Do the Dependency Conflicts in My Project Matter?

Ying Wang
Northeastern University
Shenyang, China
wangying8052@163.com

Ming Wen
The Hong Kong University of Science and Technology
Hong Kong, China
mwenaa@cse.ust.hk

Zhenwei Liu
Northeastern University
Shenyang, China
lzwneu@163.com

Rongxin Wu
The Hong Kong University of Science and Technology
Hong Kong, China
wurongxin@cse.ust.hk

Rui Wang
Northeastern University
Shenyang, China
jwm080795@163.com

Bo Yang
Northeastern University
Shenyang, China
yb9506@126.com

Hai Yu
Northeastern University
Shenyang, China
yuhai@mail.neu.edu.cn

Zhiliang Zhu*
Northeastern University
Shenyang, China
zzl@mail.neu.edu.cn

Shing-Chi Cheung*
The Hong Kong University of Science and Technology
Hong Kong, China
scc@cse.ust.hk

ABSTRACT

Intensive dependencies of a Java project on third-party libraries can easily lead to the presence of multiple library or class versions on its classpath. When this happens, JVM will load one version and shadows the others. Dependency conflict (DC) issues occur when the loaded version fails to cover a required feature (e.g., method) referenced by the project, thus causing runtime exceptions. However, the warnings of duplicate classes or libraries detected by existing build tools such as `Maven` can be benign since not all instances of duplication will induce runtime exceptions, and hence are often ignored by developers. In this paper, we conducted an empirical study on real-world DC issues collected from large open source projects. We studied the manifestation and fixing patterns of DC issues. Based on our findings, we designed Decca, an automated detection tool that assesses DC issues' severity and filters out the benign ones. Our evaluation results on 30 projects show that Decca achieves a precision of 0.923 and recall of 0.766 in detecting high-severity DC issues. Decca also detected new DC issues in these projects. Subsequently, 20 DC bug reports were filed, and 11 of them were confirmed by developers. Issues in 6 reports were fixed with our suggested patches.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**;

KEYWORDS

Empirical study, third party library, static analysis

ACM Reference Format:
Ying Wang, Ming Wen, Zhenwei Liu, Rongxin Wu, Rui Wang, Bo Yang, Hai Yu, Zhiliang Zhu, Shing-Chi Cheung. 2018. Do the Dependency Conflicts in My Project Matter?. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3236024.3236056

1 INTRODUCTION

The popularity of the Java language leads to the development of numerous Java third-party libraries [69, 96]. For example, the `Maven` repository [43] has indexed over 8.77 millions Java libraries. These libraries provide divergent functionalities and are frequently leveraged by developers to implement new projects [102]. Specifically, we investigated over 2,000 popular (i.e., over 50 stars or forks) Java projects that are randomly selected from Github [24], and found that a project directly depends on 14 different libraries on average . Besides, since a depended library might depend on other libraries, a host project would transitively depend on more libraries (e.g., 48 libraries on average for these investigated projects).

Such intensive dependencies on third-party libraries can easily lead to dependency conflicts in practice. That is, multiple versions of the same library or class are presented on the classpath. When multiple classes with the same fully-qualified name exist in a project, JVM will load one of them and shadow the others [95]. If these classes are not compatible, the program can exhibit unexpected behaviors when its components rely on the shadowed ones [72, 82].

Consider the following example. A dependency conflict issue (i.e., `#DERBY-5429` [21]) was reported to project `Apache/Derby`. A `Derby` developer found that class `JVMInfo` was included in both `derbyclient.jar` and `derby.jar`. However, if an older version of `derbyclient.jar` is specified on the classpath before `derby.jar`, the following error shown in Figure 1 will occur when the method

*Zhiliang Zhu and Shing-Chi Cheung are the corresponding authors of this paper

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3236056

`javaDump()` is triggered. This indicates that dependency conflict issues may lead to system crashes in practice.

```
java.lang.NoSuchMethodError: org/apache/derby/iapi/services/info/JVMInfo.javaDump()V
    at org.apache.derby.iapi.services.context.ContextManager.cleanupOnError(Unknown Source)
    at org.apache.derby.impl.jdbc.TransactionResourceImpl.cleanupOnError(Unknown Source)
    at org.apache.derby.impl.jdbc.EmbedConnection.<init>(Unknown Source)
    at org.apache.derby.jdbc.Driver40.getNewEmbedConnection(Unknown Source)
    at org.apache.derby.jdbc.InternalDriver.connect(Unknown Source)
```

Figure 1: A real issue `#DERBY-5429` caused by dependency conflict

Java project build tools such as `Maven` [43] and `Gradle` [26], can warn developers of duplicate JARs and classes, but they cannot identify whether the duplications are benign or harmful (e.g., causing runtime exceptions). If the duplicate classes are identical or compatible, the project runs normally even though it has unclean design or build. On the contrary, if the duplicate classes are incompatible, the warnings need to be carefully resolved. Most building tools provide their own dependency management strategies to help developers select one version of the duplicate classes during the packaging process [75, 97]. However, these tools do not guarantee loading the most appropriate class. Consequently, runtime exceptions will occur if inappropriate classes are loaded. What is worse, since existing tools do not differentiate benign from harmful warnings, developers may overlook the harmful ones and take no resolution actions, which might lead to serious consequences.

A dependency conflict issue arises when the loaded classes are not the expected ones of the project (i.e., the referenced feature set of the the project is not fully covered by the loaded classes). In practice, the dependency conflict issues can be manifested in different ways, depending on class loading mechanisms. However, no systematic studies have been made to understand the manifestation patterns of dependency conflicts and the severity of these conflicts. In this paper, we first conducted an empirical study to bridge this gap. In particular, we collected 135 real-world dependency conflict issues from the Java projects hosted on the `Apache` ecosystem [3] across 16 categories (e.g., big-data, FTP, library and testing), and explored the following two research questions. To ease presentation, we refer to the Dependency Conflict issues as DC issues.

- **RQ1 (Issue manifestation patterns):** What are the common manifestations of DC issues? Are there patterns that can be extracted to enable automated detection of these problems?
- **RQ2 (Issue fixing patterns):** How do developers fix DC issues in practice? Are there factors that affect developers' choices of different fixing solutions?

Answers to the two research questions enable us to better understand the characteristics of DC issues between a host project and its referenced third-party libraries. Specifically, we found that DC issues share three general manifestation patterns: *conflicts in library versions*, *conflicts in classes among libraries*, and *conflicts in classes between the host project and libraries* (see Section 3). The findings provide developers with guidance to help avoid, detect, and diagnose DC issues automatically. Besides, by further analyzing the patches of the fixed DC issues (128/135), we observed that there are four common fixing solutions in practice. In addition, by analyzing the discussions of the corresponding issue reports, we distilled the key information to diagnose DC issues as well as the factors affecting the efforts required for fixing the issues.

Based on the common issue manifestations and the fixing patterns revealed by our empirical study, we designed and implemented a DC detection tool using static analysis, Decca (DependEnCy Conflicts Assessment), to help avoid dependency conflict failures at runtime. Specifically, Decca can assess the severity levels of DC issues based on their impacts on the project (e.g., whether it will cause runtime failures) and maintenance costs (e.g., the required fixing efforts). To evaluate the effectiveness and usefulness of our approach, we first collect a high quality dataset of DC issues with high and low severity levels. The evaluation results show that Decca can achieve a precision of 0.923 and recall of 0.766. More importantly, we applied Decca to analyze the latest version of 30 Java projects, which include 9.8 million lines of codes, to see if it can detect unknown DC issues in the field. Decca successfully detected 466 DC issues of four different severity levels (defined in Section 4.4) from 24 projects. We further filtered out 438 benign DC issues and reported 20 bugs which contains 28 harmful issues to the corresponding developers, and 11 reported bugs (55%) have been confirmed as real DC issues, which affects the projects' reliability and maintainability. Most of the confirmed issues are identified in popular projects such as `Apache Spark` [54], `Hadoop` [27], `Beam` [13], `Google Closure-compiler` [25]. More excitingly, developers have quickly fixed 6 (55%) of them by following our suggestions. They also expressed great interests in our tool Decca. These results show the usefulness of our approach. In summary, this paper makes the following contributions:

- To the best of our knowledge, we conducted the first empirical study on DC issues in open source Java projects. Our findings help understand the characteristics of DC issues and provide guidance to related researches (e.g., compatibility and maintainability). The empirical study dataset is publicly available for research purpose.
- We proposed a dependency conflict detection approach based on the knowledge we learned from our empirical study. It can automatically detect DC issues. More importantly, it can assess their severity levels and reduce developers' efforts by filtering out those benign issues.
- We implemented our approach as an open source tool Decca[1]. The evaluation results on real world projects confirmed the effectiveness and practical usefulness of our approach.

