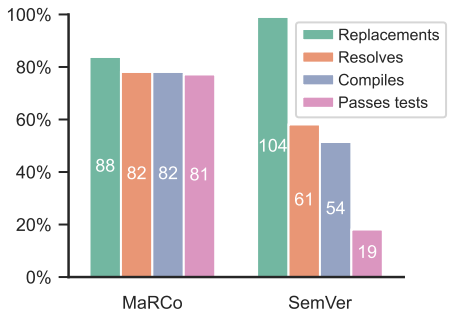


Fig. 8. Projects that resolve, compile, and pass their tests suites after range replacements.

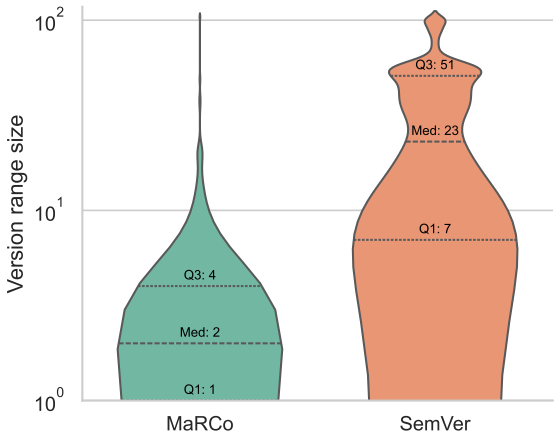


Fig. 9. Distribution of version range size generated by MARCO and SEMVER.

TABLE I
RQ3 EVALUATION METRIC RESULTS.

	MaRCo				SemVer			
Metric	Min	Max	Mean	Median	Min	Max	Mean	Median
Resolved deps	2	489.0	58.6	44.0	1	485	39.3	19.5
Change rate	0	0.5	0.1	0.0	0	1.0	0.6	0.6
Change magnitude	0	43.7	2.4	1.0	0	74.9	20.5	17.5
Upgrades	0	13.0	1.9	1.0	0	92	18.8	9.0
Upgrade steps	0	100.0	7.0	1.5	0	1888	447.8	158.0
Downgrades	0	3.0	0.1	0.0	0	1	0.0	0.0
Downgrade steps	0	131.0	2.6	0.0	0	43	0.9	0.0
Range size	1	104.0	3.9	2.0	1	101	23.0	30.4
Replacement rate	0.0	1.0	0.1	0.1	0.2	1.0	0.7	0.7

Figure 9 shows the distribution of the size of the generated version ranges for each detector. The SEMVER-generated ranges contained 1-101 versions, with a median of 23 and a mean of 30.4. The MARCO-generated ranges contained 1-104 versions, with a median of 2 and a mean of 3.9. This further confirms our expectations that SEMVER generates much wider ranges than MARCO, providing greater resolution flexibility.

Table I summarizes the evaluation metrics. The change rate

shows that SEMVER replacements caused 60% and MARCO replacements caused 10% of dependencies to resolve a different version, most of which were upgrades with SEMVER showing larger and more frequent upgrades. The replacement rate shows that, on average, 70% of resolved dependencies originated from SEMVER-generated ranges, while 13% originated from MARCO-generated ranges. This means while a compatible version range could be obtained for most dependencies using the SEMVER detector, we could only obtain a compatible version range for 13% of dependencies with the MARCO detector. The lower replacement rate reflects MARCO's stricter requirements, as it depends on the ability to locate, compile, and run dependency repositories.

To conclude, both detectors affect MAVEN's dependency resolution differently. With the MARCO detector, we replaced fewer dependency declarations (13%) across fewer projects (84%), but maintained a high success rate of 92%, demonstrating strong compatibility preservation. The SEMVER detector replaced most dependency declarations (70%) across nearly all projects (99%), but exhibits a substantially lower success rate of 18%, frequently introducing both compile and runtime incompatibilities. These results highlight a clear stability-flexibility trade-off: MARCO produces narrow, compatibility-preserving ranges that limit flexibility, whereas SEMVER produces wide ranges that frequently introduce breaking changes which limits stability. While neither approach achieves an ideal stability-flexibility balance, these results illustrate the practical boundaries of client-agnostic range replacement in maintaining reliable dependency resolution.

