Reliable Dependency Resolution Using Client-agnostic Compatibility Detection

Anonymous Authors
...
...

Abstract

Updating dependencies in a project is challenging, as new versions can introduce breaking changes or conflicts. A reliable dependency resolution should minimize such issues, but typical Maven projects suffer from undeclared dependencies and conflicting soft constraints, leading to intransparent dependencies, unstable resolution, and inflexible declarations that are hard to update. This paper presents MaRCo, a tool that improves Maven's resolution by recovering undeclared dependencies and replacing pinned versions with compatible version ranges generated via bytecode differencing and cross-version testing. Our evaluation shows that MaRCo recovers all missing dependencies in 91% of affected projects and replaces pinned versions in 78% of all projects, where 13% of dependencies resolve from the new version ranges on average. MaRCo simplifies dependency management by letting developers specify a single version while ensuring a compatible version is resolved. While the ranges are conservative, they offer a stable alternative to soft constraints and greater flexibility than hard constraints. The generated ranges provide a stable, scalable foundation to further enhance resolution flexibility through client-specific analysis.

CCS Concepts

• Software and its engineering → Software libraries and repositories.

Keywords

dependency management, client-agnostic compatibility, compatible version range generation

1 Introduction

The reuse of open-source software components as dependencies is widespread in modern software development [21]. Keeping these dependencies up-to-date is crucial, as new releases may include important bug fixes and security patches. Ideally, a *reliable* dependency manager has three key properties to support developers in this update process: the dependency resolution is *stable* to avoid an accidental introduction of breaking changes [18], *flexible* to reduce the risk of version conflicts and resolution failure [40, 44], and *transparent* by providing full and accurate dependency trees so that dependency-related issues are easy to trace [43].

Solving dependency-related issues can be time-consuming, as fixing one issue tends to cause further issues with other dependencies, a phenomenon often referred to as *Dependency Hell* [31, 41]. Two competing declaration strategies exist to control the effect of updates, which both have advantages and drawbacks: many developers opt for a restrictive *pinning* of dependency versions to avoid updates [18, 30]. This reduces the risk of breaking changes but usually leads to more outdated and vulnerable dependencies downstream [40, 44]. Other developers prefer using open *version ranges*, which increases the flexibility of the dependency resolution through automatic updates [40], but updates can be unsafe and introduce breaking changes. Developers struggle with balancing the stability of pinning and the flexibility of open ranges [14]. *Semantic Versioning* can indicate breaking changes with updates to the major version number. Unfortunately, Maven projects rarely comply with this convention [27, 32], a likely reason why 99% of version declarations are pinned soft constraints [44].

Dependency management is a general problem, but Maven-specific challenges exist. The predominately used soft constraints do not represent proper pinning. The Maven resolution does not fail for conflicts, instead, it mediates conflicts by selecting the nearest version declaration without guarantee of actual compatibility [1]. Conflicting soft constraints could therefore expose projects to unreliable dependency resolution that risks stability and flexibility issues. Additionally, Maven allows the use of transitive dependencies without explicit dependency declarations. This decreases transparency, complicates tracking of dependency-related issues, and makes the discovery of vulnerabilities more difficult [43, 44].

Several existing works have already investigated this problem and produced partial solutions to improve some of the three properties, for example, dynamic analysis using client tests [11, 17, 26, 46] and static analysis using AST [18] and call graph [45] differencing to detect incompatible updates. The most related work is Ranger [44], which replaces soft constraints with compatible ranges to achieve higher flexibility and stability. However, Ranger does not address transparency and is unfortunately not freely available for further research. All previous works share a common flaw: version compatibility is decided based on a client-specific analysis, which likely favors flexibility over stability. In contrast, developers were found to generally favor stability over flexibility [23, 30]. Client-specific analyses are often computed using client tests, requiring re-computation for every client, and since clients typically do not test their dependencies, breaking changes may be missed [18, 46].

In this paper, we investigate the state of dependency management in Maven and explore a novel resolution approach that features all three properties of reliable dependency resolution. In contrast to existing works, we are introducing a *client-agnostic* approach to identify compatible version ranges. We will answer three research questions to motivate our solution and evaluate its effectiveness.

, ,
2025. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

RQ1: How prevalent are the undeclared use of dependencies and version conflicts in the Maven ecosystem?

RQ2: How effective is a client-agnostic detection of (non-) breaking changes?

RQ3: Can MaRCo improve Maven's dependency resolution?

Our results show that both the undeclared use of transitive dependencies and conflicts in soft version declarations are common, which degrade the reliability of the dependency resolution in most Maven projects. We therefore propose MaRCo, a method to remove these smells by injecting the missing dependency declarations and replacing soft constraints with compatible version ranges. MaRCo is client-agnostic and exploits bytecode differencing to make a computationally expensive cross-version testing possible [16]. Our evaluation shows that MaRCo can recover the missing dependencies in 91% of the projects exhibiting the smell. It detects breaking changes with 0.99 recall and 0.50 precision on existing breaking change datasets, resulting in version ranges that are highly stable but less flexible. We compared the resolution outcome with and without MaRCo and found that resolution reliability improved in 78% of projects by being more stable than a soft constraint-only approach and more flexible than a hard constraint-only approach with 13% of dependencies previously using soft version constraints now resolving from the compatible version ranges on average.

Overall, this paper presents the following main contributions:

- A prevalence study of two dependency smells that degrade resolution reliability in the Maven ecosystem.
- A client-agnostic method for generating compatible ranges using bytecode differencing and cross-version testing.
- A dataset of 793 compatible dependency updates, complementing existing datasets focused on incompatible updates.

The data and code used for this paper are available at https://anonymous.4open.science/r/MaRCo/.

2 Related Work

Detecting compatible dependency updates is a well-studied problem. Verifying compatibility between two versions requires checking binary, source, and behavioral compatibility [13]. Binary and source compatibility, also called static, syntactic, or API compatibility, can be detected at compile time by static analysis tools like japicmp [24] and revapi [34] which use bytecode differencing. In contrast, behavioral, dynamic, or semantic incompatibilities occur at runtime and are difficult to detect statically [45]. As formal behavioral specifications are difficult to obtain and verify [22], tests are often used to approximate behavioral compatibility. The remainder of this section presents existing work on breaking change detection.

CompCheck [46] builds a knowledge base of client-dependency incompatibilities obtained via client tests and uses control flow graphs to check whether a client is affected by the incompatibilities, showcasing two common techniques in compatibility checkers. Breaking change detection using multiple clients' tests, or *cross-client testing*, is often motivated by the 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. [26] and DeBBI [11] 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 [17]. Ranger [44] replaces soft constraints 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 [45] 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. [18] 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 [27] 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 between two dependency versions that the client never calls, in which case the versions are considered compatible. Common for related work is that most focus on client-specific compatibility. Client-agnostic approaches, in contrast, assess compatibility without relying on client code. Maracas and Sembid are client-agnostic, but the former only considers static compatibility while the latter approximates 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 that is underexplored in related works. Mostafa et al. [25] used this method to collect incompatibilities, suggesting it may be suitable for detection, and Venegas [28] speculates it may address scalability issues in client-specific approaches. This paper therefore proposes a client-agnostic solution using bytecode differencing for static compatibility and cross-version testing for behavioral compatibility, so that compatibility decisions can be reused between clients.

