Interactive, Effort-Aware Library Version Harmonization

Kaifeng Huang
School of Computer Science and Shanghai Key Laboratory of Data Science
Fudan University
Shanghai, China

Bihuan Chen[*]
School of Computer Science and Shanghai Key Laboratory of Data Science
Fudan University
Shanghai, China

Bowen Shi
School of Computer Science and Shanghai Key Laboratory of Data Science
Fudan University
Shanghai, China

Ying Wang
School of Computer Science and Shanghai Key Laboratory of Data Science
Fudan University
Shanghai, China

Congying Xu
School of Computer Science and Shanghai Key Laboratory of Data Science
Fudan University
Shanghai, China

Xin Peng
School of Computer Science and Shanghai Key Laboratory of Data Science
Fudan University
Shanghai, China

ABSTRACT

As a mixed result of intensive dependency on third-party libraries, flexible mechanisms to declare dependencies and increased number of modules in a project, different modules of a project directly depend on multiple versions of the same third-party library. Such library version inconsistencies could increase dependency maintenance cost, or even lead to dependency conflicts when modules are inter-dependent. Although automated build tools (e.g., Maven's *enforcer* plugin) provide partial support to detect library version inconsistencies, they do not provide any support to harmonize inconsistent library versions.

We first conduct a survey with 131 Java developers from GitHub to retrieve first-hand information about the root causes, detection methods, reasons for fixing or not fixing, fixing strategies, fixing efforts, and tool expectations on library version inconsistencies. Then, based on the insights from our survey, we propose LibHarmo, an interactive, effort-aware library version harmonization technique, to detect library version inconsistencies, interactively suggest a harmonized version with the least harmonization efforts based on library API usage analysis, and refactor build configuration files.

LibHarmo is currently developed for Java Maven projects. Our experimental study on 443 highly-starred Java Maven projects from GitHub shows that i) LibHarmo detected 621 library version inconsistencies in 152 (34.3%) projects with a false positive rate of 16.8%, while Maven's *enforcer* plugin only detected 219 of them; and ii) LibHarmo saved 87.5% of the harmonization efforts. Further, 31 library version inconsistencies have been confirmed, and 17 of them have been already harmonized by developers.

[*]Bihuan Chen is the corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3409689

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**; **Software maintenance tools**.

KEYWORDS

Third-Party Libraries, Library Version Harmonization

ACM Reference Format:
Kaifeng Huang, Bihuan Chen, Bowen Shi, Ying Wang, Congying Xu, and Xin Peng. 2020. Interactive, Effort-Aware Library Version Harmonization. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages.
https://doi.org/10.1145/3368089.3409689

1 INTRODUCTION

With the increased diversity and complexity of modern systems, modular development [70] has become a common practice to encourage reuse, improve maintainability, and provide efficient ways for large teams of developers to collaborate [35]. Therefore, automated build tools (e.g., Maven) provide mechanisms (e.g., the *aggregation* mechanism in Maven [3]) to support multi-module projects for the ease of management and build. In contrast to the benefits that multi-module project brings to software development, one of the drawbacks is the complicated dependency management (colloquially termed as "dependency hell" [36]), exacerbated by the increased number of modules and the intensive dependency on third-party libraries. In this paper, we focus on the dependency management in Maven projects as Maven has dominated the build tool market for many years [60].

Problem. It is quite common that different modules of a project directly depend on the same third-party libraries. Maven provides flexible mechanisms for child modules to either inherit third-party library dependencies from parent modules (e.g., the *inheritance* mechanism [3]) or declare their own third-party library dependencies. Besides, Maven allows the version of a third-party library dependency to be explicitly hard-coded or implicitly referenced from a property which can be declared in parent modules. Therefore, *library version inconsistency* can be easily caused in practice; i.e., multiple versions of the same third-party library are directly depended on in different modules of a project. Even if the same version of a third-party library is directly depended on in different modules, the versions can be separately declared instead of referencing a common property. We refer to it as *library version false consistency* as it will turn into library version inconsistency when there is an incomplete library version update (e.g., a developer updates the version in one of the modules). Intuitively, library version inconsistency could increase dependency maintenance cost in the long run, or even lead to dependency conflicts [76] when modules are inter-dependent.

For example, an issue `HADOOP-6800` [1] was reported to the project `Apache Hadoop`, and said that "*multiple versions of the same library JAR are being pulled in .... Dependent subprojects use different versions. E.g. Common depends on Avro 1.3.2 while MapReduce depends on 1.3.0. Since MapReduce depends on Common, this has the potential to cause a problem at runtime*". This issue was prioritized as a blocker issue, and was resolved in 30 days. Developers found other library version inconsistencies, and finally harmonized the inconsistent versions of libraries `avro`, `commons-logging`, `commons-logging-api` and `jets3t` across modules `Common`, `MapReduce` and `HDFS`.

Maven's *enforcer* plugin uses a *dependency convergence* rule to detect multiple versions of the same third-party library along the transitive dependency graph; i.e., if a module has two dependencies `A` and `B`, and both depend on the same dependency, `C`, this rule will fail the build if `A` depends on a different version of `C` than the version of `C` depended on by `B`. In that sense, this rule cannot detect library version inconsistencies across modules that are not inter-dependent. Moreover, it does not provide any support on how to harmonize inconsistent library versions. As project developers have no direct control to harmonize the inconsistent library versions in transitive dependencies, we only consider direct dependencies across modules.

Approach. To better address the problem, e.g., by realizing practical solutions that are acceptable by developers, it is important to first understand developers' practices on library version inconsistencies. Therefore, we conduct a survey with 131 Java developers from GitHub to retrieve first-hand information about the root causes, detection methods, reasons for fixing or not fixing, fixing strategies, fixing efforts, and tool expectations on library version inconsistencies. 90.8% of participants experienced library version inconsistency, and 69.4% consider it as a problem in project maintenance. Our survey suggests several insights, e.g., tools are needed to proactively locate and harmonize inconsistent library versions, and such tools need to interact with developers and provide API-level harmonization efforts.

Then, inspired by the insights from our developer survey, we propose LibHarmo, the first interactive, effort-aware technique to harmonize inconsistent library versions in Java Maven projects. It works in three steps. First, it identifies library version inconsistencies by analyzing build configuration files (i.e., POM files). Second, for each library version inconsistency, it suggests a harmonized version with the least harmonization efforts (e.g., the number of calls to library APIs that are deleted and changed in the harmonized version) based on library API usage analysis and interaction with developers. Finally, if developers determine to harmonize, it refactors POM files.

We have evaluated LibHarmo on 443 highly-starred Java Maven projects from GitHub. Our experimental results have indicated that LibHarmo detected 621 library version inconsistencies in 152 (34.3%) projects, but Maven's *Enforcer* plugin only detected 219 of them. We sampled 238 library version inconsistencies with a confidence level of 95% and a margin of error of 5% for manual analysis and identified 40 (16.8%) false positives. Furthermore, LibHarmo saved 87.5% of the harmonization efforts through identifying 3 of the 24 library APIs called in projects as deleted/changed in the suggested harmonized version. Further, 31 library version inconsistencies have been confirmed, and 17 of them have been harmonized by developers.

Table 1: Survey Questions

Q1	How many years of Java programming experience do you have?
Q2	How many modules in a Java project did you participate in?
Q3	Have you ever encountered library version inconsistency?
Q4	Is library version inconsistency a problem in project maintenance?
Q5	What are the root causes of library version inconsistencies?
Q6	How did you detect library version inconsistencies?
Q7	What are the reasons of not fixing library version inconsistencies?
Q8	What are the reasons of fixing library version inconsistencies?
Q9	Which version do you use as the harmonized version to fix library version inconsistencies?
Q10	How do you fix library version inconsistencies?
Q11	How much time do you spend in fixing library version inconsistencies?
Q12	Which part of it is most time-consuming in fixing library version inconsistencies?
Q13	Is an automatic library version harmonization tool useful for library management?
Q14	Which features would be useful for an automatic library version harmonization tool?

Contributions. This paper makes the following contributions.

- We conducted a survey with 131 Java developers from GitHub to retrieve first-hand information about the practices and tool expectations on library version inconsistencies.
- We proposed the first interactive, effort-aware library version harmonization technique, LibHarmo, based on our survey insights.
- We evaluated LibHarmo on 443 Java Maven projects from GitHub, and found 621 library version inconsistencies. 31 of them have been confirmed with 17 being harmonized.

2 DEVELOPER SURVEY

Our online survey is designed for developers who participated in the development of Java Maven multi-module projects. Therefore, we selected Java Maven multi-module projects from GitHub. Then, we excluded projects whose number of stars was less than 200 in order to ensure project quality. Finally, we had 443 projects. We manually categorized the domain of 443 projects, and computed their lines of code, number of commits and number of stars. The projects spanned over 58 domains, and averagely had 178,852 lines of code, 3,923 commits, and 2,347 stars. From them, we collected 5,316 developers whose email address on their GitHub profile page was valid. We sent an email to each of the developers to clarify the library version inconsistency problem and kindly ask them to participate in our online questionnaire survey. The questions are listed in Table 1, and a complete questionnaire with options is available at our website [5]. We promised developers that their participation would remain confidential, and all reporting would be based on aggregated responses.

Our survey consists of 14 questions, covering the following seven aspects, to learn about their professional background, practices and tool expectations on library version inconsistencies. The answers to

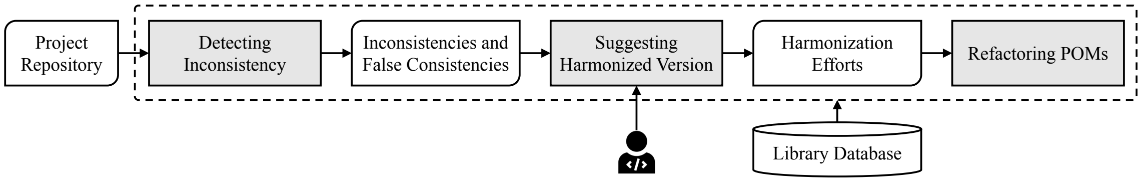


Figure 1: An Overview of LibHarmo

open questions were categorized by three authors separately; and a group discussion was conducted to reach consensus.

