Removing dependencies from large software projects: are you really sure?

Ching-Chi Chuang
Delft University of Technology
Delft, Netherlands
C.Chuang-1@student.tudelft.nl

Luís Cruz
Delft University of Technology
Delft, Netherlands
L.Cruz@tudelft.nl

Robbert van Dalen
ING
Amsterdam, Netherlands
Robbert.van.Dalen@ing.com

Vladimir Mikovski
ING
Amsterdam, Netherlands
Vladimir.Mikovski.Iotov@ing.com

Arie van Deursen
Delft University of Technology
Delft, Netherlands
Arie.vanDeursen@tudelft.nl

***Abstract*—When developing and maintaining large software systems, a great deal of effort goes into dependency management. During the whole lifecycle of a software project, the set of dependencies keeps changing to accommodate the addition of new features or changes in the running environment. Package management tools are quite popular to automate this process, making it fairly easy to automate the addition of new dependencies and respective versions. However, over the years, a software project might evolve in a way that no longer needs a particular technology or dependency. But the choice of removing that dependency is far from trivial: one cannot be entirely sure that the dependency is not used in any part of the project. Hence, developers have a hard time confidently removing dependencies and trusting that it will not break the system in production. In this paper, we propose a decision framework to improve the detection of unused dependencies. Our approach builds on top of the existing dependency analysis tool DepClean. We start by improving the support of Java dynamic features in DepClean. We do so by augmenting the analysis with the state-of-the-art call graph generation tool OPAL. Then, we analyze the potentially unused dependencies detected by classifying their logical relationship with the other components to decide on follow-up steps, which we provide in the form of a decision diagram. Results show that developers can focus their efforts on maintaining bloated dependencies by following the recommendations of our decision framework. When applying our approach to a large industrial software project, we can reduce one-third of false positives when compared to the state-of-the-art. We also validate our approach by analyzing dependencies that were removed in the history of open-source projects. Results show consistency between our approach and the decisions taken by open-source developers.**

***Index Terms*—unused dependencies, call graph generation, static analysis**

I. Introduction

Modern software systems rely on package managers to gain benefits from the increasing number and massive support of dependencies [1]. Software dependencies hosted on centralized code repositories by package managers allow software engineers to reuse code, reduce development costs and ease maintenance efforts. While the convenience of adding new collections of software dependencies speeds up software development, software projects might retain dependencies that gradually become obsolete throughout the process of development [2]. Dependencies that become obsolete and overlooked can increase complexity, decrease maintainability, and in some cases bloat software size. Thus, it is important for developers to properly clean outdated dependencies.

This is a problem taken seriously at ING Bank [3]. ING is a global company and large software organization that offers financial products and services to 38.5 million customers in over 40 countries and has 15,000 employees in software technology. Hence, at ING it is quintessential that software projects are continuously maintained and meet high-quality standards. Leaving unused dependencies in large software projects can lead to major problems downstream (e.g., in security, maintainability, scalability, etc.). However, deciding to remove a dependency can be an intimidating task: one wrong decision could make core business services temporarily unavailable.

The mainstream approaches to detecting unused dependencies rely on static dependency analysis or dynamic dependency analysis [4]. The performance of the static approaches depends on the soundness and precision of the call graph construction whereas the performance of the dynamic approaches resorts to the coverage of route collections at runtime. Hence, static approaches tend to be quicker and more scalable. However, generating a static call graph has been considered an undecidable problem [5], meaning that it is difficult to confidently say whether a dependency is being reached or not based on state-of-the-art call graph generation tools.

Several static analysis tools have been developed to remove unused but declared dependencies. For Java projects, fundamental efforts have been contributed by communities to help developers analyze dependencies statically in JDK[1] and Maven[2]. Also, advanced tools [6]–[8] have been built to address prevalent dynamic language features: reflection, dynamic proxy, and classloading [9]. Despite these efforts,

[1]https://wiki.openjdk.java.net/display/JDK8/Java+Dependency+Analysis+Tool
[2]https://maven.apache.org/shared/maven-dependency-analyzer/

finding unused dependencies of Java projects is not a trivial goal. It has been demonstrated that all state-of-the-art static analysis frameworks fail to capture complete dynamic language features in call graphs [10].

These limitations give rise to false positives of the unused dependency detection, which is a challenge when creating tools to help developers in removing these dependencies. When they receive false alerts too often, they are likely to ignore warnings, filter alerts, or turn away from using tools altogether [11]. To encourage the usage of tools, researchers start to think that it is not only beneficial to pursue the precision of tools but also critical for tool authors to present warnings from developers' perspectives [12].

The underlying principle of current static dependency analysis tools is reachability. Any dependency is classified as unnecessary if it cannot be reached from the application code. Based on this binary result i.e., reachable or unreachable, tools provide recommendations that help developers remove unused dependencies. However, we argue that the existing analysis result produced by tools shows more information than a binary recommendation. If the reachability is interpreted in more detail, for example, complexity and evolution of method calls between artifacts, tools may report the reasoning behind recommendations and allow developers to be effective in their decision-making.

Making a decision on removing a dependency is far from an easy task, especially for software in production. Developers need to balance the potential risk of making serious errors and the future benefit of saving maintenance efforts. In this regard, the reasoning behind such a decision should be adequate to minimize the potential risk; otherwise, developers will not take risks to reduce maintenance. Thus, we propose a decision framework that examines several aspects of the results generated by static analysis tools and guides developers through the decision on whether to remove a given dependency or not. We provide a public replication package with our experiments and results: https://bitbucket.org/scam2022chingchichuang/static_dependency_analysis/.

This paper is structured as follows. In Section II, we introduce a dependency analysis summary of four real-world Maven projects that motivate this work. Section III compares the differences between our work and existing literature. Section IV and V describe the decision framework and methodology applied to answer the research questions. The results are described in Section VI and their insights are discussed in Section VII. In section VIII, the threats to the validity of our work are explained. Finally, in Section IX we draw the main conclusions and share our future work.