3 Study Overview

This paper combines an empirical study of two dependency smells in the Maven ecosystem with the proposal and evaluation of MaRCo, a tool that can remove these smells. Figure 1 provides a high-level overview over the different parts.

We started by collecting 735 Maven projects from GitHub, which were further filtered based on RQ-specific requirements (1). The collected projects were created between January 1, 2023 and May 1, 2024, and had ≥10 stars and ≥50 commits. First, we study the prevalence of used undeclared dependencies and conflicting soft constraints in the collected projects (2, Section 4). We then present MaRCo and explain how it addresses these smells to improve resolution reliability (4, Section 5). Then, to assess how well the MaRCo-generated ranges preserve compatibility, we evaluate MaRCo on four existing datasets of (mostly) breaking changes (4), and a new

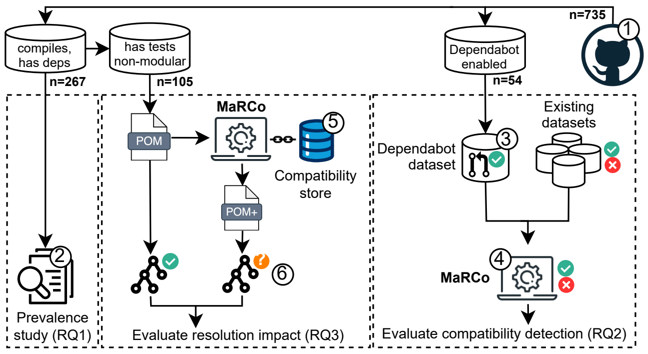

Figure 1: Study overview

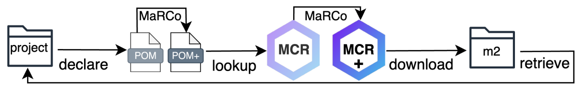

Figure 2: How MaRCo can be applied in the resolution process.

dataset of non-breaking changes which we collect to complement the existing datasets (3, Section 6). Finally, to assess MaRCo's impact on the resolution process, we measure the change in the dependency sets resolved with and without MaRCo and investigate to what extent it reduces the smells from RQ1 (6, Section 7). We conclude the study with a discussion on MaRCo's limitations and future work on the topic of reliable dependency resolution (Section 8).

4 Prevalence of Dependency Smells (RQ1)

We have identified that the use of undeclared dependencies and conflicting soft constraints are dependency smells that negatively affect the reliability of a project's dependency resolution. This section will investigate two aspects of these smells in Maven projects found on GitHub: (i) we will determine how often these two smells appear in practice and (ii) we will investigate how many traces we find of developers manually mediating conflicting soft versions, e.g., through version overriding, to understand related efforts.

We have used the *Maven Dependency Plugin* (MDP) on 266 projects that (i) had dependencies and (ii) compiled. These projects had, on average, 52.9 resolved dependencies and 61.4 unique dependency declarations.

Undeclared Dependencies and Conflicts. Used, but undeclared dependencies decrease resolution transparency, while conflicting soft versions decrease both flexibility (compared to open version ranges) and stability (compared to hard constraints). We investigate the prevalence of both to determine how many projects may have compromised reliability.

Methodology. We use MDP to extract information about the dependency declarations. The `analyze` [3] goal flags transitive dependencies that are used without explicit declaration as a direct dependency as *used undeclared*. Such undeclared dependencies reduce the transparency of the dependency tree and complicate the reasoning about dependency relations, which is why we consider them as anti-patterns in this paper.

The `tree` [4] goal lists the resolved dependency tree and provides additional information. We are interested in the flag *omitted by conflict*, which indicates an intervention of Maven where a concrete dependency declaration was omitted to avoid a conflict with an earlier declaration. This flag will only appear for soft constraints, as Maven will throw a resolution failure when hard constraints like version ranges cannot be fulfilled.

Results. We find that both dependency smells are common in our dataset. 73% had at least one instance of a used undeclared dependency, with 9.5 instances on average. However, these numbers should be interpreted as lower bounds because MDP only detects static instances of this smell. The high lower bound means that any method relying on dependency declaration modification, like MaRCo, should inject missing direct dependencies first. 66% of the projects had at least one instance of conflicting soft version declarations, with 13.0 different conflicting version declarations on average over 11.4 dependencies. For projects with conflicts, 21% of the total declared versions on average are conflicting soft versions. We also observe that projects tend to have many instances of conflicting soft version declarations (13.0 conflicting declarations per project), but each conflict is rather small in scope (1.1 conflicting declarations per conflict).

Manual Conflict Mediation. To gauge whether conflicting soft constraints cause issues for developers, we look into how frequently developers use manual mediation techniques to override Maven's mediation of conflicting soft constraints. Frequent manual mediation could suggest that Maven's nearest-first mediation strategy is insufficient and may in some cases cause issues for developers.

Methodology. A soft version conflict occurs when there are at least two soft constraints for the same dependency with different versions in a project's dependency tree. Maven's conflict mediation strategy is to resolve the version declared closest to the root while ignoring the others [1]. Maven provides two manual mediation techniques to override the resolved version:

(1) Use the `dependencyManagement` section to directly override the resolved version

(2) Exploit Maven's nearest-first mediation by declaring an unused conflicting dependency as a direct dependency

We use MDP again to find these cases of manual conflict mediation. The `analyze` goal can identify *unused declared* dependencies, which could be used to manually solve conflicts. If we are able to find conflicting version declarations for this dependency in the remaining dependency tree, we treat it as a case of *manual mitigation*. Furthermore, we check the output of the `tree` goal. If any entry in the dependency tree is marked as *managed from*, it indicates that the dependent project had declared a `dependencyManagement` section to resolve the conflict.

Results. 67% of the 174 projects with conflicting soft constraints perform manual mediation of at least one conflict, showing that manual conflict mediation is common. However, 64% also have at least one conflict that is not manually mediated. On average, projects that use manual mediation techniques seem to have more conflicting declarations (12.2) than projects with unmanaged conflicts (7.4). These results indicate that Maven's nearest-first mediation strategy is problematic specifically for projects with many conflicting declarations, and suggest the risk of introducing dependency-related issues may increase with increasing conflicting soft constraints.

Conclusion. Both dependency smells are very common in our dataset. 73% of the studied projects use undeclared dependencies and 66% had conflicting version declarations. We found proof in 67% of the affected projects that these conflicts frequently need to be manually mediated. As the discussed mediation strategies are not inherited from parents [44], manual mediation is not sustainable though, as it requires continuous effort by dependents. We can conclude that developers would substantially benefit from improvements in the dependency resolution that reduce or eliminate the two dependency smells.