Professional Background (Q1–Q4). In response to the invitation emails, 131 developers finished the questionnaire within seven days (i.e., a participation rate of 2.5%). Of all participants, 44.3% have more than 10 years of Java programming experience, 25.2% have 5 to 10 years, and 30.5% have less than 5 years. 47.3% participated in the development of over 10 modules in one project, 23.7% participated in 5 to 10 modules, and 29.0% participated in less than 5 modules. 90.8% of participants experienced library version inconsistency, and 69.4% consider it as a problem in project maintenance. We can observe that the participants have relatively good experience in modular development as well as in handling library version inconsistencies.

Root Causes (Q5). 67.1% and 65.8% named unawareness of the same library in other modules and backward incompatibility issues in library versions as the major root causes of library version inconsistencies. Different development schedule among different modules (46.1%), unawareness of the library version inconsistency problem (31.6%), and not considering library version inconsistency as a problem (23.7%) are the secondary root causes. Some minor root causes (14.5%) include bad dependency management hygiene, unawareness of new library versions, and usage difficulty with Maven.

Detection Methods (Q6). Being asked about the detection or manifestation of library version inconsistencies, bugs due to conflicting library versions [76] (72.4%) is the main way to manifest, followed by bugs due to library API behavior changes (47.4%). Manual investigation of module POM files (46.1%) is the main way to detect, followed by communication with developers of other modules (14.5%) and adoption of Maven's *enforcer* plugin (10.5%).

Reasons for Fixing or not Fixing (Q7–Q8). The participants reported four main reasons for not fixing: heavy fixing efforts due to backward incompatibility issues (45.3%), heavy fixing efforts due to intensive library API dependency (38.7%), fixing difficulty due to different development schedule in different modules (36.0%), and no serious consequence occurred (30.7%). 6.6% emphasized that they always selected to fix. On the other hand, there are three main reasons for fixing: avoiding great maintenance efforts in the long run (68.4%), ensuring consistent library API behaviors across modules (63.2%), and serious consequences occurred (e.g., bugs) (55.3%).

Fixing Strategies (Q9–Q10). When harmonizing the inconsistent library versions, 77.6% used one of the newer versions than all currently declared versions with the least harmonization efforts, but 29.0% chose one of the currently declared versions with the least harmonization efforts. Besides, 61.8% harmonized the versions in all of the affected modules, while 38.2% only harmonized the versions in some of the affected modules.

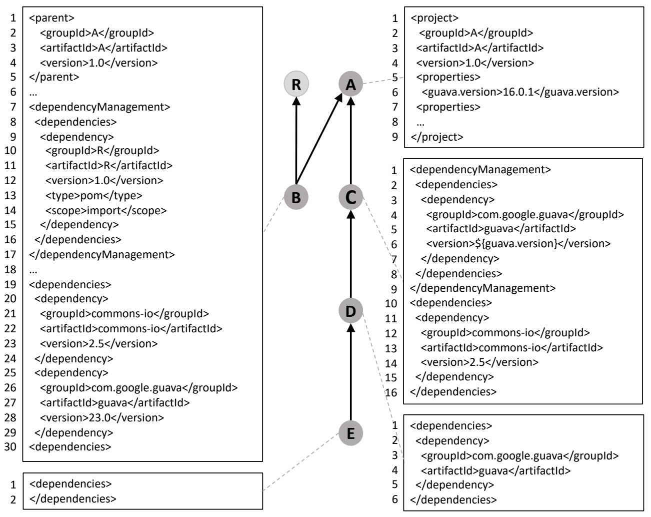
Fixing Efforts (Q11–Q12). 50.0% spent hours in fixing library version inconsistencies, 32.9% even spent days, and only 11.8% spent minutes. Besides, locating all inconsistent library versions (56.7%), determining the harmonized version (49.3%), and refactoring the source code (48.0%) are the most time-consuming steps in fixing. Other time-consuming steps are refactoring the POM files (32.0%) and verifying the fix through regression testing (6.7%).

Tool Expectations (Q13–Q14). 45.6% thought an automated library version harmonization tool would be useful, but 14.0% thought it would not be useful mostly because they already adopted Maven's *enforcer* plugin. 46.5% thought it depended on how well it would be integrated into the build process, how automated it would be, etc. With respect to the most useful feature in such a tool, detecting all library version inconsistencies (75.9%) and suggesting the harmonized version (71.4%) are the most useful ones, followed by reporting detailed API-level fixing efforts (49.1%) and refactoring the POM files (42.0%). Surprisingly, refactoring the source code (25.0%) is less useful than all the previous features.

Insights. From our survey findings, we have several insights. **I1:** tools are needed to help developers proactively locate and harmonize inconsistent library versions, as library version inconsistencies are mostly manually detected, or passively found after serious consequences. **I2:** developers should interact with such tools to determine where and whether to harmonize, as library version inconsistencies span multiple modules that have different development schedule, and might not be fixed due to heavy harmonization efforts. **I3:** such tools need to provide developers with API-level harmonization efforts, as API backward incompatibility, API dependency intensity, and API behavior consistency are key factors for developers to determine whether to harmonize. **I4:** such tools need to be integrated into the build process for the ease of adoption.

3 APPROACH

Based on the insights **I1**, **I2** and **I3** from our developer survey, we propose the first interactive, effort-aware technique, named LibHarmo, to assist developers in harmonizing inconsistent library versions (and falsely consistent library versions). As shown in Fig. 1, it takes as input a Java Maven project repository, and works in three steps. First, it detects inconsistency (Sec. 3.1). Here the challenge is to statically construct the inheritance relationships among POMs from different modules and resolve the version of library dependencies declared in these POMs. This step realizes **I1**. Second, it suggests harmonized version (Sec. 3.2) with harmonization efforts by interacting with developers. Here the goal is to distinguish whether the library APIs called in the project are deleted/changed in the suggested harmonized version so that developers can confidently determine where and whether to harmonize while saving their harmonization efforts. This step fulfills **I2** and **I3**. Finally, if developers decide to harmonize, it refactors POMs (Sec. 3.3). This step achieves **I1**. In fact, the source code also needs to be adapted to the harmonized version. We leave it to developers because i) empirical studies [18, 81] have reported that existing library



```
1  <parent>
2    <groupId>A</groupId>
3    <artifactId>A</artifactId>
4    <version>1.0</version>
5  </parent>
6  …
7  <dependencyManagement>
8   <dependencies>
9    <dependency>
10    <groupId>R</groupId>
11    <artifactId>R</artifactId>
12    <version>1.0</version>
13    <type>pom</type>
14    <scope>import</scope>
15   </dependency>
16  </dependencies>
17 </dependencyManagement>
18 …
19 <dependencies>
20  <dependency>
21   <groupId>commons-io</groupId>
22   <artifactId>commons-io</artifactId>
23   <version>2.5</version>
24  </dependency>
25  <dependency>
26   <groupId>com.google.guava</groupId>
27   <artifactId>guava</artifactId>
28   <version>23.0</version>
29  </dependency>
30 <dependencies>
```

```
1  <dependencies>
2  </dependencies>
```

```
1  <project>
2    <groupId>A</groupId>
3   <artifactId>A</artifactId>
4   <version>1.0</version>
5   <properties>
6    <guava.version>16.0.1</guava.version>
7   <properties>
8   …
9  </project>
```

```
1  <dependencyManagement>
2   <dependencies>
3    <dependency>
4     <groupId>com.google.guava</groupId>
5     <artifactId>guava</artifactId>
6     <version>${guava.version}</version>
7    </dependency>
8   </dependencies>
9  </dependencyManagement>
10 <dependencies>
11  <dependency>
12   <groupId>commons-io</groupId>
13   <artifactId>commons-io</artifactId>
14   <version>2.5</version>
15  </dependency>
16 </dependencies>
```

```
1  <dependencies>
2   <dependency>
3    <groupId>com.google.guava</groupId>
4    <artifactId>guava</artifactId>
5   </dependency>
6  </dependencies>
```

Figure 2: An Example of POM Inheritance Graph

API adaptation techniques [7, 17, 20, 21, 28, 32, 59, 69, 79, 84] have an average accuracy of 20% and ii) our survey indicates that refactoring the source code is surprisingly a less useful feature (potentially due to the low accuracy of automated techniques). Besides, LibHarmo relies on a library database (Sec. 3.4) to provide JAR files of candidate library versions for harmonization. At the current stage, LibHarmo is implemented as a prototype to demonstrate its value, and hence it is not integrated into the build process yet and does not satisfy **I4**.

3.1 Detecting Inconsistency

The first step of LibHarmo consists of three sub-steps: it generates a POM inheritance graph, analyzes the POM inheritance graph to resolve the version of library dependencies in each POM, and identifies library version inconsistencies and false consistencies.

Generating POM Inheritance Graph. Maven provides the *inheritance* mechanism [3] to inherit elements (e.g., dependencies) from a parent POM. It does not support multiple inheritance, however, it indirectly supports the concept by using the *import* scope [2]. Maven also does not allow cyclic inheritance. Therefore, the inheritance relations among POMs in a project form a directed acyclic graph. We define such a POM inheritance graph $\mathcal{G}$ as a 2-tuple $\langle \mathcal{M}, \mathcal{E} \rangle$, where $\mathcal{M}$ denotes all the POMs in a project, and $\mathcal{E}$ denotes the inheritance relations among the POMs in $\mathcal{M}$. Each inheritance relation $e \in \mathcal{E}$ is denoted as a 2-tuple $\langle m_1, m_2 \rangle$, where $m_1, m_2 \in \mathcal{M}$, and m_1 inherits m_2 (i.e., m_2 is the parent POM of m_1).

To construct $\mathcal{G}$ of a project, LibHarmo scans its repository recursively to collect all the local POMs and add them into $\mathcal{M}$. Then, for each POM m in $\mathcal{M}$, LibHarmo parses it to locate its parent POMs based on the *inheritance* mechanism and the *import* scope; i.e., LibHarmo parses the `parent` section (the *inheritance* mechanism) and the `dependencyManagement` section (the *import* scope). For each located parent POM m', an inheritance relation $e = \langle m, m' \rangle$ is generated and added into $\mathcal{E}$. As m' can be a remote POM, LibHarmo crawls it from Maven repository, and adds it into $\mathcal{M}$. $\mathcal{E}$ is constructed after all the local and remote POMs in $\mathcal{M}$ are parsed.