II. Motivating Example and Research Questions

Since there is no static analysis tool that can guarantee its finding on unused dependencies, tools usually emphasize the result with warning such as "potentially" unused dependencies to notify developers that their analysis results should be accepted with caution. Table I summarizes the analyzed results of one ING's project and three open-source projects using the state-of-the-art tool DepClean[3]. The table presents the number of used dependencies and potentially unused dependencies. Part of these projects possesses a significant number of unused dependencies, which is unusual and unexpected. We hypothesize that some of the potentially unused dependencies may be false positives – i.e., a dependency that is being incorrectly classified as unused. This happens, for example, in cases where it is difficult to collect using static analysis all the possible entry methods of a dependency. Hence, it may happen that all classes within the dependency become unreachable from the application, and tools misclassify it as unused.

TABLE I
THE SUMMARY OF DEPENDENCY ANALYSIS BY DEPCLEAN

Project Name	Number of Dependencies	
	Used	Potentially Unused*
enterprise user management app	73	71
jenkins [core v2.343]	80	16
zipkin [zipkin-server v2.23.16]	94	22
onedev [server-core v7.0.9]	176	72

*This follows the term of DepClean. In this work, we call it **flagged unused**.

To reduce false positives and improve the recommendation, we propose a framework that 1) combines different call graph tools, 2) considers the history of changes in the dependencies of a project, and 3) guides developers through the decision process before yielding a final result.

> ***RQ1****: Can we systematically combine automated and manual analyses to improve the detection of unused dependencies of software projects at ING?*

Why: As shown in the motivating example, some of the detected unused dependencies in the enterprise user management application may be false positives. Reducing false positives can save the effort of developers in decision making, which encourages them to invest time in removing dependencies with high confidence and deals with less probable ones later. Also, the recommendations of the decision framework have to be evaluated by developers, which may not be available for open-source projects. Hence, this work relies on developers at ING to judge and verify the correctness actively.

How: One enterprise user management application in the production is chosen because of its peculiar software structure which poses challenges for the state-of-the-art tool. After being presented with the suggestions of the decision framework, developers select some dependencies for testing based on their understanding of the usage of the dependencies. The result of the tests is compared to the recommendation of the decision framework.

> ***RQ2****: Is our decision framework to detect unused dependencies confirmed by the commit history of other open-source projects?*

[3]https://github.com/castor-software/depclean/releases/tag/2.0.0

Why: Since the selected enterprise user management application at ING is maintained by the production team, the available testing windows and the capacity of developers are limited. Hence, dependencies that were previously deleted in the open-source projects are valuable to evaluate the decision framework.

How: We inspect the history of three open-source projects: Jenkins, Zipkin, and Onedev. For each project, we collect all the commits that add or remove dependencies and compare the reasons behind these changes with the proposals from our decision framework.

III. Related Work

A. Dependency Analysis Tools

Previous work has conducted a study to find out the presence of unused dependencies in Maven artifacts [13]. The authors develop a tool called DepClean, which extends the maven-dependency-analyzer maintained by the Maven team. They collect a list of dependencies declared in the POM and analyze the bytecode to identify all potentially unused dependencies. They analyze the bytecode by using the ASM library[4] while capturing annotation, field, method, and limited dynamic features like class literals in each class. But when analyzing a dependency that is invoked by other dynamic features, a used dependency may be considered unused due to missing edges established by dynamic features.

Their goal is to generate a variant of the POM without those unused dependencies. The authors also applied the same tool to study how unused dependencies increase, decrease or remain stable over time in hundreds of single-module Maven projects [14]. In this work, we further extend DepClean by adding support of a static call graph tool OPAL[5]. Moreover, instead of removing unused dependencies from the POM based on a binary evaluation of the bytecode, we investigate the relationship of unused dependencies with other dependencies and provide more reasoning than the binary evaluation.

Another previous work has augmented the static reachability analysis with dynamic reachability analysis [4]. Their tool, JShrink, uses test cases to find dynamic features invoked at runtime and adds them back to amend static call graphs. Their analysis is fine-grained down to the level of method and field. To preserve behaviors after removing unnecessary bytecode, they build type dependency graphs using the ASM library to ensure type safety. Compared to their work, our analysis is coarse-grained at the artifact level and purely static. In our targeted scenario, test cases are not widely available or only include a few basic ones; therefore, JShrink cannot provide much help. Also, we intend to remove unused dependencies for modules. Instead of removing redundant bytecode in each artifact, we can notify developers to refactor when the usage of an artifact is relatively low. Additionally, the authors make use of this tool as a backend of WebJShrink, a visualization interface allowing developers to select removal options [15]. We acknowledge the idea that the decision of removing dependency should be made by developers rather than the tool's authors.

B. Static Call Graph Construction Tools

Call graph construction is a principal element of static analysis tools to determine unused dependencies. Previous work has laid out the difficulties of building a sound and precise call graph statically in Java. The main obstacle is posed by the usage of the Reflection API, a great mechanism for developers to inspect and adapt the behavior of their software in the runtime environment [9]. Since tools cannot correctly predict how software is evolving, tools simply consider all the possibilities, leading to unsoundness and imprecision. According to an empirical study [16], as many as 78% of 461 representative open-source Java projects contain at least one usage of the Reflection API. Likewise, the same study found that 21% of open-source Java projects use dynamic proxies, a proxy mechanism that brings the flexibility of forwarding method calls to different objects at runtime. This mechanism also generates a dynamic layer against static analysis tools to model [7].

Many existing static analysis frameworks have implemented numerous call graph algorithms to cope with these challenges. Previous work has conducted a comparative study of four state-of-the-art frameworks: Soot, WALA, DOOP, and OPAL [10]. Their result helps us understand the performance of frameworks w.r.t the profile of language features support and runtime costs. Overall, OPAL with the Rapid Type Analysis (RTA) call graph algorithm [17] is the most feature-completed option which enables users to solve the most prevalent Java dynamic features and APIs among 23 categories grouped from predefined 122 test cases. OPAL is also the fastest framework due to its scalability. This scalable feature ensures that data structures are immutable while call graphs are constructed in parallel [18].

Moreover, OPAL provides an API[6] to analyze merely a portion of dependencies by classifying them as **project files** or **library files**. Since this API only analyzes project files and their outgoing method calls to library files, it is efficient to analyze the provided enterprise user management application with a normal laptop [19].

Other studies have built reflective analysis [9] and dynamic proxy support [7] on top of the DOOP framework with high accuracy. However, DOOP's call graph generator is so time-consuming and memory-intensive that it is impractical for real-world usage. As far as we know, no studies have discussed the performance of applying these frameworks in industrial software. In this work, we intend to investigate how OPAL helps developers find unused dependencies in production code.

