Assessing Patch Correctness in Test-based Automated Program Repair

Jared Wang

Texas A&M University

jwangj001@tamu.edu

Mentor: Dr. Ali Ghanbari

Auburn University

azg0212@auburn.edu

Abstract

Effective bug fixing in software development relies on the precision of the patches applied to identified issues. Even if a patch passes all its tests, it may not completely resolve the underlying bug, a situation referred to as patch overfitting. This discrepancy can occur because test cases may only partially capture the intended behavior of a program. While manual inspection of these patches is often necessary, it can be a labor-intensive process. This project aims to explore these challenges by evaluating patch correctness within the Defects4J framework through mutation analysis. By examining the limitations of current automated testing methods and suggesting potential improvements, this research seeks to contribute to a better understanding of patch efficacy and advancements in software maintenance and reliability.

Introduction

The challenge of maintaining software reliability is compounded by the introduction of defects and the subsequent need for effective patches. In the context of large-scale software systems, this challenge becomes even more pronounced. Defects4J serves as a critical resource, providing a structured database of real-world bugs and their corresponding patches. This report documents our systematic investigation into patch correctness using both automated and manual analysis techniques.

Background and Related Work

Several studies have highlighted the significance of empirical data in software testing. The Defects4J framework, which encompasses a broad array of real bugs from well-known open-source projects, stands as a pivotal resource in this domain. Below are the key articles that have informed this study and provides context for the methodologies employed:

1. **Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs** (Just, Jalali, & Ernst, 2014) [1]
    - This paper presents Defects4J, an innovative database and an extensible framework that provides a collection of real-world bugs from open-source Java programs, facilitating reproducible and comparable empirical studies in software testing research. The initial version of Defects4J includes 357 real bugs extracted from five widely-used open-source Java projects: JFreeChart, Commons Lang, Commons Math, Joda-Time, and Mockito. Each bug is accompanied by a corresponding test suite, which is essential for identifying the defect and verifying the correctness of patches.
    - The motivation behind Defects4J stems from the challenges in conducting empirical studies in software testing that are reproducible and comparable. Historically, many studies have relied on synthetic bugs or small-scale examples, which do not adequately reflect the complexity and diversity of real-world software. Defects4J addresses this issue by providing a large and diverse collection of real bugs, making it easier for researchers to evaluate testing techniques and automated program repair tools in a more realistic and controlled environment.
    - Key features of Defects4J:
        i. **Real Bugs:** Defects4J includes 357 real bugs from five well-known Java programs. Each bug is documented with its corresponding faulty and fixed program versions, as well as a test suite that exposes the bug. This collection helps researchers evaluate the effectiveness of testing and repair techniques on real-world defects.
        ii. **Extensibility:** The framework is designed to be extensible, allowing new bugs to be added with minimal effort. Integration with version control systems such as Git facilitates the addition of new bugs and updates to existing ones, ensuring that the database remains current and relevant.
        iii. **Test Execution Framework:** Defects4J provides a suite of utilities for common software testing tasks, including code coverage analysis, regression testing, and fault localization. These tools help researchers conduct thorough and systematic evaluations of their testing and repair techniques.
2. **Patch Correctness in Automated Program Repair Based on the Impact of Patches on Production and Test Code** (Ghanbari and Marcus 2022) [2]

- This article focuses on the criteria for evaluating patch correctness generated by Automated Program Repair (APR) tools. Automated Program Repair (APR) systems aim to automatically generate patches for software bugs. However, these systems often produce patches that pass existing test suites without truly fixing the underlying issue, leading to the need for automatic correctness assessment. This paper introduces Shibboleth, a technique for evaluating patch correctness by analyzing the impact of patches on both production and test code. Shibboleth goes beyond traditional syntactic analysis by incorporating semantic and coverage-based assessments.
- Shibboleth assesses patch correctness through a multi-faceted approach:
    i. Syntactic Similarity: This facet measures the textual differences between the patched and unpatched code. It helps identify superficial changes that may not significantly alter program behavior.
    ii. Semantic Similarity: Using dynamic analysis, this facet evaluates how the program's behavior changes after applying the patch. By running the patched program and comparing its execution to the unpatched version, Shibboleth identifies discrepancies in behavior that may indicate an incorrect patch.
    iii. Code Coverage: Shibboleth assesses the impact of the patch on test cases, particularly focusing on branch coverage. By analyzing which parts of the code are exercised by the test suite before and after applying the patch, Shibboleth identifies patches that may have inadvertently reduced test coverage or missed critical code paths.
- The effectiveness of Shibboleth was evaluated using 1,871 patches generated by 29 Java-based APR systems for programs within the Defects4J repository. The results showed that Shibboleth outperformed state-of-the-art techniques in both ranking and classification tasks. Specifically, it demonstrated higher precision and recall in identifying correct patches, proving its robustness and reliability.