Example 3.1. Fig. 2 presents a generated POM inheritance graph, where the nodes represent POMs, the arrows represent inheritance relations, and the dotted lines link to excerpts from POMs. Here `A`, `B`, `C`, `D` and `E` are local POMs, and `R` is a remote POM. `B` has two parent POMs, `A` and `R`. In particular, `B` inherits `A` by declaring the `groudId`, `artifactId` and `version` of `A` in the `parent` section (Line 1–5 in `B`). `B` inherits `R` by declaring the `groudId`, `artifactId` and `version` of `R` in a dependency with `type` being *pom* and `scope` being *import* in the `dependencyManagement` section (Line 7–17 in `B`).

Resolving Library Dependencies. We first introduce Maven's dependency declaration mechanisms before diving into the details. The `dependencies` section contains the library dependencies that a POM declares to use, and such library dependencies will be automatically inherited by child POMs, whereas the `dependencyManagement` section contains the library dependencies that a POM declares to manage, and such library dependencies will be used/inherited only when they are explicitly declared in the `dependencies` section with their version not specified. Moreover, the version of a library dependency can be explicitly declared by a hard-coded value or implicitly declared via referencing a property. A property can be overwritten by declaring the same property with a different value.

Example 3.2. In Fig. 2, `B` declares two library dependencies `B` wants to use, and the versions are hard-coded (Line 20–29 in `B`). `C` declares one library dependency `C` wants to use (Line 10–16 in `C`); and `C` also declares one library dependency `C` wants to manage (Line 1–9 in `C`), and the version references a property, `guava.version`, which is declared in Line 5–7 in `A`. `D` automatically inherits the library dependency in Line 10–16 in `C`; and `D` also inherits the managed library dependency in Line 1–9 in `C` by explicitly declaring it in Line 1–6 in `D`. `E` inherits from `D` the two library dependencies `D` inherits from `C`.

Based on the dependency declaration mechanisms, all the library dependencies of a POM can be resolved based on the resolved library dependencies of its ancestor POMs. To ease the detection and harmonization of inconsistencies and false consistencies, we first define a library dependency d as a 6-tuple $\langle lib, ver, pro, m_{lib}, m_{ver}, m_{pro} \rangle$, where lib denotes a library, uniquely identified by its `groupId` (i.e., the organization lib belongs to) and `artifactId` (i.e., the name of lib); ver denotes the resolved version number of lib; pro denotes the property that the version of lib references, and it will be `null` when the version of lib is hard-coded; m_{lib} denotes the POM that owns lib either by declaration or inheritance; m_{ver} denotes the POM that declares the version of lib; and m_{pro} denotes the POM that declares pro.

For each POM m in $\mathcal{M}$, we resolve m's library dependencies $\mathcal{D}_m$ that are declared in m or inherited from ancestors of m. To this end, LibHarmo performs a breadth-first search on $\mathcal{G}$ to visit m and m's ancestors while following Maven's *nearest definition wins* and *first declaration wins* strategy [2]. For each visited POM, we parse each library dependency in the `dependencies` section to create a d and add d to $\mathcal{D}_m$, and analyze the `properties` and `dependencyManagement` section to resolve the unresolved version of library dependencies in $\mathcal{D}_m$. Here version range is supported by finding the highest version from our library database that satisfies the version range specification. Finally, we get all library dependencies $\mathcal{D} = \bigcup_{m \in \mathcal{M}} \mathcal{D}_m$.

Example 3.3. Table 2 presents the the process of resolving library dependencies for `E` in Fig. 2 along its inheritance hierarchy. At `E`, as `E` does not declare any library dependency, no library dependency is created. Next, at `E`'s parent `D`, `guava` is declared but its version is not

Table 2: An Example of Resolving Library Dependencies

E		
D	<guava, , , E, , >	
C	<guava, , guava.version, E, C, >	<commons-io, 2.5, null, E, C, null>
A	<guava, 16.0.1, guava.version, E, C, A>	<commons-io, 2.5, null, E, C, null>

declared. Hence, d_1 is created with lib and m_{lib} set to guava and E. Next, at D's parent C, d_1's version is declared by referencing a property. Thus, d_1's pro and m_{ver} is set to `guava.version` and C. Meanwhile, C declares `commons-io` and hard-codes its version. Thus, d_2 is created as ⟨`commons-io`, `2.5`, `null`, E, C, `null`⟩. Finally, at C's parent A, the property `guava.version` is declared, and thus d_1's ver and m_{pro} is set to `16.0.1` and A. E owns library dependencies d_1 and d_2.

Identifying Inconsistencies and False Consistencies. As we do not have direct control over remote POMs, we remove from $\mathcal{D}$ the library dependencies whose m_{lib} is a remote POM. However, it is possible that the library dependencies of local POMs are inherited from remote POMs. To detect library version inconsistencies and false consistencies, we first identify the libraries $\mathcal{L}$ from $\mathcal{D}$, i.e., $\mathcal{L} = \{d.lib \mid d \in \mathcal{D}\}$. Then, for each $lib \in \mathcal{L}$, we find all the library dependencies $\mathcal{D}_{lib} = \{d \mid d \in \mathcal{D} \wedge d.lib = lib\}$. Finally, we determine the consistency of $\mathcal{D}_{lib}$ by classifying it into the following four types.

- *Inconsistency (IC).* $\mathcal{D}_{lib}$ belongs to the type of *inconsistency* if the library dependencies in $\mathcal{D}_{lib}$ do not have the same version; i.e., $\mathcal{D}_{lib}$ satisfies that $\exists d_1, d_2 \in \mathcal{D}_{lib}, d_1.ver \neq d_2.ver$.
- *True Consistency (TC).* $\mathcal{D}_{lib}$ belongs to the type of *true consistency* if all the library dependencies in $\mathcal{D}_{lib}$ have the same version by referencing one property or inheriting from one POM; i.e., $\mathcal{D}_{lib}$ satisfies that $\forall d_1, d_2 \in \mathcal{D}_{lib}, (d_1.pro \neq null \wedge d_1.pro = d_2.pro \wedge d_1.m_{pro} = d_2.m_{pro}) \vee (d_1.m_{ver} = d_2.m_{ver})$.
- *False Consistency (FC).* $\mathcal{D}_{lib}$ belongs to the type of *false consistency* if all the library dependencies in $\mathcal{D}_{lib}$ have the same version but do not reference one property and do not inherit from one POM; i.e., $\mathcal{D}_{lib}$ satisfies that $\forall d_1, d_2 \in \mathcal{D}_{lib}, d_1.ver = d_2.ver \wedge \exists d_1, d_2 \in \mathcal{D}_{lib}, d_1.pro \neq d_2.pro \vee d_1.m_{pro} \neq d_2.m_{pro} \vee d_1.m_{ver} \neq d_2.m_{ver}$.
- *Single Library (SL).* $\mathcal{D}_{lib}$ belongs to the type of *single library* if there is only one library dependency in $\mathcal{D}_{lib}$ (i.e., $|\mathcal{D}_{lib}| = 1$).

Example 3.4. Fig. 3 presents all the resolved library dependencies of Fig. 2, which involve two libraries guava and `commons-io`. Hence, we have $\mathcal{D}_{guava} = \{d_1, d_3, d_7\}$ and $\mathcal{D}_{commons-io} = \{d_2, d_4, d_5, d_6\}$. $\mathcal{D}_{guava}$ belongs to *IC* as two different versions of guava are used, and $\mathcal{D}_{commons-io}$ belongs to *FC* because the version of `commons-io` is hard-coded in two POMs.

3.2 Suggesting Harmonized Version

On the one hand, for a false consistency $\mathcal{D}_{lib}$, the same version is already adopted in the library dependencies in $\mathcal{D}_{lib}$, but it will turn into an inconsistency if there is an incomplete library version update (e.g., a developer updates the version of some but not all of the library dependencies in $\mathcal{D}_{lib}$). In that sense, it also needs to be harmonized to become a true consistency. To reduce the harmonization efforts, we directly recommend the currently used version as the harmonized version. On the other hand, for an inconsistency $\mathcal{D}_{lib}$, we first analyze the harmonization efforts at the library API level, and then interactively suggest a harmonized version with the least efforts.

Analyzing Harmonization Efforts. If developers try to manually harmonize $\mathcal{D}_{lib}$ to a new version, they must determine whether

d_1 = <guava, 16.0.1, guava.version, E, C, A>
d_2 = <commons-io, 2.5, null, E, C, null>

d_3 = <guava, 16.0.1, guava.version, D, C, A>
d_4 = <commons-io, 2.5, null, D, C, null>

d_5 = <commons-io, 2.5, null, C, C, null>

d_6 = <commons-io, 2.5, null, B, B, null>
d_7 = <guava, 23.0, null, B, B, null>

Figure 3: The Resolved Library Dependencies of Fig. 2

each called library API in each library dependency in $\mathcal{D}_{lib}$ is deleted (i.e., its API signature does not exist, which will fail the compilation) or changed (i.e., its API signature is not changed, but its behavior is changed, which will pass the compilation) in the new version. While deleted APIs can be easily caught during compilation, changed APIs can be easily missed but may cause API breaking. Hence, we define harmonization efforts as the number of called library APIs that are deleted/changed in the harmonized version and the number of calls to library APIs that are deleted/changed in the harmonized version.

