Towards increasing the reliability of Maven's dependency resolution

Master's Thesis

Cathrine Paulsen

Towards increasing the reliability of Maven's dependency resolution

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Cathrine Paulsen
born in Bodø, Norway

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Towards increasing the reliability of Maven's dependency resolution

Author: Cathrine Paulsen
Student id: 4659732

Abstract

A reliable dependency resolution process should minimize dependency-related issues. We identify transparency, stability, and flexibility as the three core properties that define a reliable resolution process and discuss how different dependency declaration strategies affect them. To increase the reliability of Maven's dependency resolution we identify two patterns of misuse, or smells, that commonly occur in Maven projects: the presence of used undeclared dependencies and conflicting soft version constraints. We introduce and evaluate a proof-of-concept method, MARCO, designed to address these smells. MARCO increases transparency by injecting used undeclared dependencies and balances stability and flexibility by replacing soft version constraints with compatible version ranges. The version ranges are generated through a dependency-specific approach to compatibility using bytecode differencing and cross-version testing. The empirical evaluation of MARCO shows that while the ranges generated by the dependency-specific approach may be stricter than necessary, they are unlikely to contain breaking changes. Overall, we see that MARCO is able to make the resolution process slightly more reliable, affecting 13% of dependencies in 71% of projects, in a way that is more stable than a soft constraint-only approach, and more flexible than a hard constraint-only approach.

Preface

This thesis marks the end of my Master's journey, a period that has been both intellectually challenging and personally rewarding. I am deeply grateful to my supervisor, Dr. Sebastian Proksch, whose guidance, support, and encouragement to follow my curiosity has been invaluable in shaping this thesis. I also extend my heartfelt thanks to my family and friends for their continuous support throughout my degree and for believing in me when I did not. Writing the last sentence of this thesis, I feel a great sense of pride over all that I have achieved, learned, and grown during this period. Moving forward, I am excited to see what opportunities the future may hold and plan to stay forever curious.

Cathrine Paulsen
Delft, the Netherlands
July 2, 2024

Contents

List of Figures

Chapter 1

Introduction

There are several reasons a developer may want to update their dependencies: to receive updates, new features, bug fixes, or security updates. However, doing so may inadvertently introduce breaking changes or dependency conflicts, which can be time-consuming for developers to fix. As dependency trees grow, resolving dependency conflicts becomes increasingly more complex; solving a conflict in one part of the tree may make another conflict appear elsewhere. Due to the often time-consuming and frustrating nature of dependency-related problems, they are sometimes collectively referred to as *dependency hell* [44, 32]. To prevent breaking their projects, many developers simply avoid updating their dependencies altogether [30, 21]. This update aversion becomes particularly problematic in the case of security updates due to the phenomenon of vulnerability propagation in software ecosystems [43]. A project that relies on a vulnerable dependency does not only expose itself to security risks but all of its dependents as well. Proper use of dependency management systems can help reduce dependency-related problems and even improve the security of the larger ecosystem by preventing vulnerability propagation; however, improper use may also cause dependency-related problems [46]. One such dependency management system is Maven, for which we look into how we can enforce proper use to avoid dependency-related problems and define two common patterns of misuse as dependency smells.

Declaring dependencies with SemVer-compatible open version ranges is an important prerequisite for the automatic propagation of security updates [43]. Open version ranges ensure that the resolved dependency versions are as recent as possible, and the SemVer standard ensures that no breaking changes are introduced by pinning the major version. Although most Maven repositories follow SemVer conventions, breaking changes in non-major versions is still common [33, 28]. This observation may explain why 99% of Maven dependency declarations do not use open version ranges but pin a specific version as a soft version constraint (SoftVer) instead [46]; developers simply cannot trust that non-major releases are non-breaking.

The widespread use of SoftVer introduces two issues into the dependency resolution process, namely *instability* and *inflexibility*. Although SoftVer pins a specific version, there is no guarantee that the pinned version will be resolved. In the case where there are multiple SoftVer pins of the same dependency, Maven simply picks the first version it encounters while traversing the dependency tree in a breadth-first-search manner [1]. If two such conflicting SoftVer versions are incompatible, it may result in build errors, or unexpected behavior and errors at runtime due to breaking changes. This causes instability since it is unknown until runtime whether a completed dependency resolution succeeded without issues. Hard constraints require the pinned version to be resolved, causing conflicting dependency declarations to be detected during resolution [10]. While removing the instability from the process, hard constraints are even more inflexible than soft constraints. Pinning singular versions, whether they are soft or hard constraints, may unnecessarily tighten the dependency resolution constraints and is more likely to result in resolution failure when old dependencies are updated or new ones are added. Inflexible version constraints also prevent automatic dependency updates [21] and their propagation downstream [46], which can leave the library and its dependents vulnerable and buggy. Using open version ranges would increase flexibility at the cost of introducing instability into the resolution process since the sub-optimal SemVer compliance in Maven is likely to introduce breaking changes.

Besides SemVer, another aspect of Maven that becomes problematic when updating dependencies is the possibility to directly use transitive dependencies [1]. This introduces *non-transparency* into the dependency resolution process. All directly used dependencies should be declared as direct dependencies; otherwise, if a transitive dependency that is used directly is updated to an incompatible version or removed entirely, it may result in resolution, compilation, or runtime failure without a transparent cause. Increased transparency may also indirectly increase a project's security, as developers tend to upgrade vulnerable direct dependencies more often than transitive ones [46].

To mitigate these issues, we define the concept of a *reliable* resolution process as one that is *stable*, *flexible*, and *transparent*. Furthermore, we define the direct use of transitive dependencies and conflicting SoftVer constraints as dependency smells that negatively affect these properties. Different dependency declaration strategies also affect these properties differently. Hard constraints provide the strongest stability but can make upgrading or adding dependencies difficult, making them inflexible. Open version ranges provide a high degree of flexibility but may introduce breaking changes, making them unstable. Balancing these two properties is therefore important and is something developers indicate that they struggle with [16]. One method to achieve this balance is through the use of *compatible version ranges*: version ranges that are as broad as possible without introducing breaking changes. This thesis proposes an automated solution, MARCO, that injects declarations of missing direct dependencies to increase transparency and converts SoftVer declarations to compatible version ranges to balance stability and flexibility, providing developers with a more reliable dependency resolution process to mitigate dependency-related issues without having to modify the resolution process itself.

There are currently no solutions that address all three properties of reliable resolution in this way. The closest existing tool is RANGER, which replaces SoftVer constraints with compatible version ranges [46], but it does not address transparency and is not fully open-source. Generating compatible version ranges boils down to compatibility checking, or breaking change detection between two dependency version pairs, which is a well-studied problem. COMPCHECK, DEBBI [13] and Mujahid et al. [27] all use a form of cross-client regression testing to detect whether two dependency versions are behaviorally compatible.

The main problem with approaches relying on client tests is that it is not common practice for clients to test their dependencies. Clients instead trust that the dependencies themselves are well-tested [21], and client tests often fail to capture issues caused by dependency updates [47]. If the test coverage of the dependency is low, the chance of breaking changes not being caught by the tests increases. Motivated by the low coverage of client tests, UPPDATERA and SEMBID use static analysis techniques on ASTs and callgraphs to detect patterns that match behavioral breaking changes.

Common for all existing solutions is that they take a client-specific approach to compatibility, meaning that the compatibility decision is based on analyzing the client using the dependency rather than the dependency itself. A more direct but so far unexplored approach to check behavioral compatibility between dependency versions is to use cross-version regression testing using the tests of the dependency instead of the client, which is a dependency-specific approach. Both client- and dependency-specific approaches have advantages and disadvantages. Client-specific approaches that rely on client tests may have a low false positive rate but a high false negative rate when detecting breaking changes. Dependency-specific approaches that rely on dependency tests will comparatively have a higher false positive rate since clients may only use parts of the dependency but a lower false negative rate assuming that dependencies test their own behavior better than clients. A client-specific approach used to generate compatible version ranges may therefore result in a resolution that is more flexible but less stable, whereas a dependency-specific approach may be more stable but less flexible. Surveys by Mirhosseini and Parnin [24] and Pashchenko et al. [30] suggest that developers themselves also favor stability over flexibility, which encourages looking into the dependency-specific approach.

To assess the need for an automated solution like MARCO and its efficacy in improving the reliability of Maven's resolution process, the thesis investigates the following research questions (RQs):

- *RQ1: How prevalent are the dependency smells?* To increase the reliability of Maven's resolution process, MARCO removes the dependency smells associated with used undeclared dependencies and conflicting soft constraints. To motivate the need for such a solution, we investigated the actual prevalence of these issues in real-life Maven projects and found that both smells are common.

- *RQ2: How often do developers manually mediate SoftVer conflicts?* To assess whether developers can benefit from an automated solution, we investigated how often developers use version-overriding techniques to manually override Maven's conflict resolution of conflicting soft constraints and found that manual conflict mediation is relatively common and that mediated conflicts seem to involve more conflicting declarations than unmediated conflicts.

- *RQ3: How successfully can we find test suites for dependencies using GitHub linking?* MARCO generates dependency-specific compatible ranges using bytecode differencing to determine static compatibility and cross-version regression testing to determine behavioral compatibility. To perform cross-version testing, we need access to the dependency's tests. We therefore investigated whether we can accurately link a Maven dependency to its GitHub repository to increase the likelihood of locating tests, and found that GitHub linking can significantly increase the amount of tests we can find compared to test jars published on Maven Central.

- *RQ4: How effective is the dependency-specific approach in detecting breaking and non-breaking changes?* The purpose of this RQ is to measure whether the compatible ranges generated by the dependency-specific approach combining bytecode differencing and cross-version testing are likely to contain breaking changes. We found that the generated ranges are unlikely to contain breaking changes, but may be stricter than necessary compared to other client-specific approaches.
- *RQ5: How successful is the proposed solution in improving Maven's dependency resolution?* The end goal of the evaluation is to investigate whether and to what extent MARCO is able to influence Maven's resolution process to be more reliable. We applied MARCO to real-world Maven projects and compared the outcome of the resolution process before and after to see how the resolution process changed. We found that MARCO is likely to influence the resolution process towards being more reliable in a way that is more stable than a soft constraint-only approach and more flexible than a hard constraint-only approach.

The main contributions of this thesis are:

- Identified two commonly occurring dependency smells that negatively affect the reliability of a project's dependency resolution process, whose automatic removal could improve reliability in 45% of projects and reduce developer effort spent manually resolving conflicting soft constraints.
- A Maven-to-Github linking strategy which significantly increases the likelihood of locating dependency tests compared to relying on test jars published along with the dependency artifact.
- A proof-of-concept method that generates compatible version ranges using a dependency-specific approach to compatibility based on bytecode differencing and cross-version testing capable of influencing Maven's resolution process to be more reliable without modifying the resolution process itself.
- Identified and discussed several opportunities for future work on reliable dependency resolution and compatible version range generation.

The data and code used to generate the results for this thesis are available on Zenodo [31].

The remainder of this thesis is structured as follows. Chapter 2 covers related works. Chapter 3 describes MARCO, the proposed solution to remove the defined dependency smells from any Maven project. Chapter 4 presents the empirical prevalence study, covering RQ1 and RQ2. Chapter 5 provides an empirical evaluation of MARCO, covering RQ3, RQ4, and RQ5. Chapter 6 discusses key findings and limitations and provides directions for future work. Finally, Chapter 7 summarizes the thesis.

Chapter 2

Related Work

The idea of aiding developers in the dependency management process by automatically detecting compatible version updates or compatible version ranges to avoid breaking changes is not new. There are three types of compatibility in Java: binary, source, and behavioral [15]. Binary and source compatibility are also referred to as static, syntactic, or API compatibility, while behavioral compatibility is also called dynamic or semantic compatibility [21, 46]. Static incompatibilities can (in most cases) be detected at compile time by static analysis tools such as JAPICMP [25] and REVAPI [35] which use bytecode differencing. Behavioral incompatibilities relate to incompatible runtime behavior which is more difficult to detect accurately using static analysis methods [45]. Formal behavioral specifications are also rarely available and difficult to verify [23]. Tests are therefore often used to approximate behavioral compatibility.

This chapter discusses literature relevant to compatibility checking, or breaking change detection. The related works are split into two broad categories based on how they determine compatibility: using static or dynamic analysis. The chapter concludes with a discussion on client- versus dependency-specific compatibility approaches.

2.1 Dynamic Analysis

Zhu et al. [47] introduced the COMPCHECK tool for detecting breaking changes, and introduced the concept of *client-specific* compatibility. If a breaking change is found using a client-specific technique, it means that the client's use of the dependency has been analyzed and found that the breaking change is used by the client. This term is also used in this thesis, and complimented by the term *dependency-specific*, meaning that the compatibility decision is only based on analyzing the dependency and is therefore client-agnostic. To detect breaking changes, COMPCHECK maintains a knowledge base of known client-dependency incompatibilities, which is obtained via client tests. The technique of detecting breaking changes using multiple clients' tests is called *cross-client testing*, and is commonly used in related works. Control flow graphs of clients are used to determine dependency usage and a lookup is performed in the knowledge base whether the usage is related to a known incompatibility. The technique of using control flow graphs or call graphs to determine dependency usage is useful to ensure that a breaking change is actually reached by a specific client and is sometimes referred to as *reachability analysis* in related works.

Mujahid et al. [27] used cross-client testing to identify behavioral breaking changes in dependency upgrades in the NPM ecosystem. Tests are crowd-sourced from the dependents of the dependency. Because cross-client testing comes with a high computational cost, the dependents with high-coverage test suites are prioritized to reduce it. The update is emulated on the dependents, and if a previously passing test suite fails after the update then the update is deemed incompatible. This is a form of cross-client testing and is motivated by the fact that only using a single client's tests is often unreliable due to lacking test coverage. They found that crowd-sourcing tests like this could increase test coverage of the dependency from 47% to 55%, but that finding enough dependents to increase test coverage proved a limitation.

