MaRCo

Compatible Version Ranges in Maven

Paulsen, Cathrine; Proksch, Sebastian

DOI
10.1109/ICSME64153.2025.00105

Publication date
2025

Document Version
Final published version

Published in
Proceedings - 2025 IEEE International Conference on Software Maintenance and Evolution, ICSME 2025

Citation (APA)
Paulsen, C., & Proksch, S. (2025). MaRCo: Compatible Version Ranges in Maven. In *Proceedings - 2025 IEEE International Conference on Software Maintenance and Evolution, ICSME 2025* (pp. 910-914). (Proceedings - 2025 IEEE International Conference on Software Maintenance and Evolution, ICSME 2025). IEEE. https://doi.org/10.1109/ICSME64153.2025.00105

MaRCo: Compatible Version Ranges in Maven

Cathrine Paulsen
Software Technology
Delft University of Technology
Delft, the Netherlands
C.R.Paulsen@tudelft.nl

Sebastian Proksch
Software Technology
Delft University of Technology
Delft, the Netherlands
S.Proksch@tudelft.nl

2025 IEEE International Conference on Software Maintenance and Evolution (ICSME) | 979-8-3315-9587-6/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICSME64153.2025.00105

***Abstract*—Managing dependencies in Java projects is challenging: undeclared, implicit dependencies and conflicting version declarations can lead to breaking changes and unpredictable resolution. We present MARCO, a tool to improve resolution reliability in Maven. It injects missing direct dependencies and replaces pinned versions with client-agnostic compatible version ranges, which can be safely reused across clients. The ranges are obtained by combining bytecode differencing and cross-version testing to detect API and behaviorally compatible dependency versions. We demonstrate how MARCO can be used to retrieve compatible versions for specific dependencies, replace pinned versions using compatibility mappings, and execute the full pipeline to enable compatibility-aware resolution. Our preliminary evaluation shows MARCO recovers all missing dependencies for 91% of affected projects, and replaces pinned versions with stable, compatible version ranges for 13% of dependencies on average across 78% of projects. MARCO demonstrates the feasibility of scalable, compatibility-driven dependency management. The demo is available at https://youtu.be/2faDG8Cmmh0.**

***Index Terms*—dependency management, client-agnostic compatibility, compatible version range generation**

I. INTRODUCTION

Modern software relies heavily on the reuse of open-source software components to accelerate development [1]. These dependencies quickly form complex trees, as each direct dependency may introduce several transitive ones. Determining which versions of which dependencies to resolve is a challenging problem [2]. Developers therefore rely on dependency managers, like Maven for Java projects, to automate the process. However, these tools are not without flaws, and solving dependency-related issues can be time-consuming as fixing one issue often causes further issues with other dependencies, a phenomenon known as *Dependency Hell* [3].

A *reliable* dependency manager offers dependency resolution along three key dimensions: it is *stable* to prevent breaking changes [4], *flexible* to prevent version conflicts and resolution failure [5], and *transparent* by providing accurate dependency trees to facilitate debugging [6].

Maven projects often have *undeclared, implicit dependencies* [7] and *conflicting version declarations* [8]. The implicit use of transitive dependencies can result in unexpected breakages from seemingly innocent changes, like removing unused dependencies, and prevents timely vulnerability discovery [6]. Version conflicts arise when a project depends on different version pins (i.e., soft version constraints) of the same dependency. Maven mediates conflicts by resolving the nearest declaration [9], which can introduce breaking changes if the conflicting versions are incompatible. These issues make it difficult to safely maintain dependencies to keep up with bug fixes and security updates. Although *Semantic Versioning* indicates backwards compatibility within the same major version, breaking changes are still common in minor and patch versions [10]. Consequently, most Maven projects exclusively pin versions [11], increasing the risk of conflicts compared to version ranges, which enable automated but potentially unsafe updates.

