PExReport-Maven: Creating Pruned Executable Cross-Project Failure Reports in Maven Build System

Sunzhou Huang
sunzhou.huang@utsa.edu
University of Texas at San Antonio
San Antonio, Texas, USA

Xiaoyin Wang
xiaoyin.wang@utsa.edu
University of Texas at San Antonio
San Antonio, Texas, USA

ABSTRACT

Modern Java software development extensively depends on existing libraries written by other developer teams from the same or a different organization. When a developer executes the test, the execution trace may go across the boundaries of multiple dependencies and create cross-project failures (CPFs). A readable, executable, and concise CPF report may enable the most effective communication, but creating such a report is often challenging in Java ecosystems. We developed PExReport-Maven to automatically create the ideal CPF reports in the Maven build system. PExReport-Maven leverages the Maven build system to prune source code, dependencies, and the build environment to create a concise stand-alone executable CPF reproduction package from the original CPF project. The reproduction package includes the source code, dependencies, and build environment necessary to reproduce the CPF, making it an ideal CPF report. We performed an evaluation on 74 software project issues with 198 cross-project failures, and the evaluation results show that PExReport can create pruned reproduction packages for 184 out of the 198 test failures in our dataset, with an average reduction of 72.97% in Java classes. A future study will be conducted based on user feedback from using this tool to report real-world CPFs. PExReport-Maven is publicly available at https://github.com/wereHuang/PExReport-Maven. The tool demo is available on the PExReport website: https://sites.google.com/view/pexreport/home.

CCS CONCEPTS

• **Software and its engineering** → *Software maintenance tools.*

KEYWORDS

cross-project failure, test failure, failure report, maven, failure reproduction

ACM Reference Format:
Sunzhou Huang and Xiaoyin Wang. 2023. PExReport-Maven: Creating Pruned Executable Cross-Project Failure Reports in Maven Build System. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3597926.3604929

This work is licensed under a Creative Commons Attribution 4.0 International License.

ISSTA '23, July 17–21, 2023, Seattle, WA, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0221-1/23/07.
https://doi.org/10.1145/3597926.3604929

1 INTRODUCTION

Modern Java software development extensively depends on existing libraries written by other developer teams. When a developer executes the test, the execution trace may go across the boundaries of multiple dependencies and create cross-project failures (CPFs). Typically, the client and third-party developers do not share dependencies or the test environment, which makes reporting CPFs more challenging. None of the existing techniques is able to produce an ideal CPF report that is simultaneously executable, readable, and concise. In particular, code portions generated from program slicing techniques [5, 10] are typically not compilable or executable because they do not consider environment dependencies beyond source code. Software debloating [6, 7] techniques are applied directly to binary code, so the corresponding source code is difficult to acquire. Packaging the whole project will guarantee failure reproduction but lead to high storage and network transmission costs, as well as noise during debugging. This creates the trilemma of the cross-project failure report, as shown in Figure 1.

To solve the CPF trilemma, we designed a framework called PExReport in our previous work [8]. We implemented PExReport in Maven [1] build system to create a tool called PExReport-Maven for solving the trilemma in the Java ecosystem. In this paper, we reveal additional implementation details with a concrete example, describe the tool's anticipated users, and propose a plan for future study based on user feedback.

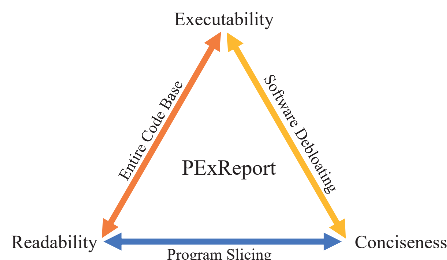

Figure 1: Cross-Project Failure Report Trilemma

2 PEXREPORT-MAVEN DESCRIPTION

2.1 Overview

Figure 2 shows an overview of PExReport-Maven's workflow. The input to PExReport-Maven is a failed test case from an existing Maven build environment, and the output is a pruned stand-alone executable failure report. PExReport contains two major phases: the collection phase to collect information about necessary Java source code, JAR dependencies, and the Maven build environment for re-executing the failed test case, and the reconstruction phase to reconstruct a stand-alone Maven project for the failed test case