To this end, for each $d \in \mathcal{D}_{lib}$, LibHarmo runs JavaParser [71] on the `src` folder that has the same prefix path to $d.m_{lib}$, with JAR files from our library database (see Sec. 3.4), to locate API calls to d. Thus, we have a set of called library APIs $\mathcal{A}_d$ and a set of library API calls C_d. Then, LibHarmo determines the candidate library versions $\mathcal{V}_d$ for harmonization from our library database which contains all the released versions of $d.lib$. Here, we compute $\mathcal{V}_d$ as the versions that are no older than the highest version in $\mathcal{D}_{lib}$ as developers tend to use newer versions but not always the newest version, as suggested by our survey. Next, for each candidate version $v \in \mathcal{V}_d$, LibHarmo locates the called library APIs in d that are deleted or changed in v. Here, a library API is deleted in v if there is no library API with the same fully qualified name in v. A library API is changed in v if its fully qualified name is not changed but the body code of the library API or the code of its transitively called methods in its static call graph is changed. LibHarmo uses Soot [74] to extract the static call graph. Hence, we decompose $\mathcal{A}_d$ into three sets $\mathcal{AD}_d^v$, $\mathcal{AC}_d^v$ and $\mathcal{AU}_d^v$, respectively representing the called library APIs in d that are deleted, changed and unchanged in v. Correspondingly, we can decompose C_d into three sets $\mathcal{CD}_d^v$, $\mathcal{CC}_d^v$ and $\mathcal{CU}_d^v$, respectively representing the calls to the library APIs in $\mathcal{AD}_d^v$, $\mathcal{AC}_d^v$ and $\mathcal{AU}_d^v$ (i.e., the calls to the deleted, changed and unchanged library APIs). Therefore, the efforts f_d^v to harmonize d to the version v can be characterized as a 6-tuple, i.e., $f_d^v = \langle \mathcal{AD}_d^v, \mathcal{AC}_d^v, \mathcal{AU}_d^v, \mathcal{CD}_d^v, \mathcal{CC}_d^v, \mathcal{CU}_d^v \rangle$. Notice that $\mathcal{AU}_d^v$ and $\mathcal{CU}_d^v$ actually represent the efforts that are saved for developers by LibHarmo because developers do not need to waste time on these called but unchanged library APIs.

Example 3.5. We use one of the detected inconsistencies in the popular project Apache Tika to demonstrate our harmonization effort analysis. This inconsistency is about library `commons-cli`, involving three modules: `tika-server`, `tika-batch` and `tika-eval`. The first module explicitly declares `commons-cli` with version 1.2. The latter two reference two properties with the same property name `cli.version` declared in their own POM file, and both of them declare version 1.4. Version 1.4 is one of the candidate version. It turns out that the number of called library APIs in version 1.2 in `tika-server` is 5, and there are 33 calls to the 5 library APIs. The 5 library APIs are all changed in version 1.4. Therefore, $|\mathcal{AD}_d^v| = 0$, $|\mathcal{AC}_d^v| = 5$, $|\mathcal{AU}_d^v| = 0$, $|\mathcal{CD}_d^v| = 0$, $|\mathcal{CC}_d^v| = 33$, and $|\mathcal{CU}_d^v| = 0$.

Interactively Recommending Harmonized Version. As revealed by our survey (see Sec. 2), developers may choose to not harmonize all inconsistent library dependencies due to various reasons (e.g., different development schedule, or heavy efforts due to API dependency intensity or backward incompatibility). Thus, LibHarmo is designed to interact with developers such that 1) developers are provided with detailed library API-level harmonization efforts f_d^v for each library dependency $d \in \mathcal{D}_{lib}$ to be harmonized into each candidate version $v \in \mathcal{V}_d$; 2) developers have the flexibility to decide which of the library dependencies $\mathcal{D}'_{lib} \subseteq \mathcal{D}_{lib}$ need to be harmonized; and 3) developers are provided with a ranked list of candidate versions based on flexible combinations of $\mathcal{AD}_d^v$, $\mathcal{AC}_d^v$, $\mathcal{AU}_d^v$, $\mathcal{CD}_d^v$, $\mathcal{CC}_d^v$ and $\mathcal{CU}_d^v$ (e.g., the default ranking is based on the summation of $|\mathcal{CD}_d^v|$ and $|\mathcal{CC}_d^v|$ over all library dependencies in $\mathcal{D}'_{lib}$) such that they can choose the harmonized version v_h with the least harmonization efforts they consider acceptable.

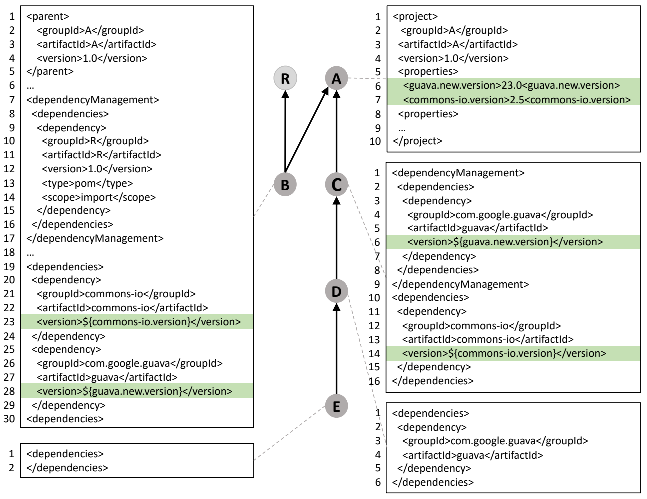
To ease the determination of $\mathcal{D}'_{lib}$, we first decompose $\mathcal{D}_{lib}$ according to $d.m_{ver}$; i.e., the library dependencies that have their version declared in the same POM m_{ver} are grouped into $\mathcal{D}_{lib}^{m_{ver}}$. $\mathcal{D}_{lib}^{m_{ver}}$ actually belongs to the type of true consistency (or single library), and should be harmonized together to still keep the consistency. For example, $\mathcal{D}_{guava}$ in Example 3.4 can be decomposed into $\mathcal{D}_{guava}^{C} = \{d_1, d_3\}$ and $\mathcal{D}_{guava}^{B} = \{d_7\}$. Based on the decomposition, we allow developers to determine which groups need to be harmonized.

3.3 Refactoring POMs

The last step of LibHarmo is to carry out the harmonization on POMs. LibHarmo can automatically refactor POMs based on the library dependencies $\mathcal{D}'_{lib}$ that developers choose from an inconsistency $\mathcal{D}_{lib}$ and the harmonized version v_h that developers choose. The POM refactoring is exactly the same for false consistencies.

The goal of our harmonization is to make $\mathcal{D}'_{lib}$ become a true consistency; i.e., all the library dependencies in $\mathcal{D}'_{lib}$ need to have their version reference a property of value v_h. To this end, LibHarmo first locates the POMs $\mathcal{M}'$ that declare the version of the library dependencies in $\mathcal{D}'_{lib}$; i.e., $\mathcal{M}' = \{d.m_{ver} \mid d \in \mathcal{D}'_{lib}\}$. On one hand, the lowest common ancestor of the POMs in $\mathcal{M}'$ on the POM inheritance graph $\mathcal{G}$ is the POM where LibHarmo newly declares a property of value v_h. On the other hand, $\mathcal{M}'$ contains the POMs where LibHarmo changes the (implicit or explicit) version declaration of lib to a reference to the newly declared property. Occasionally, the lowest common ancestor could be a remote POM that we do not have direct control, or $\mathcal{G}$ contains several sub-graphs that are not connected. Thus, LibHarmo finds several lowest common ancestors, each of which is the lowest common ancestor of some POMs in $\mathcal{M}'$, and separately applies the same refactoring process.

Finally, LibHarmo checks whether the properties that are referenced in $\mathcal{D}'_{lib}$ are referenced by the other library dependencies in $\mathcal{D}$. If not, such properties become unused after our refactoring and can be deleted. Specifically, for each library dependency $d \in \mathcal{D}'_{lib}$ that declares the version by referencing a property, LibHarmo extracts a 2-tuple $\langle d.pro, d.m_{pro}\rangle$, and checks whether there exists a library dependency d' in $\mathcal{D} - \mathcal{D}'_{lib}$ such that $d.pro = d'.pro \wedge d.m_{pro} = d'.m_{pro}$. If exists, $d.pro$ is still referenced by other library dependencies, and is kept; otherwise, LibHarmo deletes $d.pro$ from $d.m_{pro}$.



```
1  <parent>
2    <groupId>A</groupId>
3    <artifactId>A</artifactId>
4    <version>1.0</version>
5  </parent>
6  …
7  <dependencyManagement>
8   <dependencies>
9    <dependency>
10     <groupId>R</groupId>
11     <artifactId>R</artifactId>
12     <version>1.0</version>
13     <type>pom</type>
14     <scope>import</scope>
15    </dependency>
16   </dependencies>
17 </dependencyManagement>
18 …
19 <dependencies>
20  <dependency>
21    <groupId>commons-io</groupId>
22    <artifactId>commons-io</artifactId>
23    <version>${commons-io.version}</version>
24  </dependency>
25  <dependency>
26    <groupId>com.google.guava</groupId>
27    <artifactId>guava</artifactId>
28    <version>${guava.new.version}</version>
29  </dependency>
30 <dependencies>
```

```
1  <dependencies>
2  </dependencies>
```

```
1  <project>
2    <groupId>A</groupId>
3   <artifactId>A</artifactId>
4   <version>1.0</version>
5   <properties>
6    <guava.new.version>23.0<guava.new.version>
7    <commons-io.version>2.5<commons-io.version>
8   <properties>
9   …
10 </project>
```

```
1  <dependencyManagement>
2   <dependencies>
3    <dependency>
4     <groupId>com.google.guava</groupId>
5     <artifactId>guava</artifactId>
6     <version>${guava.new.version}</version>
7    </dependency>
8   </dependencies>
9  </dependencyManagement>
10 <dependencies>
11  <dependency>
12   <groupId>commons-io</groupId>
13   <artifactId>commons-io</artifactId>
14   <version>${commons-io.version}</version>
15  </dependency>
16 </dependencies>
```

```
1  <dependencies>
2   <dependency>
3    <groupId>com.google.guava</groupId>
4    <artifactId>guava</artifactId>
5   </dependency>
6  </dependencies>
```

Figure 4: An Example of Refactoring POMs

Example 3.6. Given $\mathcal{D}_{guava} = \{d_1, d_3, d_7\}$ in Example 3.4 and the harmonized version `23.0`, $\mathcal{M}'$ is computed as {`B`, `C`}, and their lowest common ancestor is `A`. Hence, a property `guava.new.version` is declared at Line 6 in `A` in Fig. 4, and `B` and `C` respectively change the version declaration at Line 28 in `B` and at Line 6 in `C` to reference the property `guava.new.version`. Moreover, as the previous property `guava.version` is not referenced by other library dependencies, it is deleted from `A`. Similarly, for the false consistency $\mathcal{D}_{commons-io} = \{d_2, d_4, d_5, d_6\}$ in Example 3.4, $\mathcal{M}'$ is computed as {`B`, `C`}. Hence, a property `commons-io.version` is declared in `A`, the lowest common ancestor of `B` and `C`; and the version declaration at Line 23 in `B` and at Line 14 in `C` is changed to reference `commons-io.version`.