2 PRELIMINARIES

2.1 Motivation

To investigate the pervasiveness of the DC problem, we collected 2,289 Java projects from Github and examined whether they contain duplicate JARs or classes using the `Maven-Dependency-Plugin` [41]. The above projects were randomly selected based on two criteria: first, it should have achieved over 50 stars or forks (popularity); second, it is built on `Maven` platform. The results show that 1,457 (63.65%) projects contain the same library of different versions, 1,003 (43.82%) projects contain duplicate classes in different libraries. Besides, 954 (41.68%) of projects are affected by both of the above two cases. These results indicate that DC problem is very common in practice.

Dependency conflicts can have serious consequences in practice. For instance, `Apache/Hadoop` significantly suffers from the

[1] link: https://deccadc.github.io/fse18/

DC issues, as mentioned in its issue report #HADOOP-11656 [28]. According to the issue report, Hadoop exposes a variety of third party libraries to its downstream clients (i.e., those projects depend on Hadoop). This caused its downstream users suffering long from the problem of dependency conflicts. Hadoop developers have been always searching for a good solution to resolve these conflicts. For instance, one developer complained about the current workaround:

"We have tried dependency harmonization in the past. It doesn't work, because different projects have different release schedules and different needs. Not to mention different communities. Also, projects like HBase want to support multiple versions of Hadoop. This means that they either have to live with mixed versions of things like Guava, Jetty, etc. or agree to never update dependencies."

This DC issue has lasted for 879 days (from Mar. 2, 2015 to Sep. 28, 2017), before it was resolved by shading (i.e, renaming) all its dependencies into a *Uber Jar* [61] with the help of Maven-Shade-Plugin [44]. However, the workaround affected 176 downstream client projects and has induced various software maintenance problems.

Our above preliminary study has shown the pervasiveness and significance of the DC problem. When conflicting JARs are detected, existing software build tools, such as Maven, generally adopt an arbitration mechanism to load one of the JARs. However, correctness is not guaranteed. These tools will also give warnings of duplicate classes. Due to lack of further analysis, many of these warnings are false positive results. For example, in our investigated projects, we observed that only 23.00% of the DC issues reported by Maven attracted developers' attention and were fixed within five subsequent releases. Besides, these tools do not analyze the impacts of the DC issues on the concerned project and maintenance costs. Consequently, developers might mistakenly overlook harmful DC issues that lead to system crashes.

2.2 Challenges

Motivated by the above observations, we aim at detecting the DC warnings and their severity levels in this study. An important objective in our study is to assess the severity levels for these detected DC warnings based on their impacts on the project and their maintenance costs. However, effective detection and assessment of DC issues need to address the following two challenges. First, the manifestations of DC issues in practice are diverse and non-deterministic. It is because that the manifestation of DC issues depends on the order in which the JARs are present on the classpath, while the classpath order of dependencies might differ across different running environments [71]. In addition, the order is also affected by the dependency management criteria of software building platforms (e.g., Maven, Gradle). To address this challenge, we conducted an empirical study to characterize the common manifestation patterns of DC issues (see Section 3). Second, there are no existing empirical evidences or tools that indicate which types of dependency conflicts are more serious than the others. Therefore, even if dependency conflicts have been manifested, we still do not know how to assess the severity levels for them. To address this challenge, we further analyzed the diverse information of the DC issue reports and their patches, to understand how to assess their impacts on the project at runtime and what information should be extracted to classify their warning severities (see Section 4).

3 EMPIRICAL STUDY

This section aims at answering our proposed research questions RQ1-2. In the following, we first present the data collection process and then discuss our empirical findings.

3.1 Data Collection Process

To understand the manifestation and fixing patterns of DC issues, we selected Java open source projects that are built by Maven from the Apache ecosystem as the subjects for our empirical study due to the following reasons. First, Apache is one of the most popular open source software ecosystem and includes different types of software. The subjects thus selected are representative. Moreover, the use of Apache projects for empirical software engineering research is common [69, 76, 79, 85, 86, 89, 99, 103]. Second, Apache projects are well-maintained. Specifically, they use the BugZilla [16] and JIRA [39] systems to track issues. The Apache Software Foundation provides official support of Git [23] mirrors for all projects. The issue reports and code repositories are open to the public, which greatly facilitates us to study the target problem [98].

For the selected Java projects, we identified confirmed DC issues in the issue tracking systems. First, we used the keywords "library", "dependency" or "compatibility", etc. to narrow the issues down to the compatibility problems between host projects and third party libraries. Then, we used the keywords "conflict" or "NoSuchMethodError", etc. to further locate the DC issues. As such searches returned noisy results, we refined these results by manual checking [83]. Eventually, we obtained 135 DC issues, and 128 of them have been fixed.

Table 1 summarizes the demographics of those projects covered by the collected 135 DC issues. They are large (up to 2,218 kLOC), popular (up to 16,398 stars), and diverse (14 different categories, e.g., big data, build management, HTTP). In the following, we analyze the 135 DC issues from the Apache projects and report our findings.

3.2 RQ1: Issue Manifestation Patterns

DC issues in Java projects can be manifested in different ways. This RQ aims at categorizing their manifestation patterns. The categorization enables us to understand how DC issues are triggered and how they can be detected. To answer RQ1, we manually investigated the collected 135 bug reports and related discussions(e.g., comments and patches). Specifically, three postgraduates first classified them independently, and then reached a consensus by discussion if the classification results are different among them.

We observed that DC issues can be manifested in three patterns: (1) *conflicts in library versions*, (2) *conflicts in classes among libraries*, and (3) *conflicts in classes between the host project and libraries*. These three patterns differ in how a DC issue is triggered. We use three examples in Figure 2 to illustrate these three patterns, respectively.

A. Conflicts in library versions (39 out of 135 issues). Suppose a host project directly uses Lib1 and Lib2:v1, and Lib1 transitively depends on Lib2:v2, where *L:v* denotes a library *L* of version *v*. According to Maven's *nearest wins strategy*, Maven chooses the version that appears at the nearest to the root (host project) of the dependency tree if there are multiple versions of the same library. As shown in Figure 2(a), the log information in the bottom is the dependency information printed by Maven. Only Lib2:v1 will be packaged during the building process. Then, a system failure will occur if Lib1 invokes the features which are not included in Lib2:v1.

Table 1: The statistics of subjects covered by collected 135 DC issues

Software	Star			Size (LOC)[1]			Categories[2]	Issue tracking systems		Severity[3]					Selected issues
	Min.	Max.	Avg.	Min.	Max.	Avg.		Jira	BugZilla	M	C	B	N	Mi	
71	10	16398	796	0.9k	2218.4k	375.3k	16/28	115	20	74	7	19	13	22	135

1. LOC denotes lines of codes; 1 K = 1000; 2. The software category refers to Apache official definition [5]; 3. M: Major; C:Critical; B:Block; N:Normal; Mi:Minor

For instance, in issue `#YARN-6414` [66], two versions of `Guava` are introduced in `YARN`. Their introduced dependency paths are: `YARN` → `Guava:21.0` and `YARN` → `Hadoop` → `Guava:11.0.2`. As a result, `Guava:11.0.2` is shadowed by `Guava:21.0`, as the latter is nearer to the host project `YARN`. From developers' discussions, we found that `Hadoop` referenced to class `LimitInputStream` which is defined in the shadowed version `Guava:11.0.2` while not defined in the loaded version `Guava:21.0`. Therefore, the project crashed with `NoClassDefFoundError`.

B. Conflicts in classes among libraries (90 out of 135 issues). Suppose that libraries `Lib1` and `Lib2` are present in the dependency tree as shown in Figure 2 (b). `Lib1` and `Lib2` include three duplicate classes *A*, *B* and *C*. Based on the `Maven`'s *first declaration wins strategy*, the duplicate classes within the first declared JAR `lib2` will shadow the ones included in `lib1`. Then, DC issue arises, if the host project referenced to the features only defined in the shadowed classes. The scenario generally occurs in two cases: (1) a fat JAR repackages a library that is already declared on the classpath; and (2) a library is renamed and unknowingly added to the classpath twice. Taking issue `#SUREFIRE-851` [59] as an example, `Maven Surefire` directly depends on libraries `Jaxws-rt` and `Gf-client`. However, library `Gf-client` defined class `SEIModel` that was incompatible with class `SEIModel` included in library `Jaxws-rt`. `Maven Surefire` referenced to method `SEIModel.getJAXBContext()` defined in `Gf-client` while not defined in `Jaxws-rt`. Unfortunately, class `SEIModel` in `Gf-client` was shadowed, as library `Jaxws-rt` declared ahead of it on the classpath. As a result, `NoSuchMethodError` occurred at runtime.

C. Conflicts in classes between host project and libraries (6 out of 135 issues). If the host project and `Lib1` include duplicate classes *A*, *B* and *C*, then only those included in `Lib1` will be included during the packaging process. However, if the features only defined in classes *A*, *B* and *C* of the host project have been referenced, system might throw exceptions or errors. The dependency analysis scope of `Maven` platform is limited to the libraries listed in its dependency configuration script (`pom.xml`). Since `pom.xml` does not declare the host project [52], `Maven` does not even notice the DC issues of such case. For example, in `#STORM-2382` [58], host project `Storm` and its dependent library `Log4j` included incompatible classes with the same names. In particular, as `Log4j` is a logging framework, several classes defined in `Log4j` have been moved into `Storm` for specific purposes during the evolution process. Then, the classes included in library `Log4j` shadowed those defined in the host project, which leaded to a runtime failure.