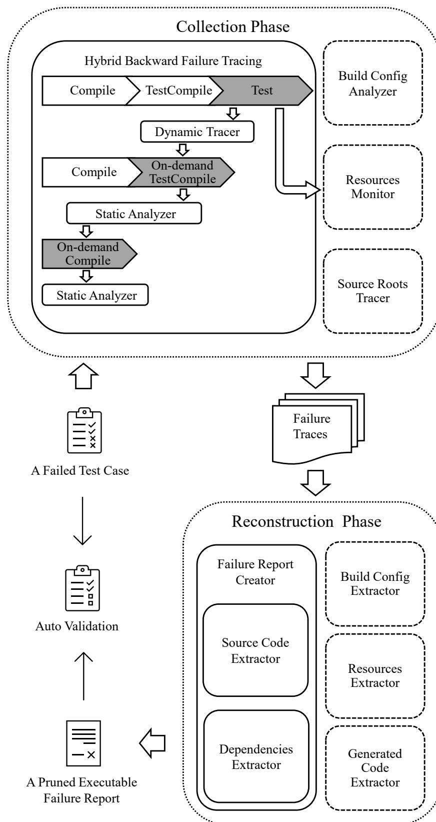

Figure 2: PExReport-Maven Workflow

based on the collected information. The failure traces are the collected information from a failed test case in the first phase. The arrows show the information flow between each component of PExReport. The reconstructed Maven project for failure reporting will be automatically validated by checking whether exactly the same error messages are triggered as in the original failure. Once the project is validated, it can serve as a reproduction package of the original failed test case and will be reported as a pruned executable failure report to the developers of dependency code.

2.2 Core Maven Phases for the Lifecycle of Test Failures

PExReport-Maven is designed based on the core phases of the Maven build system involved in the lifecycle of test failures. [4] These phases are generally executed in the order they are presented, but some phases may be omitted if not required for the current project.

- **GenerateSources** (optional): generate any source code for inclusion in compilation.
- **ProcessResources** (optional): copy and process the resources into the destination directory.
- **Compile** (mandatory): compile the source code of the project.
- **TestCompile** (mandatory): compile the test source code into the test destination directory.
- **Test** (mandatory): run tests using a suitable unit testing framework.

The three mandatory phases, namely `Compile`, `TestCompile` and `Test`, are the basic steps to reproduce a test failure used by Maven. We noticed that `compile` and `test-compile` are both used to compile source code. The main reason is to make the software source code independent of the test source code. Code generation and resource management are quite popular in complex real-world Java applications. Although they are optional tasks, we should not underestimate their role in reproducing real-world failures.

Figure 3 is a concrete example that shows the lifecycle of a Java test failure in the Maven build system. This example only involves sample Java source code and JAR dependencies. The GenerateSources and ProcessResources are excluded for simplicity.

- **Compile**: Java compiler compiles all source code in the left rounded rectangle (red) by referencing JARs.
- **TestCompile**: Java compile compiles all test source code in the middle rounded rectangle (blue) by referencing classes of *Compile* task and test JARs.
- **Test**: In the right rounded rectangle (orange), the JVM runs *ClientTest.clientTestCase* test by loading classes from *Compile* and *test* tasks.

2.3 Hybrid Backward Failure Tracing

In the collection phase, the base component is hybrid backward failure tracing, a three-step analysis that traces failure-related Java source code and JAR dependencies (Java classes or Bytecode). This component compiles and executes the failed test case in its original Maven build environment, and tracks Java class usage at *Test*, *TestCompile* and *Compile* core Maven phases. These three phases are essential for any test code to be compiled and executed, as shown in Figure 3.

The workflow of hybrid backward failure tracing is shown in the top part of Figure 2. In build process order, the compiled Java classes in *Compile* task are provided to the succeeding *testCompile* task as dependencies; after that, the compiled Java classes of *testCompile* are loaded to *test* task for execution. The build process must be irreversible to ensure that the Java classes that compile before them never depend on those that compile after them. If a test failure occurs, all the failure-related Java classes can be tracked in the reversed build process order. Since failure happens at the end of the build process (*test* task), backward tracing can be performed to obtain the failure dependency tree.

The detailed tracing of the failure example in Figure 3 is shown in the following:

- **Round 1**: executes *Compile* (all source) and *TestCompile* (all test source) tasks, then run *ClientTest.clientTestCase* in *Test* task. The dynamic tracer component records all dynamically loaded classes in the *Test* task, accordingly.
  - Input: *ClientTest.clientTestCase*
  - Output Traces:
    - From TestCompile: *ClientTest*, *ClientTestHelp* (dynamic)
    - From Compile: *ClientClass*
    - From Internal: *ClassForExternal* (client-internal-util.jar)
    - From External: *ClassLibraryFailure* (library-failure.jar)
- **Round 2**: executes *Compile* task, and *TestCompile*