5 Reliable Dependency Resolution

The results of RQ1 illustrate that reliable dependency resolution requires addressing two common issues in Maven projects: undeclared dependencies and conflicting soft constraints. Undeclared dependencies can lead to unexpected resolution failures, while conflicting soft constraints may cause inconsistencies in resolved versions and prevent updates. This section presents the Maven Compatible Range (MaRCo) toolkit, which consists of the Replacer and the Generator. MaRCo's goal is to improve the reliability of Maven's resolution process without modifying the resolution algorithm itself but by modifying dependency declarations instead. This keeps MaRCo simple since we do not have to consider how different POM configurations affect resolution. Section 5.1 explains how dependencies are declared and retrieved for resolution and how MaRCo modifies the declarations. The remaining sections detail how the Replacer and Generator work to achieve this goal.

5.1 Guiding the Resolution Process

The Maven resolution process can be simplified as follows and is illustrated in Figure 2. A project declares its direct dependencies in its POM file. Maven downloads the dependency artifacts from the Maven Central Repository (MCR) into the client's `m2` folder. The artifacts include the declared dependencies' POM files and bytecode packaged as JARs. Because the former contains the project's transitive dependencies, Maven repeats the download for each dependency's POM file. Once all dependencies are downloaded, they are retrieved for resolution. MaRCo modifies the dependency declarations used in the resolution process in two parts. First, MaRCo modifies the project's POM by replacing the direct dependencies' soft constraints with compatible version ranges. However, we also need to replace the transitive dependencies' soft constraints stored in the dependencies' POMs in the client's `m2` folder. To modify the transitive POMs we can therefore either apply MaRCo to the `m2` folder, or to (parts of) MCR.

If MaRCo is to be applied in a real-world setting the most practical solution is to use MaRCo as described in Figure 2. This would require setting up and maintaining a mirror repository to MCR containing MaRCo-replaced POMs. Using the mirror repository, the Maven resolution process will download the pre-replaced POMs, and developers would only have to apply MaRCo to their own POM. Setting up and maintaining a proper mirror repository is out of scope for this paper. To emulate the mirror repository for experiments requiring POM replacement (RQ3), full dependency replacement is instead done by applying MaRCo to all POMs in the `m2` folder. There is no difference between the two approaches from

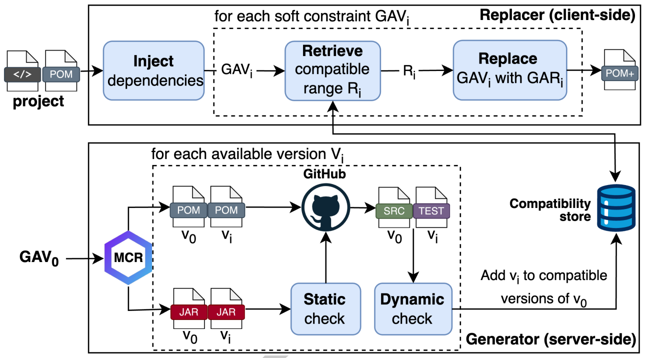

Figure 3: Overview of the MaRCo Replacer and Generator.

the perspective of the Maven resolver, so this choice should not impact the evaluation.

MaRCo achieves a more reliable dependency resolution by injecting the missing dependency declarations to increase transparency, and by replacing soft constraints with compatible ranges to balance stability and flexibility. Incorporating MaRCo into a developer's workflow should require minimal effort and overhead, and therefore consists of two components: a lightweight Replacer used by the developer client-side, and a server-side Generator, see Figure 3. The Replacer is responsible for replacing a project's POM with a modified POM containing the injected and replaced dependency declarations. It is designed so that developers would not need to change how they declare dependencies: they could 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 precomputed and reused between clients to reduce overhead.

5.2 Modifying the Dependency Declarations

The Replacer takes a Maven project as input and outputs a new POM without the previously introduced dependency smells. The smell of missing direct dependencies is removed by declaring all dependencies directly used as direct dependencies with the help of MDP [2]. Removing the smell of soft constraints involves replacing the soft constraints with compatible ranges which are fetched from the compatibility mapping precomputed by the Generator.

The compatibility mapping maps a dependency version to a list of compatible versions which is converted into the Maven range format [7]. A range is defined by the string `[lowerBound, upperBound]`, where the lower bound is the oldest and the upper bound is the newest version according to Maven's sorting algorithm [38]. The range includes all available versions between the lower and upper bounds and multiple ranges can be concatenated by commas. The compatible version list is therefore converted to a compatible range by first sorting it with Maven's sorting algorithm, then identifying the upper- and lower bounds. To identify whether there are gaps in the compatible range, we fetch the dependency's available versions from MCR and check whether all available versions between the lower and upper bounds are in our compatible version list. If there are gaps, we identify the continuous compatible ranges and concatenate them.

The Replacer works as follows. MDP's *analyze* goal [3] is first run on the project and returns the used undeclared dependencies. These declarations are then injected into the POM's `dependency` section. Because the returned dependency declarations contain soft constraints, the injection step is necessary before replacing the POM's soft constraints.

5.3 Generating Compatible Versions

To increase the reliability of the dependency resolution process, the Replacer replaces soft constraints with compatible ranges. To enable the Replacer to be lightweight, the Generator pre-computes a dependency's compatible versions using a client-agnostic compatibility approach. The Generator has two responsibilities: (i) perform compatibility checking and store the results in the compatibility mapping storage, and (ii) serve requests of specific compatibility mappings by the Replacer. As seen in Fig. 3, the MaRCo Generator runs server-side and is not intended for client use unless they want to compute and host their own compatibility mappings.

The Generator works as follows. Given a specific dependency version v (*base*), the Generator fetches the dependency's available versions av (*candidates*), and computes whether they are compatible with the base. 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. The compatibility check can fail at any step, and a candidate is only added to the mapping if it passes all steps.

Computing static and dynamic compatibility. The compatibility checking component implements a client-agnostic approach to compatibility. The compatibility check for a version pair consists of a static and a dynamic check. The static check uses japicmp, a common static compatibility checker for Java [27, 44]. The static check is run before the dynamic check because bytecode differencing is computationally cheap compared to cross-version testing and only requires access to the base and candidate JARs which are provided by MCR. 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 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 candidate's 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, then with the candidate's code. The test results of the base code establish the baseline that the test results of the candidate are compared to. The candidate is considered incompatible with the base if it returns more test failures than the baseline, and otherwise compatible.

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 [16]. 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 [6] of the (parent) POM. It then extracts the version from the `tag` [5] if available, and otherwise from the GAV. Finally, it retrieves the 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. [19] and performs similarly on the Reproducible Central dataset [33]: The algorithm successfully linked 63% of GAVs to a repository and tag with 96% accuracy, while 37% could not be linked mainly due to missing GitHub URLs in the POMs. Using this approach, we located tests for 53% of GAVs, while only 2% had test jars on MCR, showing its effectiveness for test recovery.

6 Breaking Change Detection (RQ2)

This section investigates the extent to which a client-agnostic compatibility approach can detect (non-)breaking changes as detected by client-specific approaches. The remainder of the section will introduce the datasets we use for the evaluation, our preprocessing of the data, and our evaluations of the compatibility classification of version pairs (updates) and the ability to detect compatible versions.