Chen et al. [13] introduced DEBBI, which uses a cross-client testing technique similar to Mujahid et al. [27] to detect breaking changes. Instead of prioritizing client test suites by coverage like Mujahid et al. [27] did, DEBBI prioritizes the test suites of clients that have a high API usage of the dependency under test.

He et al. [20] conducted a qualitative survey of DEPENDABOT usage among developers. DEPENDABOT provides developers with a compatibility score when opening pull requests for dependency upgrades. The compatibility score indicates how likely the update is to introduce breaking changes, and is calculated based on the fraction of other projects performing the same dependency update that have passing CI pipelines. This is a form of cross-client testing for compatibility checking. The authors found that developers express concern over the effectiveness of the compatibility scores, and that they found high-quality test suites more useful for assessing compatibility. Their conclusions suggest that if cross-client testing is used, then the tests of the client under consideration should be included and prioritized when determining compatibility. Basing compatibility for a single client based on whether other clients have breaking CI pipelines may be misleading, since other clients may not use the same parts of the library.

2.2 Static Analysis

Zhang et al. [46] presents the tool RANGER which is the closest existing solution to MARCO. The goal of RANGER is to convert SoftVer constraints into safe, compatible ranges to tackle the problem of vulnerability propagation in the Maven ecosystem. Static compatibility is checked using REVAPI and JAPICMP on the dependency JARs. If the static compatibility tools return API incompatibilities, RANGER uses reachability analysis using call graphs to check whether the client uses the incompatible sections of the API. If no static incompatibilities are reachable by the client, dynamic compatibility is checked using the static analysis tool SEMBID. If no reachable dynamic incompatibilities are found, the tests of the client are used as a final step to ensure that the client does not introduce behavioral breaking changes.

Zhang et al. [45] presents the SEMBID (Semantic Breaking Issue Detector) tool which aims to detect semantic breaking changes statically, motivated by the lacking coverage of client tests. Breaking changes are detected by semantic differencing of call graphs between two dependency versions. For any semantic change detected between the call graphs, SEMBID checks whether it matches any of the non-breaking heuristic patterns to reduce false positives. Despite this, the false positive rate of SEMBID is still relatively high at 22%. While SEMBID was able to detect breaking changes that were not detected by single client test suites, it also did not detect all breaking changes that were detected by the test suites, since the static analysis is not able to cover dynamic language features such as reflection.

Hejderup and Gousios [21] introduces the static analysis tool UPPDATERA, which aims to detect static and semantic breaking changes between Java library versions. Client tests are not used on the reasoning that they are effective only if they have adequate test coverage, which in practice is not common since clients do not tend to test their dependencies' behavior. ASTs are analyzed using AST differencing for possible semantic breaking changes using SpoonLabs/GumTree [17], and call graphs are used for reachability analysis to check whether the incompatibilities are reachable by the client to reduce false positives. They found that UPPDATERA's static approach could detect twice as many breaking changes as client tests, but struggles similarly to SEMBID with false positives caused by refactorings and over-approximated function calls.

Ochoa et al. [28] presents the MARACAS static analysis tool for detecting static breaking changes between Java library versions. MARACAS extends the existing tool JAPICMP which detects static breaking changes given two versions of a dependency. However, the static breaking changes may or may not be exposed via the API used by the client. JAPICMP is therefore extended to also factor in common annotations used in libraries to denote parts of the library that are not intended for client use. MARACAS uses this information to detect whether breaking API changes actually affect the client. If not, then they can be ignored when determining whether a specific dependency upgrade introduces breaking changes to the client to reduce false positives.

2.3 Client- versus Dependency-specific Compatibility

In addition to compatibility types, there are also two different perspectives to compatibility: the client-specific approach and the dependency-specific approach. The client-specific approach looks at the question of whether two dependency versions are compatible from the perspective of the client that uses the dependency by analyzing the client [47]. There may be incompatibilities between two library versions that the client code never calls, in which case the incompatibility is irrelevant to the client, and the versions are labeled as compatible. Common for all existing solutions is that they use a client-specific approach to compatibility. The only exception is MARACAS; however, it only consider static and not dynamic compatibility. To fully determine whether two dependency versions are compatible, we need to verify both.

A more direct but so far unexplored approach to check dynamic compatibility between dependency versions is to use cross-version regression testing using the tests of the dependency instead of the client, which is a dependency-specific approach. Mostafa et al. [26] used cross-version regression testing to collect a large number of backward behavioral incompatibilities, suggesting that this method may be effective at breaking change detection.

Venegas [42] also suggested that future work using cross-version regression testing could address scalability issues encountered in client-specific approaches.

Whether static or dynamic analysis techniques are used to determine compatibility, dependency-specific approaches are underexplored. This thesis therefore proposes an alternative solution to related works that uses both static analysis (JAPICMP) for static compatibility and dynamic analysis (cross-version testing) for dynamic compatibility from a dependency-specific compatibility perspective so that compatibility decisions can be precomputed and reused.

Chapter 3

Achieving a more reliable dependency resolution

This chapter presents the proposed solution, called the Maven Compatible Range (MARCO) toolkit, to achieve a more reliable dependency resolution. The toolkit consists of two main components: the REPLACER and the GENERATOR. The ultimate goal behind the proposed solution is to guide Maven's resolution process to be more reliable without modifying the process itself but instead modifying how dependencies are declared. Section 3.1 explains at a high level how Maven's resolution process works and how MARCO is used to guide it towards being more reliable. The remaining sections explain in more detail how the REPLACER and the GENERATOR work to achieve this goal. Section 3.2 describes how the REPLACER modifies the dependency declarations to guide the resolution process, and Section 3.3 describes how the GENERATOR computes compatible versions that the REPLACER uses to replace soft version constraints with compatible ranges.

3.1 Guiding the Resolution Process

The standard Maven resolution process can be simplified as follows and is partially illustrated in Figure 3.1. A project declares its direct dependencies in its POM file. During the project's dependency resolution process, Maven downloads the artifacts of the dependencies declared in the POM from the Maven Central Repository (MCR)[1]. The artifacts are placed in a local folder on the project's client machine called the `m2` folder. The downloaded artifacts include the packaged compiled source code (JARs) of the dependencies as well as the dependencies' POM files. The JARs are necessary so that the project can make calls to the dependency APIs, and the POM files are necessary to retrieve the project's transitive dependency declarations. Because the project needs to download all dependencies, both direct and transitive, Maven repeats the download process for each dependency's POM file.

MARCO does not modify the resolution process itself, but only the POM file that contains the dependency declarations for a given Maven project so that Maven performs its resolution algorithm more reliably. This allows the proposed solution to remain simple

[1] unless an alternative repository is configured

since we do not have to consider all the intricacies of POMs and how different configurations, such as profiles, affect the resolution process. Figure 3.1 shows an overview of how MARCO could be applied to modify the dependency declarations used by Maven's resolution process. Specifically, MARCO is used in two parts of the resolution process. First, MARCO is applied to the project's POM to create the modified POM file. This step will modify the direct dependency declarations to use compatible version ranges. However, to fully guide the resolution process we also need to modify the transitive dependency declarations, which are stored in the project's dependencies' POMs which Maven fetches from the local `m2` folder after downloading them from MCR. To modify the transitive POMs we therefore have two options: we can either apply MARCO to the `m2` folder as shown in Figure 3.2, or we can apply MARCO to (parts of) MCR to create a modified MCR that contains MARCO-replaced POMs as shown in Figure 3.1.

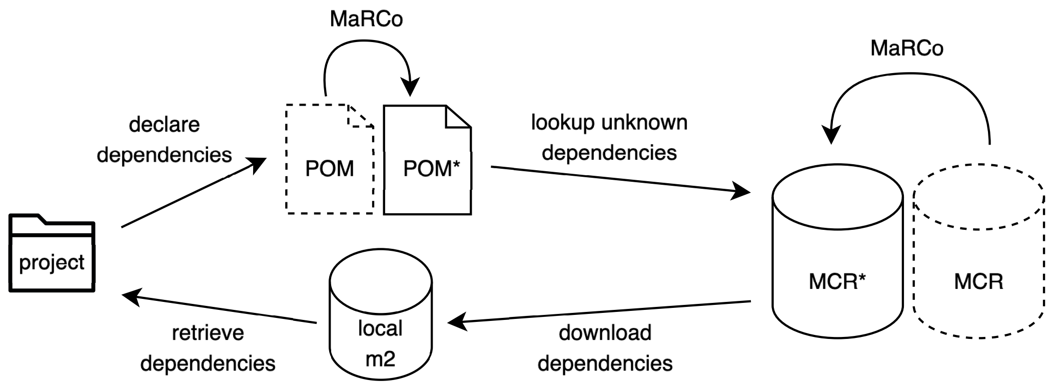

Figure 3.1: How MARCO is intended to be applied to achieve full replacement.

If MARCO is to be applied in a real-world setting then the most convenient solution for developers is to use MARCO as described in Figure 3.1. This would require setting up and maintaining an alternative mirror repository to MCR containing pre-replaced POMs of as many commonly used dependencies as possible. Developers wanting to use MARCO in their workflow could then simply add the mirror repository to their POM. The Maven resolution process will then download the pre-replaced POMs hosted by the mirror repository, 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 thesis, but could be an interesting direction for future work. To emulate the mirror repository for experiments requiring POM replacement (RQ5), full dependency replacement is instead done by applying MARCO to all the dependencies contained in the local `m2` folder, as is shown in Figure 3.2. From the perspective of the Maven resolver, there is no difference between the two solutions. Choosing one over the other will therefore have no impact on evaluation.

The MARCO toolkit helps developers achieve a more reliable dependency resolution by injecting declarations of missing direct dependencies to increase transparency, and by converting SoftVer constraints to compatible version ranges to balance stability and flexibility. Furthermore, incorporating MARCO into a developer's workflow should provide minimal

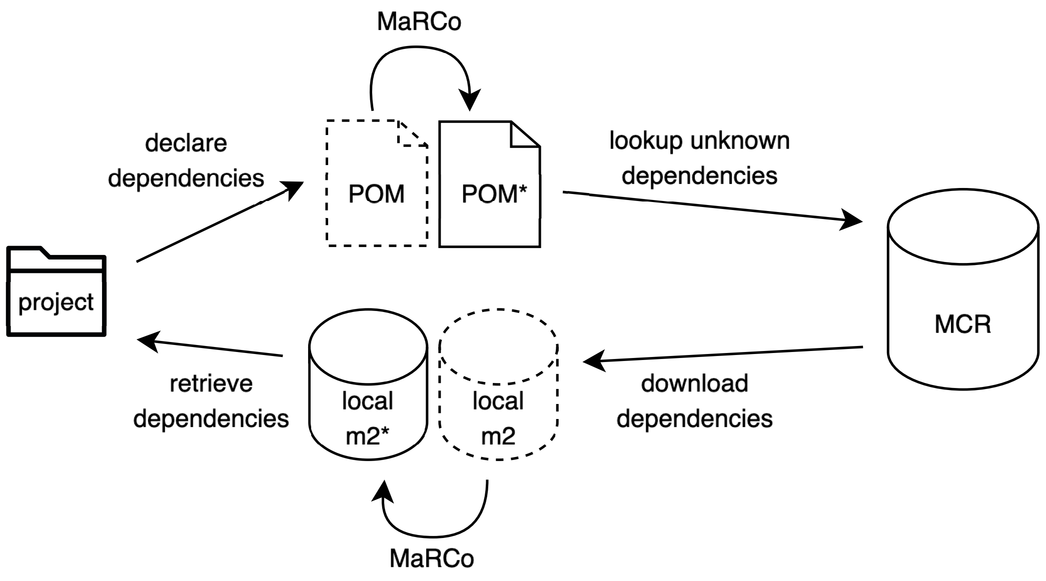

Figure 3.2: How MARCO is actually applied in the experiment requiring full replacement.

overhead and require minimal effort. The toolkit therefore consists of two components: a lightweight REPLACER which is used by the developer client-side, and a server-side GENERATOR, see Figure 3.3. The REPLACER is responsible for replacing a project's POM with a modified POM containing the injected and replaced dependency declarations. It is designed with the idea in mind that developers using MARCO would not need to change how they declare dependencies. They could pin a version they know works as a SoftVer, then apply the REPLACER to their POM and be confident that the resolved versions will be compatible with the one they pinned. To replace SoftVer constraints with compatible version ranges, the REPLACER needs access to a mapping from dependency versions to their compatible version ranges. The GENERATOR takes care of computing, storing, and serving these mappings. The compatible version ranges are generated using a dependency-specific approach to compatibility checking, meaning that the compatible versions for a specific dependency version are client-agnostic and can be pre-computed to reduce overhead.