These three manifestation patterns *A*, *B* and *C* are all ascribed to the reason that the host project references to the unexpected version of classes or libraries. More accurately, if the loaded classes or libraries do not completely cover the actually referenced feature set of the host project, runtime exceptions or errors will occur. These findings help us understand when duplicate classes or libraries are introduced in one project, which version will shadow the others based on building mechanism. In addition, these findings are crucial for us to automatically detect DC issue and distinguish the benign and harmful DC warnings as well.

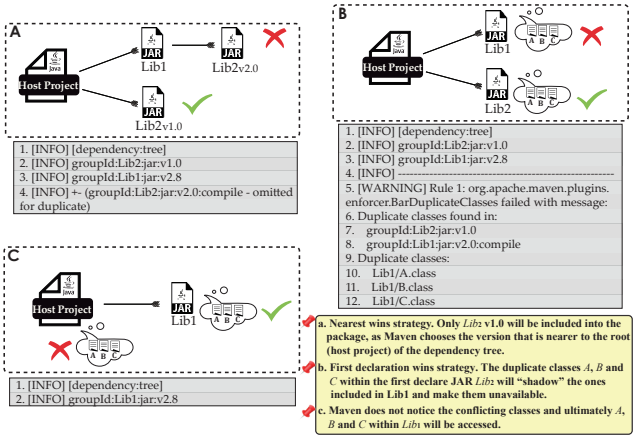


Figure 2: Issue manifestation patterns

3.3 RQ2: Issue Fixing Patterns

To answer RQ2, we further studied the issue reports and patches of the 128 fixed DC issues. We aim to study (1) whether the fixing solutions of DC issues share common patterns; (2) what factors affect the choice of different fixing solutions; and (3) how much effort developers spend in fixing these issues for different solutions.

Our study identified four common patterns that were applied to fix the DC issues:

Pattern 1: Shading the conflicting libraries (25 out of 128 solutions). `Maven-Shade-Plugin` provides the capability to package the project in an *Uber Jar* [61], including its third party libraries. It will also shade (i.e., rename) the packages of some of the libraries. In this way, developers can relocate the classes that were included in the shaded artifact to create a private copy of their bytecode (e.g., issue `#SPARK-2848`[56]). This solution allows multiple versions of the same class to be referenced by the host project.

Pattern 2: Adjusting the classpath order of dependencies (42 out of 128 solutions). The dependency order configurations provided by software build tools, play a vital role in determining a project's classpath. Especially, for `Maven`, library declaration order specified in `pom.xml` corresponds to their appearance order on the classpath [40]. In practice, forcing a particular dependency order on the classpath is a strategy commonly used by developers for fixing DC issues at a relatively low cost. For example, in issue `#HDFS-10570` [32], there were two versions of the `Netty` libraries on the `HDFS`'s classpath and the older version shadowed the newer one. A runtime failure occurred, as the loaded older version of `Netty` does not include the features referenced by `HDFS`. However, the shadowed newer version of `Netty` could cover all the referenced features. Therefore, developers solved this issue easily by reversing their declaration order in `pom.xml`. Adopting this solution to work around DC issues requires the shadowed classes define all the features referenced by host project.

Table 2: The relations between manifestation and fixing patterns

Manifestation	Pattern 1	Pattern 2	Pattern 3	Pattern 4	Other
Pattern A	5	10	18	3	1
Pattern B	20	32	30	2	1
Pattern C	0	0	3	0	3

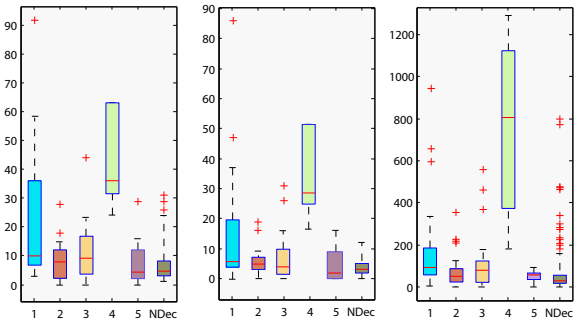
Pattern 3: Harmonizing library versions (51 out of 128 solutions). A DC issue always occurs when multiple versions of the same library coexist in one project but they are incompatible with each other. Solutions of this pattern upgrade or downgrade some of the JARs to resolve the version inconsistencies (e.g., issue `#HADOOP-7606` [31]). However, the solution could only be applied to the scenario that the harmonized version completely covers the feature set referenced by the host project. As developers discussed in issue `#HADOOP-8104` [30]:

"*The inconsistent versions of `Jackson` libraries (1.8 and 1.7.1) caused the `NoSuchMethodError`. By manually verifying, `Jackson` 1.8.8 artifact contains all the methods explicitly invoked by `Hadoop`. So we don't need the mixed versions of `Jackson`.*"

Pattern 4: Classloader customization (5 out of 128 solutions). This solution uses dynamic module system frameworks such as `OSGI` [48] and `Wildfly` [64], to allow different versions of the same libraries or classes coexist in one project by creating multiple classloaders [49, 74]. Although this solution works for most of the DC issues, it comes with additional costs and often requires developers to have a deeper understanding of the class loader mechanism. Besides, using these frameworks also requires the system undergoing a series of laborious refactoring operations. For instance, in issue `#CURATOR-200` [19], `Curator` project involved deep coupling with an older version of `Guava`, and its other libraries introduced a newer `Guava` release. As the host project referenced to the feature sets included in both of these two versions, they adopted `OSGI` technique to keep them coexist in `Curator`. Since the fixing solution is laborious and time-consuming, the developers struggled to adopt this solution after 35 rounds of discussions.

Other workarounds (5 out of 128 solutions). The remaining issues are resolved in miscellaneous ways. For instance, they printed a warning on console and disabled the unexpected behaviors caused by conflicts (e.g., issue `#YARN-5271` [65]), or used a combination of mix versions of a library to satisfy all the feature references (e.g., issue `#CXF-5132` [20]), etc.

Diagnosis and fixing efforts. Figure 3 shows the comparisons of the diagnosis and fixing efforts. In order to compare the efforts required to fix DC issues, we collected another 128 non-DC issues with the same severities as the collected DC issues aforementioned. **Pattern 1** and **Pattern 4** are applicable to the scenario that the host project references to the feature set defined in multiple versions of the same library or class, but neither of a single version can satisfy the host project's requirement. From the concerned discussions, developers first need to spend efforts in identifying the duplicate classes, the loaded classes, and their referenced classes. They then decide which solution can relocate the shadowed ones. Besides, customizing class loaders or shading the JARs are difficult to handle. Thus, the fixing efforts of **Pattern 1** and **Pattern 4** are significantly higher than those of the others. On the contrary, adjusting dependency order on classpath to eliminate conflicts is a relatively easier way to resolve the problem as shown in Figure 3. Even so, it takes more time to diagnose for DC issues than non-DC ones on average.



(a) Comments (b) Rounds (c) Duration

Figure 3: Comparison of the diagnosis and issue-fixing efforts ("1 ~ 4" = "Pattern 1 ~ 4", "5" refers to other workarounds, "NDec = non-dependency conflict issues"), where (a) concerns the number of comments for each issue, (b) concerns the number of discussion rounds during the fixing process, and (c) concerns the number of days from the time when the issue was reported to the time when it was fixed.

Factors affecting the choice of solutions. Table 2 shows the relationships between common manifestation patterns and fixing patterns. From this table, we can tell that issues that are manifested by type *A* and *B* can be fixed by all the four solution patterns, and issues manifested by *C* are mainly fixed by harmonizing library versions or modifying the code to work around the problems. By further analyzing the discussions of these issue reports, we found that the selection of these solutions are highly correlated with the following four factors: *duplicate class set*, *actual loaded feature set*, *shadowed feature set* and *referenced feature set*. The root cause for all these DC issues is that the actual loaded features is a subset of the referenced ones. If reversing the dependency declaration order on the classpath can avoid runtime exceptions (i.e., the shadowed features satisfy host project's requirements), developers prefer to adopt **Pattern 2** as a workaround to fix this issue. The reason is that it comes along with lowest cost without the requirement to change any code. Besides, keeping version consistency via **Pattern 3** is commonly adopted if the harmonized version can cover all the referenced features. Otherwise, developers have no choice but to use **Patterns 1, 4** or other workarounds to relocate the shadowed dependencies required by the host project to ensure the coexistence of the conflicting JARs.

Diagnosis and fixing costs give us hints for assigning the severity levels for DC issues. Naturally, more complicated cases should be more carefully handled to ensure the projects' long-term health and improve their maintainability.