Datasets. We evaluate MaRCo on four pre-existing datasets to get a thorough understanding of how it performs at detecting (non-) breaking changes. Because the existing datasets contain mostly breaking changes, we create the complementary Dependabot dataset of non-breaking changes. Each dataset is cleaned and prepared into a compatible format. MaRCo can either take a version pair as input and outputs the compatibility decision (classification), or it can take a single version as input and outputs its compatible versions (retrieval). All datasets are client-specific, meaning that they may contain duplicate data points across clients. Because MaRCo is client-agnostic, these duplicates are filtered out.

BUMP. The BUMP [35] benchmark, cloned on March 22, 2024, contained 571 breaking dependency updates obtained from GitHub PRs opened by Dependabot, which were rejected due to failing builds. For each data point, we extract the dependency name (groupId and artifactId) and the old and new versions defining the update. This information determines the GAVs of the base (old) and candidate (new) versions used for MaRCo's compatibility check.

We filter out duplicates and irrelevant dependency updates. We discard *plugin*-type updates because MaRCo does not replace plugin declarations, and *POM*-type updates because we do not know which dependency update in the POM caused the breaking change. Finally, each data point has a failure category. We discard *dependency version lock* and *enforcer rule* failures because they are caused by project-specific POM configurations that are irrelevant for client-agnostic compatibility. After all filtering steps, we were left with a cleaned dataset of 372 data points.

Depandabot. Similarly, the Dependabot dataset contains 793 non-breaking dependency updates which we extracted from 1,368 Dependabot PRs that were approved without changes from the 54/735

GitHub projects with Dependabot updates enabled for Maven. Assuming developers reject updates that break their projects, this serves as a reasonable heuristic for identifying compatible updates.

Uppdatera. The Uppdatera [18] dataset contains 19 breaking and non-breaking dependency updates, including a manually validated ground truth, test results of the client, and the Uppdatera results. No preprocessing was necessary.

CompCheck. The CompCheck [46] dataset contained 756 client-dependency update pairs that have been flagged by CompCheck as incompatible. After removing duplicate updates across clients, 634 unique dependency updates remain.

Ranger. Different from the others, the Ranger [44] dataset contains client-specific compatible version *ranges* for specific GAVs obtained by Ranger. The original dataset consisted of 4,107 data points. Because MaRCo is client-agnostic, we merge the ranges of duplicate GAVs across clients, resulting in 480 data points.

Each of the five datasets provides a unique perspective for our evaluation. The BUMP and Dependabot datasets help us evaluate MaRCo's ability to detect (non-)breaking changes, whereas the Uppdatera, CompCheck, and Ranger datasets allow us to compare MaRCo to client-specific solutions that cannot easily be rerun. A client-agnostic approach will overestimate breaking changes since a dependency may contain breaking changes that are unreachable by a specific client, so we expect a high recall on BUMP and a lower recall on the Dependabot dataset. On the other hand, client-specific approaches that base behavioral compatibility decisions on client tests may underestimate breaking changes since it is uncommon for clients to test their dependencies [18, 46].

Compatibility Classification of Version Pairs. Each version pair (v_{old}, v_{new}) in the BUMP, Dependabot, Uppdatera and CompCheck datasets represents a unique dependency update. Whether this update is breaking (incompatible) or non-breaking (compatible) is a binary classification problem. When generating compatible versions ranges, MaRCo starts by performing such a classification for a given version pair. Every non-breaking version v_{new} is marked as compatible with v_{old}. Applying the classification step of MaRCo on these datasets will evaluate how effective MaRCo is at classifying (non-)breaking changes.

Methodology. We evaluate the performance of MaRCo using recall and precision, which are commonly used to evaluate binary classification models [9]. The BUMP, Dependabot, and CompCheck datasets only contain one compatibility class, which is chosen as the positive label P. Uppdatera contains both compatibility classes, and we choose the breaking class as P since it is the minority class. The precision for the BUMP, Dependabot and CompCheck datasets is trivially 1.0 and therefore not reported.

Results. The outcome of whether a version pair is compatible using MaRCo can be one of three options. The pair is compatible if it passes both the static and dynamic compatibility checks. The pair is incompatible if it fails both checks. If MaRCo fails to evaluate the data point, the compatibility outcome is inconclusive. The outcome of applying MaRCo to each of the classification datasets is shown in Table 1. Figure 4 shows the causes of inconclusive data points. The evaluation metrics for each dataset are shown in Table 2.

Table 1: MaRCo results on the classification datasets.

	BUMP	Dependabot	Uppdatera	CompCheck
Total	371	793	19	634
Statically ...				
incompatible	346	262	7	511
compatible	25	455	11	123
Linked	16	304	9	67
Runnable	3	60	1	1
Dynamically ...				
incompatible	1	8	0	1
compatible	2	51	1	0
Total conclusive	349	321	9	512
Total inconclusive	22	472	10	122

Table 2: Summary of classification performance evaluations

Dataset	Class	Method	Success	Rec.	Prec.
BUMP	breaking	MaRCo	0.94	0.99	
Dependabot	non-breaking	MaRCo	0.40	0.19	
Uppdatera	both	MaRCo	0.47	1.00	0.50
		Uppdatera		1.00	0.43
		Project tests		0.25	0.50
CompCheck	breaking	MaRCo	0.81	1.00	

In the BUMP benchmark, MaRCo successfully evaluated 349/371 data points (success rate: 0.94). Out of these, MaRCo labeled 347 points correctly as incompatible and labeled two points incorrectly as compatible (recall: 0.99). The incorrect labels 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 the dependency maintainers introducing (un)intentional changes that were not caught by their tests. For example, `plexus-io:3.3.0` introduced a breaking bug [39], which was fixed in 3.3.1 and a regression test was added [12]. This means MaRCo will consider 3.3.0 incorrectly compatible with older versions prior to 3.3.1. However, it will also correctly list 3.3.1 as a compatible update for 3.2.0, so any project using 3.2.0 would resolve the bug-free 3.3.1 since it is the latest version. Assuming bugs are eventually patched, MaRCo prevents buggy updates. 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 [10], but within one of the clients [8]. This proves Hyram's law [42] by illustrating that bug fixes can break clients that rely on the buggy behavior.

For the Dependabot dataset, MaRCo successfully evaluated 322/793 data points (success rate: 0.40). Out of the these, MaRCo labeled 51 points correctly as compatible and 270 points incorrectly as incompatible (recall: 0.19). Although MaRCo assigns the majority of data points the correct label in the static and dynamic stage, the recall is low because 87% of data points are lost between static and dynamic evaluation. This causes the number of data points that are determined statically incompatible to be disproportionately high compared to the number of data points for which we could get a conclusive dynamic compatibility result. If we calculate recall separately for the static and dynamic evaluation stages, we get a static recall of 0.63 and a dynamic recall of 0.86.

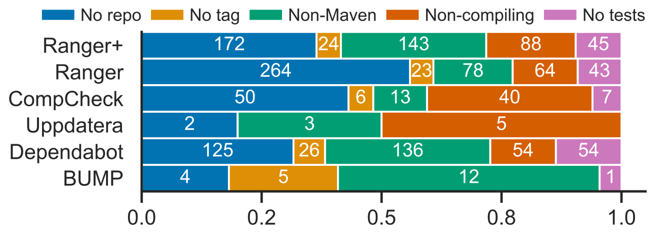

Figure 4: Causes of inconclusive evaluations.