C. Software metrics for unused dependency

Software metrics are widely used for software fault detection [20], [21]. However, only a few previous works focus

[4]https://asm.ow2.io/

[5]https://github.com/fasten-project/fasten/tree/develop/analyzer/javacg-opal

[6]https://www.opal-project.de/library/api/SNAPSHOT/org/opalj/br/analyses/Project$.html

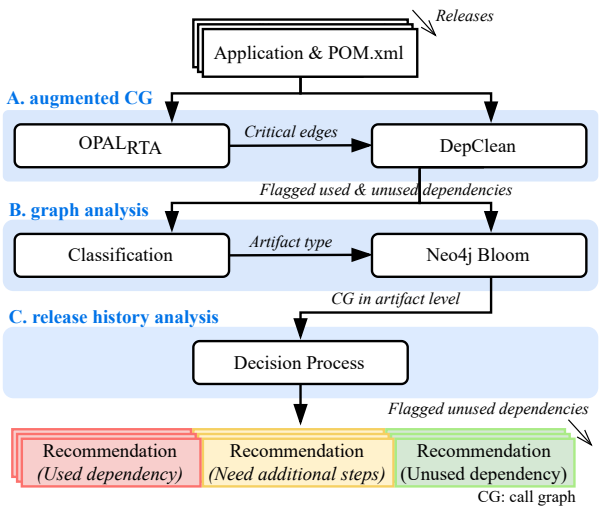
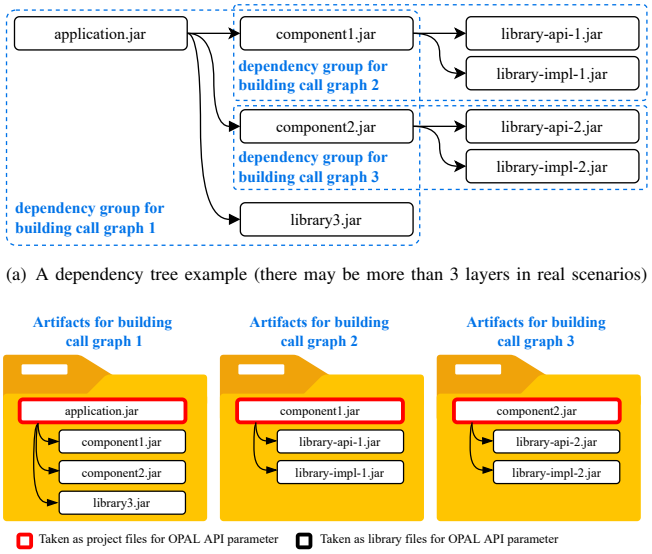


Fig. 1. Decision framework workflow



(a) A dependency tree example (there may be more than 3 layers in real scenarios)

(b) The process of separating dependencies for the OPAL call graph construction

Fig. 2. OPAL call graph constructions by the dependency tree

on metrics for the detection of unused code. Haas et al. [22] investigate whether code stability and code centrality indicate the likelihood of unnecessary code. The authors report that 34% of recommendations for unnecessary code are confirmed as true positives. Another study tries widely accepted objected-oriented metrics to predict dead code methods [23]. It concludes that LOC, WMC, and RFC are useful indicators to discover dead code. However, these previous works focus on the code level instead of the dependency level. In this work, our approach is inspired by software metrics and aims to contribute to the classification of the unused dependency for decision-making.

IV. Framework to Remove Unused Dependencies

We propose a decision framework that comprises the following three steps to remove unused dependencies:

A. **augmented CG.** Combine different call graph tools for the analysis of dependency usage.

B. **graph analysis.** Classify flagged dependencies by their relationships with other dependencies in the call graph.

C. **release history analysis.** Analyze the history of code changes related to a flagged unused dependency.

In this section, we pinpoint each of these steps as shown in Fig. 1. Rounded rectangles represent procedures for every software release. First, we build an augmented call graph by which the dependency analysis flags dependencies as used and unused. Next, we classify these flagged dependencies according to their relationships in the call graph. Last, we apply the decision process to provide recommendations for the developers.

A. Dependency analysis based on a call graph built by different tools

Software projects may contain different language features and APIs which affects the performance of the dependency analysis tools. Hence, we start by analyzing the project with DepClean to collect a preliminary result. To enhance the support of dynamic features for the static dependency analysis, we augment the call graph of DepClean with critical edges collected with OPAL+RTA.

Although OPAL$_{RTA}$ supports many dynamic features, the way of applying OPAL API to build a call graph has a great impact on the precision of the call graph. It is because OPAL has high coverage and is designed to find all the possible implementations of an interface or abstract class. If a global call graph is built by joining all dependencies with the main application code, there might be some theoretical edges created between unrelated dependencies. To avoid these edges, two factors are considered to maintain the precision of the call graph: **dependency tree**, and **critical edges**.

We use the **dependency tree** to enhance the precision of generating the global call graph. We do so because a global call graph might lead to theoretical edges that are inconsistent with the hierarchy defined by the dependency tree. Fig. 2 illustrates this process with a dependency tree of a Maven application. These dependencies are downloaded and grouped into multiple folders based on their layers in the Maven dependency tree. Next, the artifact of a parent dependency in each folder is classified as a **project file** whereas artifacts of all the child dependencies are classified as **library files** as shown in Fig. 2b. After the classification, OPAL API is applied to build a call graph per folder. In this way, we avoid inconsistent theoretical edges such as a method call from `component1` to `library-impl-2`.