3.4 Library Database

Recall that our harmonization efforts analysis (see Sec. 3.2) requests from the library database the JAR files of a library version and some newer releases of the same library. Therefore, LibHarmo crawls the JAR files of all releases of a library in a demand-driven way from Maven repository. Besides, LibHarmo regularly updates any new library releases for the libraries in our library database. Currently, our library database has 6,007 libraries and 12,595 library releases.

4 EVALUATION

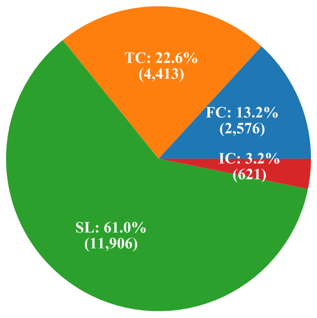
We have implemented a prototype of LibHarmo in Java and Python in a total of 14.6K lines of code. We have released the source code at our website [5]. In this section, we report our evaluation results.

4.1 Evaluation Design

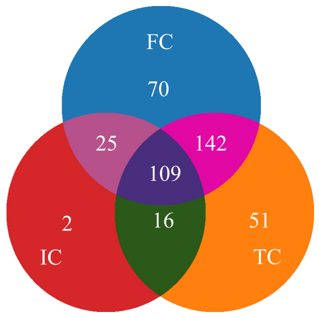
We used the same set of 443 Java Maven multi-module projects used in our survey as the dataset for our evaluation. We designed our evaluation to answer the following four research questions.

RQ1: Can LibHarmo effectively detect library version inconsistencies and false consistencies? (Sec. 4.2)

RQ2: What is the prevalence and severity of the detected library version inconsistencies and false consistencies? (Sec. 4.3 and 4.4)



(a) Distribution



(b) Intersection

Figure 5: Overall Distribution of IC, FC, TC and SL

RQ3: Can LibHarmo help to save efforts of harmonizing library version inconsistencies for developers? (Sec. 4.5)
RQ4: What is developers' feedback about LibHarmo? (Sec. 4.6)

Specifically, **RQ1** is designed to show LibHarmo's capability in detecting inconsistencies and false consistencies. To this end, we compare LibHarmo with Maven's *enforcer* plugin, and also manually analyze LibHarmo's false positive rate by statistical sampling. **RQ2** is designed to raise attention from the community to the problem of inconsistencies and false consistencies. **RQ3** is designed to report the degree of manual efforts saved by using LibHarmo. **RQ4** is designed to report the usage feedback about LibHarmo.

To this end, we ran LibHarmo against 433 projects to 1) detect all inconsistency (IC), false consistency (FC), true consistency (TC) and single library (SL) (see Sec. 3.1 for definitions), which is used to answer **RQ1** and **RQ2**, 2) analyze the harmonization efforts for each inconsistency for each candidate harmonized version, which is used to answer **RQ3**, and 3) generate a report including the previous two sets of information and send to developers, which is used to answer **RQ4**.

4.2 Effectiveness Evaluation (RQ1)

LibHarmo detected 621 ICs and 2,576 FCs, respectively affecting 152 and 346 projects. Differently, Maven's *enforcer* plugin only detected 219 of the 621 ICs. The reason is that *enforcer* only focuses on a subset of inconsistencies (i.e., inconsistencies in inter-dependent modules of a project). In addition, Maven's *enforcer* plugin does not provide support on how to harmonize ICs, while LibHarmo estimates required harmonization efforts for developers to confidently make harmonization decisions while saving harmonization efforts (see a detailed evaluation in Sec. 4.5).

False Positive Analysis. Considering the large size of detected ICs and FCs, we took a statistical sampling approach to manually analyze false positives. Specifically, we randomly sampled 238 ICs and 334 FCs. The sample size allows the generalization of our results at a confidence level of 95% and a margin of error of 5%, computed by a sample size calculator [4]. Then, four of the authors followed an open coding procedure [38] to investigate each IC and FC and categorize reasons for false positives. We found no false positive for FCs; but we found 40 false positives for ICs because developers intentionally declare multiple properties for different versions of the same library so as to provide comprehensive supports for different runtime environments. For example, project `memcached-session-manager` is a tomcat session manager that keeps sessions in memcached or Redis, for highly available, scalable and fault tolerant web applications. It declares four properties for version 6.0.45, 7.0.85, 8.5.29 and 9.0.6 for various tomcat dependencies, as it is currently working with tomcat 6.x, 7.x, 8.x and 9.x (as explained in its `README` file). However, such false positives can be easily and quickly determined by developers. Therefore, a false positive rate of 16.8% for ICs can be acceptable.

LibHarmo detected 621 inconsistencies and 2,576 false consistencies respectively in 152 and 346 projects, while Maven's *enforcer* plugin only detected 219 inconsistencies. LibHarmo had a false positive rate of 16.8% for inconsistencies, and no false positive for false consistencies.

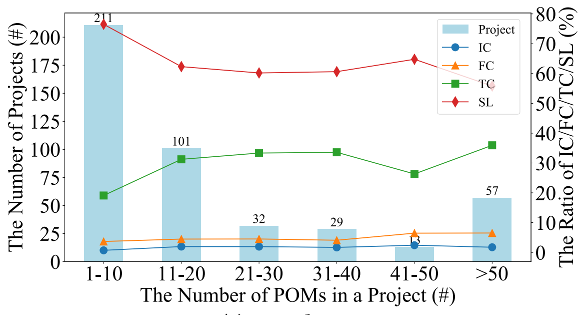
4.3 Prevalence Evaluation (RQ2)

We analyzed the prevalence of ICs and FCs by measuring the overall distribution of ICs, FCs, TCs and SLs as well as their fine-grained distribution with respect to the modular complexity of projects (approximated as the number of POMs).

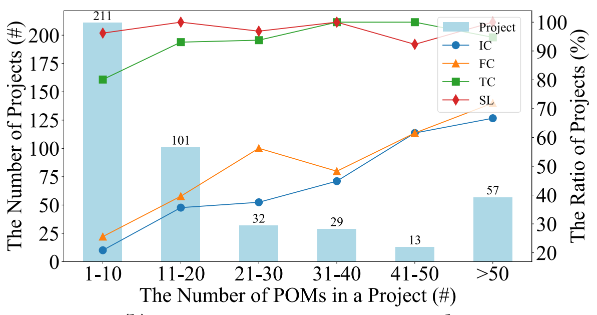
Overall Distribution. Fig. 5a reports the overall distribution of detected ICs, FCs, TCs and SLs. SLs account for 61.0%, while ICs, FCs and TCs account for 39.0%, which means that over one-third of the libraries are used across multiple modules. More specifically, TCs account for 22.6%, which are much higher than ICs and FCs, and cover 318 projects. This indicates that library version harmonization (via referencing a property) is already a practice that is adopted by many projects. Nevertheless, there are still 2,576 FCs, accounting for 13.2% and covering 346 projects. They could turn into ICs if not carefully maintained, and thus increase the burden of library maintenance. There are 621 ICs, which account for 3.2% and cover 152 projects. These results indicate that library version inconsistency and false consistency are quite prevalence in real-world projects.

Moreover, Fig. 5b reports the intersections among the projects that are affected by IC, FC and TC. Noticeably, there is a high overlap (i.e., 251 projects) between TC and FC. This indicates that while many projects adopt consistent library versions, they still leave many libraries not truly consistent. Similarly, the libraries in 51 projects are all consistent, while the libraries in 70 projects are all falsely consistent. Moreover, the overlap between FC and IC is also high (i.e., 134 projects), and most of the projects that have IC also have FC. This is potentially because that FC has a high chance to turn into IC. Furthermore, 109 projects have IC, FC and TC at the same time, which indicates that using consistent library versions is not consistently recognized across the whole development team of a project.

Fine-Grained Distribution. The bars in Fig. 6a report the total number of projects whose number of POMs is in a specific range. As we can see, nearly half (47.6%) of the projects have less than 10 POMs, and only 22.3% of projects have more than 30 POMs. These results indicate that most projects have moderate complexity in modules. The four curves in Fig. 6a report the distribution of IC, FC, TC and SL as projects' complexity in modules increases, while the four curves in Fig. 6b correspondingly present the ratio of projects that have IC, FC, TC and SL. We can see that the ratio of IC slightly increases and the ratio of projects having IC greatly increases. This indicates that as projects have more complexity in modules, library maintenance becomes more complicated, and hence there is a higher chance to introduce inconsistencies. Besides, the ratio of FC and TC does not

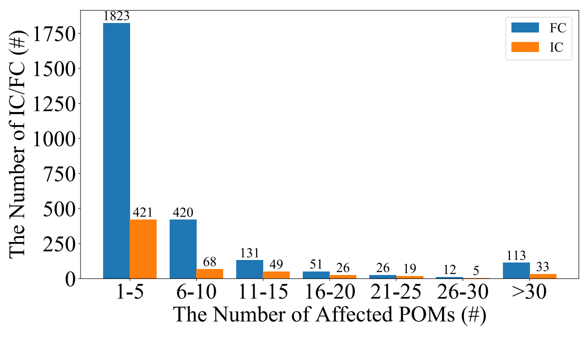


(a) Distribution

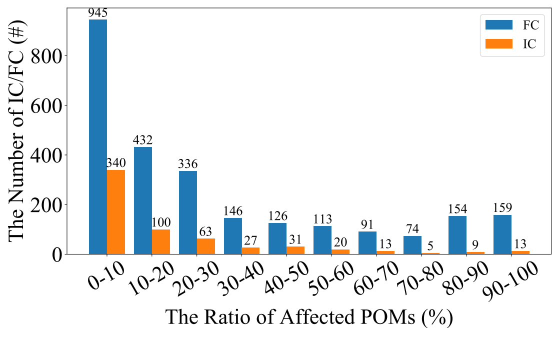


(b) Projects Having IC, FC, TC and SL

Figure 6: Distribution of IC, FC, TC and SL w.r.t. Modular Complexity of Projects



(a) Distribution across the Number of Affected POMs



(b) Distribution across the Ratio of Affected POMs

Figure 7: Distribution of IC and FC across Affected POMs

decrease, and the ratio of projects having FC and TC even increases. This indicates that although projects become more complex in modules, developers may still willing to keep library versions consistent. In that sense, LibHarmo can help developers systematically detect inconsistencies or false consistencies as early as possible.