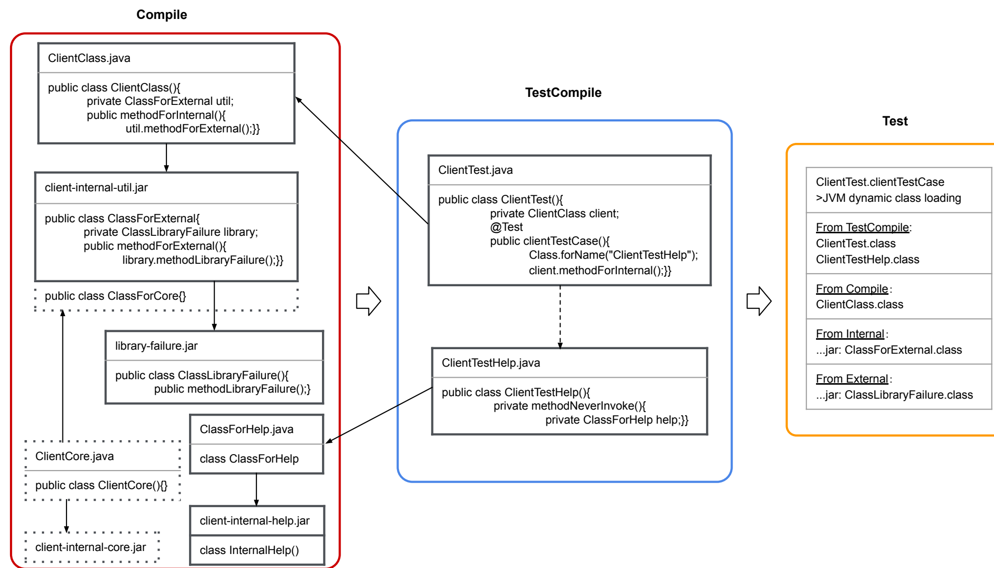

Figure 3: Maven Test Failure Example

(*ClientTest.java*, *ClientTestHelp.java*) task. The static analyzer component records all referred classes in *TestCompile* task, accordingly.
  - Input: Traces from *Round 1*
  - Output Traces:
    * From TestCompile: *ClientTest*, *ClientTestHelp* (dynamic)
    * From Compile: *ClientClass*, *ClassForHelp* (static)
    * From Internal: *ClassForExternal* (client-internal-util.jar)
    * From External *ClassLibraryFailure* (library-failure.jar)
- **Round 3**: executes *Compile* task (*ClientClass.java*, *ClassForHelp.java*) task. The static analyzer component records all referred classes in *Compile* task, accordingly.
  - Input: Traces from *Round 2*
  - Output Traces:
    * From TestCompile: *ClientTest*, *ClientTestHelp* (dynamic)
    * From Compile: *ClientClass*, *ClassForHelp* (static)
    * From Internal: *ClassForExternal* (client-internal-util.jar), *InternalHelp* (client-internal-help.jar)
    * From External: *ClassLibraryFailure* (library-failure.jar)

2.4 Enhancement Components

In addition to the base component (hybrid backward failure tracing), our tool also consists of three enhancement components to further reconstruct a reliable build environment.

- **Handling of the build configuration.** In Maven, all work is done by plugins, but most plugins are irrelevant for reproducing failure. The Build Configuration Analyzer fetches the *effective build configuration* (*help:effective-pom* [3]), which Maven uses to build the failed test at run time. The effective POM gathers all build configuration values scattered in all build configuration files and resolves configuration value overwriting among multiple configuration files based on the *nearest definition* for resolving dependencies. However, the *effective POM* is still redundant for reproducing the failed test compared with the required configuration values of the three core phases. Since phases in the Maven build process are performed by various plugins and the plugins can be attached to different build phases (e.g., the Java compiler plug-in can be attached to the Compile and TestCompile phases), we further leverage the attachment relationship to identify all the plugins that are attached to the three core build phases. It also excludes code-style checking and analysis plugins because they do not directly affect compilation or testing. The build configuration extractor collects the configuration information from these plugins and updates the failure reproduction package.
- **Handling of resource files.** In the Linux kernel, the inotify API provides a mechanism for monitoring filesystem events. We use Pyinotify [2], a Python module that wraps the inotify API, to monitor all resource activities under the project scope. As shown in Figure 2, the resource monitor only outputs files accessed at *Test* phase because the *Process resources* phase copies all resource files to the target folder. For necessary resource files accessed during the compilation process (e.g., templates of generated source code), we specially handled them by the Generated Code Extractor. Our resource monitor also ignores all files and directories generated during the build or test process because these files should not be included in the reproduction package (unnecessary and causing path conflicts). For the files copied to the target location from source locations, we do not consider them generated files because the Resource Monitor can trace back to their source copies in the original project.
- **Handling of source code generation.** The Maven build process may generate new source code in various ways, such as by creating code from template files, generating parsing code from syntax or XML files, or even directly fetching source code from remote locations. Furthermore, code generation is often implemented in third-party tools and plugins. To handle such high flexibility in code generation in a general way, we omit the code generation process and directly

include the generated source code in the failure reproduction package. At *GenerateSources* phase, the build tools use source code root paths (Source Roots) to locate all original and generated source code. In the Collection Phase, our Source Roots Tracer tracks all the accessed source code root paths from the debug information of compilation. Next, the Generated Code Extractor utilizes the paths to identify the generated source code and excludes all original source code. In addition, the build tools may also generate some source code from code annotation processing. Our Generated Code Extractor excludes such code because it will cause compilation conflicts.