Critical edges are a set of edges that must be called at runtime if they are reachable from the application code. Conversely, some edges occur frequently but may not necessarily be called at runtime. For example, some methods such as toString, hasNext, or toArray defined in JDK are implemented by so many dependencies, which makes it difficult to anticipate

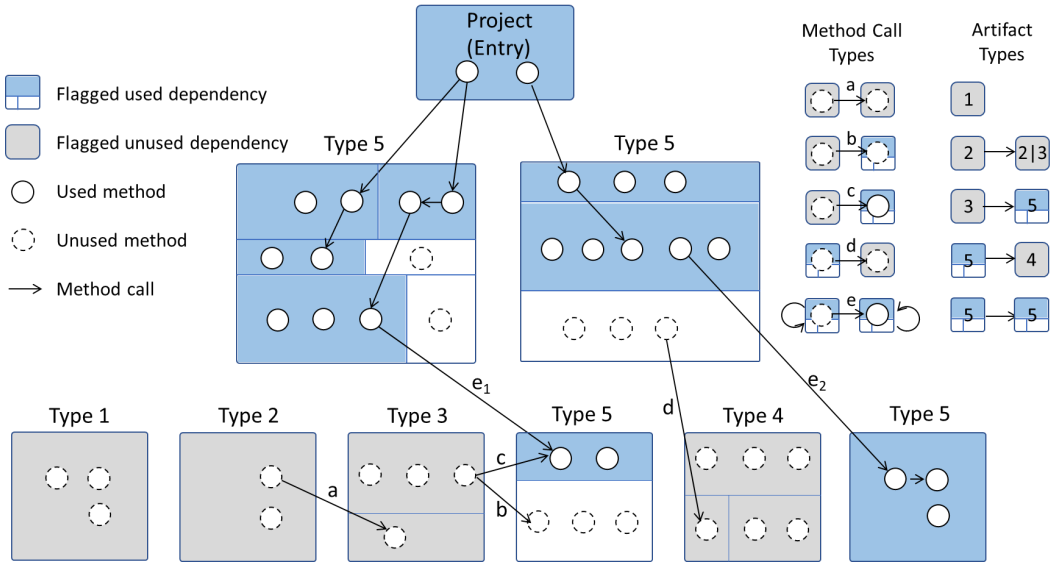


Fig. 3. Classification of flagged used and unused dependencies. The rule of classification follows the symbol of method call types and artifact types shown on the upper-right hand side. Method call types represent all the possible relationships between methods in two artifacts. Based on the incoming and outgoing method call types of an artifact, the artifact can be categorized into five types. Type 1 and Type 2 only have relationships with other flagged unused dependencies. Type 3 and Type 4 are more complicated and have relationships with flagged used dependency. Contrary to Types 1–4, we consider Type 5 as flagged used dependency and reachable from the entry code of the application.

which implementation will be executed during runtime. Hence, OPAL will create edges leading to certain dependencies simply because they implement these methods. To prevent the call graph from exploding we apply the following approximation: we remove any edges found by OPAL that point to multiple implementations of the same method. Hence, we only keep the critical edges. Since these critical edges are a unique path between a component and a target method, we are certain that the respective component will always call the target method during runtime. After collecting these critical edges from all OPAL call graphs, we add them to the call graph of DepClean. With this approximation, we avoid an overly complex call graph that takes too much time to generate and analyze.

To flag which dependency is used or not, we follow DepClean's approach. A set of entry classes in a call graph must be defined. For the enterprise user management application at ING, all the classes that handle requests are possible entries. For the open-source projects, all the classes in the source folder are used as entries. Next, entry classes are used to traverse a call graph and find all the reachable classes. If any class of a dependency is found to be accessible from entry classes, this dependency is flagged as used. Also, as long as a class contains a method accessible from entry classes, all the methods in the class are flagged as used too. An example of flagging a dependency as used or unused is presented in Fig. 3.

B. Classification of flagged dependencies according to the call graph

A call graph may expand with increased dependencies and become difficult to trace, but method calls between dependencies can be simplified according to their source and target methods as exemplified in Fig. 3. If a method can not find a route back to any entry class or used method, the method is defined to be an unused method. An unused method could exist both in flagged used and unused dependencies.

Based on the type of incoming and outgoing method calls, a flagged dependency can be classified into five types. If the dependency can be classified as more than one artifact type, we choose the one with the largest number.

- Artifact type 1 represents an isolated dependency and it has no external method call.
- Artifact type 2 indicates that a flagged unused dependency has incoming or outgoing method calls to or from other flagged unused dependencies.
- Artifact type 3 depicts that a flagged unused dependency has outgoing method calls to any flagged used dependency.
- Artifact type 4 portrays that a flagged unused dependency has incoming unused method calls from any flagged used dependency. The incoming unused method calls are not accessible from any entry classes.
- Artifact type 5 describes a flagged used dependency.

Neo4j Bloom [24] is adapted to visualize classified artifact types and their relationship to other artifacts. In Neo4j, a graph data structure is organized as nodes, relationships, and properties. The discrete nodes are connected by relationships and both of them can be described further by properties [25]. The definition of Fig. 3 is followed to set properties of nodes and relationships. For example, the artifact type is one of the properties of nodes while the method call type is one of the properties of relationships. The visualization gives an overview of which dependency is flagged unused while artifact types indicate their relationships to other dependencies. This approach may help developers comprehend the detailed information of flagged unused dependencies.

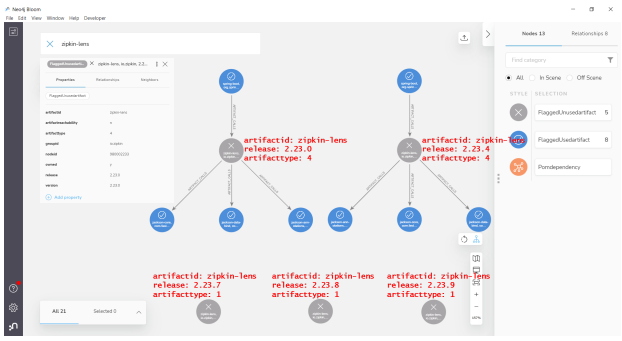


Fig. 4. Neo4j Bloom for the visualization of the call graph at the artifact level after the classification of artifact types and the relationship between artifacts.

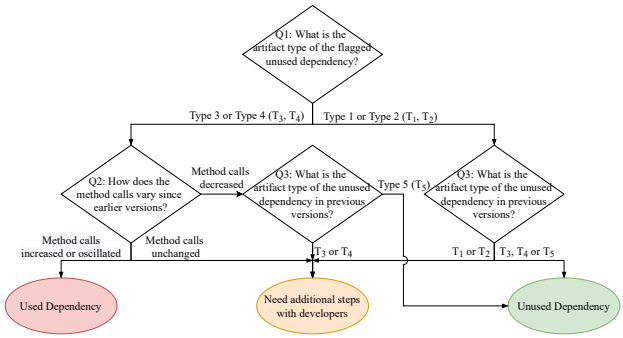


Fig. 5. Decision process for the recommendation of individual flagged unused dependencies.