> **Answer RQ_3:** Client-agnostic compatible ranges expose a stability-flexibility trade-off: MARCO-generated ranges are narrow but unlikely to introduce breaking changes (improved stability), while SEMVER-generated ranges are wider but likely to introduce breaking changes (improved flexibility).

VIII. DISCUSSION AND FUTURE WORK

This paper investigates how compatible version ranges affect dependency resolution in the MAVEN ecosystem, which predominantly relies on version pins. We first examined how conflicting version pins are mediated in practice, showing that conflicts often resolve to SEMVER-unstable versions and are frequently manually mediated. We then evaluated the impact of compatible version range replacement using the MARCO- and SEMVER-based client-agnostic compatibility detectors. The results reveal a clear trade-off: MARCO generates narrow, stable ranges that avoid breaking changes but limit flexibility, whereas SEMVER produces wider ranges that increase flexibility but often introduce breaking changes. These findings highlight both the potential and limitations of client-agnostic compatibility analysis for improving resolution stability and flexibility. This section reflects on these findings and outlines directions for future work.

Resolution reliability in practice. RQ_1 confirms related work [12] that conflicting version pins remain widespread, affecting 65% of projects, reducing resolution stability and flexibility and putting them at risk of future dependency-related issues. 23% of conflicts resolve to versions that are not guaranteed to be compatible with all conflicting pins, representing latent compatibility risks that may surface as dependents or their dependencies evolve. Even with perfect SEMVER adoption, 12% of conflicts involve version pins across different major versions, making them inherently unsolvable by SEMVER-based ranges alone. Furthermore, 67% of projects with conflicts show traces of manual conflict mediation, indicating substantial maintenance effort. Together, these findings highlight limitations of MAVEN's nearest-first mediation and motivate compatibility-aware automated mediation strategies.

Flexible or stable, not both. To enable compatibility-aware conflict mediation, we explored replacing version pins with compatible version ranges generated by the MARCO and SEMVER detectors. RQ_2 showed that both detectors identify breaking changes with similar effectiveness but exhibit opposite recall–precision trade-offs, which in RQ_3 translate into a stability–flexibility trade-off in dependency resolution. Although client-agnostic compatibility enables scalable range replacement, it cannot strike an ideal balance between stable and flexible resolution. MARCO ranges provide stability-oriented resolution, aligning better with developers' preference for stability. The detector performs comparably to the client-specific UPPDATERA detector while remaining fully client-agnostic, providing a scalable baseline for compatibility detection that enables ecosystem-wide range replacement without repeated per-client analysis. Future work could explore hybrid approaches that combine client-agnostic compatibility analysis with lightweight client-specific reachability analysis to derive more precise compatible version ranges and achieve greater flexibility. For example, by storing MARCO's client-agnostic compatibility mappings at method- instead of package-level, client-specific reachability analysis could filter the compatibility result to achieve more flexible compatible version ranges without compromising stability or scalability.

Pinning is preferable to SEMVER. Previous work has shown that SEMVER-compatible updates often contain static API incompatibilities, which the RQ_3 results confirm: SEMVER-based ranges caused compilation failure in 11% of projects. Even more concerning, the SEMVER-based ranges caused behavioral breaking changes during runtime in 65% of projects. These findings suggest that maintainers primarily follow SEMVER to communicate interface stability rather than behavioral compatibility, and help explain the developer aversion to SEMVER ranges. However, even if maintainers adhered to SEMVER perfectly in their release versions, Hyrum's law [34] implies that any change could break some client, effectively forcing most changes to become major releases. While this would substantially improve stability, it would also severely limit the flexibility provided by SEMVER ranges. To support both stable and flexible resolution, versioning would need to become more fine-grained so that clients can determine whether upgrades are breaking with respect to their specific usage. Because such

granularity is unrealistic to maintain manually, this motivates future work on automatic compatibility-aware versioning.