Automated tools could reduce the risk of future dependency-related issues by recovering missing dependencies and replacing pins with verified compatible ranges. Prior work addresses parts of this problem, mostly through breaking change detection to classify version pairs as (in)compatible. Most are client-specific solutions, meaning they rely on analyzing the *client*, or dependent, of the dependency, often using the test suites of one or more clients [12–15], or rely on static analysis of the dependency [4, 16]. However, static analysis struggles to detect behavioral changes due to refactoring and reflection, and clients generally have poor test coverage of their dependencies [4]. Furthermore, results from client-specific tools are not reusable and do not scale well, especially for less popular dependencies with fewer clients. Ranger [11] combines these solutions, replacing version pins with compatible ranges generated via static analysis of the dependency and execution of client tests, but inherits the same limitations. In addition, it is not fully open source, which limits replicability.

We complement existing work with MARCO, a tool for Maven projects that recovers missing dependency declarations and replaces version pins with compatible version ranges to improve the resolution of conflicting dependencies. MARCO detects API compatibility via static analysis (bytecode differencing), and behavioral compatibility using the tests of the dependency instead of the client. As a result, the compatibility results are *client-agnostic* and can be reused. We document and demonstrate three practical use cases that may be of interest to both researchers and practitioners:

- Generating compatible versions for specific dependency versions (GAVs), used to evaluate breaking change detectors or to inform compatible upgrades and downgrades.
- Replacing version pins with ranges across the dependency tree using MARCO-generated or custom compatibility map-

2576-3148/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSME64153.2025.00105

pings to explore compatibility-aware resolution strategies.

- Executing the full pipeline, replacing all version pins with compatible ranges to enable resolving version conflicts in a compatibility-preserving way.

MARCO[1] is available in our replication package [17], which also contains 1252 reusable compatibility decisions generated by MARCO for 323 GAVs. The remainder of the paper is structured as follows. Section II describes MARCO's implementation. Section III shows different use case applications. Finally, Section V discusses the main evaluation results, limitations, and future work.

II. MARCO

In this section, we present the *Maven Compatible Range toolkit* (MARCO). Maven resolves a project's dependencies by recursively retrieving metadata from the declared dependencies' POM files from the *Maven Central Repository* (MCR). This process constructs a dependency tree of direct and transitive dependencies. Maven then resolves conflicts using a nearest-first strategy, where the version declared closest to the root of the tree in breadth-first-search order is selected. This strategy can produce unexpected outcomes when dependencies are undeclared or version pins are incompatible. MARCO improves this process by rewriting the dependency declarations to guide Maven toward compatibility-preserving resolutions without altering the resolver itself.

To achieve this, MARCO consists of two components: the GENERATOR, which computes compatibility mappings between dependency versions, and the REPLACER, which rewrites POMs by injecting missing declarations and replacing pinned versions with verified compatible ranges (see Figure 1). Separating the components ensures that incorporating MARCO into a developer's workflow requires minimal effort and overhead. MARCO allows developers to pin a version as a soft constraint, then apply the REPLACER to their POM and be confident that the resolved versions are compatible with the one they pinned. To replace soft constraints, the REPLACER requires a mapping from dependency versions to compatible ranges. The GENERATOR takes care of computing, storing, and serving these mappings. The compatible ranges are client-agnostic, meaning that the compatible versions for a specific dependency can be pre-computed and reused between clients to reduce overhead.

A. Breaking change detection

The GENERATOR pre-computes a dependency's compatible versions using a client-agnostic compatibility approach, which allows the REPLACER to be lightweight. The GENERATOR has two responsibilities: (i) perform compatibility checking and store the results, and (ii) serve requests of specific compatibility mappings by the REPLACER. Figure 1 shows that the GENERATOR runs server-side and is not intended for client use.

The compatibility mappings are created as follows. Given a specific dependency version v (*base*), the GENERATOR fetches the dependency's available versions av (*candidates*), and

[1]https://github.com/CathrinePaulsen/MaRCo-toolkit

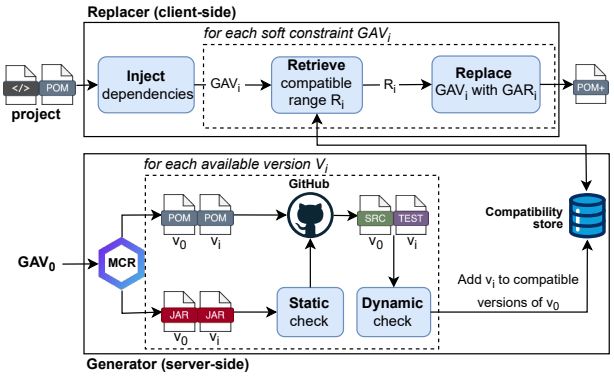

Fig. 1. Overview of the MARCO REPLACER and GENERATOR.