Fig. 4 demonstrates how to use the artifact classification and Neo4j Bloom to observe the usage of one flagged dependency in the Zipkin project. In earlier versions 2.23.0 and 2.23.4, this flagged dependency is classified as artifact type 4. However, in the later version since 2.23.7, the flagged dependency is classified as artifact type 1. It shows that incoming and outgoing method calls of this flagged unused dependency decrease, which may be evidence of true unused dependency. This way, the complexity of the call graph is simplified while preserving high-level relationships between flagged used and unused dependencies. Hence, developers are provided with essential information but not overwhelmed.

C. Analysis of the history of code changes related to a flagged unused dependency

The high-level classification of a flagged dependency also helps us describe the code changes related to a dependency across versions. Analyzing and comparing to earlier versions might find crucial evidence that increases the confidence of flagged unused dependencies. To convey the confidence in removing a flagged unused dependency based on the change in its usage, Fig. 5 provides a decision tree that depicts the decision process and its recommendations. Hence, developers may rely on these recommendations to prioritize which flagged unused dependency could be removed. Three questions are pinpointed as follows:

Q1 **What is the artifact type of the flagged unused dependency?** Every unused JAR artifact is categorized by the complexity of its relation with other artifacts. The flagged dependency that has more method calls across artifacts like type 3 and type 4 is handled differently in comparison to the flagged dependency with few method calls across artifacts.

Q2 **How do the method calls vary since previous versions?** If there is any method call removed since previous versions, this may be an indication of an unnecessary dependency. On the contrary, if the number of method calls increase compared to previous versions, the dependency should be retained. However, if the connection is unchanged throughout previous versions, it requires extra effort to distinguish the usage of the dependency. It is because the call graph tool is not sound, and possible to miss some features. To be safe, developers have to decide if it is required to be investigated further.

Q3 **What is the artifact type of the flagged unused dependency in previous versions?** This question compares both types 1,2 and types 3,4 to their artifact types in the earlier version. For example, if their artifact type in previous versions is type 5, it is obvious that some of the method calls have been removed since earlier versions. In this case, the flagged dependency is recommended as unused. On the other hand, if the artifact type is the same as in previous versions, the recommendation is needing additional steps with developers.

Generally speaking, the decision process is designed to be strict with giving recommendations to remove a flagged dependency. When there is evidence that method calls have reduced since an earlier version such as from type 5 to type 1–4, the flagged dependency is recommended as unused. On the other hand, when some method calls are observed to be increased or varied, the flagged dependency is considered as used (i.e. a false positive). For a flagged dependency that has no method calls changed in earlier versions, the flagged dependency needs additional steps with developers before taking further actions.

V. Methodology

A. Evaluation of the decision framework at ING

We selected an enterprise user management application in the production for the evaluation because it is legacy software that certainly contains unused dependencies. The application is packaged as an EAR file which includes 144 Jar dependencies, and 43 of them are maintained by different groups at ING. The class and line of code of dependencies are 3050 and 211657 respectively. The project migrated from an inhouse-developed platform to Maven in 2019. Hence, there are 41 releases available to evaluate the decision framework.

To evaluate the recommendation, we guide developers through the results of our decision framework and ask them to provide their input: whether they agree with the recommendation and what is the main reason. We discuss with

three developers who develop or maintain the project. Two developers have worked on these projects for more than 4 years. One developer has maintained this project for almost 2 years. All of them have more than 5 years of working experience on Java projects.

We remove the dependencies approved by developers from the project and execute the existing system and functionality tests. If the tests pass, we deploy our changes to the test environment of the software. This triggers a set of additional ING-specific checks that maintainers have to perform to validate the changes. Meanwhile, we also collect the system log to spot any unusual behavior – e.g., an error message. In the absence of any issue or concern from developers, we assume that the dependency can be successfully removed and our code changes can be merged into production.

This process is time-consuming and is taken very seriously by developers at ING. To use their time efficiently, we opt for removing multiple dependencies in the same merge request.

B. Evaluation of the decision process in open-source projects

For the open-source projects, we select three projects: Jenkins, Zipkin, and Onedev. Jenkins is used in previous work [13] to evaluate DepClean. We select Zipkin and Onedev from a set of 50 projects collected with the search tool *SEART*[7]. With SEART, we retrieve 50 top Java projects available on Github that have more than 5k stars, more than 50 releases, and were active in the first quarter of 2022. These projects are then filtered according to the following criteria: 1) the project uses maven to manage dependencies (e.g., Gradle projects are discarded), 2) the project has a single main module that we can use for analysis (we do not support multi-module analysis), and 3) the project has at least 5 commits that remove dependencies.

We then inspect all the commits that remove or add dependencies in every release of the selected projects. For the removed dependencies, we examine the code changes and commit messages to assess why they are removed. For the added dependencies, we check whether that dependency is brought back in later releases. Any commit that does not reveal the reason for removing a dependency is discarded. We then determine the reasons behind removing dependencies by reading the commit messages and inspecting code differences. We then divide commits into three groups (R_1) replace dependency, (R_2) remove code and dependency, and (R_3) only remove dependency. Reasons R_1 and R_2 imply that the dependency is still needed by the project and some other reason lies in its removal from the dependency specification (e.g., security issues, API migration, etc.). Reason R_3 clearly indicates that the dependency is unused – hence, we compare these cases with the recommendations provided by our framework (cf. Fig. 5).

VI. Results

In this section, we present the result collected using the proposed methodology for two research questions.

[7]https://seart-ghs.si.usi.ch/

TABLE II
Dependency analysis result of the project at ING

Dependency Analysis Tool	Number of Flagged Dependencies	
	Used	Potentially Unused
DepClean	73	71[a]
DepClean + OPAL	85	59[b]
DepClean + OPAL + Decision process	98	46[c]

[a] From which 26 are direct, 45 transitive, and 0 inherited.
[b] From which 16 are direct, 43 transitive, and 0 inherited.
[c] From which 16 are direct, 30 transitive, and 0 inherited; we conclude that there are 10 `unused` and 36 `need additional steps with developers`.