3.2 Modifying the Dependency Declarations

The goal of the REPLACER is to modify the dependency declarations in the POM in such a way that Maven's dependency resolution process is more transparent, flexible, and stable. As seen in Fig. 3.3, the REPLACER is the lightweight client-side tool used by developers. It takes a Maven project as input and outputs a new, modified POM without the previously introduced dependency smells of missing direct dependencies and SoftVer version constraints. The smell of missing direct dependencies is removed by declaring all dependencies directly used as direct dependencies with the help of the Maven Dependency Plugin

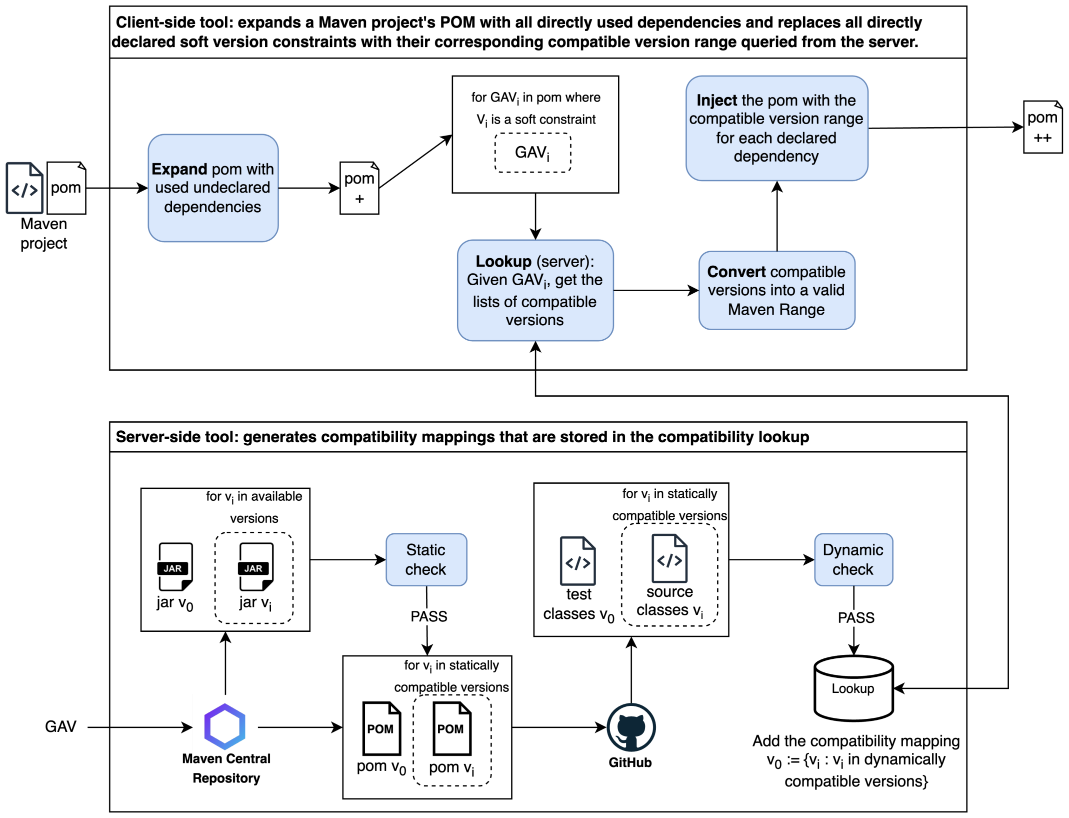
Client-side tool: expands a Maven project's POM with all directly used dependencies and replaces all directly declared soft version constraints with their corresponding compatible version range queried from the server.
pom
Maven project
Expand pom with used undeclared dependencies
pom +
for GAV$_i$ in pom where V$_i$ is a soft constraint
GAV$_i$
Lookup (server): Given GAV$_i$, get the lists of compatible versions
Convert compatible versions into a valid Maven Range
Inject the pom with the compatible version range for each declared dependency
pom ++
Server-side tool: generates compatibility mappings that are stored in the compatibility lookup
for v$_i$ in available versions
jar v$_0$
jar v$_i$
Static check
PASS
for v$_i$ in statically compatible versions
test classes v$_0$
source classes v$_i$
Dynamic check
PASS
Lookup
GAV
Maven Central Repository
for v$_i$ in statically compatible versions
pom v$_0$
pom v$_i$
GitHub
Add the compatibility mapping v$_0$:= {v$_i$: v$_i$ in dynamically compatible versions}

Figure 3.3: Overview of the MARCO REPLACER (top) and GENERATOR (bottom)

[4]. Removing the smell of SoftVer version constraints involves replacing the SoftVer version constraints with compatible version ranges which are fetched from the compatibility mapping pre-computed by the GENERATOR.

The compatibility mapping maps a dependency version to a list of compatible versions. This list of compatible versions must be converted into a format that Maven recognizes as a range, which is defined by the Maven Version Range Specification [8]. Simplified, a range is defined by the string `[lowerBound, upperBound]`, where the lower bound is the lowest version and the upper bound is the highest version according to Maven's version sorting algorithm [40]. The range includes all available versions between the lower bound and upper bound, and multiple ranges can be concatenated by commas to allow for gaps. The compatible version list is therefore converted to a compatible version range by first sorting the list using Maven's sorting algorithm, then identifying the upper- and lower bounds. To identify whether there are gaps in the compatible version range, we must also fetch the available versions of the dependency from MCR and compare whether all available versions between the identified lower and upper bounds are in our compatible version list. If there are gaps, we identify the continuous compatible ranges and concatenate them as described previously.

Compatibility checking using cross-version regression testing is computationally heavy.

Because the compatibility checking does not need to be re-computed for each client, we keep all the heavy computation steps associated with compatibility checking in the server-side GENERATOR so that the client-side REPLACER remains lightweight and provides no significant overhead to the developer's workflow.

The algorithm behind the REPLACER is shown in Alg. 1. The Maven Dependency Plugin's *analyze* goal [2] is first run on the Maven project and returns the used undeclared dependencies, i.e., the missing direct dependency declarations. These declarations are then injected into the `dependency` section of the POM. The missing dependency declarations returned by the *analyze* goal contain SoftVer version constraints. The injection step is therefore also necessary before replacing the SoftVer version constraints to ensure that all of them are replaced. Important to note is that the Maven Dependency Plugin is only able to detect *static* instances of used undeclared dependencies. If the project uses dynamic language features, there may be instances of used undeclared dependencies that are not detected and therefore not injected. This is discussed further in Chapter 6.

Algorithm 1 POM range-enhancement by replacing SoftVer versions via lookup of pre-computed mappings of SoftVer version to compatible version ranges

```
Input: Maven project ($p$)
Output: Range-enhanced POM

                                                        ▷ Inject missing dependencies
1: $missing\_deps$ ← CALL mvn dependency:analyze
2: $POM$ ← inject $POM$ with $missing\_deps$

                                                        ▷ Replace SoftVer
3: for each $(dep, softver\_version) \in POM$ do
4:     $range$ ← LOOKUPRANGE($dep$, $softver\_version$)
5:     $POM$ ← replace $softver\_version$ with $range$ in $POM$
6: end for

7: return $POM$
```

3.3 Generating Compatible Versions

To increase the reliability of the dependency resolution process, the REPLACER replaces soft version constraints with compatible version ranges. To enable to REPLACER to remain lightweight, the GENERATOR pre-computes a dependency's compatible versions using a dependency-specific compatibility approach. The GENERATOR has two responsibilities: perform compatibility checking and store the results in the compatibility mapping storage; and serve any lookup requests of specific compatibility mappings by the REPLACER. As seen in Fig. 3.3, the MARCO GENERATOR runs server-side and is not intended to be directly used by developers unless they want to compute and host their own compatibility mappings. The remainder of this section will describe how the GENERATOR computes compatible versions.

Given a specific dependency version called the *base* version, the GENERATOR fetches all *candidate* versions, which are the dependency's currently available versions from Maven Central, and computes whether they are compatible with the base. If the candidate is compatible, it is added to the base version's compatibility mapping. The compatibility mapping of a specific version v of a dependency with available versions av, can be expressed as follows:

$$v \mapsto \{v_i \,|\, v_i \in av \text{ and is compatible with } v\} \tag{3.1}$$

Checking whether a candidate version v_i is compatible with the base version v involves three main steps that are performed in order: the static compatibility check, the Maven-to-GitHub linking, and the dynamic compatibility check. The compatibility check can fail at any of the three steps, and a version is only added to the compatibility mapping if it passes all three steps. The algorithm behind the GENERATOR is shown in Alg. 2.

Computing static and dynamic compatibility The compatibility checking component implements a dependency-specific approach to compatibility. The compatibility check for a version pair consists of a static and a dynamic compatibility check. To determine whether two versions are compatible, there are three types of compatibility to check: source, binary, and behavioral compatibility [15]. The static check checks for source and binary compatibility using JAPICMP [25], a commonly used static compatibility checker for Java, used by for example RANGER [46] and MARACAS [28].

The dynamic check checks for behavioral compatibility, which is approximated using regression testing of the candidate version's source code on the base version's test code. This is a dependency-specific approach to behavioral compatibility using cross-version regression testing. To run the candidate version's code on the base version's tests, we create a new Maven project which contains the following: the compiled source code of the candidate, the compiled test code of the base, and a combined POM. The combined POM uses the candidate's POM as a base since we are preparing to run the candidate's source code. Because the base version's tests may involve test dependencies and test suites can change between versions, we need to make sure that we only include the test dependencies of the base version in the combined POM. To determine the behavioral compatibility between a base version and a candidate version, the base version's tests are run in two stages, first with the base version's code, then with the candidate version's code. The passing tests on the base code establish the baseline that the test results of the candidate are compared to. If running the tests on the candidate code results in more failures than the base code, we consider the candidate incompatible with the base and otherwise compatible.

The static check is run before the dynamic check because bytecode differencing is computationally cheap compared to cross-version regression testing and only requires access to the base and candidate JARs which are provided by MCR. A candidate version must be both statically and dynamically compatible with its base to be compatible. Therefore, if the candidate is statically incompatible we can skip the expensive dynamic check since the candidate will be incompatible regardless of the dynamic outcome.

Locating tests using Maven-to-GitHub linking The dynamic check requires the compiled source code of the candidate, the compiled test code of the base, and a combined POM to run. While the compiled source code can be extracted from the JARs used for the static check, and the POMs are available on MCR, finding the test code poses a challenge since test jars are not commonly available on MCR [19]. If we cannot find the dependency's test code, we cannot run the dynamic check, and as a result we cannot determine the candidate's compatibility. Limiting the dependencies we can evaluate compatibility to only those that publish test jars on MCR may significantly reduce the applicability of MARCO. The GitHub linking component between the static and dynamic checks aims to address this problem, based on the assumption that we are likely to find the test code of a dependency in the same repository that contains its source code. Given a specific version of a dependency (GAV), the GitHub linking algorithm finds the GitHub repository and tag via information stored in the GAV's POM. The algorithm is split into the following steps:

1. *Finding the GitHub repository*. Maven provides a Source Code Management (`scm`) tag which can be included in POMs to link to the dependency's source code repository [6]. If a GitHub link is found in the `scm` section, the GitHub repository is extracted from the link. If no GitHub repository or `scm` section is found, but the POM links a parent POM, the process is repeated for the parent POM.

2. *Finding the version string*. If we managed to extract a GitHub repository, we need to find the GitHub tag that most likely corresponds to the dependency version. The `scm` section may contain a `tag` sub-tag, which indicates which tag in the source code repository the GAV corresponds to [5]. If the `tag` is present, this is used as the version string. If `tag` is not present or does not lead to finding a matching tag in the next step, the process is repeated using the version contained in the GAV as the version string.

3. *Matching the version string to a GitHub tag*. To match the version string (`<version>`) obtained from the POM with a GitHub tag, the linking algorithm constructs possible candidate tag names combining the version string with the following commonly used GitHub tagging patterns: `<artifactId>-<version>`, `v<version>`, `r<version>`, and `<version>`. To find the GitHub tag, the algorithm first tries to look up the candidate tag names via the GitHub API. If no tag is found via the lookup (exact match), the algorithm then performs an inexact match over all tags in the repository. The inexact match checks whether any of the candidate tag names occur as a substring in any of the GitHub tags. If there are multiple matches, the inexact match selects the shortest overall string to avoid picking extensions not present in the original version string, such as `-beta`.

Algorithm 2 Given a SoftVer version v_0 and dependency dep, generate a compatible version range.

```
 1: function GenerateRange($dep$, $v_0$)
 2:     Find $(dep, v_0)$ on the Maven Central Repository (MCR)
 3:     Fetch $available\_versions$ by looking up $dep$ on MCR
 4:     $range \leftarrow \emptyset$
 5:     for each $v \in available\_versions$ do
 6:         $incompatibilities \leftarrow$ StaticCheck($jar_{v_0}$, $jar_v$)
 7:         if $incompatibilities \neq \emptyset$ then
 8:             continue (reject $v$)
 9:         end if
10:         $incompatibilities \leftarrow$ DependencyDynamicCheck($v_0$, $v$)
11:         if $incompatibilities \neq \emptyset$ then
12:             continue (reject $v$)
13:         end if
14:         $range \leftarrow range \cup v$
15:     end for
16:     return $range$
17: end function

18: function DependencyDynamicCheck($v_0$, $v$)
19:     Get $v_0$ test suite and $v$ source code via GitHub.
20:     Run the tests of $v_0$ with the source code of $v$
21:     $incompatibilities \leftarrow$ list of test failures
22:     return $incompatibilities$
23: end function
```

Chapter 4

Empirical Prevalence Study

In Chapter 1, we introduced two dependency smells that negatively affect the reliability of a project's dependency resolution process: the direct use of transitive dependencies and conflicting soft version declarations. This chapter investigates how widespread these two dependency smells are, and how often developers use version-overriding techniques to manually override Maven's conflict resolution process by analyzing the POM files of Maven projects collected from GitHub. To this end, we define the following research questions:

- *RQ1: How prevalent are the dependency smells?*
  - *RQ1.1: How prevalent is the direct usage of transitive dependencies?*
  - *RQ1.2: How prevalent are SoftVer conflicts?*
- *RQ2: How often do developers manually mediate SoftVer conflicts?*

The answers to these questions motivate whether developers can benefit from an automated solution that removes these dependency smells to increase resolution reliability. If the smells frequently occur and developers are often manually mediating conflicts, an automated solution that removes these smells for them to increase reliability could alleviate dependency effort spent manually resolving dependency-related issues associated with these dependency smells.

4.1 Dataset

To answer RQ1 and RQ2 we need data on the declared, resolved, and mediated dependencies of real-life Maven projects. To this end, we created a dataset using the Maven Dependency Plugin to analyze the dependencies of a collection of Maven projects found on GitHub. This section describes the methodology used for collecting and analyzing the projects to create the dataset used to answer the research questions. An overview of the methodology is visualized in Figure 4.1.