4 DEPENDENCY CONFLICT DIAGNOSIS

Despite extensive warnings given by software build tools, dependency conflicts still exist and remain unresolved in 68.46% the popular `Github` projects as revealed in Section 2.1. The reason is that the existing build tools detect DC issues at library or class level without analyzing the impacts of these issues on program behaviors. As such, these tools do not differentiate the benign issues from the harmful ones. This motivates us to propose an automated approach to analyze DC issues at a finer granularity and provide

assessment of their severity levels. Based on our findings in RQ1, the benign and harmful DC issues are determined by the difference between the loaded and actually referenced classes. Combining our observed manifestation patterns and dependency management criteria of building platform, we can automatically detect the multiple versions of classes or libraries, and identify the loaded version and the shadowed ones on the classpath. Then, we can differentiate the benign from harmful DC issues. As discussed in RQ2, these issues can be further subdivided based on their diagnosis and fixing costs. In this manner, we can help developers highlight the harmful DC issues to avoid runtime exceptions or errors. In the following, we first formulate the DC problem and then elaborate on our detection algorithm and severity assessment strategy.

4.1 Problem Formulation

To ease presentation, we let p denote a host project. The set of classes defined by the developers of the host project is called the host class set, which is denoted as $\mathcal{H}$. The set of third-party libraries referenced by the host project is denoted as $\mathcal{B}$. For each library $l_i \in \mathcal{B}$, it defines a set of classes C_{l_i}, where i indexes a third-party library. Multiple versions of a library are uniquely indexed. We denote all the classes defined in $\mathcal{B}$ as C, specifically, $C = \cup_{\forall l_i} C_{l_i}$. Then, the set of all the classes that might appear on the classpath is denoted $\mathcal{K} = \mathcal{H} \cup C$. For each class $c_i \in \mathcal{K}$, we define function $f(c_i)$ to extract the set of all the features in c_i. A feature refers to a method in this study. Based on the above definitions, we can formally define the dependency conflict problem among the classes in $\mathcal{K}$ as follows. Note that our formulation focuses on harmful dependency conflicts.

Definition 1. (Dependency Conflict). Let c_1 and c_2 be any two classes specified on the classpath. If specifying these two classes with different orders on the classpath can lead to different program behaviors, we define it as a dependency conflict.

According to Java's class loading mechanism, a classpath determines the locations where to find the required classes during runtime. If multiple classes with the same fully-qualified name are specified on the classpath, only one of them will be loaded based on the loading mechanism of the build tool (i.e., as revealed in our empirical study) and the others will be shadowed. To detect dependency conflict issues in practice, we first need to identify those classes C_m that have multiple versions specified on the classpath. In order to assess the severities of the detected DC issues, we need to analyze those features defined in these duplicate classes as well as the features referenced by the host project as revealed by our empirical study in Section 3. Therefore, for each of the class $c_i \in C_m$, we further introduce the following concepts:

Definition 2. (Duplicate Class Set). Suppose there are m ($m > 1$) versions of class c_i declared on classpath. We define the duplicate class set as $\mathcal{D}_i = \{c_{ij} | m > 1, 1 \leq j \leq m\}$, where c_{ij} represents the j-th version of class c_i.

Definition 3. (Referenced Feature Set). Let $\mathcal{RH}$ be the feature set directly or indirectly referenced by the host classes $\mathcal{H}$. We denote the feature set in $\mathcal{D}_i$ as $\mathcal{RD}_i = \cup_{\forall c_{ij} \in \mathcal{D}_i} f(c_{ij})$. For $c_i \in C_m$, the referenced feature set by the host classes is $\mathcal{R}_i = \mathcal{RH} \cap \mathcal{RD}_i$.

Definition 4. (Loaded Feature Set). Suppose $c_{il} \in \mathcal{D}_i$ is the actual loaded version of class c_i. We then define the actual loaded feature set as $\mathcal{L}_i = f(c_{il})$.

Definition 5. (Shadowed Feature Set). Suppose $c_{il} \in \mathcal{D}_i$ is the actual loaded version of class c_i. We then define the feature set $\mathcal{S}_i = \cup_{\forall c_{ij} \in \mathcal{D}_i \setminus c_{il}} f(c_{ij})$ as the shadowed feature set.

For each $c_i \in C_m$, we can obtain the above four different sets. Based on the observations from the empirical study, A DC issue arises when the actual loaded feature set $\mathcal{L}_i$ does not subsume the referenced feature set $\mathcal{R}_i$. Therefore, we say that there is a DC issue for p if the following condition satisfies:

$$\mathcal{R}_i \nsubseteq \mathcal{L}_i, \exists c_i \in C_m \quad (1)$$

4.2 Overview

Based on our previous observations, we propose a static analysis technique involving four steps as shown in Figure 4. First, it extracts the library dependency tree by analyzing the library dependency management script (e.g, `pom.xml, build.gradle`). Second, it identifies duplicate libraries or classes based on the dependency tree and bytecode (JAR or class files). Third, it deduces the *loaded*, *shadowed* feature set based on the class loading rules of build tools. In our study, we focus on the `Maven` class loading mechanisms observed in the empirical study. However, our approach can be easily generalized to other build environment, by adapting the corresponding class loading rules. Furthermore, the *referenced* feature sets can be obtained via static analysis. Finally, it detects DC issues based on the deduced feature sets and assesses their severity levels according to their impacts on the system and maintenance costs.

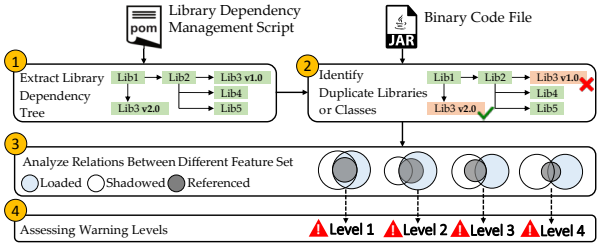


Figure 4: Architecture of DECCA

4.3 Detecting DC Issues

According to our problem formulation in Section 4.1, the key to detect DC issues is to identify the *duplicate*, *referenced*, *loaded* and *shadowed* feature sets first, and then to check whether the condition of the root cause formulated in Equation 1 is satisfied.

The findings of our empirical study towards the manifestation patterns enable us to identify the loaded and shadowed feature sets automatically. First, according to **Pattern A**, if different versions of same libraries are in one project, the version that appears nearest to the host project in the dependency tree will shadow the others and make them unavailable. Second, if different libraries contain the same classes, then the classes included in the first declared JAR will shadow the others, based on **Pattern B**. Finally, as dependency analysis scope of `Maven` is limited to the libraries listed within its `pom.xml`, if host project and the libraries include the same classes, then the ones in the libraries will be loaded refers to **Pattern C**.

Identifying the duplicate classes is straightforward since we only need to check whether a class with the same fully-qualified name has appeared on the classpath for multiple times. However, when multiple versions of classes coexist in the project, one key problem is to detect the referenced feature set $\mathcal{R}_i$.

To address this problem, we designed an algorithm, which takes the following two parts as inputs: (1) duplicate class set $\mathcal{D}_i$, and (2) dependency management script of the project under analysis. The output of the algorithm is the referenced feature set $\mathcal{R}_i$. Specifically, the algorithm contains the following steps:

- Initializes the referenced feature set $\mathcal{R}_i$ as an empty collection.
- Finds the path set $\mathcal{P}_i$ from the host project to each library l_i that contains duplicate class $c_i \in \mathcal{D}_i$ on the dependency tree LT.
- Extracts the reference relationships based on class names using static analysis.
- Analyzes the libraries on each path $p_t \in \mathcal{P}_i$ separately, since static analysis does not know which version is the referenced version if different versions of the same class coexist in bytecode.
- Performs the following tasks for each path $p_t \in \mathcal{P}_i$: (1) Identifies the boundary feature set $\mathcal{B}_s$. Boundary feature set $\mathcal{B}_s$ represents the feature set defined in the host project that directly depends on host project's succeeding library on path p_t [87], where host project's succeeding library is the library after the host project on path p_t. (2) Identifies the boundary feature set $\mathcal{B}_e$ defined in library l_i and directly depended by the preceding library of l_i on path p_t, where the preceding library of l_i is the library before l_i on path p_t . (3) For each method $m_t \in \mathcal{B}_s$, if m_t directly or transitively depends on $m_k \in \mathcal{B}_e$, performs $\mathcal{R}_i \leftarrow \mathcal{R}_i \cup \mathcal{M}_k \cup m_k$, where $\mathcal{M}_k$ is the feature set used by m_k.

Decca is implemented as a `Maven` plugin based on `Soot` framework. It leverages `Soot`'s program dependency graph and call graph APIs to identify the referenced feature set. It is well-known that statically constructing sound and complete call graphs and program dependence graphs for Java language is challenging due to the language features such as dynamic binding and reflections [73, 90]. This factor affects Decca's detection precision (See Section 5.1).