All results for the Uppdatera dataset are listed in Table 3. Our definitions of true/false positive/negative labels differ slightly from the the original dataset: in our case, true positives (TP) are pairs correctly identified as incompatible, false positives (FP) are incorrectly identified as incompatible, true negatives (TN) are correctly identified as compatible, and false negatives (FN) are incorrectly identified as compatible.

MaRCo successfully evaluated 9/19 data points (success rate: 0.47). On the evaluated data points, MaRCo has a recall of 1.0 and a precision of 0.50, Uppdatera has a recall of 1.0 and precision of 0.43, and project tests have a recall of 0.25 and a precision of 0.50. MaRCo agreed with the ground truth in 5 (56%) cases. The 4 incorrectly evaluated cases were false positives due to static incompatibilities that were unreachable by the project performing the update. Despite the high false positive rate (4/9), the false negative rate is 0, which shows that MaRCo is better suited for stability over flexibility.

For the data points where MaRCo disagreed with the ground truth, it performs similarly to Uppdatera and better than project tests. MaRCo did not catch more correct decisions than Uppdatera, confirming our initial intuition that client-agnostic approaches overestimate breaking changes. Two data points were inconclusive due to their GitHub repositories being unavailable. The remaining inconclusive data points (8) were caused by failing to run the dynamic compatibility check: the GitHub repositories either used non-Maven build systems (3), or could not be compiled (5).

For the CompCheck dataset, MaRCo successfully evaluated 512/634 data points (success rate: 0.81). All data points were correctly labeled as incompatible, resulting in a recall of 1.0. Most of the data points (81%) were correctly labeled as incompatible by the static compatibility check, but a large part (55%) of the remaining 123 points became inconclusive before reaching the dynamic compatibility check. Figure 4 shows that the majority of these data points can either not be linked to a repository (50) or not be compiled (40).

Retrieval of Compatible Versions. The Ranger dataset maps a specific dependency version (GAV) to its compatible range. The compatible range can be expanded to a list of compatible versions by retrieving the available versions within the range, and evaluation on this dataset can be modeled as a retrieval problem. Applying MaRCo on the Ranger dataset will evaluate how effective it is at retrieving all compatible versions for a given GAV.

Methodology. We evaluate MaRCo's performance on the retrieval task using recall and precision, which are also commonly used to evaluate retrieval models [9, 36]. Let V_m and V_r be the set of compatible versions retrieved by MaRCo and Ranger, respectively. We define recall as $|V_m \cap V_r| \,/\, |V_r|$ and precision as $|V_m \cap V_r| \,/\, |V_m|$.

Each data point in the Ranger dataset maps a GAV to a range. We provide the GAV as input to MaRCo and record the range it outputs. The ranges are converted to version lists by looking up the available versions on MCR between the lower and upper bounds. We only considered versions available before April 1, 2023 to avoid MaRCo including versions in its ranges that Ranger did not evaluate.

Results. MaRCo only successfully evaluated 8/480 data points (success rate: 0.02). We show the full evaluations for these data points in Table 4. Figure 4 shows the causes of inconclusive data points. Because the amount of inconclusive data points caused by the GitHub repository not being found is so high (55%), we used deps.dev [29] to manually restore the missing repositories. However, the restoration did not result in the successful evaluation of more data points. Instead, the main causes of inconclusive data points shifted from link (no repo or tag) to run errors (non-Maven, non-compiling, or no tests). Noticeably, 30% of data points failed to run because they used non-Maven build systems.

For the data points MaRCo did evaluate, the recall is generally higher than the precision with a macro average of 0.66 and 0.52, respectively. This means MaRCo was in general able to retrieve most versions deemed compatible by Ranger, but also included versions that Ranger did not. In other words, the ranges generated by MaRCo are more flexible than those by Ranger even though we would expect the opposite since Ranger is client-specific. This could be because Ranger avoids versions with the Log4Shell vulnerability, and because MaRCo includes downgrades, which Ranger does not.

Conclusions. MaRCo's client-agnostic compatibility approach is effective at detecting breaking changes with high recall ($\geq$0.99) across all breaking change datasets. While it detects non-breaking changes with a static recall of 0.63 and a dynamic recall of 0.86, the overall recall is low (0.19) due to the dynamic check failing to evaluate a majority of data points. We therefore conclude that the client-agnostic approach is effective, but limited by a low success rate when detecting non-breaking changes. This problem is further exacerbated by dependencies that use non-Maven build systems.

As expected, MaRCo overestimates breaking changes as seen by the low false negative and high false positive rates on the Uppdatera dataset, sacrificing precision for recall. The high false positive rate reduces the size of the ranges resulting in lower flexibility, but the low false negative rate improves stability since the ranges are unlikely to contain breaking changes.

7 Impact on Maven's Resolution (RQ3)

We compare the dependency trees of the GitHub projects before and after applying MaRCo to assess how it affected the dependency resolution processes of the projects. To see whether MaRCo has any effect on the dependency smells, we also rerun the RQ1 experiments on the projects after applying MaRCo. Since RQ2 showed that MaRCo has a high false positive but a low false negative rate for detecting breaking changes, we expect it to influence the resolution process to favor stability over flexibility.

Methodology. We apply MaRCo to the projects with dependencies that compile and have tests so that we can verify that no static or dynamic breaking changes were introduced. We discard multi-module projects to simplify POM replacement, resulting in 105 eligible projects. Before we can apply the Replacer to the projects, we first need to use the Generator to compute the compatibility mappings

Table 3: Comparison of the compatibility evaluations based on manual evaluation, project tests, the Uppdatera tool, and MaRCo