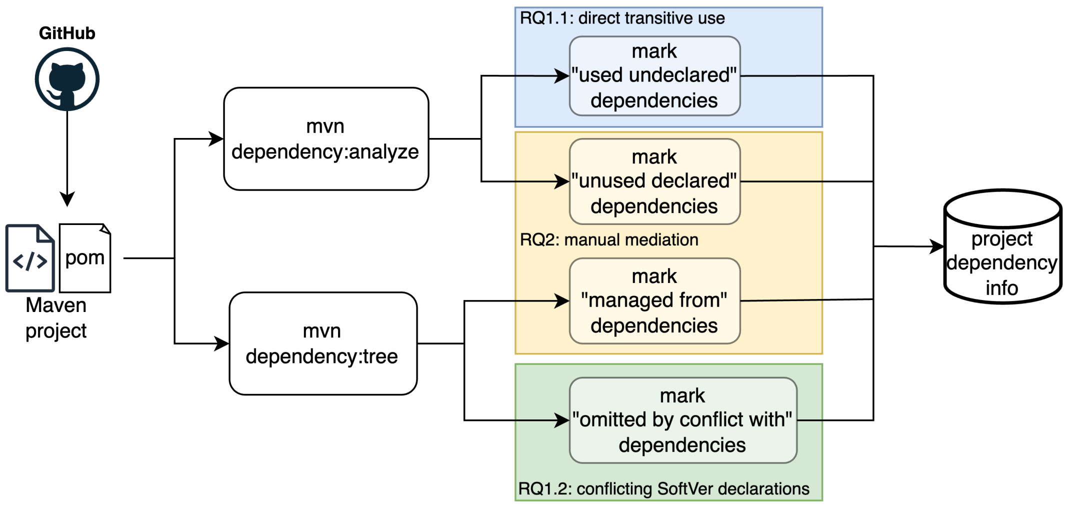
GitHub
pom
Maven project
mvn dependency:analyze
mvn dependency:tree
RQ1.1: direct transitive use
mark "used undeclared" dependencies
mark "unused declared" dependencies
RQ2: manual mediation
mark "managed from" dependencies
mark "omitted by conflict with" dependencies
RQ1.2: conflicting SoftVer declarations
project dependency info

Figure 4.1: Overview of the methodology used for the Empirical Prevalence Study.

Methodology Maven projects were collected on June 2, 2024, using the GitHub API with the following filters: created since January 1, 2023, Java as language, and at least 20 stars. The projects were received from the API in ascending order of their creation date, and the collection was stopped after 800 projects due to time and space constraints. From this initial selection, projects that did not have a POM file were removed to filter out non-Maven projects. 362 projects with POM files remained after filtering, of which we were able to successfully run the Maven Dependency Plugin on 226 projects.

The Maven Dependency Plugin was run on the collected projects using two goals: `dependency:analyze` [2] and `dependency:tree` [3]. The *analyze* goal gives information about whether the project has dependencies that are *used undeclared* or *unused declared*, which is used to answer RQ1.1 and RQ2, respectively. The *tree* goal with the `-Dverbose` flag shows the full dependency tree of the project, including conflicting or overridden dependency declarations. If there are conflicting SoftVer version declarations of the same dependency, these are marked by the Maven Dependency Plugin as *omitted by conflict*. The Maven Dependency Plugin also marks dependencies as *managed from* if a dependency's version has been overridden by the `dependencyManagement` section. The *omitted by conflict* and *managed from* information from the *tree* goal is used to answer RQ1.2 and RQ2, respectively. The dependencies and the information obtained from the Maven Dependency Plugin are then stored in a data store for further analysis to answer each research question.

Results The final dataset consists of 226 projects with an average of 60.8 dependency declarations and 57.3 resolved dependencies per project. The averages of resolved and declared dependencies are not equal because Maven will only resolve one version for each declared dependency, but the dependency tree may contain more than one conflicting SoftVer version declaration for the same dependency.

4.2 RQ1: Prevalence of Dependency Smells

This section answers the question *How prevalent are the dependency smells?* (RQ1), which consists of answering the following two subquestions:

- *RQ1.1: How prevalent is the direct usage of transitive dependencies?*
- *RQ1.2: How prevalent are SoftVer conflicts?*

The presence of the smell in RQ1.1 negatively affects the reliability of the resolution process by decreasing transparency, while the smell in RQ1.2 decreases flexibility compared to open-version ranges and decreases stability compared to hard constraints. The removal of these smells could therefore help increase the reliability of a Maven project's resolution process, and we investigate their prevalence to determine how many Maven projects have potentially compromised reliability due to the presence of these smells. This section will first explain why the methodology shown in Figure 4.1 is sound, followed by the results for RQ1.1 and RQ1.2.

Methodology To answer RQ1.1 we investigate the prevalence of the direct usage of transitive dependencies which are the same dependencies that the plugin marks as *used undeclared*: A *used undeclared* dependency is a dependency that is used directly but not declared as a direct dependency, which is possible if it exists as a transitive dependency in the project's dependency tree. It should be noted that the Maven Dependency Plugin is a static analysis tool and will only detect static instances of *used undeclared* dependencies. There may be other dynamic instances of *used undeclared* dependencies that are not detected by the *analyze* goal due to the use of dynamic language features, such as the Java Reflection API. This is discussed further in Chapter 6.

To answer RQ1.2, we look into how prevalent conflicting SoftVer declarations[1] are, which are the dependency declarations that the plugin marks as *omitted by conflict*. The only other declaration method in Maven besides SoftVer is version ranges, which result in resolution failure when there are conflicting declarations rather than *omitted by conflict*. A hard constraint is a special case of a range containing only one version, so these will also not be marked as *omitted by conflict*. We can therefore be confident that SoftVer declarations are the only dependency declarations that are marked as *omitted by conflict*.

Results Based on our sample of 226 projects, we find that both dependency smells described by RQ1.1 and RQ1.2 are common. 45% of projects had at least one instance of direct usage of a transitive dependency (RQ1.1), with 7.8 instances of this smell on average. Because the Maven Dependency Plugin only detects the static instances of RQ1.1, these numbers should be interpreted as lower bounds. This means that the actual prevalence may be even higher. How to determine a more accurate estimation of the actual prevalence of the direct usage of transitive dependencies is discussed further in Chapter 6 as a possible direction for future work. Regardless, the high lower bound of the prevalence of *used undeclared*

[1]A SoftVer declaration is a dependency declaration that contains a soft version constraint

dependencies means that any method that relies on modifying dependency declarations, like MARCO, should consider injecting the missing direct dependencies first.

45% of projects also had at least one instance of conflicting SoftVer declarations (RQ1.2), with 16.8 instances of conflicting version declarations on average over 5.3 unique dependencies. For projects with conflicts, 28% of the total dependency declarations on average are conflicting SoftVer constraints. From these results, we also see that projects tend to have many instances of conflicting SoftVer constraints (16.8 conflicting declarations per project), but each conflict is rather small in scope (3.2 conflicting declarations per conflict).

4.3 RQ2: Prevalence of Manual Conflict Mediation

Now that we have found that conflicting SoftVer constraints are common, we investigate how often developers manually mediate SoftVer conflicts using version-overriding techniques. Specifically, this section answers the question *How often do developers manually mediate SoftVer version conflicts?* (RQ2). Frequent manual mediation could suggest that Maven's nearest-first conflict resolution strategy is not sufficient and can cause dependency-related issues that developers need to address via manual conflict resolution. As in the previous section, this section will first explain why the methodology shown in Figure 4.1 is sound, followed by the results for RQ2.

Methodology A SoftVer conflict occurs when there are at least two SoftVer constraints for the same dependency that declare different versions in a project's dependency tree. In other words, a SoftVer conflict involves at least one instance of conflicting SoftVer declarations. Maven can only resolve one of the declared versions and will ignore the others. Maven's nearest-first conflict resolution strategy is to pick the version declaration that is the closest to the root. Maven provides two manual mediation techniques in case a developer would want to override the dependency version resolved by Maven's conflict resolution:

1. Use the `dependencyManagement` section to directly override the resolved version.
2. Declare transitive dependencies as direct dependencies to take advantage of Maven's nearest-first resolution strategy to control which version Maven resolves.

The first manual mediation technique using `dependencyManagement` can be detected by inspecting the dependencies that the Maven Dependency Plugin marks as *managed from*. The second manual mediation technique using direct declarations of transitive dependencies can be detected by looking at the dependencies the Maven Dependency Plugin marked as *unused declared* that also have declarations marked as *omitted by conflict* in the verbose dependency tree. If the *unused declared* dependency also occurs in a conflict, we can be more certain that the developer intentionally declared the dependency to control the mediation of the conflict, as opposed to the developer simply forgetting to remove a dependency that is no longer used.

It should be noted that both version-overriding techniques are not robust as they are not automatically inherited by the dependents [46]: `dependencyManagement` in transitive dependencies is ignored, and version-overriding by declaring an *unused direct* dependency will also be ignored by dependents if there is another version for the same dependency declared closer to the root. This means that manual version-overriding is not a sustainable way to deal with dependency-related issues caused by conflicting SoftVer declarations and requires continuous developer effort to maintain.

Results From the initial 226 projects in the dataset, RQ1 found that 103 contained at least one SoftVer conflict. Out of these 103 projects, we see that manual mediation of SoftVer conflicts is relatively common with 21% of the projects performing manual mediation of at least one conflict. The vast majority (98%) of the projects have at least one conflict that is not manually mediated. Another observation is that projects that use manual mediation techniques seem to have more conflicting declarations (12.55) on average than projects with unmanaged conflicts (7.2). This may indicate that Maven's nearest-first conflict resolution strategy may be causing issues for projects that have many conflicting declarations that developers feel the need to manually resolve themselves. It would be interesting to see if this observation still holds over a larger project sample, as it suggests that the risk of introducing dependency-related issues like breaking changes that require manual effort to resolve may increase with the increased presence of conflicting SoftVer constraints.

Chapter 5

Empirical Evaluation

This chapter evaluates the proposed solution (MARCO) described in Chapter 3 on empirical data by answering RQ3, RQ4, and RQ5, which are defined as follows:

- *RQ3: How successfully can we find test suites for dependencies using GitHub linking?*
- *RQ4: How effective is the dependency-specific approach at detecting breaking and non-breaking changes?*
- *RQ5: How successful is the proposed solution at improving Maven's dependency resolution?*

Each RQ has its own section, where its methodology and results are described. RQ3 and RQ4 evaluate the two core components of the MARCO GENERATOR in isolation: the GitHub linking algorithm (RQ3), and the dependency-specific approach for breaking change detection (RQ4). RQ5 evaluates MARCO as a whole by applying it to real Maven projects and investigating MARCO's applicability and effect on Maven's dependency resolution process with respect to flexibility and stability.

All computations were performed on a system with amd64 CPU architecture running Ubuntu 22.04.4, Maven version 3.9.6, and Java version 17.0.10. Running the same experiments on a different system will likely produce different results.

5.1 RQ3: Locating Dependency Tests

This section answers RQ3: *How successfully can we find test suites for dependencies using GitHub linking?* To generate the compatible version range for a specific dependency version (GAV), the GENERATOR relies on cross-version testing using the test suite of the dependency. This is part of the dependency-specific approach to breaking change detection and requires access to the dependency's test suite. However, it is not common practice to publish test jars on Maven Central [19]. If the dependency has its source code publicly available, we assume we are likely to find its tests where the source code is located. To locate the source code of a dependency, and therefore also hopefully its test suite, the GENERATOR implements a GitHub linking algorithm that links the dependency's GAV to the likely GitHub repository and commit SHA that produced the JAR published on Maven Central. Answering this RQ consists of looking into the following subquestions:

- *RQ3.1: How likely are we to find a GitHub link for a given GAV?*
- *RQ3.2: Are we more likely to find test source code on GitHub than test jars on Maven Central?*
- *RQ3.3: How certain can we be that the source code on GitHub corresponds to the source code that was used to produce the JAR on Maven Central?*

5.1.1 Methodology

Reproducible Central [34] is used as the ground truth dataset to evaluate the GitHub linking algorithm. The dataset was cloned on January 24, 2024, from `https://github.com/jvm-repo-rebuild/reproducible-central`. At this time, Reproducible Central contained reproducible build specifications for 3119 GAVs over 611 Maven projects (GAs). Each build specification includes the GitHub repository and tag (or commit SHA) that was used to reproduce the GAV's published Maven artifact. Build specifications where the repository or tag field is empty or unclear due to missing variable replacements were removed. 3094 valid data points were extracted from the initial 3119 build specifications.

To evaluate the GitHub linking algorithm (RQ3.1, RQ3.3), we provide it the GAV of each ground truth data point and check whether the matched repository and tag match the ground truth repository and tag. The evaluation was run on May 13, 2024. Because GitHub repositories and tags may be removed at any time, later runs may yield different results.

To find whether tests are more likely to be found as test jars on Maven Central or as test suites on GitHub (RQ3.2), we look up whether the artifact has a test jar on Maven Central and whether the matched GitHub repository at the matched tag has a test suite that is detectable by the Maven Surefire Plugin [7] which Maven uses to run tests. The test jar is found by checking if the artifact's repository on Maven Central has a file ending in `-tests.jar`. The test suite is found by checking if the GitHub repository has any files following the same default naming patterns used by the Surefire Plugin to detect tests: `Test*.java`, `*Test.java`, `*Tests.java`, and `*TestCase.java` [9].

5.1.2 Results

The results of applying the GitHub linking algorithm to the ground truth dataset are shown in Table 5.1. If the linking algorithm managed to produce a GitHub repository and tag for a GAV, the result is labeled as a *match success* and otherwise as *match failure*. Match successes are further categorized as *correct match* if the matched repository and tag matched the ground truth, and *incorrect match* otherwise. Match failures are similarly categorized into *repo failure* if no repository was found or *tag failure* if no tag was found.

The algorithm successfully produced a GitHub repository and tag combination for 1944 out of 3094 (63%) of the GAVs. 94% of the 1944 matches were matched using exact match

Table 5.1: Results of the Maven-GitHub linking test.

	Match success		Match failure	
Total GAVs	**Correct match**	**Incorrect match**	**Repo failure**	**Tag failure**
3094	1874	70	1101	49

Table 5.2: How many of the Reproducible Maven GAVs have test jars available on Maven Central, and test suites available on GitHub.