4.4 Assessing DC Severity Levels

`Maven` detects duplicate instances as warnings at the library or class level, which is coarse-grained. Besides, it generates all DC warnings without distinguishing their severity levels. If the referenced feature set defined in the duplicate classes are compatible and consistent, they have no serious effects on the host project at runtime. Therefore, it is difficult for developers to distinguish benign warnings from harmful ones (e.g., causing runtime exceptions) if the generated warnings do not provide extra severity information. Decca addresses this issue in two steps. First, it analyzes the host project and the third-party libraries at a finer granularity, which is at the feature level, to assess whether those duplicate instances are harmful or not driven by our findings on the manifestation patterns. Second, it estimates the maintenance efforts required by the detected issues driven by our findings on the fixing solutions to further augment the issues with severity levels. Then, Decca will assign a DC issue to one of the four severity levels, which are defined as follows.

Level 1: $\mathcal{R}_i \subseteq \mathcal{L}_i$ and $\mathcal{R}_i \subseteq \mathcal{S}_i$. In this case, the referenced feature set is a subset of the actual loaded feature set. Besides, the shadowed version completely covers the feature set used by the host project. This indicates that any orders of the specification of these duplicate classes on the classpath will not induce serious runtime errors. Therefore, this is a benign conflict and will not affect the system reliability at runtime.

Level 2: $\mathcal{R}_i \subseteq \mathcal{L}_i$ and $\mathcal{R}_i \nsubseteq \mathcal{S}_i$. In this case, the referenced feature set is a subset of the actual loaded feature set. However, the shadowed feature set does not cover the referenced feature set. It is considered as a potential risk for system reliability since different orders of the specifications of these duplicate classes on the classpath (e.g., in different running environment or building platform) might induce runtime errors.

Level 3: $\mathcal{R}_i \nsubseteq \mathcal{L}_i$ and $\mathcal{R}_i \subseteq \mathcal{S}_i$. It is a harmful conflict, as the actual loaded feature set does not consume the referenced feature set. The runtime errors will occur when the expected feature cannot be accessed. However, in this case, the shadowed feature set completely cover the feature set referenced by host project. Therefore, it can be solved by adjusting the dependency order on the classpath, without changing any source code.

Level 4: $\mathcal{R}_i \nsubseteq \mathcal{L}_i$ and $\mathcal{R}_i \nsubseteq \mathcal{S}_i$. It is a harmful conflict, as the actual loaded feature set does not cover the referenced feature set. Besides, the shadowed feature set does not consume the referenced feature set neither. Therefore, this type of conflicts can not be easily resolved by adjusting the dependency orders on the classpath. To solve these issues, it requires more efforts to ensure the multiple versions of classes could be referenced by the host project.

DC issues detected as **Level 1** and **Level 2** are benign ones since they will not cause system failures in the current version. However, those issues at **Level 2** might cause runtime errors potentially if the dependency orders have been changed. DC issues detected as **Level 3** and **Level 4** are harmful ones as they can lead to system crashes under the current configurations. These severity levels provide developers with a deeper understanding of the detected DC issues, and guidance to fix these bugs. For instance, DC issues at **Level 2** can be potentially avoided by fixing **Pattern 3** (harmonizing library versions) and **Pattern 2** (adjusting the classpath order of dependencies) can be used as the workaround of the **Level 3** DC issues. However, DC issues at **Level 4** require more diagnosis efforts since they need to be solved by fixing **Pattern 1** (shading the conflicting libraries) or **Pattern 4** (classloader customization).

5 EVALUATION

We evaluate the effectiveness and usefulness of Decca using real-world open source projects against two research questions:

- **RQ3 (Effectiveness):** How effective can Decca detect real DC issues and assess their severity levels?
- **RQ4 (Usefulness):** Can Decca detect unknown DC issues in real-world projects and facilitate developers in diagnosing them?

To study RQ3, we first collected a high quality dataset which contains high-severity (i.e., **Level 3** and **4**) and low-severity (i.e., **Level 1** and **2**) DC issues. We then apply Decca to this dataset to see if DC issues with high-severity can be detected.

To study RQ4, we randomly selected 30 projects from `Github` and built their latest releases on `Maven` platform. Then, we applied Decca to these projects to detect unknown DC issues and assess their severity levels. Especially, we filtered out the **Level 1** issues, which will not lead to the system failures no matter how the classpath order of dependencies is changed. For the identified issues with **Levels 2**, **3** and **4** severities, we reported them to developers using the corresponding bug tracking systems and evaluated the usefulness of Decca based on developers' feedbacks.

5.1 RQ3: Effectiveness of Decca

Data collection. We constructed the dataset from the 2,289 Java projects as mentioned in Section 2.1. Decca can detect DC issues in four severity levels. However, it is difficult to collect the ground truth about the exact severity levels of DC issues. In practice, developers often consider DC issues in severity at two levels: (1) high-severity issues (include **Level 3** and **4** DC issues), which will cause runtime exceptions or errors; and (2) low-severity issues (include **Level 1** and **2** DC issues), which should not cause unexpected behaviors, exceptions or errors if the project and dependent libraries remain unchanged. This categorization information would be easily identified from bug reports or log message of commits. Therefore, this evaluation mainly focus on whether Decca can distinguish high-severity issues from low-severity ones. To guarantee the quality of labeling high-severity and low-severity issues, we chose the fifth latest release of each project, since we assumed that the time interval from fifth latest release to now is sufficient for developers to diagnose DC issues and give solutions (fix or not). We constructed the dataset of high-severity issues by selecting from the fixed DC issues, since developers typically assigned severe issues with higher priority to fix. However, it does not mean that all fixed issues must be high-severity since developers might also fix low-severity ones to avoid potential issues arise in future maintenance (e.g., DC issues at **Level 2**). Therefore, we use the following criterion to select high-severity issues: (1) the fixing patch is linked to a bug report which records a runtime exception or compilation error; or (2) the commit log message explicitly mentioned that it fixed a harmful DC issue (i.e., causing runtime exceptions or compilation error). We collect the dataset of low-severity DC issues from the unfixed ones. However, not being fixed so far does not necessarily mean that issue is low severity since it might take developers a lot of time to diagnose the issue. It has been found that bugs are usually repaired within 1 to 2 years across different projects since they were introduced to the project [78]. Therefore, we only select those DC issues that have not been fixed for the next 24 months as low-severity ones. The other DC issues are ignored in our evaluation to avoid introducing noises, since we do not have high-confidence to judge whether their severities are high or low. In this manner, we collected 47 high-severity and 172 low-severity DC issues as the ground truth dataset. The data collection process involves two graduate students: one identifying the high-severity and low-severity DC instances and the other verifying the results.

Metrics. We use *Recall*, *Precision*, and *F-measure* to evaluate the performance of Decca, which are defined by the following metrics:

True Positive (TP): the conflict identified as a high-severity issue (i.e., **Level 3** or **Level 4**) is a high-severity issue.

False Positive (FP): the conflict identified as a high-severity issue (i.e., **Level 3** or **Level 4**) is a low-severity issue.

True Negative (TN): the conflict identified as a low-severity issue (i.e., **Level 1** or **Level 2**) is a low-severity issue.

False Negative (FN): the conflict identified as a low-severity issue (i.e., **Level 1** or **Level 2**) is a high-severity issue.

Based on the above four metrics, we can obtain the *Recall*, *Precision*, and *F-measure* as follows:

$$Precision = TP/(TP + FP) \quad (2)$$

$$Recall = TP/(TP + FN) \quad (3)$$

$$F\text{-}measure = 2 \times Precision \times Recall/(Precision + Recall) \quad (4)$$

Precision evaluates whether Decca can detect high-severity issues precisely. *Recall* evaluates the capability of Decca in detecting all the high-severity issues. *F-measure* combines the *Precision* and *Recall* together [100].

Results. The experimental results show that Decca identified 39 high-severity DC instances with a *Precision* of 0.923, a *Recall* of 0.766 and a *F-measure* of 0.837. Based on the results, we can conclude that Decca can effectively detect the DC issues with severity levels. We further investigated the reason why Decca generated false positive and negative cases of Decca. We found that, this is mainly because static analysis cannot accurately deal with the Java language features such as dynamic method dispatching (virtual function invocations) and reflections [73, 90]. For instance, project `Apache/Metamodel` introduced conflicting JARs `Httpclient` 4.4.1 and 4.5.2, and version 4.4.1 shadowed 4.5.2. By static analysis, host project referenced to 423 features of library `Httpclient`, but the loaded version 4.4.1 only defined 378 of them. Decca assesses this DC issue as **Level 4** severity. However, this is a false positive. By further checking, we found that these uncovered 45 (i.e., 423 − 378) features will never be executed at runtime. This discrepancy is mainly because we adopt conservative static analysis to construct call graph and include some false call edges in the virtual function call sites. In another example, libraries `xercesImpl` and `xml-apis` both included class `SingleSignOnFactory`, which caused a DC issue in project `Apache/Oodt`. Based on Decca' detection, the loaded class defined in `xml-apis` could fully cover the referenced feature set. As a result, our tool assigned **Level 1** severity to this instance. In fact, Decca ignored a case that a method in the shadowed class was used by host project via reflection mechanism, which leaded to a false negative warning.