PR	GA	Old version	New version	Manual	Tests	Uppdatera	MaRCo
spotify/dbeam#189	org.apache.avro:avro	1.9.1	1.9.2	N	FP	FP	FP (stat. incompat.)
airsonic/airsonic#1622	org.apache.commons:commons-lang3	3.9	3.10	N	TN	FP	FP (stat. incompat.)
bitrich-info/xchange-stream#570	com.pubnub:pubnub-gson	4.31.0	4.31.1	N	TN	FP	- (no Maven)
CROSSINGTUD/CryptoAnalysis#245	org.eclipse.emf:org.eclipse.emf.common	2.15.0	2.18.0	N	TN	FP	FP (stat. incompat.)
dbmdz/imageio-jnr#84	com.github.jnr:jnr-ffi	2.1.12	2.1.13	N	TN	FP	- (no compile)
dnsimple/dnsimple-java#23	com.google.code.gson:gson	2.2.4	2.8.6	N	TN	TN	FP (stat. incompat.)
smallrye/smallrye-config#289	io.smallrye.common:smallrye-common-expr.	1.0.0	1.0.1	N	TN	TN	- (no compile)
dropwizard/metrics#1567	org.jdbi:jdbi3-core	3.12.2	3.13.0	N	TN	TN	- (no compile)
s4u/pgverify-maven-plugin#96	io.github.resilience4j:resilience4j-retry	1.3.1	1.4.0	N	TN	TN	- (no Maven)
UniversalMediaServer/UniversalMediaServer#1987	org.apache.commons:commons-text	1.3	1.8	P	FN	TP	TP (dyn. incompat.)
CSUC/wos-times-cited-service#36	org.apache.httpcomponents:httpclient	4.5.11	4.5.12	N	TN	FP	- (no GitHub)
Grundlefleck/ASM-NonClassloadingExtensions#25	org.ow2.asm:asm-analysis	7.0	8.0.1	N	TN	FP	- (no GitHub)
RohanNagar/lightning#211	io.dropwizard:dropwizard-auth	1.3.17	2.0.8	P	TP	FP	TP (stat. incompat.)
zalando/riptide#932	io.micrometer:micrometer-core	1.3.6	1.4.1	P	FN	TP	TP (stat. incompat.)
pinterest/secor#1273	com.amazonaws:aws-java-sdk-s3	1.11.763	1.11.764	N	TN	TN	- (no compile)
michael-simons/neo4j-migrations#60	io.github.classgraph:classgraph	4.8.68	4.8.71	N	TN	TN	TN (compatible)
zaproxy/crawljax#115	org.apache.commons:commons-lang3	3.3.2	3.10	P	FN	TP	TP (stat. incompat.)
hub4j/github-api#793	com.squareup.okio:okio	2.5.0	2.6.0	N	TN	TN	- (no Maven)
http://zalando/logbook#750	io.netty:netty-codec-http	4.1.48.Final	4.1.49.Final	N	TN	TN	- (no compile)

Table 4: Comparison of the ranges produced by Ranger and MaRCo on the Ranger dataset

	Ranger		MaRCo			
GAV	Range	#	Range	#	Recall	Precision
com.indoqa:indoqa-boot:0.12.0	[0.12.0, 0.16.0]	5	[0.10.0, 0.16.0]	7	1.00	0.71
eu.unicore.security:securityLibrary:5.3.1	[5.3.1, 5.3.2]	2	[5.3.0, 5.3.6]	6	1.00	0.33
org.dhatim:dropwizard-sentry:2.0.25-2	[2.0.25-2, 2.0.26-1]	2	[2.0.25, 2.1.2-4]	33	1.00	0.06
org.robotframework:jrobotremoteserver:4.0.1	[4.0.1, 4.1.0]	2	[4.0.0, 4.1.0]	3	1.00	0.67
org.spdx:spdx-tools:2.2.5	[2.2.5, 2.2.6]	2	[2.2.5, 2.2.5]	1	0.50	1.00
com.helger.photon:ph-oton-bootstrap4-stub:8.3.2	[8.3.2, 8.3.3]	2	[8.2.5, 8.3.2]	8	0.50	0.13
com.hotels:waggle-dance-api:3.9.8	[3.9.8, 3.10.12]	16	[3.9.0, 3.9.9]	8	0.13	0.25
org.biojava:biojava-core:4.2.0	[4.2.0, 6.0.2]	47	[4.2.0, 4.2.7]	8	0.17	1.00

for all dependencies, and then we need to apply the Replacer to the dependencies' POMs. Only after the compatibility mappings have been computed and the dependency POMs have been replaced can we apply the Replacer to the projects. To determine how the resolution process has changed we calculate different metrics using the dependency trees before and after applying the Replacer to the projects. Applying MaRCo to the previously collected projects involves the following steps:

(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.
(4) Re-build the projects, re-run the tests, and re-generate the new dependency trees. The trees, compilation, and test results are then used to compute the evaluation metrics.

The process is repeated until the resolution result is stable. Finally, to assess MaRCo's impact on the dependency smells, we rerun the RQ1 experiments on projects where at least one dependency declaration was successfully replaced.

Evaluation metrics. We define several metrics to measure how the dependency resolution process has changed for each project. We also look at the sizes of the generated ranges as larger ranges provide more flexible resolution.

Success rate. The fraction of projects that still resolve, compile, and pass their tests after having at least one dependency declaration replaced. Low success rates indicate a less flexible resolution process if there is increased resolution failure, or less stable if there is increased compilation or test failure.

Replacement rate. The fraction of resolved dependencies that originate from a replaced dependency declaration. MaRCo's impact on the resolution depends on how many declarations it replaces.

Downgrades (and Upgrades). The number of GAs for which a lower (or higher) version is resolved. *Downgrade steps* (and *upgrade steps*) are then the number of versions each downgrade (or upgrade) is behind (or ahead) of the previously resolved version.

Change rate. The fraction of GAs that resolve to different versions.

Change magnitude. is the total magnitude of version changes and is defined as the sum of upgrade and downgrade steps.

Together, *Change Rat*e and *Change Magnitude* indicate how flexible the resolution process is, since it measures how much the newly resolved versions deviate from the previously resolved versions.

Results. MaRCo replaced at least one dependency declaration in 88/105 projects. The resulting metrics are shown in Table 5. After a

Table 5: RQ3 evaluation metric results

Metric	Min	Max	Mean	Median
Resolved dependencies	2	489.0	58.6	44.0
Change rate	0	0.5	0.1	0.0
Change magnitude	0	43.7	2.4	1.0
Upgrades	0	13.0	1.9	1.0
Upgrade steps	0	100.0	7.0	1.5
Downgrades	0	3.0	0.1	0.0
Downgrade steps	0	131.0	2.6	0.0
Range size	1	104.0	3.9	2.0

replacement of, on average, 13% resolved dependencies, 82 projects could be resolved and compiled, and 81 passed their tests, which represents a success rate of 92% (81/88) for the replacements. The high success rate indicates that the ranges are unlikely to introduce breaking changes, and flexible enough to not cause resolution failures in most cases. However, this number may be artificially high due to the relatively low average replacement rate of 13%.

There is a large variation in the number of versions included in the generated ranges, with sizes from 1-104. While the majority of the ranges only contained a single version, 40% also contained 2-6 versions with an overall average of 3.82. 6% of dependencies resolve a different version than before with an average change magnitude of 2.42, showing added flexibility in the resolution process. On average, upgrades were more common than downgrades, and upgrades were on average larger in magnitude (7.01 versions versus 2.61).

After re-running the RQ1 experiments on the projects with successful replacements, we see that projects with undeclared dependencies were reduced by 91%, while projects with conflicts increased by 59%. Rerunning the experiments without injection showed no significant change in conflicts. The increase in conflicts with injection shows that it improves transparency by making previously hidden conflicts explicit. The low replacement rate is likely the main cause of MARCo's inability to reduce conflicts. To see whether an increased replacement rate would help, we re-applied MARCo to the projects using only statically compatible versions and reran the experiments again. To avoid breaking the Maven tooling used by MARCo, we did not replace declarations of plexus-utils, commons-collections, and velocity, and did no replacements in org.apache.maven dependencies, affecting 13 declarations over 8 projects. 20 projects still resolved with injection, 34 without. Unmanaged conflicts and undeclared dependencies were reduced by 100%. Projects with managed conflicts were unchanged, however, due to lacking range support in `dependencyManagement`: the newest version in the range is resolved regardless of whether another version would have no conflicts.