Total GAVs	**Maven test jar**	**GitHub test suite**
3094	62	1644

lookup, showing that the chosen common practice patterns are effective. For the GAVs where a GitHub combination was found, 1874 (96%) matched the ground truth and 70 (4%) did not. Out of the 1944 GAVs for which the algorithm found a match, Table 5.2 shows that 1644 (85%) had test suites, while only 62 (2%) out of all GAVs had test jars on Maven Central. This indicates that using GitHub linking is useful to increase the applicability of MARCO by increasing the chance of finding tests for a given GAV.

The algorithm failed to produce a GitHub repository and tag combination for 1150 (37%) of the GAVs. The majority (96%) of match failures are repo failures, which occur when there is no `scm` section in a POM or no GitHub link in the `scm` section. The linking algorithm would therefore greatly benefit from more Maven projects actively using and maintaining their `scm` tags. The algorithm could also be extended with support for other `scm` vendors than GitHub, such as GitLab, or by utilizing dependency databases like deps.dev [29] which stores information about GAVs such as source code repository links.

Incorrect matches and tag failures were manually inspected and categorized into failure categories to determine the cause of failure. These results are shown in Table 5.3. The categories are defined as follows:

Inconclusive The evaluation of the data point is inconclusive. The algorithm finds the tag that according to naming conventions logically corresponds to the GAV, however, Reproducible Central reports a different commit SHA not connected to any tag. For these failures, tag linking may not return the exact source code the artifact was built from.

Disagreement The algorithm gives a repository and/or tag that does not equal the ground truth. These are actual incorrect matches that are correctly labeled as incorrect.

Agreement The algorithm and ground truth give different tags, but both tags point to the same commit. These are actually correct matches that are falsely labeled as incorrect.

Invalid The ground truth repository or tag no longer exists so the data point cannot be evaluated.

Table 5.3: Breakdown of failure categories from the Reproducible Central test.

	Total	**Inconclusive**	**Disagreement**	**Agreement**	**Invalid**
Incorrect match	70	61	1	8	-
Tag failure	49	-	33	-	16

Out of the 70 incorrect matches, 8 are actually correct, resulting in 62 (3%) incorrect matches. If the algorithm finds a matching repo-tag combination, the source code is therefore highly likely to be what the Maven artifact was built from and is therefore suitable to use to determine breaking changes.

It should be noted that the 611 GAs included in Reproducible Central do not necessarily form a representative sample of the more than 500,000 GAs contained on Maven Central [38][1]. Furthermore, Reproducible Central only covers GAs that have public GitHub repositories, and not every GA on Maven Central does. In practice, the match success rate on Maven Central is therefore likely to be lower than on the Reproducible Central dataset.

To conclude, we find that GitHub linking is effective at increasing the number of GAVs for which we can find tests. A GitHub match was successfully found for 63% of GAVs (RQ3.1). The GitHub linking strategy found test suites for 53% of GAVs (83% of successfully linked GAVs), which is substantially higher than the 2% of GAVs we found test jars for on Maven Central (RQ3.2). Although the overall match failure is relatively high (and likely higher on Maven Central), it still results in a significantly larger number of test suites found than test jars. Furthermore, we can be confident that the source code for the given repository-tag combination corresponds to the source code that was used to produce the GAV's JAR on Maven Central: out of 1944 matches, Reproducible Central only reported a different commit for 3% of the GAVs (RQ3.3).

5.2 RQ4: Detecting Breaking Changes

This section answers RQ4: *How effective is the dependency-specific approach at detecting breaking and non-breaking changes?* The motivation behind this RQ is to see how the dependency-specific approach implemented by the MARCO GENERATOR[2] measures up to client-specific approaches in practice, and how well the dependency-specific approach can detect breaking (and non-breaking) changes detected by client-specific approaches. Because the dependency-specific approach is independent of the client code, it will overestimate breaking changes since a dependency may contain breaking changes that are not reached by a specific client's code. On the other hand, client-specific approaches that base behavioral compatibility decisions on client tests may underestimate breaking changes since it is not common practice to test your dependencies [21, 47].

To answer the RQ, we evaluate the MARCO on the following five datasets:

[1]as of January 26, 2024

[2]the terms MARCO and MARCO GENERATOR are used interchangeably in this section

The BUMP benchmark (n=372) [36] contains dependency version pairs that contain reproducible client-specific breaking changes obtained from the rejected Dependabot pull requests of GitHub projects that have failing builds or tests. Because the BUMP benchmark only contains breaking changes, any method that simply labels all version pairs as breaking would perform well on it. Because a dependency-specific approach will overestimate breaking changes, we expect the recall to be high on the BUMP benchmark. To get a better indication of how well MARCO detects breaking changes, we therefore collected a complementary dataset using a similar methodology to BUMP but for non-breaking changes, the Dependabot dataset.

The Dependabot dataset (n=1087) contains dependency version pairs that contain non-breaking changes collected from Dependabot pull requests of GitHub projects that were merged with no changes based on the assumption that developers are not likely to merge Dependabot pull requests if they break their projects. Because a dependency-specific approach overestimates breaking changes, we expect the false positive rate of MARCO to be relatively high on this dataset. A high false positive rate means the resolution process will be less flexible, but as long as the false negative rate is low developers can at least be confident that it is stable.

The UPPDATERA dataset (n=19) [21] is a qualitative dataset containing both breaking and non-breaking client-specific dependency updates. The dataset contains three evaluations per data point: a manual ground truth evaluation, the project test suite evaluation, and the UPPDATERA evaluation which is based on AST differencing and call graph analysis.

The COMPCHECK dataset (n=634) [47] is a quantitative dataset of version pairs that contain client-specific breaking changes according to the evaluation by COMPCHECK which uses cross-client tests and client call graphs.

The RANGER dataset (n=480) [46] differs from the other datasets which contain compatibility decisions for version pairs. This dataset instead contains client-specific compatible version ranges for specific dependency versions (GAVs) that avoid vulnerable versions of the log4j-core dependency in their direct and transitive dependencies, which allows us to compare the dependency-specific ranges generated by MARCO and the client-specific ranges generated by RANGER. Breaking changes are detected using a combination of client tests and call graphs.

Evaluating MARCO on these five datasets should give us a thorough understanding of how the dependency-specific performs at detecting breaking (and non-breaking) changes. The BUMP and Dependabot datasets will give an overall evaluation of how well MARCO can detect breaking and non-breaking changes at the client level, whereas the remaining three datasets of UPPDATERA, COMPCHECK, and RANGER also serve as comparisons with existing client-specific solutions. While it would be interesting to compare the performance of MARCO, UPPDATERA, COMPCHECK, and RANGER on the BUMP and Dependabot datasets, not all tools are available or easily re-run. We therefore instead run MARCO on the evaluation datasets provided by their papers.

The methodology for how each dataset is prepared before MARCO can be applied is covered in Section 5.2.1. The evaluation of MARCO is then split into two categories depending on the nature of the dataset: whether the dataset contains data points that represent a classification or retrieval problem. Section 5.2.2 evaluates MARCO on the BUMP, Dependabot, UPPDATERA, and COMPCHECK datasets, which covers how effective MARCO is at classifying a dependency update as breaking or non-breaking. Section 5.2.3 evaluates MARCO on the RANGER dataset, which covers how effective MARCO is at retrieving all compatible versions for a given dependency version. Finally, Section 5.2.4 concludes the overall results and answers the research question.

5.2.1 Preparing the Datasets

Before MARCO can be applied, each dataset is cleaned and prepared into a format that MARCO understands. 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 a list of compatible versions (retrieval). Common for all existing datasets is that they are client-specific, meaning that they may contain duplicate data points that represent the same dependency version or dependency version pair, just on different client projects. Because MARCO is dependency-specific and therefore client-agnostic, the duplicate dependency updates are filtered out. The specific details on how each dataset was obtained, cleaned, and prepared are presented below.

The BUMP benchmark The benchmark was cloned from `https://github.com/chains-project/bump` on March 22, 2024, and contained 571 data points. Before we can evaluate MaRCo on BUMP, we need to clean it since not all data points are relevant for the evaluation. For each data point, BUMP contains a json metadata file with the necessary information we need under the key `updatedDependency`. From this key we extract the name of the dependency which consists of its groupId and its artifactId, and the 'old' and 'new' versions that define the update. This information will be used to find the GAVs of the base (old) and candidate (new) versions used to run MARCO's compatibility check. 65 duplicate data points were filtered out so that the remaining 506 data points each represent a unique dependency update.

Dependency updates that are not relevant for the evaluation of MARCO were then filtered out. First, data points with the *plugin* and *POM* update types were removed. Plugin updates are removed since MARCO only replaces dependency declarations and not plugin declarations. A POM-type dependency imports a set of one or more dependencies from an external POM. An update to this POM may therefore result in multiple GAV updates, and we do not know which specific GAV update caused the breaking change the data point represents. Finally, Each data point has a failure category describing the cause of the breaking change which includes compilation failures, test failures, dependency resolution failures, dependency version lock failures, and enforcer rule failures. Dependency version locks and enforcer rules are configurations defined in a specific project's POM and are therefore not relevant for a dependency-specific breaking change approach, these are therefore filtered out. After all filtering steps, we were left with a cleaned dataset of 372 data points, or 65% of the original dataset.

The Dependabot dataset Because the BUMP benchmark only contains data points representing breaking changes, we create the Dependabot dataset to serve as a complimentary dataset that represents non-breaking changes using a similar but less extensive approach than BUMP. We collect Dependabot PRs that perform a single dependency update to a POM, and that were approved with no changes. Assuming that developers do not approve Dependabot updates that break their projects often, this should be a relatively safe and simple heuristic as to whether an update is compatible. First, GitHub projects were collected using the GitHub API with the following filters: created between January 1, 2023, and March 5, 2024, Java as programming language, at least 20 stars, and at least 50 commits. Projects that did not have a `dependabot.yml` file with Dependabot enabled for Maven were then filtered out. For each remaining project, we collected the pull requests opened by Dependabot that updated a POM file and were merged without any changes. From the collected pull requests, we create the update data points by extracting the GAV information from the title using the following title pattern: 'bump `groupId:artifactId` from `oldVersion` to `newVersion`'. In the end, 1087 data points representing single dependency updates were collected from 1748 pull requests over 85 projects.

The UPPDATERA dataset The original dataset was taken from Table 3 in Hejderup and Gousios [21] and contained 22 client-dependency update pairs taken from Dependabot pull requests. 19 of the data points contained a ground truth class label corresponding to a compatibility decision (S/compatible or U/incompatible), the others were filtered out. No further processing or filtering steps were necessary before applying MARCO.

The COMPCHECK dataset The original dataset contained 758 client-dependency update pairs. Two data points were removed: one specified a range for its 'old' version, for which we do not know what resolved version COMPCHECK based the compatibility evaluation on; the other had no 'old' or 'new' version given. 18 other data points had malformed version numbers and had to be manually corrected. For example, the GAV com.google.guage:guava:20 was given while the version 20 does not exist, although 20.0 does. All 18 corrected data points followed this pattern of only providing the major version as the version string. If the version string is not exact MARCO will not be able to evaluate the data point. From the remaining 756 data points, 634 unique dependency upgrades over 430 unique GAs were extracted and evaluated by MARCO.

The RANGER dataset The original dataset was collected by Zhang et al. [46] to evaluate RANGER. The original dataset consisted of multiple CSV files containing a total of 4107 data points. The data points contained compatible version ranges for specific GAVs for specific GitHub projects. Because MARCO only cares about the GAV and not the project it is used in, 480 data points were extracted which corresponds to the number of unique GAVs found in the original dataset. Each data point then maps a GAV to a compatible range generated by RANGER.

5.2.2 Compatibility Classification of Version Pairs

The BUMP, Dependabot, UPPDATERA and COMPCHECK datasets contain data points that are version pairs (v_{old}, v_{new}) that represent a single dependency version update, and a compatibility decision whether the update is breaking (incompatible) or non-breaking (compatible). This information can be modeled as a binary classification problem: given a dependency update, classify it as breaking or non-breaking. As the first step in generating compatible version ranges, MARCO also performs this classification of a given version pair. If the version pair is classified as non-breaking, then v_{new} is added to the compatible versions of v_{old}. Applying the classification step of MARCO on these datasets will therefore evaluate how effective MARCO is at classifying breaking and non-breaking changes.

Methodology We evaluate the performance of MARCO using accuracy, recall[3], and precision[4], which are common metrics used for the evaluation of binary classification models [11] and are defined as follows:

$$accuracy = \frac{TP+TN}{P+N}, \quad recall = \frac{TP}{TP+FN}, \quad precision = \frac{TP}{TP+FP} \tag{5.1}$$

The BUMP, Dependabot, and COMPCHECK datasets only contain one compatibility class, which is chosen as the positive label P. The P for BUMP and COMPCHECK therefore refers to the breaking compatibility class, whereas P for Dependabot is the non-breaking compatibility class. Since UPPDATERA contains both compatibility classes, we choose the breaking compatibility class as P since it is the minority class label. Precision is only meaningful for the UPPDATERA dataset which has two class labels, and is therefore not reported for the BUMP, Dependabot and COMPCHECK datasets for which it is trivially 1.0.

Results The outcome of determining 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 5.4. Inconclusive data points for which MARCO was unable to determine the compatibility class were manually inspected, and Table 5.5 shows the causes of inconclusive data points. The evaluation metrics for each dataset are shown in Table 5.6. The remainder of this section explains the result for each dataset in more detail.

The BUMP benchmark Out of the 371 data points, MARCO was able to successfully evaluate 349 of them, resulting in a success rate of 0.94. Out of the 349 successfully

[3]also called Positive Predictive Value
[4]also called Sensitivity

Table 5.4: MARCO results on the compatibility classification datasets.

	BUMP	Dependabot	Uppdatera	CompCheck
Total	371	1087	19	634
Statically incompatible	346	390	7	511
Statically compatible	25	576	11	123
Linked	16	374	9	67
Runnable	3	78	1	1
Dynamically incompatible	1	11	0	1
Dynamically compatible	2	67	1	0
Total conclusive	349	468	9	512
Total inconclusive	22	629	10	122

Table 5.5: Breakdown of the reasons why MARCO could not dynamically evaluate inconclusive datapoints in each classification dataset.