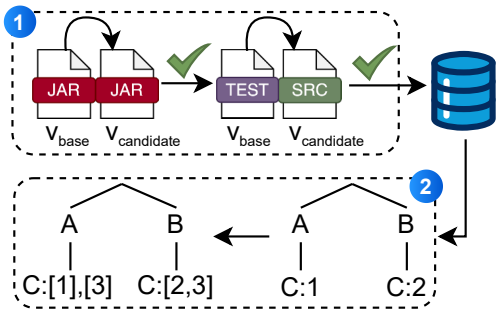

Fig. 2. The GENERATOR pre-computes the compatible versions (1), which the REPLACER uses to expand version pins into ranges (2).

computes whether they are compatible with the base v. The compatibility mapping is expressed as follows: $v \mapsto \{v_i \mid v_i \in av \text{ and is compatible with } v\}$. Checking whether a candidate v_i is compatible with the base v involves three main steps performed in order: the static compatibility check, the Maven-to-GitHub linking, and the dynamic compatibility check.

Computing compatibility. The compatibility check for a version pair consists of a static and a dynamic check, as shown in step 1 of Figure 2. The static check uses JAPICMP [18], a common static compatibility checker for Java [10, 11]. Given the base and candidate JARs, JAPICMP reports whether there are any binary or source incompatible changes, meaning the candidate is statically incompatible with the base. A candidate must be both statically and dynamically compatible with its base to be compatible. Therefore, if the candidate is statically incompatible, we skip the dynamic check. The dynamic check checks for behavioral compatibility approximated by cross-version testing, i.e., by running the base's test code with the candidate's source code. To this end, we create a new Maven project with the compiled source code of the candidate, the compiled test code of the base, and a combined POM. The combined POM is the candidate's POM with the test dependencies replaced by the base's test dependencies since we are running the base's tests. To determine the behavioral compatibility between the base and the candidate, the base's tests are run in two stages: first with the base's code to

establish a baseline, then with the candidate's code. The candidate is considered compatible with the base if it passes the same tests as the baseline, and otherwise incompatible.

Locating tests via GitHub linking. The dynamic check requires the candidate's compiled source code, the base's compiled test code, and a combined POM. While MCR provides the compiled source code and POMs, tests are rarely published [19]. We therefore implement a GitHub linking component to recover more dependency tests. Given a dependency version (GAV), the linking algorithm retrieves the GitHub repository and tag via the GAV's POM. First, it extracts the repository from the GitHub URL in the `scm`-section [20] of the (parent) POM. It then extracts the version from the `tag` [21] if available, and otherwise from the GAV. Finally, it retrieves the repository tag by looking up the following patterns with the GitHub API: *<artifactId>-<version>*, and *{v,r,}<version>*. This approach is similar to Keshani et al. [22] and performs similarly on the Reproducible Central dataset [23]: The algorithm successfully linked 63% of GAVs to a repository tag with 96% of tags matching the ground truth. Using this approach, we located tests for 53% of the GAVs on Reproducible Central, while only 2% had test JARs on MCR, showing its effectiveness for test recovery.

B. Modifying Dependency Declarations

The REPLACER transforms project POMs in two ways: (i) it injects missing direct dependencies identified using the *analyze* goal of the *maven-dependency-plugin* [24], and (ii) it replaces pinned versions with compatible ranges based on the pre-computed mappings. Because the missing dependency declarations returned in the first step contain soft constraints, the injection must be performed before replacing the POM's soft constraints. In the second step, the REPLACER extracts the GAV of each dependency declaration that contains a soft version constraint, looks the GAV up in the compatibility mappings that were pre-computed by the GENERATOR, and fetches the list of compatible versions as shown in step 2 in Figure 2. The list is converted into the Maven range format [25] using Maven's version sorting algorithm [26], and the soft version constraint is replaced by the range.