A. Can we systematically combine automated and manual analyses to improve the detection of unused dependencies of software projects at ING? (RQ1)

The usage of dependencies is analyzed and presented in Table II which compares the results collected from different dependency analysis tools. The baseline is provided by DepClean which flags 73 used and 71 unused dependencies. After augmenting the DepClean call graph with critical OPAL edges, another 12 dependencies become accessible from the entry classes. Hence, the number of flagged used dependencies increases to 85 while the number of flagged unused dependencies decreases to 59.

Next, all the 59 flagged unused dependencies are classified as the corresponding artifact types and are compared to their classification in the earlier versions. By doing so, we find that the number of method calls in another 13 flagged unused dependencies has increased since earlier versions. Hence, these 13 flagged unused dependencies are recommended as ***used*** by the decision process. As a consequence, the number of potentially unused dependencies lowers again to 46. Within these 46 flagged unused dependencies, there are 10 flagged unused dependencies whose method calls reduce since an earlier version, so these 10 flagged unused dependencies are recommended accordingly as ***unused*** by the decision process. For the rest 36 flagged unused dependencies, their method calls are relatively stable, which calls for ***additional steps with developers***.

Table III summarizes the recommendations of the decision process and developers' feedback on 59 potentially unused dependencies flagged by DepClean+OPAL. For the recommendation as ***used dependencies***, developers agree with eight of them because they know the functionalities of these eight dependencies and are certain about their use cases. On the other hand, they are clueless about the other five recommendations since all these five dependencies are transitive and are barely noticed.

For the recommendations that ***need additional steps with developers***, the majority of them are declined by developers to remove due to several reasons. Firstly, some of their functionalities are known and needed by developers. Secondly, developers avoid excluding currently unused transitive dependencies in case they may become used after the dependency upgrade in the future. Thirdly, the dependency may aim to package other file formats such as javascript instead of adding java

TABLE III
SUMMARY OF THE RECOMMENDATIONS OF THE DECISION PROCESS AND DEVELOPERS' FEEDBACK.

Recommendation	Developers' Decisions	Developers' Reasons	Number
Used dependencies (13)*	Agree with the recommendation	The functionalities of these dependencies are known and needed.	8
	Not sure about the recommendation	These are transitive dependencies and functionalities are unknown.	5
Need additional steps with developers (36)	Do not remove dependencies	The functionalities of these dependencies are known and needed.	8
		These are transitive dependencies and may become used in the future.	18
		The dependencies only contain javascript code.	4
	Can be removed	These dependencies have not changed for years.	6
Unused dependencies (10)	Do not remove dependencies	These are transitive dependencies and may become used in the future.	6
		These dependencies may be used in edge cases.	1
		The functionalities of these dependencies are known and needed.	2
	Can be removed	This is a duplicated dependency.	1

*In total, there were 98 flagged used artefacts. We selected the 13 artifacts that had been flagged as unused in the early stages of the analysis but were then discarded by our decision process.

TABLE IV
SYSTEM AND FUNCTIONALITY TEST FOR SELECTED DEPENDENCIES BASED ON THE RECOMMENDATION AND DEVELOPERS' FEEDBACK.

Recommendation	Developers' Reasons for Tests	Dependency Removing Tests
Used dependencies (3/13)*	Verify that dependencies are indeed used.	Test=3; Pass=0; Fail=3
Need additional steps with developers (6/36)*	Can be removed	Test=6; Pass=3; Fail=3
Unused dependencies (1/10)*	Can be removed	Test=1; Pass=1; Fail=0

*Only partial dependencies were selected for dependency-removing tests.

class files. In addition to the majority, developers decide that 6 dependencies can be removed since they have not used them for years. For the recommendation as ***unused dependencies***, 9 of them are declined by developers to remove because of the future upgrade of transitive dependencies, possible usages by edge cases, and the necessity of functionalities. Only 1 dependency in these recommendations is accepted by developers due to the existence of a duplicated dependency.

The result of the system and functionality test for dependency removal is presented in Table IV. Only 10 out of 59 unused dependencies flagged by DepClean+OPAL are forwarded to dependency removal tests. It is because many dependencies are declined by developers to remove due to safety concerns. For the recommendation of ***used dependencies***, three dependencies are selected for the tests, and all of them cause some failures of functionalities as expected. For the dependencies that ***need additional steps with developers***, six dependencies are chosen while half of them fail the tests. For the recommendation of ***unused dependencies***, one dependency is picked and passes the test.

B. Is our decision framework to detect unused dependencies confirmed by the history of other open-source projects? (RQ2)

The results of the consistency between the reason for removing a dependency in the open-source projects and the recommendation of the decision process are presented in Table V. We divide each dependency removal by the different reasons behind that change: R_1, R_2, R_3 (as explained in Section V-B).

For each reason for removing dependencies, the dependency usage immediately before being removed can be inferred as ground truth to compare the recommendations of our proposed decision process and DepClean. For any recommendation that is inconsistent with the reasons for being removed, the result is labeled with an asterisk symbol in Table V.

In sum, the results show that only 3 recommendations from our decision process are inconsistent with the actual reasons for being removed. This result is far better than DepClean's recommendation, where 21 recommendations are not reflected in the commit history. In addition, there are 11 dependencies that our decision process can not determine their usage. In those cases, we need additional steps with developers.

VII. DISCUSSION

In this section, we answer our research questions and discuss the implications of the results.

A. Can we systematically combine automated and manual analyses to improve the detection of unused dependencies of software projects at ING? (RQ1)

The combination of automated and manual analyses reduces false positives and helps developers prioritize the tests of unused dependencies. Results presented in Table II show that 12 dependencies that were initially flagged as unused by DepClean become used after we augment the call graph with critical OPAL edges. We conjecture three reasons that may explain this. First, the provided project heavily relies on dynamic proxies to invoke various implementations of services – this is the case for 10 out of 12 dependencies that implement such services. The dynamic proxy is one of the dynamic features in the Java language that is supported by OPAL. Second, there are 2 out 12 dependencies that become used not because they implement dynamic features but because they are direct dependencies used by some of the

TABLE V
EVALUATION OF THE DECISION PROCESS BY CHECKING THE REASONS OF REMOVING DEPENDENCIES IN THE OPEN-SOURCE PROJECTS.

