**Abstract**—Modern software systems heavily rely on external libraries developed by third-parties to ensure efficient development. However, frequent library upgrades can lead to compatibility issues between the libraries and their client systems. In this paper, we introduce **COMPsuite**, a dataset that includes 123 real-world Java client-library pairs where upgrading the library causes an incompatibility issue in the corresponding client. Each incompatibility issue in **COMPsuite** is associated with a test case authored by the developers, which can be used to reproduce the issue. The dataset also provides a command-line interface that simplifies the execution and validation of each issue. With this infrastructure, users can perform an inspection of any incompatibility issue step-by-step for a more detailed investigation. We make **COMPsuite** publicly available to promote open science. We believe that various software analysis techniques, including compatibility checking, debugging, and regression test selection, can benefit from **COMPsuite**. The demonstration video of **COMPsuite** is available at https://www.youtube.com/watch?v=7DQGsGs_65s.

**Index Terms**—Incompatibility issue, software libraries, dataset

I. INTRODUCTION

Modern software systems are becoming increasingly complex due to the need for integrating various components developed by different teams or organizations. These components are often subject to continuous evolution, and as a result, ensuring that new upgrades to third-party libraries do not cause any compatibility issues with the existing software system is a challenging task. The complexity of these systems and the number of dependencies involved make it difficult to anticipate and identify incompatibilities that may arise from updates to external components. Incompatibility issues resulting from upgrades to external components can compromise the reliability of software systems, potentially leading to significant financial losses for the organizations that rely on these systems.

Many techniques have been proposed to address third-party library compatibility issues, including regression testing [1], static analysis [2], dependency conflict detection [3], and client-specific compatibility checking [4]. These techniques address library compatibility issues in different dimensions and have been evaluated with their own isolated datasets.

An excellent dataset has the potential to serve as a valuable reference for future research in this field. However, composing the dataset requires intricate manual validation, e.g., confirming whether the cause of a test failure is due to runtime exception, assertion violation, or other reasons. Therefore, we propose **COMPsuite**, the first incompatibility issue dataset focusing on library behavioral incompatibility with concrete reproducible test cases. Each test case is isolated and validated, enabling the direct manifestation of the incompatibilities.

**COMPsuite** comprises 123 real-world Java client-library pairs such that upgrading any library results in incompatibility issues for the corresponding client. Every incompatibility issue in **COMPsuite** contains a test case created by developers, allowing for the reproduction of the issue. On top of this dataset, we also developed an automated command-line interface, which streamlines all processes of the reproduction, such as downloading and compiling a project, running target tests and re-running the tests after a library upgrade. With this infrastructure, users may reproduce an incompatibility issue programmatically with minimal efforts.

**Contribution.** We make the following contributions:

1) We construct a dataset, **COMPsuite**, including 123 reproducible, real-world client-library pairs that manifest incompatibility issues when upgrading the library. These data points originate from 88 clients and 104 libraries.

2) We created an automated command-line interface for the dataset. With this interface, users can programmatically replicate incompatibility issues from the dataset with a single command. The interface also offers separate commands for each step of the reproduction of incompatibility issues.

We envision that **COMPsuite** to be used to evaluate various program analysis techniques, including compatibility checking, module-level regression testing selection, and debugging techniques. More detailed information can be found in Section IV. The dataset and tool are available at: https://github.com/compsuite-team/compsuite.

II. DATASET CREATION

A. Subjects Selection

To ensure the representativeness and reproducibility of the **COMPsuite** dataset, we focus on including high-quality and popular client projects and libraries. The selection of client projects was sourced from GitHub, a widely recognized online community for hosting open-source codebases. To ensure the inclusion of the most popular projects, we systematically sorted all the available projects in descending order based on their number of stars on GitHub and selected the target clients
from the top of the list. The selection of libraries was sourced from Maven Central, which hosts 33.5M of Java libraries and their associated binaries, making it a widely used repository of libraries for Java API and library research. We include a library in the dataset only if it has more than 100 usages (i.e., clients) on Maven Central. Our selection criteria aimed to ensure the inclusion of popular and widely used client projects and libraries in the dataset, thereby maximizing its relevance and usefulness to the research community.

Among the highly-rated client projects, our selection criteria focused on those that use Maven as their build systems, given its widespread adoption and maturity. Maven provides a standardized approach to managing Java projects and their dependencies, where each library dependency in a Maven client project is represented as an item in a pom.xml file, making it easy to identify and edit library versions programmatically. Furthermore, Maven offers built-in functionality for running unit tests and generating test reports, which simplifies the identification and diagnosis of incompatibility issues arising from test executions. Since Maven projects typically rely on Maven Central as their centralized repository for hosting and downloading libraries, the process of obtaining and managing libraries in our dataset is simplified.