To conclude, MARCo can improve Maven's dependency resolution to some extent, with 13% of dependencies in 78% of projects resolving from the MARCo-generated version ranges which are unlikely to contain breaking changes, thereby increasing stability. However, most ranges are hard constraints, which lowers flexibility compared to soft constraints. Furthermore, due to the low replacement rate, we are unable to observe reductions in conflicts. By only considering static compatibility to boost the replacement rate, MARCo eliminates all unmanaged conflicts, but increases resolution failure. From this, we conclude that the client-agnostic approach can provide a more reliable resolution that favors stability, but it results in too inflexible constraints.

8 Discussion and Future Work

This paper provides the design and evaluation of MARCo, a toolkit that influences Maven's dependency resolution towards being more reliable to decrease dependency-related issues and improve developer efficiency. We found that used undeclared dependencies and conflicting soft version constraints are common dependency smells that negatively affect resolution transparency, flexibility, and stability. MARCo therefore increases transparency by injecting the missing dependencies and balances stability and flexibility by replacing soft version constraints with client-agnostic compatible version ranges. This section reflects on the findings and limitations of this work and provides possible future work.

Inconclusive dynamic check reduces impact. MARCo's main limitations relate to its impact. The fewer dependencies it can evaluate for compatibility, the fewer dependency declarations it can replace, and the less impact it will have on the dependency resolution. These issues originate from the dynamic compatibility check, which requires locating, compiling and running dependency tests. First, linking a GAV to its GitHub repository via its POM is not always possible. Second, compiling arbitrary Maven projects from GitHub is not trivial [19]. Future work using cross-version testing should therefore consider these issues and could investigate static methods to approximate behavioral compatibility or test generation when dependency tests are unavailable.

Compatibility is subject to test coverage. Whether the generated compatible ranges are free of behavioral breaking changes depends on the quality of the available test suite. Low-quality test suites will catch fewer breaking changes than extensive ones, as seen by the two breaking changes in the BUMP benchmark 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 [15]. Test generation could not only reduce the number of false positives caused by low coverage but also the number of inconclusive compatibility decisions due to unavailable tests. Tests may also cover internal API methods.Future work could exclude tests covering the internal API to reduce false dynamic incompatibilities, similar to how MARACAS exclude internal API sections using annotations to reduce false static incompatibilities.

Client-agnostic compatibility limits flexibility. RQ2-3 showed that MARCo provides stable ranges that are stricter than necessary, reducing the potential flexibility of the dependency resolution. This is logical since not every client is affected by every breaking change. By extending the client-agnostic approach with client-specific analysis, the ranges could be expanded to improve flexibility without compromising stability. Future work could therefore consider such hybrid approaches that combine the scalability and stability provided by the client-agnostic approach and the flexibility provided by client-specific approaches. For example, a hybrid approach could use MARCo's client-agnostic approach to build a knowledge base of incompatibilities. Reachability analysis could then be employed to see whether the client reaches any parts of the dependency that contain breaking changes. The precomputed client-agnostic compatible ranges can then be expanded with previously incompatible versions with unreachable breaking changes.

Dynamic language features cause missed injections. MaRCo improves the transparency of the dependency resolution process by injecting used undeclared dependencies, but its effectiveness is limited by MDP which cannot detect undeclared dependencies used via dynamic language features such as Java's Reflection API. Because as much as 78% of Java programs use reflection [20], the prevalence of missing declarations is likely higher than RQ1 shows. Used undeclared dependencies may cause resolution, compilation, or run failure if its transitive declaration disappears or changes to a non-compatible version. Because we observe a low failure rate in RQ3, we believe missing injections do not threaten the evaluation of MaRCo; however, future work could consider how we can improve detection. Slimming [37] detects unused declared dependencies while considering reflection usage, and could perhaps be used to detect used undeclared dependencies as well.

Scaling to ecosystem-level deployment. MaRCo currently performs full dependency replacement by replacing the dependency POMs in the client's `m2` folder. Section 5 presented a more practical solution, using a mirror repository of MCR with pre-replaced POMs. How to efficiently set up and maintain this mirror repository for a large ecosystem with frequent dependency releases is another direction for future work. Furthermore, the compatibility mappings are generated using regression testing, which is computationally expensive [16]. Future work could consider looking into regression test selection techniques to optimize catching as many breaking changes as possible with the least computational overhead.

Hidden efforts. As we have seen in RQ1, many soft conflicts need manual mediation, yet, the actual effort that developers have to invest to handle intransparent or inflexible dependency management remains invisible in the released artifacts. These issues typically arise much earlier in the development process and must be addressed long before a library is published in repositories such as Maven Central. Future work should conduct a field study to quantify the true cost of inadequate tooling and understand its impact on developers.

Research-friendly tooling. Some software ecosystems are more open to experiments with dependency management. For example, NPM allows the registration of custom resolvers, which enables researchers and developers to test different resolution strategies without disrupting their entire development process. In contrast, Maven tightly integrates the default dependency resolution into many of its built-in plugins, which makes experimentation significantly more challenging.

For this paper, we have explored the idea of mutating the local `.m2` folder by altering POM files. While this approach works in the controlled setting of an experiment, it is not a practical solution since the folder is shared by all Maven projects on the system. Further complicating the matter, the `mvn` build command itself relies on dependencies from this folder, meaning that any mutation affects not only the studied target projects, but also the build tooling, which introduces significant confounding factors.

Future work should improve the study of dependency resolution, either by developing better ways to integrate with existing tools or by exploring ecosystems that reduce engineering overhead and allow researchers to focus on dependency management challenges rather than tool integration.

Threats to validity. MaRCo may contain bugs affecting the correctness of its output. Although MaRCo is unit tested to verify expected behavior, the tests are not exhaustive. The reproducibility of our results may be affected by software, hardware, and temporal dependencies. Some projects may only resolve, compile, or run on specific platforms or hardware. Because GitHub repositories and tags are mutable, future GitHub linking may produce different results. We have therefore provided hardware, software, and time details for the experiments. The representativeness of the datasets impacts our findings' generalizability. RQ1 and RQ3 use relatively small samples of Maven projects which may not fully represent the overall population. However, our preliminary results reveal interesting findings motivating a larger-scale prevalence study. Furthermore, the five datasets used to evaluate MaRCo show consistent results, increasing confidence in our findings.

9 Summary

We define transparency, stability, and flexibility as core properties of a reliable resolution process that mitigates dependency-related issues, and used undeclared dependencies and conflicting soft constraints as two common smells that negatively affect these properties. We found that developers often manually resolve conflicting soft constraints, especially in projects with many conflicting constraints. This implies having conflicting soft constraints risks dependency-related issues requiring manual effort to fix. MaRCo increases transparency by injecting missing dependencies and balances stability and flexibility by replacing soft constraints with compatible ranges. Modifying the dependency declarations in this manner could simplify dependency management for developers, allowing them to specify one version while the resolution considers all compatible versions. MaRCo's evaluation shows that the generated ranges are unlikely to contain breaking changes, and can be used as a scalable basis for client-specific approaches to expand the precomputed compatible ranges to improve flexibility. We hope these results can contribute to further discussions and future work on reliable dependency resolution.