5.2 RQ4: Usefulness Of Decca

Decca successfully identified 466 DC issues from 24 projects among all the 30 projects analyzed. Among these instances, 438 (93.9%) of them are at **Level 1**, 20 (4.2%) of them are at **Level 2**, 4 (0.08%) of them are at **Level 3**, and 4 (0.08%) of them are at **Level 4**. After filtering out the benign issues with **Level 1**, we reported issues with **Level 2**, **3** and **4** to their corresponding bug tracking systems. Our bug report includes the following information:

(1) Severity: concerning the severity options of bug tracking system, if the detected DC instances are assessed as **Level 1** or **Level 2**, we labeled them as "`minor`" issues, otherwise as "`major`" issues. Note that, we filter out most of the **Level 1** issues. However, to confirm our expectation in handling **Level 1** issues, we randomly sampled four issues and reported them to developers.

(2) Root cause: we listed the library pairs including duplicated classes, or different versions of the same libraries in the project. More importantly, we provided the differences between the feature set of the actually loaded classes and that of the referenced ones.

(3) Fixing suggestions: according to our findings of empirical study (Section 3.3), developers prefer to adopt fixing **Pattern 2** (adjusting the classpath order of dependencies) and **Pattern 3** (harmonizing library versions) to solve DC issues as they requires less efforts. Therefore, for the DC issues with **Level 2** and **Level 3** severities, we suggested developers to fix the issues by harmonizing library versions or adjusting the classpath dependency order

Table 3: Experimental subjects and checking results

ID	Project	Category	Revision	Size (LOC)	Star	Fork	Availability	Severity level				Bug ID
								L1	L2	L3	L4	
1	Spark [54]	Big data	8077bb0	130.0k	16262	15050	Apache/Github	40	1	0	0	SPARK-23509♢
2	Beam [13]	Big data	a750128	337.0k	1722	1038	Apache/Github	17	2	0	0	BEAM-3690♢
3	Bahir [11]	Extension tool	6ea42a8	0.9k	152	102	Apache/Github	22	0	1	1	BAHIR-159♣
4	Wicketstuff/Core [63]	Container	5cc41f5	228.5k	314	293	Apache/Github	16	1	0	0	Issue #621♢
5	Javasoze clue [38]	Command	23c9da4	2.8k	103	37	Github	18	1	0	0	Issue #61
6	ActiveMQ Artemis [8]	Network server	f6c5408	557.8k	271	326	Apache/Github	24	0	0	0	-
7	Apex Core [6]	Platform	4fb580f	87.0k	277	161	Apache/Github	34	0	0	0	-
8	Ignite [33]	OSGI	4e86660	2218.4k	1505	848	Apache/Github	7	0	0	0	-
9	Wicket [62]	Web framework	b728c69	352.5k	412	293	Apache/Github	2	0	0	0	-
10	Google/Closure-Compiler [25]	JS compiler	900251b	427.6k	4005	777	Github	4	1	0	0	Issue #2815♠
11	Orientdb [47]	Database	56ab1ac	496.3k	3366	718	Github	8	0	0	1	Issue #8111♠
12	Cm [18]	Web application	9e6f45b	19.k	12	6	Github	5	0	0	1	Issue #1♢
13	Brooklyn [15]	Cloud	48dbcc3	276.1k	69	47	Apache/Github	20	0	1	0	BROOKLYN-581
14	CarbonData [17]	Big data	9f2884a	127.9k	612	391	Apache/Github	25	4	0	0	CARBONDATA-2169
15	Prestodb [53]	Big data	89fed3a	0.8k	15	22	Github	16	1	0	0	Issue #29
16	Solr [57]	Network Server	d32048c	31.7k	295	207	Github	10	1	0	0	DATASOLR-447
17	tomcat exporter [60]	Exporter	70ac377	0.9k	19	10	Apache/Github	10	2	0	0	Issue #8
18	Hadoop Common [27]	Database	1e85a99	2042.8.k	5883	3987	Apache/Github	16	0	0	1	HADOOP-15261♢
19	Oozie [45]	Big data	9e662c7	198.6k	364	327	Apache/Github	25	0	1	0	OOZIE-3185♠
20	Accumulo [1]	Database	d98843b	563.8k	343	197	Apache/Github	33	1	0	0	ACCUMULO-4812♣
21	Eclipse jetty [22]	Debugging	b71cd70	375.9k	1868	1134	Github	6	2	0	0	Issue #2232
22	Parquet [50]	Big data	b82d962	0.9k	550	465	Apache/Github	2	1	0	0	PARQUET-1236♢
23	Apex Malhar [6]	Big data	0d98d05	243.7k	110	149	Apache/Github	34	1	0	0	APEXMALHAR-2556
24	Atlas [10]	Framework	6770091	123.4k	33	32	Apache/Github	44	1	1	0	ATLAS-2437

♢:The issues have already been fixed. ♠:The issues were confirmed and being fixed in process.
♣:They have been confirmed as DC issues, and they are suggested to be solved by the upstream third party libraries.

based on their specific scenarios. For **Level 4** issues, we suggested them to use fixing **Pattern 1** (shading the conflicting libraries) to work around these problems.

Altogether, we reported 20 DC bugs including 28 issues (issues in one project were combined into one bug report). As shown in Table 3, 11 bugs (55%) were confirmed by developers as real issues within a few days. 6 out of the 11 confirmed bugs (55%) were quickly fixed using our suggestions, 3 confirmed bugs (30%) are in the process of being fixed, and the other 2 confirmed bugs are to be resolved by the developers of upstream third party libraries. By further analysis, we found that those non-severe issues (**Level 2**) have a low confirmation rate (30%). Meanwhile, the severe issues (**Level 3 and 4**) have a higher confirmation rate (75%), which is within our expectation. The 10 unconfirmed issues are mainly due to the inactive maintenance of the corresponding release versions.

5.2.1 Feedback on Reported Issues. For the 6 fixed issues, the developers agreed that the detected conflicts could bring risks to software reliability or maintainability, and they also invited us to upload patches to resolve the issue. In particular, 2 out of the 6 issues (i.e., `HADOOP-15261` [29] and `Issue #1` [34]) are detected as **Level 4** by Decca. The quick feedbacks from the corresponding developers are within our expectations since these issues are very serious and can cause runtime errors. The remaining four are detected as **Level 2** (i.e., `BEAM-3690` [14], `Issue #621` [36], `SPARK-23509` [55] and `PARQUET-1236` [51]). Although these issues might not cause runtime errors at the moment, to avoid potential errors in the future, developers still fixed them after reviewing the different feature sets between the conflicting JARs.

Three issues (i.e., `Issue #2815` [35], `Issue #8111` [37] and `OOZIE-3185` [46]) have been confirmed and are being fixed in process up till now. For example, `Google Closure-Compiler` library contains library `com.google.code.gson`. However, its downstream project `Wicketstuff Core` also includes the library `com.google.code.gson` of a newer version, which introduces a dependency conflict. This issue was detected as **Level 2** DC issue. We reported this issue to `Google Closure-Compiler` as `Issue #2815`, and it was confirmed and supported by the developers:

"*I'm encountering this problem as well. Is there a way to build the closure compiler JAR without any dependencies packaged in it?*"

Two DC issues were confirmed, but developers considered that they should be solved by its upstream third party libraries (i.e., `BAHIR-159` [12] and `ACCUMULO-4812` [2]). For example, developers explained in `ACCUMULO-4812`:

"*The conflicting JARs are being brought in through `Hadoop` or `ZooKeeper`'s classpath, and should be addressed by their packaging.*"

In addition, we randomly selected four DC issues at **Level 1** to report (e.g., `ARTEMIS-1674` [9] and `APEXCORE-805` [7]), since we would like to see whether developers care about these issues. As we expected, the developers would not like to revise their source code urgently until they cause failures, although they acknowledged these issues. These results indicate that Decca can help developers filter out benign DC issues and highlight the harmful ones.

5.2.2 Feedback on DECCA. Besides confirming our reported issues, several developers expressed interests in our tool Decca, which is encouraging. For example, we received the following feedback in `BEAM-3690`:

"*This seems like a handy report, is the tool you used to identify this error open source? I am curious to give it a try (also for other stuff).*"

"*Related, but not the same: I have tried turning on dependency convergence in the `Maven-enforcer-plugin`. We need the same for gradle to ensure long-term health and protect from regressions. Maybe the tool that generated this fine-grained conflicts report can also fail the build? That would be nice.*"

The above comments are made from experienced developers [4] in the `Apache Beam` community. Currently, existing plugins, such as `Maven-enforcer-plugin` [42] aforementioned can identify DC issues. However, Decca still attracts developers' attention. The feedbacks indicate that when reporting DC issues, it is crucial to

provide developers with severity levels and more-detailed conflicting information. Our tool Decca is able to achieve this. The above results and feedbacks from developers show that the information (e.g., severity levels) provided by Decca is useful for developers to diagnose the DC issues in practice.

6 THREATS TO VALIDITY

Our study is subject to the following major threats to validity.

DC issue selection. The DC issue selection may threaten the validity of our empirical study results since the keyword searching strategy can introduce noises. To reduce this threat, we manually inspected, re-examined and cross-validated all the collected DC issues independently to assure the data quality.