	BUMP		Dependabot		Uppdatera		CompCheck	
Reason for error	**Link error**	**Run error**	**Link error**	**Run error**	**Link error**	**Run error**	**Link error**	**Run error**
No GitHub repo found	4		168		2		50	
No GitHub tag found	5		34		0		6	
Not a Maven project		12		139		3		13
Did not compile		0		76		5		40
Has no runnable tests		1		81		0		7
Total	9	13	202	296	2	8	56	55

Table 5.6: Summary of classification datasets and performance evaluations.

Dataset	Samples	Break	No break	Method	Success Rate	Accuracy	Recall	Precision
BUMP	372	✓		MARCO	0.94	0.93	0.99	
Dependabot	1087		✓	MARCO	0.43	0.06	0.17	
Uppdatera	19			MARCO	0.47	0.56	1.0	0.50
		✓	✓	UPPDATERA		0.33	1.0	0.43
				Project tests		0.56	0.25	0.50
CompCheck	634	✓		MARCO	0.81	0.81	1.0	

evaluated data points, MARCO labeled 347 points correctly as incompatible and labeled 2 points incorrectly as compatible, resulting in a recall of 0.99.

The two points incorrectly labeled as compatible were the update of org.codehaus.plexus:plexus-io from 3.2.0 to 3.3.0 and the update of net.minidev:json-smart from 2.4.8 to 2.4.9. Upon manual inspection, these two data points are examples of the same edge case that the dependency-specific approach is sensitive to wrongly classify: i.e. the test suite of the base version is lacking in coverage and the candidate version introduces a bug (or intentional change) that is not caught by the base test suite.

plexus-io:3.3.0 introduced a breaking bug [41], which was not caught by the test suite of 3.2.0. This bug was later fixed in 3.3.1 and a test was added to catch the bug in later releases [14]. This means that versions containing this bug (3.3.0) could be compatible with versions prior to the test being added in 3.3.1, but not after. However, running MARCO on 3.2.0 with candidates 3.3.1 shows that 3.3.1 is compatible with 3.2.0, so that a project using MARCO with 3.2.0 pinned would resolve the bug-free 3.3.1 (if possible) since it is the later version in the generated range. The assumption for this edge case not being a problem is that bugs are eventually detected and fixed in new patch releases. And since Maven will resolve the latest version it can from a range, the ranges essentially enable automatic bug updates.

The update of json-smart showcases a similar problem: 2.4.9 intentionally introduced a depth limit as a security measure [12]. While not a bug, it is an example of how client projects may rely on behavior that is not tested by the dependency. If the dependency developers change this behavior, it may cause breaking changes in the client as was the case here where the specific project that performs this update in the BUMP benchmark contained a test that exceeded the newly introduced depth limit.

The Dependabot dataset Out of 1087 collected data points, MARCO was able to successfully evaluate 468 of them, resulting in a success rate of 0.43. Out of the 468 successfully evaluated data points, MARCO labeled 67 points correctly as compatible and 401 points incorrectly as incompatible, resulting in a recall of 0.06. Although MARCO assigns the majority of data points the correct label in the static and dynamic stage, the recall is low because 86% 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 0.60 and a dynamic recall of 0.86.

The UPPDATERA dataset Because this dataset only contains 19 data points, we provide the entire dataset in Table 5.7. All data provided in the table, except from the MARCO column, is taken from the original paper [21]. Because we do not use all of the original class labels, we have slightly changed the definitions of the true/false positive/negative labels compared to the ones given in the original dataset: true positives (TP) are pairs correctly identified as incompatible, false positives (FP) are pairs incorrectly identified as incompatible, true negative (TN) are pairs correctly identified as compatible, and false negatives (FN) are pairs was incorrectly identified as compatible. Rows are marked green when MARCo agrees with the manual ground truth evaluation. Yellow, orange, and red rows are updates for which MARCo disagrees with the ground truth but agrees with two, one, or none of the other evaluation methods, respectively. Rows are marked white if MARCo did not manage to determine compatibility.

The MARCo GENERATOR was able to successfully evaluate 9 out of the 19 data points, resulting in a success rate of 0.47. Based on the subset of data points that were successfully evaluated, MARCo achieves a recall of 1.0 and a precision of 0.50. On the same subset UPPDATERA achieved a recall of 1.0 and precision of 0.43, while project tests achieved a recall of 0.25 and a precision of 0.50.

Out of the 9 successfully evaluated data points, MARCo agreed with the ground truth evaluation in 5 (56%) of the cases. The incorrectly evaluated 4 cases were all false positives due to MARCo reporting static incompatibilities that were not relevant (i.e. not reachable) by the specific project performing the dependency upgrade. In these cases, a client-specific approach would be beneficial. Despite the high false positive rate (4/9), the false negative rate is 0. This shows that MARCo is better suited when favoring stability over flexibility.

For the data points where MARCo disagreed with ground truth, MARCo performs similarly to UPPDATERA and better than project tests. Compared to UPPDATERA, MARCo did not catch more correct decisions but that was also not expected since we expect a dependency-specific approach to overestimate breaking changes. Indeed, for the update of com.google.code.gson:gson from version 2.2.4 to 2.8.6, MARCo did overestimate it as breaking while both client-specific approaches did not.

The two data points that were inconclusive due to the GitHub linking failing were manually inspected, and the two dependencies indeed did not have GitHub repositories. The remaining 8 inconclusive data points were caused by MARCo not being able to run the dynamic compatibility check because the linked GitHub repositories either used non-Maven build systems, or MARCo was unable to compile the repository.

The COMPCHECK dataset The MARCo GENERATOR was able to successfully evaluate 512 data points, resulting in a success rate of 0.81. All of the successfully evaluated data points were labeled correctly as incompatible, resulting in a recall of 1.0. Since the success rate is fairly high, we did not perform manual restoration of GitHub repository links. Although most of the data points (81%) were correctly labeled as incompatible by the static compatibility check, a large part (55%) of the remaining 123 points becomes inconclusive before reaching the dynamic compatibility check. Table 5.5 shows that the majority of data points become inconclusive because they could not be linked to a GitHub repository (50), or because MARCo could not compile it (40).

Table 5.7: Comparison of the compatibility evaluations based on manual evaluation, project test suite, the Uppdatera tool, and MaRCo. The reason for a decision is given between brackets for MaRCo.

PR	GA	Old version	New version	Manual	Project 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-expression	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)

5.2.3 Retrieval of Compatible Versions

Instead of mapping version pairs to a compatibility decision, 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: given a GAV, retrieve all compatible versions. Applying MARCO on the RANGER dataset will evaluate how effective it is at retrieving all compatible versions for a given GAV.

Methodology Similarly to the evaluation of MARCO's performance on the classification task, we evaluate the performance of MARCO on the retrieval task using recall and precision, which are also common metrics used for the evaluation of retrieval models [37, 11]. Let V_{marco} and V_{ranger} be the set of compatible versions retrieved by MaRCo and RANGER, respectively. We define recall and precision as follows:

$$recall = \frac{|V_{marco} \cap V_{ranger}|}{|V_{ranger}|}, \quad precision = \frac{|V_{marco} \cap V_{ranger}|}{|V_{marco}|} \tag{5.2}$$

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. Because a range only includes the lower- and upper bounds of compatible versions, we need to unroll the range into a list of compatible versions to be able to calculate recall and precision. The RANGER and MARCO-generated ranges are converted to lists by looking up the available versions on Maven Central between the lower- and upper bounds of the range. Because the RANGER dataset was created April 1, 2023, only available versions published on Maven Central before this date were considered by MARCO to prevent it from including versions in its ranges that RANGER did not evaluate.

Results The MARCO GENERATOR was applied to 480 data points in this dataset, but only managed to successfully evaluate 8 of them, resulting in a very low success rate of

Table 5.8: Comparison of the ranges produced by Ranger and MaRCo on the Ranger dataset. The number after the range indicates how many versions are contained in the range.

GAV	Ranger Range (#)	MaRCo Range (#)	Recall	Precision
com.indoqa:indoqa-boot:0.12.0	[0.12.0, 0.16.0] (5)	[0.10.0, 0.16.0] (7)	1.0	0.71
eu.unicore.security:securityLibrary:5.3.1	[5.3.1, 5.3.2] (2)	[5.3.0, 5.3.6] (6)	1.0	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)	1.0	0.06
org.robotframework:jrobotremoteserver:4.0.1	[4.0.1, 4.1.0] (2)	[4.0.0, 4.1.0] (3)	1.0	0.67
org.spdx:spdx-tools:2.2.5	[2.2.5, 2.2.6] (2)	[2.2.5, 2.2.5] (1)	0.5	1.0
com.helger.photon:ph-oton-bootstrap4-stub:8.3.2	[8.3.2, 8.3.3] (2)	[8.2.5, 8.3.2] (8)	0.5	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	1.0
Macro average			0.66	0.52

Table 5.9: Breakdown of the reasons why a datapoint could not be dynamically evaluated in the Ranger dataset. The percentages are calculated based on the whole dataset of 480 GAVs.

	Before link restoration		**After link restoration**	
Reason	**Link error**	**Run error**	**Link error**	**Run error**
No GitHub repo found	264 (55%)		172 (36%)	
No GitHub tag found	23 (5%)		24 (5%)	
Not a Maven project		78 (16%)		143 (30%)
Did not compile		64 (13%)		88 (18%)
Has no runnable tests		43 (9%)		45 (9%)
Total	287 (60%)	185 (38%)	196 (40%)	276 (58%)

0.02. Because only 8 data points were evaluated, we show the full evaluations for these data points in Table 5.8. Table 5.9 lists 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%), deps.dev [29] was used to manually restore the missing GitHub repositories. In total there were 264 GitHub repository failures from 62 unique GAs. However, five GAs were responsible for >50% of the failures, with one GA being responsible for 23% of failures, as can be seen in Figure 5.1. The latter's repository was found on deps.dev, but no longer exists on GitHub. Out of the top five GAs causing failures, only one GA had its GitHub repository link restored. In total, 92 data points with GitHub repository failures were manually restored and MARCO re-evaluated these data points with the manually provided repository link. However, the manual restoration did not result in more data points being successfully evaluated. Instead, the reasons for inconclusive data points mainly shifted from link to run errors. Noticeably, 30% of data points could not be run because they did not use Maven as build system (e.g., Gradle, SBT). MARCO cannot determine compatibility unless it is able to compile the GAV and run tests on it using Maven.

For the data points MARCO did manage to evaluate, we see that the recall is in general higher than the precision. For the retrieval of compatible versions, this means that MARCO

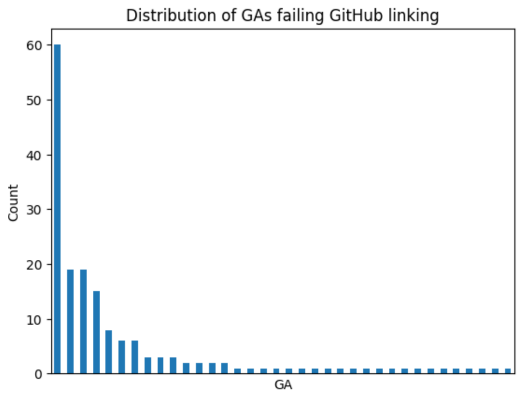

Figure 5.1: Distribution of GAs failing GitHub linking in the RANGER dataset

was in general able to retrieve most versions deemed compatible by RANGER, but the lower precision is caused by MARCO also including versions that RANGER did not. In other words, it seems the MARCO-generated ranges are more flexible than those provided by RANGER even though we would expect RANGER's ranges to be wider due to its client-specific approach. This could be explained by the fact that RANGER's ranges also avoid versions of the log4j-core dependency that contain the Log4Shell vulnerability. Another reason for the wider MARCO ranges is that downgrades are also included which RANGER does not include. This adds to flexibility since MARCO can in some cases provide stable downgrades in case upgrades are not possible.

5.2.4 Conclusions

To conclude, we observe that MARCO has a high recall ($\geq$0.99) on all classification datasets for breaking changes, as expected. The success rate is also higher on these datasets because a lot of data points are deemed incompatible at the static check which does not require compiling or running tests. MARCO has a low false negative rate and a high false positive rate for detecting breaking changes. This is also confirmed by the high recall and lower precision on the UPPDATERA dataset, showing that while false positives (breaking) are likely, false negatives (non-breaking) are unlikely. This suggests MARCO will affect the resolution process in a way that favors stability over flexibility. In a broader context, a high false positive rate is favorable over a high false negative rate when stability is favored over flexibility. The high false positive rate reduces the size of the ranges, resulting in lowered flexibility, but compensates with a low false negative rate which gives confidence in stability since the ranges are not likely to contain breaking changes.

When it comes to retrieving all compatible versions for a given dependency version, MARCO achieved a higher recall than precision in general. This shows that although MARCO in general retrieved most of the compatible versions, it also included versions that were not included by RANGER. This indicates that the MARCO-generated ranges are more flexible than those provided by RANGER even though we would expect RANGER's ranges to be wider due to its client-specific approach. The reason for this could be because

RANGER avoids vulnerabilities, but also because MARCO includes downgrades in the compatible versions for added flexibility, which RANGER does not. More flexible ranges could also mean that MARCO includes breaking versions, but the evaluation on the classification task shows that this is unlikely.

Furthermore, MARCO produced two false negative breaking changes on the BUMP benchmark which shows that the dependency-specific approach is affected by the quality of the dependency's test suite just like the client-specific test-based approaches depend on the quality of the client's test suite. If a dependency lacks test coverage, bugs may not be caught by the dependency's test suite which will cause MARCO to produce false negatives.