References

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

[2] Apache Maven Project. 2023. Apache Maven Dependency Plugin. https://maven.apache.org/plugins/maven-dependency-plugin/. accessed 04-Jun-2024.

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

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

[5] Apache Maven Project. 2023. Maven SCM Plugin - scm:tag. https://maven.apache.org/scm/maven-scm-plugin/tag-mojo.html. accessed 20-Jan-2024.

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

[7] Apache Maven Project. 2024. Version Range Specification. https://maven.apache.org/enforcer/enforcer-rules/versionRanges.html. accessed 26-Jun-2024.

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

[9] Gürol Canbek, Tugba Taskaya Temizel, and Seref Sagiroglu. 2022. PToPI: A comprehensive review, analysis, and knowledge representation of binary classification performance measures/metrics. *SN Computer Science* 4, 1 (2022), 13.

[10] Uriel Chemouni. 2023. Release V 2.4.9. https://github.com/netplex/json-smart-v2/releases/tag/2.4.9. accessed 02-Jun-2024.

[11] Lingchao Chen, Foyzul Hassan, Xiaoyin Wang, and Lingming Zhang. 2020. Taming behavioral backward incompatibilities via cross-project testing and analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 112–124.

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

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

[14] Jens Dietrich, David Pearce, Jacob Stringer, Amjed Tahir, and Kelly Blincoe. 2019. Dependency versioning in the wild. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories*. 349–359.

[15] Gordon Fraser and Andrea Arcuri. 2011. 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*. 416–419.

[16] Alex Gyori, Owolabi Legunsen, Farah Hariri, and Darko Marinov. 2018. Evaluating regression test selection opportunities in a very large open-source ecosystem. In *2018 IEEE 29th International Symposium on Software Reliability Engineering*. 112–122.

[17] Runzhi He, Hao He, Yuxia Zhang, and Minghui Zhou. 2023. Automating dependency updates in practice: An exploratory study on github dependabot. *IEEE Transactions on Software Engineering* (2023).

[18] Joseph Hejderup and Georgios Gousios. 2022. Can we trust tests to automate dependency updates? a case study of java projects. *Journal of Systems and Software* 183 (2022), 111097.

[19] Mehdi Keshani, Tudor-Gabriel Velican, Gideon Bot, and Sebastian Proksch. 2024. AROMA: Automatic Reproduction of Maven Artifacts. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 836–858.

[20] Davy Landman, Alexander Serebrenik, and Jurgen J Vinju. 2017. Challenges for static analysis of java reflection-literature review and empirical study. In *2017 IEEE/ACM 39th International Conference on Software Engineering*. 507–518.

[21] Adrienn Lawson and Stephen Hendrick. 2023. Global Spotlight 2023: Survey-based insights into the global landscape of open source trends, sustainability challenges, and growth opportunities. The Linux Foundation.

[22] Stephen McCamant and Michael D Ernst. 2004. Early identification of incompatibilities in multi-component upgrades. In *European Conference on Object-Oriented Programming*. 440–464.

[23] Samim Mirhosseini and Chris Parnin. 2017. Can automated pull requests encourage software developers to upgrade out-of-date dependencies?. In *2017 32nd IEEE/ACM international conference on automated software engineering*. 84–94.

[24] Martin Mois. 2024. japicmp. https://siom79.github.io/japicmp/. accessed 02-Jun-2024.

[25] Shaikh Mostafa, Rodney Rodriguez, and Xiaoyin Wang. 2017. 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*. 215–225.

[26] Suhaib Mujahid, Rabe Abdalkareem, Emad Shihab, and Shane McIntosh. 2020. Using others' tests to identify breaking updates. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 466–476.

[27] Lina Ochoa, Thomas Degueule, Jean-Rémy Falleri, and Jurgen Vinju. 2022. Breaking bad? semantic versioning and impact of breaking changes in maven central: An external and differentiated replication study. *Empirical Software Engineering* 27, 3 (2022), 61.

[28] Lina Maria Ochoa Venegas. 2023. Break the Code?: Breaking Changes and Their Impact on Software Evolution. [Phd Thesis 1 (Research TU/e / Graduation TU/e), Mathematics and Computer Science].

[29] Open Source Insights. [n. d.]. Understand your dependencies. https://deps.dev/. accessed 02-Jun-2024.

[30] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. 2020. A qualitative study of dependency management and its security implications. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*. 1513–1531.

[31] Tom Preston-Werner. [n. d.]. Semantic Versioning 2.0.0. https://semver.org/. accessed 03-Jun-2024.

[32] Steven Raemaekers, Arie van Deursen, and Joost Visser. 2017. Semantic versioning and impact of breaking changes in the Maven repository. *Journal of Systems and Software* 129 (2017), 140–158.

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

[34] revapi.org. 2023. Revapi. https://revapi.org/. accessed 02-Jun-2024.

[35] Frank Reyes, Yogya Gamage, Gabriel Skoglund, Benoit Baudry, and Martin Monperrus. 2024. BUMP: A Benchmark of Reproducible Breaking Dependency Updates. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering*. 159–170.

[36] Amit Singhal. 2001. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.* 24, 4 (2001), 35–43.

[37] Xiaohu Song, Ying Wang, Xiao Cheng, Guangtai Liang, Qianxiang Wang, and Zhiliang Zhu. 2024. Efficiently Trimming the Fat: Streamlining Software Dependencies with Java Reflection and Dependency Analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–12.

[38] The Apache Software Foundation. [n. d.]. 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.

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

[40] Ying Wang, Peng Sun, Lin Pei, Yue Yu, Chang Xu, Shing-Chi Cheung, Hai Yu, and Zhiliang Zhu. 2023. PLUMBER: Boosting the Propagation of Vulnerability Fixes in the npm Ecosystem. *IEEE Transactions on Software Engineering* (2023).

[41] Wikipedia. 2024. Dependency hell. https://en.wikipedia.org/wiki/Dependency_hell. accessed 03-Jun-2024.

[42] Hyram Wright. [n. d.]. Hyrum's Law. https://www.hyrumslaw.com/. accessed 21-Nov-2024.

[43] Boming Xia, Tingting Bi, Zhenchang Xing, Qinghua Lu, and Liming Zhu. 2023. 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)*. 2630–2642.

[44] Lyuye Zhang, Chengwei Liu, Sen Chen, Zhengzi Xu, Lingling Fan, Lida Zhao, Yiran Zhang, and Yang Liu. 2023. Mitigating persistence of open-source vulnerabilities in maven ecosystem. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering*. 191–203.

[45] Lyuye Zhang, Chengwei Liu, Zhengzi Xu, Sen Chen, Lingling Fan, Bihuan Chen, and Yang Liu. 2022. 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*. 1–12.

[46] Chenguang Zhu, Mengshi Zhang, Xiuheng Wu, Xiufeng Xu, and Yi Li. 2023. Client-Specific Upgrade Compatibility Checking via Knowledge-Guided Discovery. *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023), 1–31.