III. APPLICATION

The MARCO REPLACER and GENERATOR can be used independently or together for different use cases. In this section, we will present three primary use cases: compatible version retrieval with the GENERATOR, dependency declaration replacement with the REPLACER, and enabling compatibility-aware resolution with the complete MARCO workflow. Each component is a separate Python module and is run via the entry points `marco-replacer` and `marco-generator`.

A. Retrieving Compatible Versions

The GENERATOR is a standalone component that can generate the compatible versions for a specific GAV. This can be useful for evaluations against other breaking change detectors, for practitioners to provide insight into compatible versions when upgrading or downgrading specific dependencies, or for library maintainers to inform users about compatible versions.

```
$ marco-generator -g com.fasterxml.jackson.core -a jackson-
    databind -v 2.15.1

com.fasterxml.jackson.core:jackson-databind:2.15.1 has
    compatible versions ['2.15.4', '2.15.3', '2.15.2',
    '2.15.1'] out of candidate versions ['2.16.0', '2.16.0-
    rc1', '2.15.4', '2.15.3', '2.15.2', '2.15.1', '2.15.0',
     '2.15.0-rc3', '2.15.0-rc2', '2.15.0-rc1', '2.14.3']

$ jython client/client/range_converter.py --compatible
    2.15.4 2.15.3 2.15.2 2.15.1 --available 2.16.0 2.16.0-
    rc1 2.15.4 2.15.3 2.15.2 2.15.1 2.15.0 2.15.0-rc3
    2.15.0-rc2 2.15.0-rc1 2.14.3

[2.15.1,2.15.4]
```

Listing 1. Generating the compatible versions for a specific GAV.

```
{
    "com.fasterxml.jackson.core:jackson-databind:2.15.1":
        ["2.15.1", "2.15.2", "2.15.3", "2.15.4"],

    "com.google.http-client:google-http-client-gson:1.43.1":
        ["1.43.0", "1.43.1", "1.43.2", "1.43.3", "1.44.1",
     "1.44.2", "1.45.0"],

    "com.google.cloud:google-cloud-core-http:2.9.4":
        ["2.8.23", "2.8.24", "2.8.26", "2.8.27", "2.8.28",
     "2.9.0", "2.9.1", "2.9.2", "2.9.3", "2.9.4"]
}
```

Listing 2. The compatibility mappings are stored as a JSON object.

We illustrate this use case by generating the compatible version range for `jackson-databind:2.15.1` in Listing 1. The flag `max_candidates` determines how many version upgrades and downgrades to consider, and `stop_after_n` stops the compatibility check once it finds n consecutive incompatible upgrades or downgrades. The resulting list of compatible versions can be printed to the console or stored in JSON format to facilitate reuse. The JSON maps a GAV to a list of compatible versions, as shown in Listing 2.

B. Replacing Dependency Declarations

The REPLACER replaces pinned versions in a project's POM with ranges constructed from the available compatibility mappings. These mappings do not need to stem from the GENERATOR but could also be generated by another breaking change detector, which can be useful for researchers who want to compare tools or experiment with compatibility-aware resolution strategies.

The REPLACER is invoked via `marco-replacer` and takes a POM file as input. Listing 3 shows the result of invoking the REPLACER on a real Maven project's POM file, with the replaced and inserted declarations marked in green.

C. Full Pipeline: Compatibility-aware Resolution

When used end-to-end, MARCO enables compatibility-aware resolution through automatic modification of dependency declarations. In the previous use cases, we have shown how the GENERATOR can be invoked to retrieve the compatible versions for a single GAV, and how the REPLACER can be invoked to replace a single POM. Version conflicts are a result of conflicting soft constraints declared throughout the

```
<properties>
  <jacksonVersion>2.13.5</jacksonVersion>
  <openLineageVersion>0.25.0</openLineageVersion>
</properties>

<!-- Truncated... -->

<dependency>
  <groupId>io.openlineage</groupId>
  <artifactId>openlineage-java</artifactId>
  <version replaced_value="${openLineageVersion}">
      [0.24.0,0.28.0]</version>
</dependency>
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-core</artifactId>
  <version replaced_value="${jacksonVersion}">[2.13.0-rc2
      ,2.13.5]</version>
</dependency>
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
  <version replaced_value="${jacksonVersion}">
      [2.13.4.1,2.13.5]</version>
</dependency>
<dependency>
  <groupId>com.google.code.findbugs</groupId>
  <artifactId>jsr305</artifactId>
  <version replaced_value="3.0.2">[2.0.3,3.0.2]</version>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
  <version replaced_value="1.7.36">[1.7.29,1.7.36]</version>
</dependency>
<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>aws-java-sdk-core</artifactId>
  <version inserted="true">1.12.362</version>
</dependency>
```