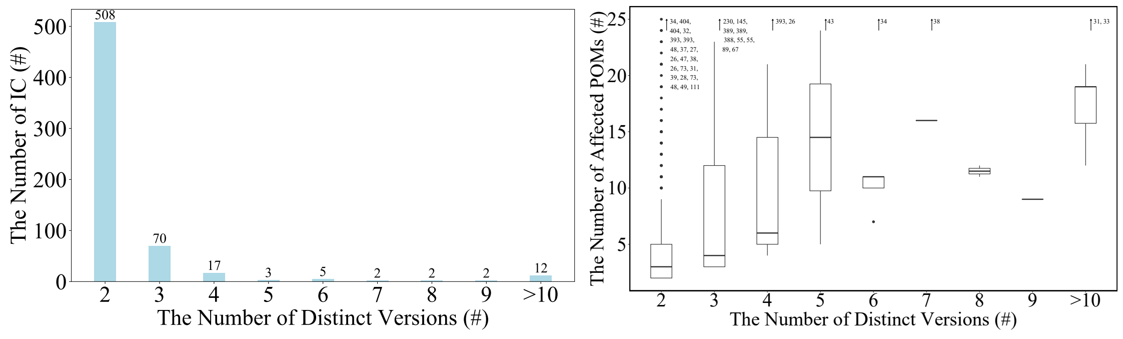
While the practice of true consistencies is widely adopted, inconsistencies and false consistencies are still quite common in real-world projects. As projects have more complexity in modules, it becomes more likely to introduce inconsistencies.

4.4 Severity Evaluation (RQ2)

We analyzed the severity of a detected inconsistency or false consistency (i.e., $\mathcal{M}_{lib}$) in terms of four indicators: 1) the number of POMs that are affected (i.e., $|\mathcal{M}_{lib}|$), 2) the ratio of POMs that are affected (i.e., $|\mathcal{M}_{lib}|$ / $|\mathcal{M}|$), 3) the number of distinct versions declared in $\mathcal{M}_{lib}$, and 4) whether the versions of library dependencies in $\mathcal{M}_{lib}$ are all explicitly declared (i.e., hard-coded), all implicitly declared (i.e., via referencing a property), or declared in a mixed way. The third indicator is only applicable for inconsistencies as false consistencies only have one version. The higher the first three indicators, the more versions are simultaneously adopted in more POMs, and thus the more severe the inconsistency or false consistency. For the fourth indicator, we regard explicit declaration is more severe than mixed declaration and implicit declaration because it indicates that developers seem to be unaware to harmonize library versions by a property. We report the aggregated result over all consistencies or false consistencies for each of the four indicators.

Affected POMs. Fig. 7a presents the distribution of IC and FC with respect to the number of affected POMs. 67.8% of ICs and 70.8% of FCs affect less than five POMs, and 21.3% of ICs and 12.9% of FCs affect more than ten POMs. On the other hand, Fig. 7b reports the distribution of IC and FC with respect to the ratio of affected POMs. 70.9% of ICs and 53.5% of FCs affect less than 20% of POMs, while 9.7% of ICs and 22.9% of FCs affect more than 50% of POMs. These results indicate that most ICs and FCs affect a relatively small number of POMs, but still around one-tenth of ICs and one-fifth of FCs could involve a relatively large number of POMs.

Distinct Versions. Fig. 8a reports the distribution of distinct versions in inconsistencies. We can see that 81.8% of ICs only have two distinct versions, and only 3.7% of ICs have more than five distinct versions. Moreover, we generated a box plot for each bar in Fig. 8a to measure the affected POMs. The result is reported in Fig. 8b, where the arrows indicate higher outliers that we hide to enhance the comprehension of the box plots. As the number of distinct versions in ICs increases, the number of affected POMs increases. With regard to the ICs that have two distinct versions, the median number of affected POMs is around three. This indicates that most ICs are still manageable if developers want to harmonize them. Still, there are 80 outliers in the first box plot, and some of them can affect around 400 POMs. We looked into these 80 outliers, and found that in 72 (90.0%) outliers, more than 80% of the POMs use one version, while less than 20% of the POMs use the other version. More interestingly, in 58 (72.5%) outliers, one of the distinct versions is only used in one POM. One potential reason is that developers have to use a specific version in a minority of POMs to avoid the heavy API backward incompatibility in them. However, if inconsistencies can be detected at the first time they are introduced, such technical debt will not



(a) Distinct Versions **(b) Distinct Versions w.r.t. Affected POMs**

Figure 8: Distribution of Distinct Versions in IC

be accumulated. This is also how LibHarmo can help developers to reduce long-term maintenance cost. Another potential reason is that developers are unaware of the minority of POMs that use a distinct version due to the complex POM inheritance graph. In that sense, automated tools like LibHarmo are needed.

Version Declaration. Fig. 9 shows the distribution of version declarations (i.e., explicit declaration (EX), implicit declaration (IM) and mixed declaration (MX)) for IC and FC. It turns out that 94.1% of FCs declare versions by hard-coding. This means that developers need to change all the affected POMs at the same time to keep these FCs consistent rather than turning such FCs into ICs, which is actually a huge but avoidable maintenance cost. Besides, 36.1% of ICs declare versions by hard-coding. This shows that hard-coding version numbers is probably not a good practice, and it tends to introduce inconsistencies. 63.9% of ICs include implicitly declared versions. This means that developers already have the sense to declare versions by referencing a property for reducing library maintenance cost.

67.8% of ICs and 70.8% of FCs affect less than five POMs. 81.8% of ICs only have two distinct versions, affecting a median number of three POMs. 36.1% of ICs and 94.1% of FCs declare all versions by hard-coding. Overall, the severity of ICs and FCs is relatively not high, indicating potentially low fixing efforts for developers to mitigate ICs and FCs.

4.5 Efforts Evaluation (RQ3)

We analyzed the harmonization efforts for each of the 621 inconsistencies for each candidate harmonized version. We report the results for the candidate version with the least harmonization efforts selected based on our default ranking (see Sec. 3.2). Fig. 10 shows two box plots (one denotes the number of APIs, and the other denotes the number of API calls) for deleted, changed, unchanged, and total library APIs that are called. Overall, 190 (30.6%) ICs have no harmonization efforts; i.e., all the invoked library APIs are not changed in the suggested harmonized version. In the remaining 431 ICs, on average, 1 and 2 of the 24 called library APIs are respectively deleted and changed in the suggested harmonized version, affecting 1 and 12 of the 63 library API calls, as indicated by the green diamonds. In other word, by using LibHarmo, developers only need to focus on the potential incompatibilities in the 3 deleted/changed APIs without worrying about the other 21 (87.5%) unchanged APIs. Hence, 87.5% of the manual efforts can be saved for developers by using LibHarmo.

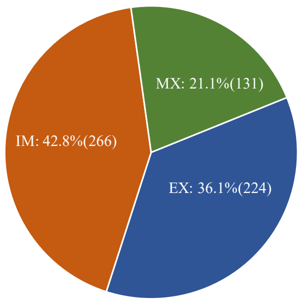
Moreover, LibHarmo provides developers with the 13 callsites of the 3 deleted/changed APIs to ease the harmonization. These results also indicate that the harmonization efforts with respect to the number of deleted and changed APIs seem small, which is actually good news as developers are more likely to mitigate inconsistencies. However, the actual harmonization efforts depend on how the deleted or changed APIs affect the business logic. Therefore, we choose to provide developers with detailed API-level reports to assist them in determining where and whether to harmonize.

Besides, as LibHarmo can automatically refactor POMs, such efforts are also saved for developers. To evaluate the effectiveness of POM refactoring, four of the authors followed an open coding procedure [38] to manually analyze the refactored POMs for the randomly sampled 238 ICs and 334 FCs in Sec 4.2, and found that refactored POMs were all correct. This owes to the fact that LibHarmo is designed to refactor POMs in such a systematic and non-invasive way that does not change the inheritance relationship among POMs.

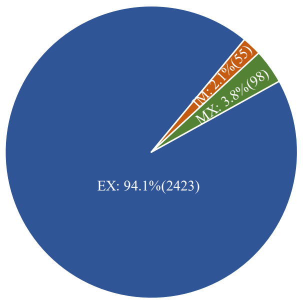
In 190 (30.6%) ICs, all the called library APIs are not changed in the suggested harmonized version. In the remaining 431 ICs, on average, 3 of the 24 called library APIs are deleted/changed, affecting 13 library API calls. Overall, LibHarmo helps to save 87.5% of the harmonization efforts. The harmonization efforts are relatively small but the true efforts are application-specific.

4.6 Developer Feedback (RQ4)

To understand developers' feedback about LibHarmo, we targeted 621 inconsistencies in 152 projects, and sent our automatically generated report to the developers of these projects. Some sample reports can be found at our website [5]. In one month, 39 developers replied. 12 of them explicitly commented that version inconsistency is certainly a problem for library maintainers, and our tool and report are useful; e.g., "*the problem you're describing is very real, and I have encountered it myself in my day-to-day job several times*", "*keep up the good work with your harmonization tool. It definitely sounds interesting!*", "*the cool reports here helped me find one real issue, thanks!*", and "*we can run your tool as part of our CI (to prevent future regressions)*". 9 of them did not comment on our tool, but only discussed the detected inconsistencies, 12 of them asked us to fill a pull request to fix the problem, 3 of them intentionally used inconsistent versions, and 3 of them were no longer Java developers or no longer in charge of the projects.



(a) IC



(b) FC

Figure 9: Version Declaration Distribution in IC and FC

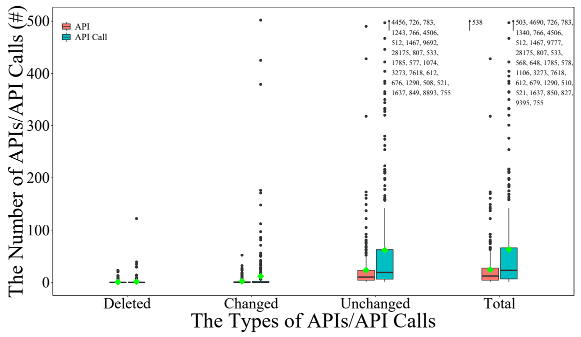


Figure 10: Harmonization Efforts

Besides, 31 inconsistencies from 26 projects have been confirmed and 17 of them have been fixed. As we crawled the project repositories several months before our report, 4 developers asked us to re-generate the report for their current repositories, and we are still waiting for their feedback. The others are still under discussion.