Table I presents the top 5 client projects and libraries in the COMPSUITE dataset, ranked by popularity. For each client project, we provide information on its lines of code (LoC) and the number of stars it has received on GitHub, while for each library, we include its number of usages by other projects from Maven Central. In total, COMPSUITE comprises 123 incompatible client-library pairs. These pairs encompass 88 distinct clients and 104 libraries altogether. On average, the affected clients have 2.5K stars on GitHub and 358.7K lines of code, while incompatible libraries have 4.8K usages on Maven Central. Thus, we believe that the incompatibility issues present in the COMPSUITE dataset have a significant impact on a large number of codebases and can affect many users of the libraries, either directly or indirectly.

To ensure that all client projects in the dataset are executable and the runs are reproducible, we performed a series of checks on each project. First, we checked out the project to the version (SHA) at the time of the dataset creation, referred to as the base version. Next, we ran the standard Maven project compilation command to verify if the project compiles successfully. If the project fails to compile, we excluded it from the dataset. Subsequently, we ran the standard Maven test command to execute all the tests in the project, ensuring that all tests pass on the base version. We excluded any project that fails to pass tests at this stage. Finally, we only included the projects that successfully compile and pass all tests on the base version, thereby ensuring that the dataset only consists of projects which can be executed and whose executions can be reproduced.

B. Data Collection

We collected the data following the below procedures. Figure 1 visualizes the overall architecture of COMPSUITE. In the upper left portion of Fig. 1, we illustrate the approach taken by COMPSUITE to identify incompatibilities between a client project and its dependent libraries. Specifically, for each client project on its base version, we upgraded each of its dependent libraries and tested if the upgrade caused any test failures. Our intuition behind this approach is that since all the tests in the client passed on the base version, if upgrading any library causes a test failure, that library upgrade must have introduced incompatibility issues. We refer to the test that flips from passing to failing as an incompatibility-revealing test.

To automatically upgrade the libraries and run the tests, we utilized the Maven Versions Plugin. For a given client project, we scanned its dependency list using this plugin to identify all the libraries that had newer versions available on Maven Central. If a library had a newer version, we marked it as upgradable. Next, for each upgradable library, we used the plugin to upgrade it by updating the pom.xml file to the most recent version on Maven Central. We then re-executed the test suite of the client. If any tests failed during this run, we marked the client-library pair as having an incompatibility issue and marked the test as an incompatibility-revealing test of this issue. It is crucial to note that we only upgraded one library at a time to isolate failures caused by different libraries. To ensure the accuracy and dependability of the dataset, we carried out a manual verification process for each identified incompatibility issue. In particular, we carefully examined the test failure messages and reports to confirm that they were indeed caused by the upgraded library. For each incompatible client-library pair, we selected a single incompatibility-revealing test to be included in the final dataset. In cases where a client-library pair had multiple incompatibility issues, we chose the one that we deemed most representative and easy to comprehend.

Finally, we persisted the metadata of all the selected incompatibility issues in a collection of json files. Figure 2 presents the metadata of an incompatibility issue in the COMPSUITE dataset. The data schema includes the ID of the issue, client project name, SHA of the client base version, URL of the client project, library name, versions of the old and new
Incompatibilities

Incompatibilities

Metadata

1. "id": "i-49",
2. "client": "wasabi",
3. "sha": "9f2a25f92e49c3044d787320e2d22e15317a8e2",
5. "lib": "org.apache.httpcomponents:httpclient",
6. "old": "4.5.1",
7. "new": "4.5.10",
8. "test": "DefaultRestEndPointTest#testGetRestEndPointURI",
9. "submodule": "modules/export",
10. "test_cmd": "mvn
  \rightarrow org.apache.maven.plugins:maven-surefire-plugin:2.20:test --fn
  \rightarrow -Drat.ignoreErrors=true -DtrimStackTrace=false
  \rightarrow -Dtest=DefaultRestEndPointTest#testGetRestEndPointURI"

For instance, in Fig. 1, the client project employs version 2.2.1 of the org.restlet.jse-org.restlet library on its \( V_{base} \) and version 3.0-M1 on its \( V_{incomp} \). In cases where multiple libraries exhibit incompatibility issues in the client project, we not only create different branches for each library with its name, but also generate a \( V_{incomp} \) version tag for each, with accompanying annotations that denote the corresponding library name and version, as depicted in Fig. 1.

The \( V_{incomp} \) tag for each client-library pair also specifies the specific test that can reveal the incompatibility issue during its run. Following Maven’s convention, the test name is formatted as TestClassName#testMethodName. By simply copying the text from the tag, users can easily run the incompatibility-revealing test on the \( V_{incomp} \) version and observe the incompatibility issue. On the \( V_{base} \) version, all tests should pass. This design aims to simplify the usage of COMP SUITE and make it more accessible and user-friendly.