Project Name	Reasons for Removing Dependencies	Dependency Usage Immediately Before Being Removed (GT)		Recommendations by the Decision Process				Recommendations by DepClean	
		Used (44)	Unused (12)	Used (37)	Unused (8)	Need additional steps with developers (11)		Used (29)	Unused (27)
						$T_1\|\|T_2$	$T_3\|\|T_4$		
Jenkins core (25)	Replace dependency(R_1)	3	0	3	0	0	0	3	0
	Remove code and dependency(R_2)	12	0	12	0	0	0	12	0
	Only remove dependency(R_3)	0	10	2*	8	0	0	2*	8
Zipkin server (23)	Replace dependency(R_1)	1	0	0	0	0	1	0	1*
	Remove code and dependency(R_2)	20	0	15	0	2	3	7	13*
	Only remove dependency(R_3)	0	2	1*	0	0	1	1*	1
Onedev server (8)	Replace dependency(R_1)	1	0	1	0	0	0	1	0
	Remove code and dependency(R_2)	7	0	3	0	2	2	3	4*
	Only remove dependency(R_3)	0	0	0	0	0	0	0	0

*The recommendations contradict to the commit history (GT).
GT: ground truth, $T_1||T_2$: Artifact type1 or type2, $T_3||T_4$: Artifact type3 or type4

previous 10 dependencies. Thirdly, it is noticed earlier that many of the frequent-occurred OPAL edges are overestimated by method implementations such as toString, hasNext, and toArray. However, none of these 12 dependencies become used due to overestimated method implementations since only critical edges are accepted. Hence, depending on the application, the degree to which our augmented analysis brings benefits will change. In particular, we anticipate that our augmentation brings more value to applications that rely heavily on dynamic features of Java, which is the case of the software project we study at ING.

The results also show the usefulness of the decision process. For the recommendation of ***used dependencies***, developers agreed that 8 out of 13 dependencies are used. For another 5 out of 13 dependencies that developers are uncertain about, they consider these dependencies as used because they are all transitive dependencies and the functionalities are unknown. When the functionalities of unused dependencies are unknown, removing them may cause potential errors in the application. For the dependencies that ***need additional steps with developers***, the majority of these dependencies are declined by developers to remove. This fact indicates that it is necessary for the dependency analysis tool to offer this kind of recommendation rather than merely providing a binary recommendation (used/unused). For the recommendation of ***unused dependencies***, developers are more concerned about considering them unused. Developers only accept this recommendation for the dependency that is duplicated.

To evaluate the recommendation and developers' feedback, ten dependencies are selected for system and functionality tests in Table IV. After developers remove 3 dependencies recommended as ***used***, each of them causes different failures which include error messages in the system log and functionality breaks at the server. In other words, these dependencies are verified as actually used in the application. Hence, it shows that the decision process can indeed help us reduce false positives. For dependencies that ***need additional steps with developers***, the result shows that developers may not always be correct on the usage of the dependencies. Three out of 6 dependencies that developers approve tests cause functional errors after they are removed. Hence, when the decision process recommends needing additional steps with developers, developers indeed need to take more efforts to investigate. For the only ***unused dependencies*** accepted by developers, it passes the test as expected. Therefore, with the help of the decision process, developers may prioritize how they plan to test and remove the unused dependencies.

However, the current design of the decision framework has a limitation and would falsely recommend a dependency as unused in some circumstances. For example, two dependencies are recommended as unused but are considered as needed by developers in Table III. It is because we only select critical edges in the OPAL call graph. When we examine the OPAL call graph, we find that all target methods of these 2 dependencies occur more than once. Since we only select critical edges to augment the call graph, the edges created by these 2 dependencies are ignored. Another corner case that is not covered by our approach is when transitive dependencies are directly called by the application. However, we do not observe this corner case in our data.

B. Is our decision framework to detect unused dependencies confirmed by the history of other open-source projects? (RQ2)

The recommendations of the decision process are consistent with the history and are cautious not to remove false positives. The results show different fingerprints of three open-source projects that can help us check if the recommendations of the decision process are consistent with the history.

For the recommendation of ***used dependencies*** by the decision process, 34 of 37 recommendations are correct since R_1, R_2 indicate that those dependencies are still used or maintained. Specifically, if a dependency is replaced by another dependency or removed along with some source code, it means that the removed dependency is still used before they are removed. Hence, the recommendation of not removing them is correct. On the other hand, 3 of 37 recommendations contradict the developers' reasons for removing dependencies. However, when the detail of the commit history is investigated, it is found that these dependencies are removed either because they are duplicated [26] or because they become provided

[27], meaning that they are still used before they are removed. Thus, the suggestion of the decision process matches all of the commit histories in this category. For the recommendation of ***unused dependencies*** by the decision process, all of them match the history of 8 commits in the Jenkins project that only remove unused dependencies.

For the recommendation that ***needs additional steps with developers*** by the decision process, 10 out of 11 dependencies are used according to the ground truth of commit history. However, these 10 dependencies are unreachable from the entry classes, which means there are some missing method calls to these 10 dependencies in the call graph. In this circumstance, DepClean flags these 10 dependencies as unused. In contrast, our decision process does not falsely classify them as unused because artifact types in previous releases are also considered. Since the decision framework does not find evidence to support whether the dependency is used or unused, the decision process is cautious and recommends taking additional steps with developers.

Likewise, when DepClean is used to decide the usage of these dependencies, the analysis result shows that 18 dependencies in the Zipkin and Onedev project are considered unused; however, these recommendations contradict the ground truth inferred from the commit history. In contrast, since our decision process considers the history changes of method calls and artifact types, some of these 18 dependencies are recommended as used by the decision process while others need additional steps with developers. In this manner, even though the call graph can not capture some dynamic features in the Java language, the decision process does not wrongly suggest developers remove false positives. This feature is crucial for the production environment.

VIII. Threats to validity

A. Construct

The approach to augmenting the call graph is designed according to the context of the provided ING project. Since the provided web application is developed a decade ago, they use the feature of the dynamic proxy to conveniently invoke various services before the technique of dependency injection becomes popular. For the project developed in recent years, the context of the software development must have changed and our approach should be adjusted to fit the different context. Specifically, the call graph construction tool, the mechanism of selecting critical edges, and the questions of the decision process may need to be adapted for the targeted project.

B. Internal

The system and functionality test rely on the experience of developers and the identification of error messages in the system log. However, even the senior developers may not know all the details in the dependencies maintained internally at ING. Also, the time allocated for the tests is limited, so some dependencies have to be tested within a batch. Although the result is expected to be the same as being tested individually, this premise has not been verified yet. Moreover, it is assumed that the error caused by the removed dependency can be triggered in a short time, which may not always be the case. Some faults may exist in the system for a long time without causing error messages.