Interestingly, a developer from hadoop confirmed that adopting consistent libraries is one of their common practice, but "*people neglect to do this; when that's found we will pull the explicit version declaration out and reference from hadoop-project; adding the import there if not already present. Therefore any duplicate declaration of a dependency with its own <version> field in any module other than hadoop-project is an error. Your dependency graphs are helpful here*". It is also worth mentioning that 4 developers commented that they also cared about inconsistencies in transitive dependencies, but also said that "*it is also very hard to fix, since the source code is not owned by me*". This is why we only focus on direct dependencies.

Nearly half of the responded developers thought that our tool and report are useful. 31 inconsistencies have been confirmed and 17 of them has been harmonized.

4.7 Discussions

Threats to Survey. First, we chose an online survey with GitHub developers instead of a face-to-face interview study with industrial developers, because it is difficult to recruit industrial developers for interviews at a reasonable cost, and an online study allows us to recruit a relatively large number of developers. Second, we decided to not offer compensation but kindly ask participants to voluntarily take the survey. As a result, we expected that GitHub developers who were really interested in library version inconsistencies and well motivated would participate in this survey. This instead could improve the quality of our survey to avoid potential cases that participants only wanted the compensation but answered haphazardly.

Threats to Evaluation. First, as we have not integrated our tool into the build process, we generated reports about inconsistencies and sent reports to relevant developers for obtaining their feedback instead of letting developers directly use our tool. While this may not get first-hand information from developers (especially for the interaction part), it relieves the burden of developers to install our tool and only focuses on the results. We believe this can help us obtain more feedback. As we have got positive feedback from developers, we are developing LibHarmo as a Maven plugin such that LibHarmo can detect the introduction of inconsistency at the first place without accumulating technical debt. Second, to be honest, we only obtained limited responses from open-source projects. We are collaborating with two interested industrial partners to deploy our tool into their CI.

Limitations. First, due to the well-known limitation of static analysis, our generated API call graphs can be unsound (e.g., due to reflection), which affects the precision of our API-level harmonization efforts analysis and **RQ3**. We will explore a combination of static analysis and dynamic analysis to improve the precise. Second, we currently only target Maven Java projects as Maven is the most widely-used build tool for Java projects, but there are other automated build tools like Gradle and Ant. Given the positive feedback from developers, we plan to support other automated build tools.

5 RELATED WORK

Library Analysis. Patra et al. [61] analyzed JavaScript library conflicts caused by the lack of namespaces in JavaScript, and proposed ConflictJS to first apply dynamic analysis to identify potential conflicts and then use targeted test synthesis to validate them. Wang et al. [76] analyzed manifestation and fixing patterns of dependency conflicts in Java, and developed Decca to detect dependency conflicts. Wang et al. [77] also proposed Riddle to generate crashing stack traces for detected dependency conflicts. Maven's *enforcer* plugin, Decca and Riddle are focused on library version inconsistencies that might have harmful consequence (e.g., bugs), and Decca and Riddle can be seen as an advanced version of *enforcer* by further locating and triggering the bugs. However, as indicated by our survey (**Q8**), harmful consequence is one of the commonest reasons for fixing library version inconsistencies, and the other top two reasons are to avoid great maintenance efforts in the long run and to ensure consistent library API behaviors across modules. Motivated by this result, LibHarmo is designed to have a wider scope in the sense that it also detects library version inconsistencies that will not cause harmful consequence. It is valuable to extend LibHarmo to distinguish harmful and unharmful library version inconsistencies. On the other hand, *enforcer*, Decca and Riddle support transitive library dependencies, while LibHarmo does not because project developers often have no direct control to harmonize the inconsistent library versions in transitive library dependencies. Moreover, LibHarmo provides API-level harmonization effort estimation for developers to confidently fix library version inconsistencies, while *enforcer*, Decca and Riddle do not.

Cadariu et al. [16] proposed an alerting tool to notify developers about Java library dependencies with vulnerabilities. Mirhosseini and Parnin [56] compared the usage of pull requests and badges to notify outdated npm packages. These approaches only detect the inclusion of vulnerable libraries. To determine if the vulnerable library code is in the execution path of a project, Plate et al. [62] applied dynamic analysis to check whether the vulnerable methods were executed by a project; and Ponta et al. [63] extended it by combining dynamic analysis with static analysis. We will consider vulnerabilities as another factor when recommending harmonized versions.

Bloemen et al. [12] analyzed the evolution of the Gentoo package dependency graph, while Kikas et al. [39] and Decan et al. [25] compared the evolution of dependency graphs in different ecosystems. Kikas et al. [39] and Decan et al. [24] also investigated the impact of security vulnerabilities on the dependency graph. Zimmermannet et al. [85] further modeled maintainers and vulnerabilities into the dependency graph in the npm ecosystem, and systematically analyzed the risk of attacked packages and maintainers and vulnerabilities. LibHarmo can be extended to support library version inconsistency analysis on the ecosystem-level dependency graph.

To the best of our knowledge, no previous work has systematically investigate library version inconsistency.

API Evolution. Many studies have investigated API evolution to analyze how developers react to API evolution [13, 33, 52, 67, 68], how APIs are changed and used [80], how API stability is measured [65], how API stability affects Android apps' success [49], how refactoring influences API breaking [27, 40, 44], how and why developers break APIs [15, 37, 83], how API breaking impacts client programs [66, 82], etc. Moreover, several advances have been made to detect API breaking. Previous work mostly uses theorem proving [29, 30, 45, 50, 51] or symbolic execution [58, 73], but has scalability issues when detecting breaking APIs in real-life program. Recently, testing techniques have been used to detect breaking APIs. Gyori et al. [31] relied on regression tests, while Soares et al. [72] generated new tests to detect behavior changes in refactored APIs. Mezzetti et al. [53] and Møller and Torp [57] targeted Node.js libraries, and used model-based testing to detect type-related breaking (i.e., changes to API signatures). Similarly, Brito et al. [14] used heuristics to statically detect type-related changes in Java libraries. However, it is an open problem to detect behavior changes when API signatures are not changed but the API bodies are changed. We will extend such approaches to improve the precision of our effort analysis.

API Adaptation. Several advances have been made to adapt client programs to API evolution based on change rules. Change rules can be manually written by developers [7, 17], automatically recorded from developers [32], derived by API similarity matching [84], mined from API usage changes in libraries themselves [20, 21] and client programs [28, 59, 69], and extracted by a combination of some of these methods [79]. Several empirical studies [18, 81] have found that these methods achieved an average accuracy of 20%. More accurate API adaptation techniques are needed and can be integrated into LibHarmo.

Library Empirical Studies. A large body of studies has been focused on characterizing the usage and update practice of libraries in different ecosystems, e.g., the usage trend and popularity of libraries and APIs [6, 8, 9, 22, 34, 42, 46, 48, 54, 55, 64, 78], the practice of updating library versions [10, 43], the latency of updating library versions [19, 23, 41, 47], and the reason of updating or not updating library versions [10, 11, 26, 43]. Our prior study [75] analyzed usages, updates and risks in third-party libraries, and motivated the problem of library version inconsistency. To the best of our knowledge, we are the first to systematically understand this problem.

6 CONCLUSIONS

In this paper, we have conducted a survey with 131 Java developers from GitHub to collect the first-hand information about practices on library version inconsistencies. Using our survey insights, we have proposed LibHarmo to harmonize inconsistent library versions in Java Maven projects. Our evaluation on 443 Java Maven projects has shown promising results. In the future, we will integrate LibHarmo into the build process, and we plan to turn our data, currently available at our website [5], into archived open data.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (Grant No. 61802067).

REFERENCES

[1] [n.d.]. *HADOOP-6800.* Retrieved March 01, 2020 from https://issues.apache.org/jira/browse/HADOOP-6800

[2] [n.d.]. *Introduction to the Dependency Mechanism.* Retrieved March 01, 2020 from https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html

[3] [n.d.]. *Introduction to the POM.* Retrieved March 01, 2020 from https://maven.apache.org/guides/introduction/introduction-to-the-pom.html

[4] [n.d.]. *Sample Size Calculator.* Retrieved March 01, 2020 from https://www.surveysystem.com/sscalc.htm

[5] [n.d.]. *LibHarmo.* Retrieved March 01, 2020 from https://libharmo.github.io

[6] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. 2017. Why do developers use trivial packages? an empirical case study on npm. In *FSE*. 385–395.

[7] Ittai Balaban, Frank Tip, and Robert Fuhrer. 2005. Refactoring Support for Class Library Migration. In *OOPSLA*. 265–279.

[8] Veronika Bauer and Lars Heinemann. 2012. Understanding API usage to support informed decision making in software maintenance. In *CSMR*. 435–440.

[9] Veronika Bauer, Lars Heinemann, and Florian Deissenboeck. 2012. A structured approach to assess third-party library usage. In *ICSM*. 483–492.

[10] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2013. The evolution of project inter-dependencies in a software ecosystem: The case of apache. In *ICSM*. 280–289.

[11] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2015. How the Apache community upgrades dependencies: an evolutionary study. *Empirical Software Engineering* 20, 5 (2015), 1275–1317.

[12] Remco Bloemen, Chintan Amrit, Stefan Kuhlmann, and Gonzalo Ordóñez-Matamoros. 2014. Gentoo package dependencies over time. In *MSR*. 404–407.

[13] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2016. How to Break an API: Cost Negotiation and Community Values in Three Software Ecosystems. In *FSE*. 109–120.

[14] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. 2018. APIDiff: Detecting API breaking changes. In *SANER*. 507–511.

[15] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. 2018. Why and how Java developers break APIs. In *SANER*. 255–265.

[16] Mircea Cadariu, Eric Bouwers, Joost Visser, and Arie van Deursen. 2015. Tracking known security vulnerabilities in proprietary software systems. In *SANER*. 516–519.

[17] Kingsum Chow and David Notkin. 1996. Semi-automatic Update of Applications in Response to Library Changes. In *ICSM*. 359–368.

[18] Bradley E Cossette and Robert J Walker. 2012. Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. In *FSE*. 55.