Limitations of cross-version testing. The effectiveness of MARCO-based range replacement depends on the behavioral compatibility check, which requires locating, compiling, and executing dependency tests. Because tracing a dependency to its GITHUB repository is not always possible and compiling arbitrary projects from GITHUB is non-trivial [35], some compatibility evaluations remain inconclusive, accounting for up to 52% of cases in the HYBRID dataset. Even when tests are available, their coverage affects the reliability of the compatibility decisions. Low-quality test suites will catch fewer breaking changes than extensive ones, as seen by the two breaking changes in the HYBRID dataset in RQ_2 that were wrongly labeled as non-breaking. To increase confidence in the compatibility decisions obtained via cross-version testing, future work could try to increase coverage through test generation techniques like EVOSUITE [36], which could reduce both false positives caused by low coverage and inconclusive evaluations due to missing tests. Future work could also exclude tests covering the internal API to reduce false dynamic incompatibilities, similar to how MARACAS [11] excludes internal API sections using annotations to reduce false static incompatibilities.

Scaling to ecosystem-level deployment. In our experiments, full range replacement is achieved by modifying dependency POMs in the local `m2` folder. While effective in a controlled setting, this approach is not practical for ecosystem-level deployment. A more scalable alternative could instead use a mirror of MCR that provides pre-replaced POMs, although efficiently maintaining such a mirror for a rapidly evolving ecosystem remains an open challenge. Furthermore, the compatibility mappings generated by the MARCO detector rely on regression testing, which is computationally expensive [37]. Future work could explore regression test selection to optimize detection of breaking changes while minimizing computational overhead. A distributed model in which library maintainers generate and publish compatibility mappings for their own releases could further improve scalability and simplify the generation and maintenance of global compatibility data, enabling compatibility-aware resolution for the entire ecosystem.

Threats to validity. The MARCO detector links dependencies to their GITHUB repositories, and because repositories and tags are mutable, future linking may produce different results. We therefore provide a replication package containing the datasets and instructions to recreate the experiments.

RQ_1 and RQ_3 analyze relatively small samples of MAVEN projects which may limit generalizability. However, RQ_1 found that 65% of projects contain conflicts, closely aligning with a larger study (n=2289) reporting 64% [12], suggesting our sample is reasonably representative.

RQ_3 evaluates regressions using project tests, although RQ_2 shows that tests may fail to detect some dependency-induced breaking changes. While some regressions may remain undetected, the large relative stability-flexibility differences between detectors remain meaningful and align with the recall–precision trade-off observed in RQ_2.

RQ_1 investigates whether conflicts are manually mediated, but not why. Manual mediation may occur for reasons unrelated to compatibility, such as avoiding vulnerable versions. A preliminary check using `OSV.dev` [38] showed that 93% of managed conflicts (vs. 84% of unmanaged conflicts) do not resolve to versions fixing known CVEs in the conflicting pins, suggesting vulnerability mitigation is unlikely to be the dominant driver. Regardless of motivation, manual mediation introduces maintenance overhead [4], and its prevalence supports our broader conclusion that MAVEN's nearest-first mediation does not eliminate the need for manual mediation.

Our project selection required successful compilation to analyze dependency usage in RQ_1 and regressions in RQ_3, which inadvertently excluded multi-module projects and may affect generalizability. In principle, the same analysis could treat each module as a separate unit of analysis [39]. Because built modules install as regular artifacts in the `m2` folder, the RQ_3 range replacement applies without major modification.

IX. SUMMARY

The reuse of open-source software components as dependencies is widespread in modern software development and updating dependencies remains challenging. SEMVER is used in practice to infer compatible updates without client context, while recent work has combined static bytecode differencing and dynamic cross-version testing to identify compatible versions. We compare these client-agnostic approaches and investigate how the derived compatible version ranges affect the stability and flexibility of MAVEN dependency resolution.