C. External

Although the history of open-source projects is applied to confirm the recommendation of the decision framework, the system and functionality test can not be executed in the open-source projects as being done in the project at ING. In addition, only a few open-source projects regularly remove unnecessary dependencies like Jenkins. For most of the studied open-source projects, developers remove dependencies usually when the dependency is replaced or the software updates. This fact presents a difficulty for us to gain more data on unused dependencies to support our decision process.

IX. Conclusions and future work

In this work, a decision framework to reduce false positives of unused dependency detection is proposed. The decision framework extends the state-of-the-art dependency analysis tool DepClean and analyzes one industrial Maven project at ING along with three open-source projects.

For the project at ING, it is found that the augmented call graph helps reduce 12 false positives out of 71 unused dependencies detected. Also, the decision process based on the classification of the relationship between dependencies helps reduce 13 false positives. Hence, a decision framework of these two approaches filters out one-third of false positives of unused dependencies. The decision process further categorizes the remaining two-thirds of unused dependencies according to their release history, which allows developers to decide which dependency could be removed with ease or not.

Furthermore, the recommendations of the decision process are verified to be consistent with the reasons for removing dependencies in three selected open-source projects. Even though the dynamic feature of Java hampers the accuracy of the dependency analysis tool and creates false positives, the decision process relies on the changes in the relationship between dependencies and successfully points out 18 dependencies that could have become false positives.

In future work, our decision framework can be extended in different ways: improve the precision of OPAL when building a call graph for large software projects; analyze hierarchical multi-module Maven projects and see how the decision process needs to be adjusted; expand our methodology with other call graph tools to enhance the soundness to a great extent. Furthermore, it would be interesting to expand the study to understand how the visualization tool helps developers understand why dependencies are classified as unused.

Acknowledgment

The authors would like to thank Maarten Groot, Megha Mahajan, and Sonal Katole for their willing contributions to this project.

REFERENCES

[1] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, vol. 24, no. 1, pp. 381–416, 2019.

[2] N. Harrand, A. Benelallam, C. Soto-Valero, F. Bettega, O. Barais, and B. Baudry, "Api beauty is in the eye of the clients: 2.2 million maven dependencies reveal the spectrum of client–api usages," *Journal of Systems and Software*, vol. 184, p. 111134, 2022.

[3] "ING," https://www.ing.nl/particulier/index.html, accessed: 2022-08-01.

[4] B. R. Bruce, T. Zhang, J. Arora, G. H. Xu, and M. Kim, "Jshrink: In-depth investigation into debloating modern java applications," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 135–146.

[5] P. Boldi and G. Gousios, "Fine-grained network analysis for modern software ecosystems," *ACM Transactions on Internet Technology (TOIT)*, vol. 21, no. 1, pp. 1–14, 2020.

[6] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.

[7] G. Fourtounis, G. Kastrinis, and Y. Smaragdakis, "Static analysis of java dynamic proxies," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 209–220.

[8] D. Helm, F. Kübler, M. Reif, M. Eichberg, and M. Mezini, "Modular collaborative program analysis in opal," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 184–196.

[9] Y. Li, T. Tan, and J. Xue, "Understanding and analyzing java reflection," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 2, pp. 1–50, 2019.

[10] M. Reif, F. Kübler, M. Eichberg, D. Helm, and M. Mezini, "Judge: Identifying, understanding, and evaluating sources of unsoundness in call graphs," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 251–261.

[11] N. Imtiaz, A. Rahman, E. Farhana, and L. Williams, "Challenges with responding to static analysis tool alerts," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 245–249.

[12] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at google," *Communications of the ACM*, vol. 61, no. 4, pp. 58–66, 2018.

[13] C. Soto-Valero, N. Harrand, M. Monperrus, and B. Baudry, "A comprehensive study of bloated dependencies in the maven ecosystem," *Empirical Software Engineering*, vol. 26, no. 3, pp. 1–44, 2021.

[14] C. Soto-Valero, T. Durieux, and B. Baudry, "A longitudinal analysis of bloated java dependencies," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1021–1031.

[15] K. Macias, M. Mathur, B. R. Bruce, T. Zhang, and M. Kim, "Webjshrink: a web service for debloating java bytecode," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1665–1669.

[16] D. Landman, A. Serebrenik, and J. J. Vinju, "Challenges for static analysis of java reflection-literature review and empirical study," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 507–518.

[17] S. Romano and G. Scanniello, "Exploring the use of rapid type analysis for detecting the dead method smell in java code," in *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2018, pp. 167–174.

[18] M. Eichberg and B. Hermann, "A software product line for static analyses: the opal framework," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, 2014, pp. 1–6.

[19] L. Sui, J. Dietrich, A. Tahir, and G. Fourtounis, "On the recall of static call graph construction in practice," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 1049–1060.

[20] J. Wang, S. Wang, and Q. Wang, "Is there a" golden" feature set for static warning identification? an experimental evaluation," in *Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement*, 2018, pp. 1–10.

[21] L. N. Q. Do, J. Wright, and K. Ali, "Why do software developers use static analysis tools? a user-centered study of developer needs and motivations," *IEEE Transactions on Software Engineering*, 2020.

[22] R. Haas, R. Niedermayr, T. Roehm, and S. Apel, "Is static analysis able to identify unnecessary source code?" *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 29, no. 1, pp. 1–23, 2020.

[23] G. Scanniello, "An investigation of object-oriented and code-size metrics as dead code predictors," in *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, 2014, pp. 392–397.

[24] "Neo4jBloom," https://neo4j.com/product/bloom/, accessed: 2022-07-29.

[25] S. Raemaekers, A. Van Deursen, and J. Visser, "The maven repository dataset of metrics, changes, and dependencies," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 221–224.

[26] "Commit history of removing duplicated dependency," https://github.com/jenkinsci/jenkins/commit/b37dc3242475f1ee5f605deec2954a6b8b07bb64.

[27] "Commit history of removing provided dependency," https://github.com/jenkinsci/jenkins/commit/63c7c2f70392ad2a8d6aed83d6593e27995017e5.