Listing 3. Dependency declarations modified by MARCO.

dependency tree, and replacing soft constraints in a single POM will therefore not solve them. To enable compatibility-aware resolution, we need to apply the REPLACER to the direct and transitive dependencies' POMs as well to ensure all declarations across the dependency tree are replaced.

This can be useful to practitioners who are interested in compatibility-aware resolution and want a minimal effort solution. MARCO allows developers to pin whatever version they are using during development, while the replacement of pins to compatible ranges ensures that a compatible version is resolved. However, this option may work best for developer-controlled dependencies with guaranteed test access. For practical large-scale use, MARCO is limited by the availability of runnable test suites, which are not available for many libraries. A partial replacement of declarations will improve the resolution reliability to some extent, but it is unlikely to resolve all version conflicts for a given project unless it only depends on dependencies that MARCO can locate. The demo shows how MARCO can be applied in this use case.

IV. DATA COLLECTION

We conducted a preliminary evaluation of MARCO to assess the reliability of its generated ranges, and its applicability and resolution impact in practice. To this end, we required a dataset of representative Maven projects to which we could apply MARCO, and a dataset of (in)compatible dependency version pairs.

Real-world projects. We sampled 105 Maven projects from GitHub. Our selection criteria were that the projects have been created between January 1, 2023 and May 1, 2024 to get current libraries at the time of evaluation. We filtered for projects with 10+ stars and 50+ commits to avoid toy projects. Furthermore, we required the projects to compile, have dependencies, include a test suite, and use a single-module Maven structure. The projects include both simple and complex dependency trees, with a minimum, maximum, and median size of 2, 489, and 44 dependencies, respectively. Applying MARCO to these projects and their dependencies resulted in compatibility mappings for 323 dependencies (GAVs), covering 1252 unique version pairs.

(In)compatible version pairs. We used two datasets for compatibility evaluation. First, *BUMP* [27] is a curated dataset containing 372 incompatible updates (i.e., version pairs) across 122 libraries (GAs) that cause resolution, compilation, or test failures in 114 Maven projects. Second, to better assess the performance of MARCO's ability to detect non-breaking changes, we collected *Dependabot updates* from the 105 projects. We mined Dependabot PRs that updated a single Maven dependency and were merged without changes as a heuristic for non-breaking updates. This yielded 481 compatible updates across 167 libraries and 43 Maven projects.

To facilitate future work on compatibility inference and dependency resolution, the replication package [17] includes the list of 105 selected projects, the dataset of compatible version pairs, and the generated compatibility mappings.

V. EVALUATION

We have performed a preliminary evaluation of MARCO to assess its effectiveness and applicability in practice. This section summarizes key results and reflects on limitations and opportunities for future work.

Compatibility detection. To determine whether we can trust the MARCO-generated version ranges to contain actual compatible versions, we evaluated MARCO's ability to classify breaking changes using the two datasets of (in)compatible version pairs from Section IV. MARCO achieved high recall for detecting breaking changes (0.99), but low recall for non-breaking changes (0.16). This suggests that MARCO produces stable ranges that are unlikely to contain breaking changes, although the ranges may not contain all compatible versions. This result is expected and by design: the datasets define compatibility based on whether it is (in)compatible for a specific client, while MARCO does not consider specific client use. The breaking changes computed by MARCO are therefore conservative to fit every client, whereas a client may not be affected by every breaking change. Future work could explore computing compatibility at a finer granularity, e.g., method-level, allowing filtering for compatible versions based on client usage. This would combine the precision of client-specific and the efficiency of client-agnostic compatibility analysis.