Ground truth dataset collection for evaluation. Collecting the ground truth dataset is challenging and can be a threat to the evaluation results. To avoid introducing noises in our evaluation dataset, we collected the high-severity set from those DC issues that been fixed in the subsequent releases. Moreover, we only keep those instances that the fixing patches are linked with bug reports or the fixing commit logs explicitly mention them as harmful DC issues. We collected the low-severity set from those DC issues that have not been fixed in their subsequent releases and no related DC issue reported in their projects during the next 24 months.

Limitation for detecting diverse types of DC issues. This paper focuses on crash DC issues due to referencing the shadowed features or classes. However, in some cases, the conflict could be caused by the changes in semantics, performance or other attributes of the duplicated libraries presented on the classpath. `Decca` does not analyze the above manifestations of DC issue, which may affect the validity of detection.

7 RELATED WORK

Library Evolution: Third-party libraries keep evolving in order to fix previous bugs, add new features and etc. Bavota et al. analyzed the evolution of the Java libraries of the `Apache` ecosystem, consisting of 147 projects, for a period of 14 years [68, 69]. Specifically, they analyzed how upgrades of a project will affect related projects and what reasons will drive a host project to upgrade its dependencies. Businge et al. analyzed the evolution of the `Eclipse` third-party plugins [70]. In particular, they studied the source compatibilities between the third-party plug-ins and the Eclipse SDK releases. The evolving of a library is embodied in the evolution of its defined features (i.e., APIs) at a finer granularity. Therefore, many researches focused on studying API stability during library evolution [81, 84, 91, 93, 94]. Specifically, Robbes et al. investigated how developers take actions to API evolutions of third-party libraries [93]. Due to the fast evolution of libraries, different versions of a library might be incompatible. Raemaeker et al. analyzed the relationship between version numbers and the binary compatibilities of third-party libraries [92]. By empirical study, they found that features defined in an older version are often unavailable in a newer version. As a result, dependencies conflicts might occur due to such incompatibilities if multiple versions are included in one project. However, no existing works focus on the manifestations and diagnosis of dependencies conflict issues.

Library Analysis: It is important for a host project to manage its depended libraries since they are evolving. Musco et al. proposed an approach to model the relationships between software system and third-party libraries to better understand software evolution [87]. In their study, they introduced the concept of software system boundary [87], which is also adopted in our paper to identify the feature set referenced by host project. Annosi et al. proposed a framework to facilitate developers in upgrading third-party libraries [67]. Since upgrading libraries will inevitably affect existing framework, their approach is designed mainly based on risk management. Ouni et al. proposed an approach called LibFinder [88], which helps developers find third-party libraries in developing. Kikas et al. analyzed the dependency network structures for multiple languages [77] and found that many third-party libraries are transitively introduced to a project. Therefore, they suggest that developers should look more carefully when using a third-party library to understand what dependencies are exactly included. Kula et al. proposed an visualization tool to investigate the history of library update [80] using the statistics of dependencies extracted from the Maven repository. By leveraging their tool, developers can easily identify outdated libraries. Yano et al. also proposed an visualization tool to investigate the popular coupling usages of third-party libraries [101]. They found that library `Http-Client 3.1` and `Collections 3.2.1` are frequently used together. The coupling relations revealed by their studies can help developers reduce the risks of using incompatible libraries. However, none of the above studies investigated the dependency conflict issues caused by the incompatibilities of third-party libraries, which is well investigated in this study.

8 CONCLUSION AND FUTURE WORK

In this paper, we first conducted an empirical study to understand and characterize DC issues between host project and third-party libraries. We investigated 135 real DC issues collected from 71 Java projects of `Apache` ecosystem to understand their common manifestations and fixing strategies. Based on our empirical findings, we formulate the dependency conflict problem and its root cause. Furthermore, we designed and implemented an automated technique Decca to detect DC issues and assess their severity levels. The evaluation results show that Decca can achieve a *Precision* of 0.923, a *Recall* of 0.766, and a *F-measure* of 0.837. We also applied Decca to detecting new DC issues on the latest version of other large Java projects. Encouragingly, developers confirmed and fixed the reported bugs detected by Decca. They also showed great interests in our proposed tool. These feedbacks from developers demonstrate the practical usefulness of Decca.

In future, we plan to design effective techniques to help developers automatically repair DC issues. Another plan is to extend our approach to other build frameworks such as `Gradle`.

ACKNOWLEDGMENTS

The authors express thanks to the anonymous reviewers and HKUST CASTLE members for their constructive comments. Part of the work was conducted during the first author's internship at HKUST in 2018. The work is supported by the Hong Kong RGC/GRF grant 16202917, MSRA grant, the National Natural Science Foundation of China (Grant Nos. 61374178, 61603082 and 61402092) and the Fundamental Research Funds for the Central Universities (Grant No. N171704004).