Using the forked client repositories and version tags provided in the COMP SUITE dataset, users can easily reproduce any incompatibility issue by checking out to \( V_{incomp} \) and running the corresponding incompatibility-revealing test. To compare the behaviors of the client with compatible and incompatible library versions, users can run the incompatibility-revealing test on both \( V_{base} \) and \( V_{incomp} \) and compare the test outcomes. This allows for a clear understanding of the impact of the library upgrade on the client behaviors.

A. Exploring an Incompatibility Issue

To ensure the reproducibility of incompatibility issues and to facilitate the demonstration of such issues, we have annotated checkpoints in the version histories of the client projects and provided tags that guide users to explore any incompatibility issues present in the COMP SUITE dataset.

As illustrated on the right-hand side of Fig. 1, our approach to handling incompatible client-library pairs involved creating a fork of the original client project for each identified pair, while preserving all code and version history information. To mark the base version of the project, we utilized the \texttt{git tag} command, designating it as \( V_{base} \). Subsequently, we developed a patch to upgrade the library from its old version to its new version, a simple process that can be accomplished with a single line change in the \texttt{pom.xml} file for Maven projects. This patch was then applied to the \( V_{base} \) version, resulting in a new version that we identified as \( V_{incomp} \). Notably, the only difference between \( V_{base} \) and \( V_{incomp} \) lies in the library version used: the old (compatible) version is utilized on \( V_{base} \) while the new (incompatible) version is utilized on \( V_{incomp} \).
(which is configurable). Then, it checks out to the base version, compiles the code, and runs the incompatibility-revealing test. Next, it upgrades the library to the new version, reruns the same test, and reports any failure information to the user.

We also provide a set of commands that break down the entire cycle of incompatibility exploration into separate steps:

1. python main.py --download i-56
2. python main.py --compile i-56
3. python main.py --testold i-56
4. python main.py --testnew i-56

We provide other commands to inspect different aspects of the incompatibility issues from the COMP SUITE dataset. A complete list of these commands can be found on COMP SUITE’s website at https://github.com/compsuite-team/compsuite.

IV. APPLICATION SCENARIOS

We anticipate that both researchers and practitioners can benefit from COMP SUITE to facilitate their investigations and research on errors and test failures induced by library upgrades. COMP SUITE supports the evaluation of various program analysis techniques such as software upgrade compatibility checking, debugging, and module-level regression test selection.

- **Compatibility Checkers and Detectors.** There are three main categories of existing compatibility checking and detection in Java: i) Techniques for detecting API-breaking changes [5]. ii) Techniques for detecting behavioral incompatibilities that cause test failures when a library is upgraded in a client [6]. iii) Techniques for detecting dependency conflicts that exhibit inconsistent semantics between libraries due to class path shading [3]. Developers of techniques in the first two categories can use COMP SUITE as a benchmark to evaluate their tools’ performance. They can run their tools on the COMP SUITE dataset and compare the results with the incompatibility issues present in the dataset. For dependency conflicts detecting techniques, users can slightly modify COMP SUITE by placing both old and new libraries on the class path and checking if the issues can be detected.

- **Module-Level Regression Test Selection.** Regression test selection (RTS) aims to reduce the cost of regression testing by selecting a subset of tests that may change the behavior due to code changes on each program version [7]. Authors of module-level RTS techniques can use COMP SUITE to assess the safety of their approaches. A safe module-level RTS technique should select all the corresponding incompatibility-revealing test cases when the library changes.

- **Debugger.** Existing debugging techniques for Java, include symbolic execution, delta debugging, fault localization, etc. These techniques aim to identify the root cause of errors or failures in software. Developers of debugging techniques can use COMP SUITE as a dataset of compatibility bugs to evaluate their techniques’ ability to perform root cause analysis by identifying the corresponding library change that caused the compatibility issue.

V. RELATED WORK

To cater to the requirements of various research endeavors, numerous outstanding datasets have been made available to date. Just et al. [8] introduced Defects4J, a database supplies actual bugs, fixed program versions, and corresponding test suites. Bui et al. [9] introduced Vul4J focusing on Java vulnerabilities. Jezek et al. [10] released their synthetic corpus of compatibility issues that simulates program evolution. There are also many datasets cater for other research domains and ecosystems [11].

Distinct from the previously discussed datasets, COMP SUITE is the first dataset emphasizes the incompatibility issues caused by Java library behavior changes. This type of issues are prevalent and difficult to detect. We believe that a dataset targeting the library upgrade incompatibility issue will contribute to the advancement of the associated technologies.

VI. CONCLUSION

This paper presents COMP SUITE, a dataset containing 123 real-world Java client-library pairs where library upgrades cause compatibility issues in the corresponding clients. On top of it, we developed a command-line interface, COMP RUNNER, which allows users to quickly check incompatibility issues with a single command and reproduce an incompatibility programmatically for in-depth analysis. We believe that various program analysis techniques like library compatibility checking, debugging, and regression test selection, may benefit from our dataset.

ACKNOWLEDGEMENT

This research is supported by the NTU Start-up Grant.

REFERENCES