Finally, the Dependabot, RANGER and UPPDATERA results indicate that the low success rate is the main threat to MaRCo's effectiveness rather than the dependency-specific approach for static and dynamic compatibility. The low success rate is particularly problematic for detecting non-breaking changes because non-breaking changes need to pass both the static and dynamic compatibility checks. Most data points that pass the static check become inconclusive due to not being able to run the dynamic check due to the complications related to building and running the dynamic check. This problem is further exacerbated by dependencies that are released on Maven Central that do not use Maven as the build system.

5.3 RQ5: Impact on Maven's Dependency Resolution

This section answers RQ5: *How successful is the proposed solution at improving Maven's dependency resolution?* The purpose of this RQ is to evaluate MARCO as a whole, by applying both the MARCO REPLACER to GitHub projects and the MARCO GENERATOR to their dependencies. 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. Based on the results of RQ4, which found that MARCO generates ranges that have a high false positive but a low false negative rate for breaking changes, we expect MARCO to influence the resolution process to favor stability over flexibility.

5.3.1 Methodology

The methodology used to evaluate this RQ is separated into three steps. First, we select the GitHub projects we will apply MARCO to. Second, before we can apply the REPLACER to the GitHub projects, we first need to use the GENERATOR to compute the compatibility mappings 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 GitHub projects. Finally, to determine how the resolution process has changed we calculate different metrics using the dependency trees before and after applying the REPLACER to the GitHub projects. Each step is explained in further detail in the following paragraphs.

Project selection GitHub projects were collected using the GitHub API using the following filters: created between January 1, 2023, and May 5, 2024, Java as programming language, at least 20 stars, and at least 50 commits. For the sake of feasibility to make POM replacement easier, projects without a POM file in its root directory and multi-modular projects were discarded so that the POM we want to replace is easy to locate. Projects that we could not compile or did not have at least one runnable test were also discarded, since we want to check whether applying MARCO introduced breaking changes to the project or not. Compilation failures would indicate static breaking changes, while failing previously passing tests would indicate behavioral breaking changes. After all filtering steps, 102 projects remained.

Applying MARCO Applying MARCO to the previously collected projects involves the following steps:

1. Generate the verbose dependency tree for each selected project. The act of building the projects and generating their dependency trees will download all required dependencies into the local `m2` folder.
2. For each declared dependency (GAV) found in the dependency trees, use the GENERATOR to compute their compatibility mappings.
3. Apply the REPLACER to the selected projects to replace their POMs.
4. Apply the REPLACER to the POMs in the `m2` folder to replace the POMs of all dependencies.
5. Re-build the selected project using the `mvn test-compile` phase, re-run the tests, and re-generate the new verbose dependency tree. The dependency trees, compilation and test results are then used to compute the various metrics used for evaluation.
6. If new dependencies are downloaded during the building in the previous step, redo all previous steps until no new dependencies are downloaded. In this particular experiment, no new dependencies were encountered after repeating the steps twice.

Evaluation metrics To measure how the dependency resolution process has changed, we define 9 metrics which are described below. In addition to these metrics, which are calculated for each project, we also look into the size distribution of the ranges generated by MARCO to get a further indication of flexibility. The larger the ranges are in general, the more flexible the resolution process will be.

Success rate The fraction of projects that still resolve, compile, and pass their test suites after having at least one of their dependency declarations (direct or transitive) replaced by MARCO. A very low success rate indicates that MARCO has made the resolution process less flexible if there is increased resolution failure, or less stable if there is increased compilation or test failure.

Resolution failure means that there are conflicting dependency declarations that declare versions that are not compatible with each other (according to MARCO). Assuming that the project's main branch is not in a broken state to begin with, this indicates that there may possibly be latent breaking changes in the project that it does not currently reach. If the resolution process has become less flexible due to the version ranges being too strict, adding a client-specific approach to widen the compatible version ranges may be necessary to reach an acceptable success rate. Because the static compatibility check should ensure compilation, no compilation failures are expected. Failing project test suites indicate that there are breaking changes in the MARCO-provided version range that were not caught in the dependency's test suite.

Replacements and Replacement rate Replacements are the total number of resolved dependencies that originate from a dependency declaration that was replaced by a MARCO-generated range. To keep track of replacements, the REPLACER adds the attribute `replaced` to the version tags it replaces. The replacement rate is then the fraction of resolved dependencies that are replacements. MARCO's impact on the resolution process depends on how many dependencies it is able to replace. If these metrics are low, it suggests the impact is limited.

Downgrades and Upgrades The number of GAs for which a lower or higher version has been resolved after applying MARCO, respectively.

Downgrade steps and Upgrade steps The sum of how many versions each downgrade or upgrade is behind or ahead of the previously resolved version, respectively.

Change rate and Change magnitude The change rate refers to the fraction of dependency GAs that resolve different versions after applying MARCO. The change magnitude of a project describes the total magnitude of version changes, and is defined as the sum of total upgrade and downgrade steps. Change rate and change magnitude give an indication of how flexible the resolution process is, since it measures how much the newly resolved versions deviate from the previously resolved versions.

5.3.2 Results

The results of computing the metrics are shown in Figure 5.3. Out of the 102 projects that passed the selection process, MARCO was able to replace at least one dependency declaration (direct or transitive) for 73 of them (71%), with 13% of a project's dependencies replaced on average. Out of the 73 projects with at least one replacement, all of them still resolved and compiled while 71 (97%) passed their test suites after replacement. The high success rate (97%) after replacement means that breaking changes are not likely to be introduced and that the ranges are flexible enough to not cause resolution failures. However, this number may be artificially high due to the relatively low average replacement rate of 13% and should therefore be regarded as an upper-bound.

5% of dependencies resolve a different version than before with an average version change magnitude of 2.57, showing added flexibility in the resolution process. On average, downgrades were more common than upgrades, and downgrades were more likely to be larger in magnitude as well. While upgrading may be viewed as more desirable from the perspective of bug fixes and security updates, downgrades may have been necessary to avoid breaking changes. Because the average change magnitude is non-zero, that means that the generated ranges must contain more than one version on average. To get a better understanding of how the ranges contribute to the change magnitude, we calculated the range size distribution in Figure 5.2. We see that there is a large variation in the number of versions included in the MARCO-generated ranges, with sizes from 1 to 104 versions. The majority of the ranges only contained one version, which is the equivalent of hard version constraints. However, many also contained up to five versions with an average of 3.82 versions.

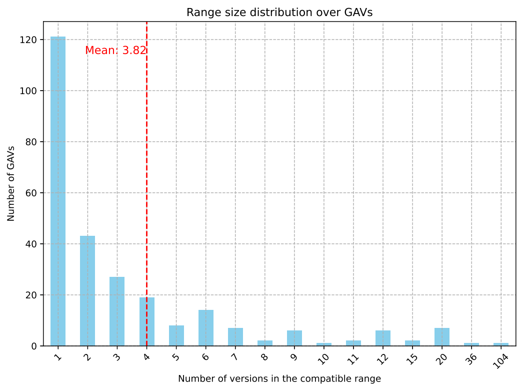

Figure 5.2: Size distribution of ranges computed by MaRCo

The collection, computation, and evaluation steps for this RQ exceeded 20 hours. The majority of the time (16 hours) was spent by the GENERATOR computing the compatibility mappings, which included cross-version testing of over 3500 dependencies. Replacing 102 project POMs using the REPLACER only took 10 minutes, confirming that MARCO is lightweight on the client side, and the computationally expensive compatibility mappings can be pre-computed and reused by multiple clients.

To conclude, MARCO was able to replace 13% of resolved dependencies on average for 71% of selected projects. The majority of ranges resulted in hard version constraints, which lowers flexibility compared to SoftVer constraints, but increases stability. 97% of projects with replacements did not suffer breaking changes, so the replaced ranges provide a high degree of stability. All in all, as seen by the previous RQs as well, MARCO favors stability over flexibility. MARCO is therefore likely to make a Maven project's resolution process more stable than a SoftVer-only approach, but also more flexible than a hard-constraint-only approach.

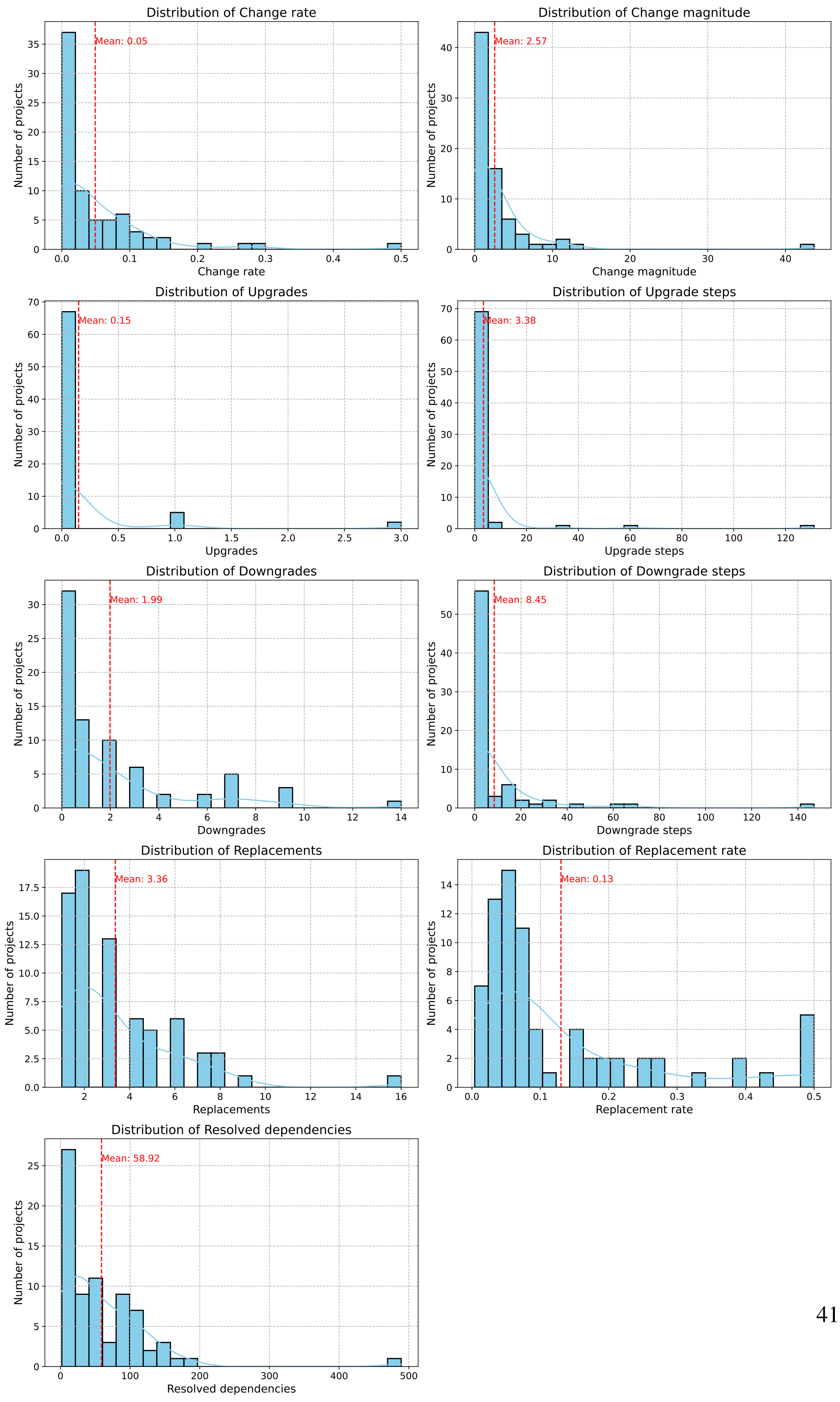

Figure 5.3: Distributions and averages for each evaluation metric

Chapter 6

Discussion and Future Work

This thesis provides the design and evaluation of MARCO, a toolkit that influences Maven's dependency resolution process towards being more reliable to decrease dependency-related problems and improve developer efficiency. We defined transparency, flexibility, and stability as core properties of a reliable dependency resolution process, and found that used undeclared dependencies and conflicting soft version constraints are common dependency smells that negatively affect these properties. MARCO therefore increases transparency by injecting the missing dependencies and balances stability and flexibility by replacing soft version constraints with dependency-specific compatible version ranges. This chapter summarizes and reflects on the key findings and limitations of the work presented in this thesis, and provides several potential directions for future work that address these findings and limitations.

Applicability issues due to inconclusive dynamic check The main limitations of MARCO relate to its applicability. The fewer dependencies MARCO can fully evaluate for compatibility, the fewer dependency declarations it can replace, and the less impact it will have on the overall dependency resolution process. While the dependency-specific approach shows promise in its ability to balance flexibility and stability in a way that is more scalable and lightweight on the client-side than client-specific approaches, future work should address its applicability issues. The applicability issues originate from the dynamic compatibility check, which requires locating, compiling and running dependency tests. Firstly, RQ3 found that the majority (95%) of link failures were caused by the POM file not containing any GitHub link. Future work relying on Maven-to-Github linking should therefore consider using other methods in addition to the POM file, such as looking up repository information using dependency databases like deps.dev [29], or implement search heuristics related to the dependency's groupId and artifactId. Secondly, RQ4 found that many compatibility evaluations become inconclusive due to failing to compile or run tests. Compiling any given Maven project found on GitHub is not trivial, as the projects may require specific or complex build instructions. Projects may also have external or environmental dependencies that are missing when MARCO attempts to compile and run the tests. Future work using cross-version testing to determine behavioral compatibility should therefore consider how to improve compilation success, for example by investigating whether static analysis methods like UPPDATERA [21] can be adjusted to be dependency-specific and used to approximate behavioral compatibility when dependency tests cannot be run.