3 ENVISIONED USERS

The following users can make use of PExReport-Maven:

- Developers of Java applications who use the Maven build system and third-party libraries (open-source or proprietary) can use our tool to report test failures.
- All software defect datasets suffer from software breakages that are mostly related to software dependencies. Dependency caching can effectively prevent software breakages and ensure long-term reproducibility. [11] Because PExReport-Maven can preserve Maven build environments (e.g., required dependency caching) and necessary portions of Java projects for reproducing CPFs, our tool is useful for researchers who want to share their CPF datasets with pruned build environments. For example, our tool has been validated on the Sensor dataset, which is a dependency conflict dataset. [8, 9]

4 USING THE TOOL

4.1 Download

PExReport have been uploaded to a open source repository on GitHub. It can be download use following `git` command.

```
git clone https://github.com/wereHuang/PExReport-Maven
```

4.2 Create a Working Environment

(1) Use a Linux machine; the tool is verified in Ubuntu 22.04 LTS. This tool also supports containerized environments with Linux host machines.

(2) Install Maven 3 and pip for Python3.

```
sudo apt update
&& sudo apt install maven python3-pip -y
```

(3) Install Python packages using `requirements.txt` under the root folder of PExReport-Maven.

```
sudo pip install -r requirements.txt
```

(4) Install the customized Maven Archetype for PExReport-Maven. Change the working directory to pexreport-archetype, then execute:

```
mvn clean install
```

(5) Install Java 8 (required for test)

```
sudo apt install openjdk-8-jdk -y
```

4.3 Command Line Usage

(1) A CPF can be triggered by the following Maven command:

```
mvn clean test -Dtest=TEST_NAME
```

(2) Pass the test name, path of the source project, group ID for internal dependencies, and report name of output to `per.py`:

```
per.py -n TEST_NAME -s SOURCE -g GROUPID -t TARGET
```

(3) The CPF report named TARGET will be created, which is a reproduction package for the original CPF. In the directory of the generated CPF report, use the same Maven command that triggered the original CPF to reproduce.

5 EVALUATION AND STUDIES

Our tool has been evaluated on executability and conciseness in our prior work [8]. The evaluation of 74 software project issues with 198 CPFs achieved a high reproduction rate for 184 out of 198 CPFs, with an average reduction rate of 72.97% on Java classes.

We plan to further collect user feedback to understand how much our tool can help developers report real-world test failures. This study could also assist us in answering the question, "How could the community enhance this tool?"

6 CONCLUSIONS

An executable test case is one of the most desirable features of failure reports. When reporting *cross-project failures (CPFs)* to library developers, a test case is even more helpful because code is a natural way to describe interactions between library code and client code. In this paper, we present PExReport-Maven, a tool to automatically create pruned executable CPF reports for reporters using the Maven build system, and solve the CPF report trilemma in the Java ecosystem. The future study will be conducted based on user feedback from using this tool for real-world test failure reporting.

REFERENCES

[1] 2002. Apache Maven. https://maven.apache.org/.
[2] 2015. seb-m/pyinotify. https://github.com/seb-m/pyinotify.
[3] 2022. Apache Maven help:effective-pom. https://maven.apache.org/plugins/maven-help-plugin/effective-pom-mojo.html.
[4] 2023. Introduction to the Build Lifecycle. https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html.
[5] Hiralal Agrawal and Joseph R. Horgan. 1990. Dynamic Program Slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*. 246–256.
[6] Bobby R Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. 2020. 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*. 135–146.
[7] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective program debloating via reinforcement learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 380–394.
[8] Sunzhou Huang and Xiaoyin Wang. 2023. PExReport: Automatic Creation of Pruned Executable Cross-Project Failure Reports. In *Proceedings of the 45th International Conference on Software Engineering*.
[9] Ying Wang, Rongxin Wu, Chao Wang, Ming Wen, Yepang Liu, Shing-Chi Cheung, Hai Yu, Chang Xu, and Zhi-liang Zhu. 2021. Will Dependency Conflicts Affect My Program's Semantics. *IEEE Transactions on Software Engineering* (2021).
[10] Mark Weiser. 1984. Program slicing. *IEEE Transactions on software engineering* 4 (1984), 352–357.
[11] Hao-Nan Zhu and Cindy Rubio-González. 2023. On the Reproducibility of Software Defect Datasets. In *Proceedings of the 45th International Conference on Software Engineering*.

Received 2023-05-18; accepted 2023-06-08