[19] Joël Cox, Eric Bouwers, Marko van Eekelen, and Joost Visser. 2015. Measuring dependency freshness in software systems. In *ICSE*, Vol. 2. 109–118.

[20] Barthelemy Dagenais and Martin P Robillard. 2009. SemDiff: Analysis and recommendation support for API evolution. In *ICSE*. 599–602.

[21] Barthélémy Dagenais and Martin P Robillard. 2011. Recommending adaptive changes for framework evolution. *ACM Transactions on Software Engineering and Methodology* 20, 4 (2011), 19.

[22] Coen De Roover, Ralf Lammel, and Ekaterina Pek. 2013. Multi-dimensional exploration of api usage. In *ICPC*. 152–161.
[23] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the Evolution of Technical Lag in the npm Package Dependency Network. In *ICSME*. 404–414.
[24] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the Impact of Security Vulnerabilities in the Npm Package Dependency Network. In *MSR*. 181–191.
[25] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering* 24, 1 (2019), 381–416.
[26] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. 2017. Keep Me Updated: An Empirical Study of Third-Party Library Updatability on Android. In *CCS*. 2187–2200.
[27] Danny Dig and Ralph Johnson. 2006. How Do APIs Evolve? A Story of Refactoring: Research Articles. *J. Softw. Maint. Evol.* 18, 2 (2006), 83–107.
[28] Mattia Fazzini, Qi Xin, and Alessandro Orso. 2019. Automated API-usage update for Android apps. In *ISSTA*. 204–215.
[29] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 2014. Automating regression verification. In *ASE*. 349–360.
[30] Benny Godlin and Ofer Strichman. 2013. Regression verification: proving the equivalence of similar programs. *Software Testing, Verification and Reliability* 23, 3 (2013), 241–258.
[31] Alex Gyori, Owolabi Legunsen, Farah Hariri, and Darko Marinov. 2018. Evaluating Regression Test Selection Opportunities in a Very Large Open-Source Ecosystem. In *ISSRE*. 112–122.
[32] Johannes Henkel and Amer Diwan. 2005. CatchUp! Capturing and replaying refactorings to support API evolution. In *ICSE*. 274–283.
[33] André Hora, Romain Robbes, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, and Marco Tulio Valente. 2015. How do developers react to API evolution? The Pharo ecosystem case. In *ICSME*. 251–260.
[34] Andre Hora and Marco Tulio Valente. 2015. apiwave: Keeping track of API popularity and migration. In *ICSME*. 321–323.
[35] Humble, Jez, and David Farley. 2010. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. Pearson Education.
[36] Michael Jang. 2006. *Linux Annoyances for Geeks: Getting the Most Flexible System in the World Just the Way You Want It.* O'Reilly Media, Inc.
[37] Kamil Jezek, Jens Dietrich, and Premek Brada. 2015. How Java APIs break–an empirical study. *Information and Software Technology* 65 (2015), 129–146.
[38] Shahedul Huq Khandkar. 2009. *Open coding.* Technical Report. University of Calgary.
[39] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. 2017. Structure and evolution of package dependency networks. In *MSR*. 102–112.
[40] Miryung Kim, Dongxiang Cai, and Sunghun Kim. 2011. An Empirical Investigation into the Role of API-level Refactorings During Software Evolution. In *ICSE*. 151–160.
[41] Raula Gaikovina Kula, Daniel M German, Takashi Ishio, and Katsuro Inoue. 2015. Trusting a library: A study of the latency to adopt the latest maven release. In *SANER*. 520–524.
[42] Raula Gaikovina Kula, Daniel M German, Takashi Ishio, Ali Ouni, and Katsuro Inoue. 2017. An exploratory study on library aging by monitoring client usage in a software ecosystem. In *SANER*. 407–411.
[43] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies? *Empirical Software Engineering* 23, 1 (2018), 384–417.
[44] Raula Gaikovina Kula, Ali Ouni, Daniel M. German, and Katsuro Inoue. 2018. An Empirical Study on the Impact of Refactoring Activities on Evolving Client-used APIs. *Inf. Softw. Technol.* 93, C (2018), 186–199.
[45] Shuvendu K Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *CAV*. 712–717.
[46] Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. 2011. Large-scale, AST-based API-usage analysis of open-source Java projects. In *SAC*. 1317–1324.
[47] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. 2017. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. In *NDSS*.
[48] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. An investigation into the use of common libraries in android apps. In *SANER*. 403–414.
[49] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2013. API Change and Fault Proneness: A Threat to the Success of Android Apps. In *ESEC/FSE*. 477–487.
[50] Stephen McCamant and Michael D. Ernst. 2003. Predicting Problems Caused by Component Upgrades. In *ESEC/FSE*. 287–296.
[51] Stephen McCamant and Michael D Ernst. 2004. Early identification of incompatibilities in multi-component upgrades. In *ECOOP*. 440–464.
[52] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An Empirical Study of API Stability and Adoption in the Android Ecosystem. In *ICSM*. 70–79.
[53] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. 2018. Type regression testing to detect breaking changes in Node. js libraries. In *ECOOP*.
[54] Yana Momchilova Mileva, Valentin Dallmeier, Martin Burger, and Andreas Zeller. 2009. Mining trends of library usage. In *IWPSE-Evol*. 57–62.
[55] Yana Momchilova Mileva, Valentin Dallmeier, and Andreas Zeller. 2010. Mining API Popularity. In *Testing – Practice and Research Techniques*. 173–180.
[56] Samim Mirhosseini and Chris Parnin. 2017. Can automated pull requests encourage software developers to upgrade out-of-date dependencies?. In *ASE*. 84–94.
[57] Anders Møller and Martin Toldam Torp. 2019. Model-based testing of breaking changes in Node. js libraries. (2019).
[58] Federico Mora, Yi Li, Julia Rubin, and Marsha Chechik. 2018. Client-specific Equivalence Checking. In *ASE*. 441–451.
[59] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Jr., Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. 2010. A Graph-based Approach to API Usage Adaptation. In *OOPSLA*. 302–321.
[60] Eugen Paraschiv. 2018. *The State of Java in 2018.* Retrieved March 01, 2020 from https://www.baeldung.com/java-in-2018
[61] Jibesh Patra, Pooja N Dixit, and Michael Pradel. 2018. ConflictJS: finding and understanding conflicts between JavaScript libraries. In *ICSE*. 741–751.
[62] Henrik Plate, Serena Elisa Ponta, and Antonino Sabetta. 2015. Impact assessment for vulnerabilities in open-source software libraries. In *ICSME*. 411–420.
[63] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. 2018. Beyond Metadata: Code-Centric and Usage-Based Analysis of Known Vulnerabilities in Open-Source Software. In *ICSME*. 449–460.
[64] Dong Qiu, Bixin Li, and Hareton Leung. 2016. Understanding the API usage in Java. *Information and software technology* 73 (2016), 81–100.
[65] Steven Raemaekers, Arie van Deursen, and Joost Visser. 2012. Measuring software library stability through historical version analysis. In *ICSM*. 378–387.
[66] Steven Raemaekers, Arie van Deursen, and Joost Visser. 2017. Semantic versioning and impact of breaking changes in the Maven repository. *Journal of Systems and Software* 129 (2017), 140–158.
[67] Romain Robbes, Mircea Lungu, and David Röthlisberger. 2012. How do developers react to API deprecation?: the case of a smalltalk ecosystem. In *FSE*. 56:1–56:11.
[68] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. 2016. On the reaction to deprecation of 25,357 clients of 4+ 1 popular Java APIs. In *ICSME*. 400–410.
[69] Thorsten Schäfer, Jan Jonas, and Mira Mezini. 2008. Mining framework usage changes from instantiation code. In *ICSE*. 471–480.
[70] Schlosser, Gerhard, and Günter P. Wagner. 2004. *Modularity in development and evolution.* University of Chicago Press.
[71] Nicholas Smith, Danny van Bruggen, and Federico Tomassetti. 2017. JavaParser: Visited. *Leanpub, oct. de* (2017).
[72] Gustavo Soares, Rohit Gheyi, Dalton Serey, and Tiago Massoni. 2010. Making program refactoring safer. *IEEE software* 27, 4 (2010), 52–57.
[73] Anna Trostanetski, Orna Grumberg, and Daniel Kroening. 2017. Modular demand-driven analysis of semantic difference for program versions. In *SAS*. 405–427.
[74] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot: A Java bytecode optimization framework. In *CASCON*. 13–.
[75] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. 2020. An Empirical Study of Usages, Updates and Risks of Third-Party Libraries in Java Projects. In *ICSME*.
[76] Ying Wang, Ming Wen, Zhenwei Liu, Rongxin Wu, Rui Wang, Bo Yang, Hai Yu, Zhiliang Zhu, and Shing-Chi Cheung. 2018. Do the Dependency Conflicts in My Project Matter?. In *ESEC/FSE*. 319–330.
[77] Ying Wang, Ming Wen, Rongxin Wu, Zhenwei Liu, Shin Hwei Tan, Zhiliang Zhu, Hai Yu, and Shing-Chi Cheung. 2019. Could I Have a Stack Trace to Examine the Dependency Conflict Issue. In *ICSE*. 572–583.
[78] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. 2016. A look at the dynamics of the JavaScript package ecosystem. In *MSR*. 351–361.
[79] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. 2010. Aura: a hybrid approach to identify framework evolution. In *ICSE*. 325–334.
[80] Wei Wu, Foutse Khomh, Bram Adams, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2016. An exploratory study of api changes and usages based on apache and eclipse ecosystems. *Empirical Software Engineering* 21, 6 (2016), 2366–2412.
[81] Wei Wu, Adrien Serveaux, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2015. The impact of imperfect change rules on framework api evolution identification: an empirical study. *Empirical Software Engineering* 20, 4 (2015), 1126–1158.
[82] Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. 2017. Historical and impact analysis of API breaking changes: A large-scale study. In *SANER*. 138–147.
[83] Laerte Xavier, Andre Hora, and Marco Tulio Valente. 2017. Why do we break APIs? first answers from developers. In *SANER*. 392–396.
[84] Zhenchang Xing and Eleni Stroulia. 2007. API-evolution support with Diff-CatchUp. *IEEE Transactions on Software Engineering* 33, 12 (2007), 818–836.
[85] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *USENIX Security*.