Dependency-specific compatibility is client-agnostic RQ4 and RQ5 showed that MARCO is likely to provide stable ranges that are stricter than necessary, reducing the potential flexibility of the dependency resolution process. This is logical, as the dependency-specific approach to compatibility checking is client-agnostic and clients may or may not be affected by breaking changes in a dependency. By extending the dependency-specific approach with a client-specific approach, the ranges could be expanded to allow for more flexibility without compromising stability. To achieve this, future work could consider using hybrid approaches that combine the scalability and stability provided by the dependency-specific approach and the flexibility provided by client-specific approaches. For example, a hybrid approach could use MARCO's dependency-specific approach to build a knowledge base of incompatibilities. Client-specific techniques such as reachability analysis could then be employed to see whether the client reaches any parts of the dependency's API that contain breaking changes, as used by RANGER [46] and UPPDATERA [21]. The method used by MARACAS [28] which allows ignoring breaking changes in internal parts of the API not intended for client use could also be considered. The pre-computed dependency-specific compatible ranges can then be expanded with previously incompatible versions where the breaking changes are not reachable by the client. This overall approach is similar to that of COMPCHECK [47], except the knowledge base would be built by a dependency-specific instead of a client-specific approach.

Dynamic language features causing missed injections MARCO increases the transparency of a project's dependency resolution process by injecting used undeclared, or missing, dependencies. However, the increase in transparency is limited by the Maven Dependency Plugin which is used to detect and inject the dependencies. The plugin cannot detect undeclared dependencies that are used via dynamic language features such as Java's Reflection API. Landman et al. [22] found that 78% of Java programs had at least one usage of the Reflection API that is problematic for static analysis tools, such as the Maven Dependency Plugin. RQ1 found that 45% of projects have at least one missing dependency but since reflection usage in Java programs is common, the actual prevalence is likely higher and MARCO is likely not injecting all missing dependencies. Used undeclared dependencies may cause resolution, compilation, or test failure when updating a project's dependencies if the undeclared dependency's transitive declaration disappears or changes to a non-compatible version. Because we do not observe a high rate of resolution, compilation, or test failures in RQ5, we do not believe missing injections pose a significant threat to the evaluation of MARCO; however, future work could consider how we can improve the detection of used undeclared dependencies that the Maven Dependency Plugin cannot detect due to dynamic language features. Song et al. [39] developed SLIMMING, which debloats dependencies while taking into account dependency usage through reflection. If SLIMMING can detect unused declared dependencies, their method can perhaps also be used to detect used undeclared dependencies due to reflection.

Compatibility result is subject to test coverage Whether the computed compatible ranges are actually compatible and do not contain undetected breaking changes will depend on the quality of the dependency's test suite. A dependency with a low-quality test suite will likely catch fewer breaking changes than a dependency with an extensive test suite, as seen by the two breaking changes in the BUMP benchmark that MARCo wrongly labeled as non-breaking in RQ4. If a test suite has high coverage, we are more confident that the test suite adequately describes the expected behavior of the dependency. To increase the confidence in the compatibility decisions obtained via cross-version testing, future work could investigate how test generation techniques such as EVOSUITE [18] could be used to increase test coverage to an acceptable threshold. Test generation could not only reduce the number of false positives produced by cross-version testing when test coverage is lacking, but also the number of inconclusive compatibility decisions caused by failure to locate or run dependency tests.

Scaling to ecosystem-level deployment The current evaluation of MARCo performs full dependency replacement by replacing the dependency POMs on the client's `m2` folder. A more practical solution was presented in Chapter 3, where a mirror repository of Maven Central is created containing pre-replaced POMs. How to efficiently set up and maintain this mirror repository for a large ecosystem with frequent dependency releases such as Maven is another direction for future work. Besides the mirror repository, the compatibility mappings also need to be maintained as new dependency versions are released. The mappings are generated using regression testing which is computationally expensive [19]. Future work could consider looking into regression test selection techniques to optimize catching as many breaking changes as possible with the least computation overhead.

Threats to validity Finally, we discuss potential threats to validity that may impact the correctness, reproducibility, or generalizability of our findings, along with the measures we have taken to mitigate these threats.

Like most software, the MARCo toolkit may contain bugs that may affect the correctness of its output and, consequently, its evaluation. We have therefore performed unit testing of MARCo's core functionalities. While these unit tests help verify expected behavior, they are not exhaustive and may not cover all possible inputs. Another threat to correctness is the usage of dependency test suites to determine compatibility since the compatibility decision relies on the quality of the test suite. Low test coverage could cause incompatible versions to be falsely labeled as compatible. MARCo mitigates this threat to some extent by not relying solely on test suites for compatibility, but also on bytecode differencing to catch static incompatibilities.

The reproducibility of our results may be affected by software, hardware, and temporal dependencies. Certain Maven projects may only resolve, compile, or run on specific platforms, software, or hardware configurations. Furthermore, GitHub linking is affected by the mutable nature of GitHub, where repositories and tags may no longer be accessible in the future. To address this we have provided the most relevant details on the hardware and software used, as well as when a particular experiment was performed.

The representativeness of the datasets used in this thesis impacts the generalizability of our findings. First, RQ1, RQ2, and RQ5 use datasets with a relatively small sample size of Maven projects on GitHub created since 2023. This sample may not be representative of the larger, and older, population of Maven projects and the findings may therefore not be fully generalizable. For example, RQ2 found over a sample of 226 projects that projects with manual conflict declarations have on average more conflicting soft constraints than projects that do not. It would be interesting to see if this observation still holds over a larger sample, as it suggests that the risk of introducting dependency-related issues that require manual effort to resolve may increase with the increased presence of conflicting SoftVer constraints. Second, RQ3 used the Reproducible Central dataset to evaluate the GitHub linking, which is a sample of 611 GAs with public GitHub repositories and may not be representative of Maven Central. The linking success rate should therefore be regarded as an upper bound. However, the primary goal of using Reproducible Central was to assess the accuracy of the linking rather than the ability to produce links. Additionally, our finding of the low test jar availability is also supported by a previous study using a more representative sample [19]. Finally, RQ4 uses five different datasets to evaluate MARCO's effectiveness in detecting breaking changes and shows consistent results so that we can be relatively confident in our findings.

Chapter 7

Summary

Dependency-related issues not automatically resolved by dependency managers can be time-consuming for developers to resolve manually. We define transparency, stability, and flexibility as the core properties of a reliable resolution process that aims to mitigate these issues. We identify used undeclared dependencies and conflicting soft version constraints as two common smells that negatively affect these properties. We found that developers often manually resolve conflicting soft version constraints, and manual resolution is more common in projects that have many conflicting constraints. This implies that having conflicting soft version constraints risks introducing dependency-related problems that require developer effort to manually resolve. We developed an automated toolkit, MARCO, which serves as a proof-of-concept that the reliability of Maven's dependency resolution can be improved by changing how dependencies are declared without modifying the underlying resolution mechanism itself. MARCO increases transparency by injecting missing dependencies and balances stability and flexibility by replacing soft constraints with compatible version ranges. Modifying the dependency declarations in this manner could simplify the dependency management process for developers by only requiring them to specify one version, while the resolution considers all compatible versions. The evaluation of MARCO shows that the pre-computation of dependency-specific compatibility using bytecode differencing and cross-version testing is a promising method that can lower the client-side overhead of generating compatible version ranges. Clients can trust that the compatibility result is unlikely to contain breaking changes, and can be used as a solid basis for client-specific approaches to expand the pre-computed compatible ranges in a scalable manner and further increase reliability through increased flexibility. We hope these results can contribute to further discussions and future work towards more reliable dependency resolution.

Bibliography

[1] Apache Maven Project. Introduction to the Dependency Mechanism, 2002-2023. URL https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html. Accessed 2023-11-13.

[2] Apache Maven Project. dependency:analyze, October 2023. URL https://maven.apache.org/plugins/maven-dependency-plugin/analyze-mojo.html. Accessed 2024-05-20.

[3] Apache Maven Project. dependency:tree, October 2023. URL https://maven.apache.org/plugins/maven-dependency-plugin/analyze-mojo.html. Accessed 2024-05-20.

[4] Apache Maven Project. Apache Maven Dependency Plugin, October 2023. URL https://maven.apache.org/plugins/maven-dependency-plugin/. Accessed 2024-06-04.

[5] Apache Maven Project. Maven SCM Plugin - scm:tag, March 2023. URL https://maven.apache.org/scm/maven-scm-plugin/tag-mojo.html. Accessed 2024-01-20.

[6] Apache Maven Project. Maven SCM Plugin - Usage, April 2024. URL https://maven.apache.org/scm/maven-scm-plugin/usage.html. Accessed 2024-05-20.

[7] Apache Maven Project. Maven Surefire Plugin, January 2024. URL https://maven.apache.org/surefire/maven-surefire-plugin/index.html. Accessed 2024-06-07.

[8] Apache Maven Project. Version Range Specification, May 2024. URL https://maven.apache.org/enforcer/enforcer-rules/versionRanges.html. Accessed 2024-06-26.

[9] Apache Maven Project. Inclusions and Exclusions of Tests, January 2024. URL https://maven.apache.org/surefire/maven-surefire-plugin/examples/inclusion-exclusion.html. Accessed 2024-06-07.

[10] Apache Software Foundation. Maven 2.0 Design Documents: Dependency Mediation and Conflict Resolution, 2006. URL `https://cwiki.apache.org/confluence/display/MAVENOLD/Dependency+Mediation+and+Conflict+Resolution`. Accessed 2023-11-13.

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

[12] Uriel Chemouni. Release V 2.4.9, March 2023. URL `https://github.com/netplex/json-smart-v2/releases/tag/2.4.9`. Accessed 2024-06-02.

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

[14] codehaus-plexus. Release Plexus IO 3.3.1, May 2022. URL `https://github.com/codehaus-plexus/plexus-io/releases/tag/plexus-io-3.3.1`. Accessed 2024-06-02.

[15] Joe Darcy. Kinds of Compatibility, 2021. URL `https://wiki.openjdk.org/display/csr/Kinds+of+Compatibility`. Accessed 2023-11-13.

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

[17] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 313–324, 2014.

[18] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419, 2011.

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

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

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

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

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

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

[25] Martin Mois. japicmp, March 2024. URL `https://siom79.github.io/japicmp/`. Accessed 2024-06-02.

[26] Shaikh Mostafa, Rodney Rodriguez, and Xiaoyin Wang. Experience paper: a study on behavioral backward incompatibilities of java software libraries. In *Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis*, pages 215–225, 2017.

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

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

[29] Open Source Insights. Understand your dependencies. URL `https://deps.dev/`. Accessed 2024-06-02.

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

[31] Cathrine Paulsen. MaRCo reproduction package. Zenodo, June 2024. URL `https://doi.org/10.5281/zenodo.12625158`.

[32] Tom Preston-Werner. Semantic Versioning 2.0.0. URL `https://semver.org/`. Accessed 2024-06-03.

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

[34] Reproducible Builds. Reproducible Builds for Maven Central Repository. URL `https://github.com/jvm-repo-rebuild/reproducible-central`. Accessed 2024-06-03.

[35] revapi.org. Revapi, 2023. URL `https://revapi.org/`. Accessed 2024-06-02.

[36] Frank Reyes, Yogya Gamage, Gabriel Skoglund, Benoit Baudry, and Martin Monperrus. Bump: A benchmark of reproducible breaking dependency updates. *arXiv preprint arXiv:2401.09906*, 2024.

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

[38] Sonatype, Inc. Statistics for the Central Repository. URL `https://search.maven.org/stats`. Accessed 2024-01-20.

[39] Xiaohu Song, Ying Wang, Xiao Cheng, Guangtai Liang, Qianxiang Wang, and Zhiliang Zhu. 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*, pages 1–12, 2024.

[40] The Apache Software Foundation. Class ComparableVersion (documentation). URL `https://maven.apache.org/ref/3.5.2/maven-artifact/apidocs/org/apache/maven/artifact/versioning/ComparableVersion.html`. Accessed 2024-06-26.

[41] Plamen Totev. Symbolic links to directories are not recognized as directories, April 2022. URL `https://github.com/codehaus-plexus/plexus-io/issues/71`. Accessed 2024-06-02.

[42] Lina Maria Ochoa Venegas. Break the code?: Breaking changes and their impact on software evolution. [Phd Thesis 1 (Research TU/e / Graduation TU/e), Mathematics and Computer Science]. Eindhoven University of Technology., 2023.

[43] Ying Wang, Peng Sun, Lin Pei, Yue Yu, Chang Xu, Shing-Chi Cheung, Hai Yu, and Zhiliang Zhu. Plumber: Boosting the propagation of vulnerability fixes in the npm ecosystem. *IEEE Transactions on Software Engineering*, 2023.

[44] Wikipedia. Dependency hell, March 2024. URL `https://en.wikipedia.org/wiki/Dependency_hell`. Accessed 2024-06-03.

[45] Lyuye Zhang, Chengwei Liu, Zhengzi Xu, Sen Chen, Lingling Fan, Bihuan Chen, and Yang Liu. Has my release disobeyed semantic versioning? static detection based on semantic differencing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.

[46] Lyuye Zhang, Chengwei Liu, Sen Chen, Zhengzi Xu, Lingling Fan, Lida Zhao, Yiran Zhang, and Yang Liu. Mitigating persistence of open-source vulnerabilities in maven ecosystem. *arXiv preprint arXiv:2308.03419*, 2023.

[47] Chenguang Zhu, Mengshi Zhang, Xiuheng Wu, Xiufeng Xu, and Yi Li. Client-specific upgrade compatibility checking via knowledge-guided discovery. *ACM Transactions on Software Engineering and Methodology*, 32(4):1–31, 2023.

Appendix A

Glossary

This appendix gives an overview of frequently used terms and abbreviations.

GA: A dependency is identified in the Maven ecosystem by the combination of its `groupId` and `artifactId`, often abbreviated to GA.

GAV: A specific version of a dependency in the Maven ecosystem is identified by its unique artifact coordinate, or GAV, which is the combination of its `groupId`, `artifactId`, and `version` string [1].

MCR: The Maven Central Repository, the official public repository where Maven artifacts are published for public use [2].

POM: A Project Object Model, which is an XML file that contains a Maven project's dependency declarations and other information required by Maven to build it[3].

[1]`https://maven.apache.org/repositories/artifacts.html`
[2]`https://central.sonatype.com/`
[3]`https://maven.apache.org/guides/introduction/introduction-to-the-pom.html`