Revisiting the Research Questions. We found that conflicts often resolve to versions that are unstable under SEMVER assumptions and that manual mediation is common, indicating that MAVEN's nearest-first mediation strategy contributes to unstable and inflexible resolution and manual maintenance effort (RQ_1). We evaluated how well the client-agnostic MARCO and SEMVER detectors identify compatible versions (RQ_2) and how the resulting version ranges affect dependency resolution (RQ_3). We found a fundamental stability-flexibility trade-off inherent to client-agnostic compatible ranges: they either introduce frequent breaking changes (SEMVER) or they are narrow but compatibility-preserving (MARCO).

Impact. These results show the potential of compatible version ranges to improve resolution reliability, but also that client-agnostic detection alone cannot yield ranges that are simultaneously stable and flexible despite enabling scalable range generation. This motivates future work on hybrid approaches combining the scalability of client-agnostic compatibility with the precision of client-specific reachability analysis.

ACKNOWLEDGMENT

This research was supported by the National Growth Fund through the Dutch 6G flagship project "Future Network Services".

References

[1] A. Lawson and S. Hendrick, "Global spotlight 2023: Survey-based insights into the global landscape of open source trends, sustainability challenges, and growth opportunities," The Linux Foundation, October 2023.

[2] J. Hejderup and G. Gousios, "Can we trust tests to automate dependency updates? a case study of java projects," *Journal of Systems and Software*, vol. 183, p. 111097, 2022.

[3] Y. Wang, P. Sun, L. Pei, Y. Yu, C. Xu, S.-C. Cheung, H. Yu, and Z. Zhu, "Plumber: Boosting the propagation of vulnerability fixes in the npm ecosystem," *IEEE Transactions on Software Engineering*, 2023.

[4] L. Zhang, C. Liu, S. Chen, Z. Xu, L. Fan, L. Zhao, Y. Zhang, and Y. Liu, "Mitigating persistence of open-source vulnerabilities in maven ecosystem," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering*, 2023, pp. 191–203.

[5] B. Xia, T. Bi, Z. Xing, Q. Lu, and L. Zhu, "An empirical study on software bill of materials: Where we stand and the road ahead," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 2630–2642.

[6] I. Pashchenko, D.-L. Vu, and F. Massacci, "A qualitative study of dependency management and its security implications," in *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, 2020, pp. 1513–1531.

[7] J. Dietrich, D. Pearce, J. Stringer, A. Tahir, and K. Blincoe, "Dependency versioning in the wild," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories*, 2019, pp. 349–359.

[8] S. Mirhosseini and C. Parnin, "Can automated pull requests encourage software developers to upgrade out-of-date dependencies?" in *2017 32nd IEEE/ACM international conference on automated software engineering*, 2017, pp. 84–94.

[9] T. Preston-Werner, "Semantic Versioning 2.0.0," https://semver.org/, accessed 03-Jun-2024.

[10] 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.

[11] L. Ochoa, T. Degueule, J.-R. Falleri, and J. Vinju, "Breaking bad? semantic versioning and impact of breaking changes in maven central: An external and differentiated replication study," *Empirical Software Engineering*, vol. 27, no. 3, p. 61, 2022.

[12] 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*, 2018.

[13] Apache Maven Project, "Introduction to the Dependency Mechanism," https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html, 2002-2023, accessed 13-Nov-2023.

[14] L. Chen, F. Hassan, X. Wang, and L. Zhang, "Taming behavioral backward incompatibilities via cross-project testing and analysis," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 112–124.

[15] S. Mujahid, R. Abdalkareem, E. Shihab, and S. McIntosh, "Using others' tests to identify breaking updates," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 466–476.

[16] R. He, H. He, Y. Zhang, and M. Zhou, "Automating dependency updates in practice: An exploratory study on github dependabot," *IEEE Transactions on Software Engineering*, 2023.