Library coverage. In terms of applicability, MARCO was able to compute compatibility for 94% of breaking change and 49% of non-breaking change data points. This shows a limitation with cross-version testing, which requires MARCO to locate runnable test suites for the GAV under test, which is not always possible. Future work could mitigate this issue by employing test generation or static analysis where dependency tests are unavailable or lack coverage.

Resolution impact. We applied MARCO to the 105 projects from Section IV to determine whether it can replace enough version pins to affect the resolution outcome. We found that it recovered all missing dependencies for 91% of affected projects. In 78% of the projects, it replaced 13% of the dependency pins with compatible ranges. These findings suggest MARCO is able to improve resolution reliability to some extent, but a replacement rate of 13% is too low to realistically eliminate all version conflicts in practice.

Overall, our preliminary evaluation suggests that MARCO can enable compatibility-aware resolution with minimal manual effort. However, its effectiveness depends on the availability and quality of dependency test suites, which currently limits its applicability to well-tested, developer-controlled dependencies. For example, low-coverage test suites may cause behavioral breaking changes to be missed and flaky tests could incorrectly label some versions as incompatible.

VI. SUMMARY

MARCO aims to improve resolution reliability by recovering missing dependencies and replacing version pins with compatible version ranges, allowing Maven to resolve version conflicts by selecting a compatible version instead of the nearest one. Future work will address MARCO's limitations regarding library coverage and compatibility precision. First, automated test generation would improve coverage of libraries with missing or low-quality test suites. Second, computing compatibility results at a finer granularity would enable post-filtering based on client usage, resulting in broader ranges while preserving the efficiency of pre-computed, client-agnostic compatibility data. We believe MARCO demonstrates the feasibility and value of client-agnostic compatibility data as a foundation for scalable compatibility-driven dependency management.

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, 2023.

[2] P. Abate, R. Di Cosmo, G. Gousios, and S. Zacchiroli, "Dependency solving is still hard, but we are getting better at it," in *IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2020.

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

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

[5] 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.

[6] 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 *IEEE/ACM International Conference on Software Engineering*, 2023.

[7] D. Jayasuriya, S. Ou, S. Hegde, V. Terragni, J. Dietrich, and K. Blincoe, "An extended study of syntactic breaking changes in the wild," *Empirical Software Engineering*, 2025.

[8] 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 *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018.

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

[10] 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*, 2022.

[11] 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 *IEEE/ACM International Conference on Automated Software Engineering*, 2023.

[12] L. Chen, F. Hassan, X. Wang, and L. Zhang, "Taming behavioral backward incompatibilities via cross-project testing and analysis," in *ACM/IEEE International Conference on Software Engineering*, 2020.

[13] S. Mujahid, R. Abdalkareem, E. Shihab, and S. McIntosh, "Using others' tests to identify breaking updates," in *International Conference on Mining Software Repositories*, 2020.

[14] 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.

[15] 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*, 2023.

[16] 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 *IEEE/ACM International Conference on Automated Software Engineering*, 2022.

[17] C. Paulsen and S. Proksch, "Marco: Compatible version ranges in maven," https://doi.org/10.5281/zenodo.15971038, 2025, IEEE International Conference on Software Maintenance and Evolution.

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

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

[20] Apache Maven Project, "Maven SCM Plugin - Usage," https://maven.apache.org/scm/maven-scm-plugin/usage.html, accessed 20-May-2024.

[21] ——, "Maven SCM Plugin - scm:tag," https://maven.apache.org/scm/maven-scm-plugin/tag-mojo.html, accessed 20-Jan-2024.

[22] M. Keshani, T.-G. Velican, G. Bot, and S. Proksch, "Aroma: Automatic reproduction of maven artifacts," *Proc. ACM Software Engineering*, vol. 1, no. FSE, 2024.

[23] Reproducible Builds, "Reproducible Builds for Maven Central Repository," https://github.com/jvm-repo-rebuild/reproducible-central, accessed 03-Jun-2024.

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

[25] ——, "Version Range Specification," https://maven.apache.org/enforcer/enforcer-rules/versionRanges.html, accessed 26-Jun-2024.

[26] The Apache Software Foundation, "Class ComparableVersion (documentation)," https://maven.apache.org/ref/3.5.2/maven-artifact/apidocs/org/apache/maven/artifact/versioning/ComparableVersion.html, accessed 26-Jun-2024.

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