REFERENCES

[1] 2018. Accumulo. https://accumulo.apache.org/. Accessed: 2018-02-28.
[2] 2018. ACCUMULO 4812. https://issues.apache.org/jira/browse/ACCUMULO-4812. Accessed: 2018-02-28.
[3] 2018. Apache. http://www.apache.org/. Accessed: 2018-02-28.
[4] 2018. Apache contributors. https://github.com/apache/beam/graphs/contributors. Accessed: 2018-02-28.
[5] 2018. Apache project category. https://projects.apache.org/projects.html?category. Accessed: 2018-02-28.
[6] 2018. Apex. http://apex.apache.org/. Accessed: 2018-01-12.
[7] 2018. APEXCORE 805. https://issues.apache.org/jira/browse/APEXCORE-805. Accessed: 2018-02-28.
[8] 2018. Artemis. https://activemq.apache.org/artemis/. Accessed: 2018-01-12.
[9] 2018. ARTEMIS 1674. https://issues.apache.org/jira/browse/ARTEMIS-1674. Accessed: 2018-02-28.
[10] 2018. Atlas. https://atlas.apache.org/. Accessed: 2018-01-12.
[11] 2018. Bahir. http://bahir.apache.org/. Accessed: 2018-02-28.
[12] 2018. BAHIR 159. https://issues.apache.org/jira/browse/BAHIR-159. Accessed: 2018-02-28.
[13] 2018. Beam. https://beam.apache.org/. Accessed: 2018-02-28.
[14] 2018. BEAM 3690. https://issues.apache.org/jira/browse/BEAM-3690. Accessed: 2018-02-28.
[15] 2018. Brooklyn. https://brooklyn.apache.org/. Accessed: 2018-01-12.
[16] 2018. Bugzilla. https://www.bugzilla.org/. Accessed: 2018-02-28.
[17] 2018. Carbondata. https://carbondata.apache.org/. Accessed: 2018-01-12.
[18] 2018. Cm. https://github.com/Jibesh97/cm. Accessed: 2018-01-12.
[19] 2018. CURATOR-200. https://issues.apache.org/jira/browse/CURATOR-200. Accessed: 2018-02-28.
[20] 2018. CXF 5132. https://issues.apache.org/jira/browse/CXF-5132. Accessed: 2018-02-28.
[21] 2018. DERBY-5429. https://issues.apache.org/jira/browse/DERBY-5429. Accessed: 2018-02-28.
[22] 2018. Eclipse jetty. https://www.eclipse.org/jetty/. Accessed: 2018-01-12.
[23] 2018. Git. http://git-scm.com/. Accessed: 2018-02-28.
[24] 2018. Github. https://github.com/. Accessed: 2018-02-28.
[25] 2018. Google closure compiler. https://developers.google.com/closure/compiler/. Accessed: 2018-02-28.
[26] 2018. Gradle. https://gradle.org/. Accessed: 2018-02-28.
[27] 2018. Hadoop. http://hadoop.apache.org/. Accessed: 2018-02-28.
[28] 2018. HADOOP-11656. https://issues.apache.org/jira/browse/HADOOP-11656. Accessed: 2018-02-28.
[29] 2018. HADOOP 15261. https://issues.apache.org/jira/browse/HADOOP-15261. Accessed: 2018-02-28.
[30] 2018. HADOOP 8104. https://issues.apache.org/jira/browse/HADOOP-8104. Accessed: 2018-02-28.
[31] 2018. HADOOP7606. https://issues.apache.org/jira/browse/HADOOP-7606. Accessed: 2018-02-28.
[32] 2018. HDFS 10570. https://issues.apache.org/jira/browse/HDFS-10570. Accessed: 2018-02-28.
[33] 2018. Ignite. https://ignite.apache.org/. Accessed: 2018-01-12.
[34] 2018. Issues #1. https://github.com/Jibesh97/cm/issues/1. Accessed: 2018-02-28.
[35] 2018. Issues #2815. https://github.com/google/closure-compiler/issues/2815. Accessed: 2018-02-28.
[36] 2018. Issues 621. https://github.com/wicketstuff/core/issues/621. Accessed: 2018-02-28.
[37] 2018. Issues #8111. https://github.com/orientechnologies/orientdb/issues/8111. Accessed: 2018-02-28.
[38] 2018. Javasoze clue. https://github.com/javasoze/clue/. Accessed: 2018-01-12.
[39] 2018. Jira. https://www.atlassian.com/software/jira/. Accessed: 2018-02-28.
[40] 2018. Maven classpath. https://maven.apache.org/shared/maven-archiver/examples/classpath.html. Accessed: 2018-02-28.
[41] 2018. Maven Dependency Plugin. http://maven.apache.org/components/plugins/maven-dependency-plugin/. Accessed: 2018-02-28.
[42] 2018. Maven enforcer plugin. http://maven.apache.org/enforcer/maven-enforcer-plugin/. Accessed: 2018-02-28.
[43] 2018. Maven repository. https://maven.apache.org/. Accessed: 2018-02-28.
[44] 2018. Maven shade plugin. http://maven.apache.org/plugins/maven-shade-plugin/. Accessed: 2018-02-28.
[45] 2018. Oozie. http://oozie.apache.org/. Accessed: 2018-02-28.
[46] 2018. OOZIE 3185. https://issues.apache.org/jira/browse/OOZIE-3185. Accessed: 2018-02-28.
[47] 2018. Orientdb. https://orientdb.com/why-orientdb/. Accessed: 2018-01-12.
[48] 2018. OSGI. https://www.osgi.org/. Accessed: 2018-02-28.
[49] 2018. OSGI classloaders. http://moi.vonos.net/java/osgi-classloaders/. Accessed: 2018-02-28.
[50] 2018. Parquet. https://parquet.apache.org/. Accessed: 2018-01-12.
[51] 2018. PARQUET1236. https://issues.apache.org/jira/browse/PARQUET-1236. Accessed: 2018-02-28.
[52] 2018. POM reference. https://maven.apache.org/pom.html/. Accessed: 2018-02-28.
[53] 2018. Prestodb. https://prestodb.io/. Accessed: 2018-01-12.
[54] 2018. Spark. http://spark.apache.org/. Accessed: 2018-02-28.
[55] 2018. SPARK 23509. https://issues.apache.org/jira/browse/SPARK-23509. Accessed: 2018-02-28.
[56] 2018. SPARK 2848. https://issues.apache.org/jira/browse/SPARK-2848. Accessed: 2018-02-28.
[57] 2018. Spring data solr. http://projects.spring.io/spring-data-solr/. Accessed: 2018-01-12.
[58] 2018. STORM2382. https://issues.apache.org/jira/browse/STORM-2382. Accessed: 2018-02-28.
[59] 2018. SUREFIRE 851. https://issues.apache.org/jira/browse/SUREFIRE-851. Accessed: 2018-02-28.
[60] 2018. Tomcat exporter. https://github.com/nlighten/tomcat_exporter. Accessed: 2018-01-12.
[61] 2018. Uber JAR. https://imagej.net/Uber-JAR. Accessed: 2018-02-28.
[62] 2018. wicket. https://wicket.apache.org/. Accessed: 2018-01-12.
[63] 2018. Wicketstuff. http://wicketstuff.org/. Accessed: 2018-02-28.
[64] 2018. Wildfly. http://wildfly.org/. Accessed: 2018-02-28.
[65] 2018. YARN 5271. https://issues.apache.org/jira/browse/YARN-5271. Accessed: 2018-02-28.
[66] 2018. YARN 6414. https://issues.apache.org/jira/browse/YARN-6414/. Accessed: 2018-02-28.
[67] Maria Carmela Annosi, Massimiliano Di Penta, and Genny Tortora. 2012. Managing and assessing the risk of component upgrades. In *Product Line Approaches in Software Engineering (PLEASE), 2012 3rd International Workshop on*. IEEE, 9–12.
[68] 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 *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE, 280–289.
[69] 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.
[70] John Businge, Alexander Serebrenik, and Mark van den Brand. 2012. Survival of Eclipse third-party plug-ins. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 368–377.
[71] Nicolas Geoffray, Gaël Thomas, Charles Clément, and Bertil Folliot. 2008. A lazy developer approach: Building a JVM with third party software. In *Proceedings of the 6th international symposium on Principles and practice of programming in Java*. ACM, 73–82.
[72] James Gosling. 2000. *The Java language specification*. Addison-Wesley Professional.
[73] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. 1997. Call graph construction in object-oriented languages. *ACM SIGPLAN Notices* 32, 10 (1997), 108–124.
[74] Richard S Hall and Humberto Cervantes. 2004. An OSGi implementation and experience report. In *Consumer Communications and Networking Conference, 2004. CCNC 2004. First IEEE*. IEEE, 394–399.
[75] Hubert Klein Ikkink. 2015. *Gradle Dependency Management*. Packt Publishing Ltd.
[76] Sascha Just, Rahul Premraj, and Thomas Zimmermann. 2008. Towards the next generation of bug tracking systems. In *Visual languages and Human-Centric computing, 2008. VL/HCC 2008. IEEE symposium on*. IEEE, 82–85.
[77] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. 2017. Structure and evolution of package dependency networks. In *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*. IEEE, 102–112.
[78] Sunghun Kim and E James Whitehead Jr. 2006. How long did it take to fix bugs?. In *Proceedings of the 2006 international workshop on Mining software repositories*. ACM, 173–174.
[79] Andrew J Ko, Brad A Myers, and Duen Horng Chau. 2006. A linguistic analysis of how people describe software problems. In *Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on*. IEEE, 127–134.
[80] Raula Gaikovina Kula, Coen De Roover, Daniel German, Takashi Ishio, and Katsuro Inoue. 2014. Visualizing the evolution of systems and their library dependencies. In *Software Visualization (VISSOFT), 2014 Second IEEE Working Conference on*. IEEE, 127–136.
[81] Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. 2011. Large-scale, AST-based API-usage analysis of open-source Java projects. In *Proceedings of the 2011 ACM Symposium on Applied Computing*. ACM, 1317–1324.
[82] Sheng Liang and Gilad Bracha. 1998. Dynamic class loading in the Java virtual machine. *Acm sigplan notices* 33, 10 (1998), 36–44.
[83] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and detecting performance bugs for smartphone applications. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. 1013–1024.

[84] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An empirical study of api stability and adoption in the android ecosystem. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE, 70–79.

[85] Wen Ming, Chen Junjie, Wu Rongxin, Hao Dan, and Cheung Shing-Chi. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering (ICSE 2016)*.

[86] Audris Mockus, Roy T Fielding, and James D Herbsleb. 2002. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11, 3 (2002), 309–346.

[87] Vincenzo Musco, Martin Monperrus, and Philippe Preux. 2014. A Generative Model of Software Dependency Graphs to Better Understand Software Evolution. *arXiv preprint arXiv:1410.7921* (2014).

[88] Ali Ouni, Raula Gaikovina Kula, Marouane Kessentini, Takashi Ishio, Daniel M German, and Katsuro Inoue. 2017. Search-based software library recommendation using multi-objective optimization. *Information and Software Technology* 83 (2017), 55–75.

[89] James W Paulson, Giancarlo Succi, and Armin Eberlein. 2004. An empirical study of open-source and closed-source software products. *IEEE Transactions on Software Engineering* 30, 4 (2004), 246–256.

[90] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2016. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience* 46, 9 (2016), 1155–1179.

[91] Steven Raemaekers, Arie van Deursen, and Joost Visser. 2012. Measuring software library stability through historical version analysis. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 378–387.

[92] Steven Raemaekers, Arie Van Deursen, and Joost Visser. 2014. Semantic versioning versus breaking changes: A study of the maven repository. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*. IEEE, 215–224.

[93] Romain Robbes, Mircea Lungu, and David Röthlisberger. 2012. How do developers react to api deprecation?: the case of a smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 56.

[94] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. 2017. On the reaction to deprecation of clients of 4+ 1 popular Java APIs and the JDK. *Empirical Software Engineering* (2017), 1–40.

[95] Alexander Stuckenholz. 2005. Component evolution and versioning state of the art. *ACM SIGSOFT Software Engineering Notes* 30, 1 (2005), 7.

[96] Ferdian Thung, David Lo, and Julia Lawall. 2013. Automated library recommendation. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*. IEEE, 182–191.

[97] Balaji Varanasi and Sudha Belida. 2014. Maven Dependency Management. In *Introducing Maven*. Springer, 15–22.

[98] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 226–237.

[99] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating bugs from software changes. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, 262–273.

[100] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. 2011. Relink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 15–25.

[101] Yuki Yano, Raula Gaikovina Kula, Takashi Ishio, and Katsuro Inoue. 2015. VerXCombo: An interactive data visualization of popular library version combinations. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*. IEEE Press, 291–294.

[102] Huan Yu, Xin Xia, Xiaoqiong Zhao, and Weiwei Qiu. 2017. Combining Collaborative Filtering and Topic Modeling for More Accurate Android Mobile App Library Recommendation. In *Proceedings of the 9th Asia-Pacific Symposium on Internetware*. ACM, 17.

[103] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schroter, and Cathrin Weiss. 2010. What makes a good bug report? *IEEE Transactions on Software Engineering* 36, 5 (2010), 618–643.