[17] C. Zhu, M. Zhang, X. Wu, X. Xu, and Y. Li, "Client-specific upgrade compatibility checking via knowledge-guided discovery," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, pp. 1–31, 2023.

[18] L. Zhang, C. Liu, Z. Xu, S. Chen, L. Fan, B. Chen, and Y. Liu, "Has my release disobeyed semantic versioning? static detection based on semantic differencing," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[19] C. Paulsen and S. Proksch, "MaRCo: Compatible Version Ranges in Maven," in *41st International Conference on Software Maintenance and Evolution*, 2025.

[20] ——, "The Effect of Compatible Version Ranges on Maven Dependency Resolution Stability and Flexibility (Replication Package)," https://doi.org/10.5281/zenodo.20846969, 2025.

[21] J. Darcy, "Kinds of Compatibility," https://wiki.openjdk.org/display/csr/Kinds+of+Compatibility, 2021, accessed 13-Nov-2023.

[22] M. Mois, "japicmp," https://siom79.github.io/japicmp/, March 2024, accessed 02-Jun-2024.

[23] revapi.org, "Revapi," https://revapi.org/, 2023, accessed 02-Jun-2024.

[24] S. McCamant and M. D. Ernst, "Early identification of incompatibilities in multi-component upgrades," in *European Conference on Object-Oriented Programming*, 2004, pp. 440–464.

[25] S. Mostafa, R. Rodriguez, and X. Wang, "Experience paper: a study on behavioral backward incompatibilities of java software libraries," in *Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis*, 2017, pp. 215–225.

[26] L. M. Ochoa Venegas, "Break the code?: Breaking changes and their impact on software evolution," [Phd Thesis 1 (Research TU/e / Graduation TU/e), Mathematics and Computer Science], 2023.

[27] F. Reyes, Y. Gamage, G. Skoglund, B. Baudry, and M. Monperrus, "BUMP: A benchmark of reproducible breaking dependency updates," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2024, pp. 159–170.

[28] Apache Maven Project, "dependency:tree," https://maven.apache.org/plugins/maven-dependency-plugin/analyze-mojo.html, October 2023, accessed 20-May-2024.

[29] K. Rybnikov and T. Schraitle, "Python helper for Semantic Versioning," https://pypi.org/project/semver/, 2013, accessed 19-Oct-2025.

[30] P. Totev, "Symbolic links to directories are not recognized as directories," https://github.com/codehaus-plexus/plexus-io/issues/71, April 2022, accessed 02-Jun-2024.

[31] codehaus-plexus, "Release Plexus IO 3.3.1," https://github.com/codehaus-plexus/plexus-io/releases/tag/plexus-io-3.3.1, May 2022, accessed 02-Jun-2024.

[32] U. Chemouni, "Release V 2.4.9," https://github.com/netplex/json-smart-v2/releases/tag/2.4.9, March 2023, accessed 02-Jun-2024.

[33] btrplace/scheduler, "Bump json-smart from 2.4.8 to 2.4.9," https://github.com/btrplace/scheduler/pull/441, March 2023, accessed 02-Jun-2024.

[34] H. Wright, "Hyrum's Law," https://www.hyrumslaw.com/, accessed 21-Nov-2024.

[35] M. Keshani, T.-G. Velican, G. Bot, and S. Proksch, "Aroma: Automatic reproduction of maven artifacts," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 836–858, 2024.

[36] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.

[37] A. Gyori, O. Legunsen, F. Hariri, and D. Marinov, "Evaluating regression test selection opportunities in a very large open-source ecosystem," in *2018 IEEE 29th International Symposium on Software Reliability Engineering*, 2018, pp. 112–122.

[38] Google, "A distributed vulnerability database for Open Source," https://osv.dev/, accessed 04-Mar-2026.

[39] R. Lu, L. Zhang, K. Li, M. Zhang, and Y. Chen, "Minimizing breaking changes and redundancy in mitigating technical lag for java projects," *arXiv preprint arXiv:2511.06762*, 2025.