3. **Mutation Testing Advances: An Analysis and Survey** (Papadakis, Kintis, Zhang, Jia & Traon, 2019) [3]
    - This article delves into the concept of Mutation Testing. Mutation testing is a fault-based software testing technique that involves modifying a program's source code in small ways to create mutants. These mutants simulate common programming errors. The goal is to evaluate the effectiveness of a test suite by determining how many of these mutants it can detect and eliminate. Over the years, significant advances have been made in mutation testing, making it a more practical and powerful tool for software quality assurance.
    - Mutation testing is built on two main concepts:

i. Mutants: These are versions of the original program with small, deliberate changes (mutations) introduced. Each mutant represents a potential fault in the program.
ii. Mutation Operators: These are rules or patterns used to create mutants. Common mutation operators include changing arithmetic operators (e.g., replacing + with -), altering logical operators (e.g., replacing && with ||), and modifying control flow statements (e.g., changing if conditions).

- Mutation testing is a powerful technique, but it has drawbacks:
  i. Computational Cost: Generating and executing a large number of mutants can be computationally expensive, especially for large codebases.
  ii. Equivalent Mutants: Some mutants are functionally equivalent to the original program and cannot be detected by any test suite. Identifying and handling these equivalent mutants is a significant challenge as it requires manual effort.
  iii. Test Suite Effectiveness: Ensuring that the test suite is comprehensive enough to detect subtle faults introduced by mutants requires careful design and inspection.

4. **Automated Classification of Overfitting Patches with Statically Extracted Code Features** (Ye, Gu, Martinez, Durieux, & Monperrus, 2021) [4]
   - This article introduces Overfitting Detection System (ODS), a machine learning approach that determines the likelihood of a patch being overfit. In the other words, ODS is a teachable machine, and this system is composed to 2 phases:
     i. Training Phase – In this phase is where researchers feed data with known correctness of the patches into the machine so that machine can predict new and unseen patches.
     ii. Prediction Phase - The machine predicts the overfitting probability of the inputted patch based on the known data during this phase.
   - This system has 3 features:
     i. Code Description Features: These features describe the properties of the code itself, such as the number of lines, types of statements, and other structural characteristics like operator types.
     ii. Repair Pattern Features: These features capture the patterns used to repair the code. They might include the types of modifications made, such as variable changes, method invocations, or control flow alterations. The aim is to understand the common patterns in overfitting patches.
     iii. Contextual Syntactic Features: These features consider the context in which code modifications occur. For example, they might look at the surrounding code to see how the patch integrates with the existing codebase. This helps in understanding whether the patch fits well syntactically within its context.

- Based on previous experiments, we can see that ODS is a reliable method in identifying overfitting patches. Experiments have shown that it has:
    i. Higher Accuracy: ODS shows higher accuracy in identifying overfitting patches compared to the baseline methods.
    ii. Improved Precision and Recall: The model demonstrates significant improvements in precision and recall, indicating fewer false positives and false negatives.
    iii. Better F1-Score and ROC-AUC: ODS achieves superior F1-scores and ROC-AUC values, showcasing its robustness and reliability in patch classification.

5. **Evaluating Representation Learning of Code Changes for Predicting Patch Correctness** (Tian, Liu, Koyuncu, Li, Klein, & Bissyande, 2020) [5]
    - This article investigates the use of representation learning to predict the correctness of patches in software code. The primary goal is to improve the accuracy of automated program repair (APR) systems by leveraging advanced machine learning techniques to understand and predict which patches are likely to be correct. Unlike traditional machine learning, this approach uses neural networks to enhance the accuracy of predicting whether a patch is correct.
    - Some key elements with this approach are:
        i. **Token Embeddings**: Using token-level embeddings to capture the low-level syntactic elements of code changes.
        ii. **Abstract Syntax Trees (ASTs)**: Leveraging tree-based representations to encapsulate the hierarchical structure of code.
        iii. **Graph-Based Representations**: Utilizing graphs to model the dependencies and relationships within the code.
    - Significant findings with this method:
        i. Token Embeddings: Models using token embeddings show significant improvements in capturing syntactic patterns, but struggle with deeper semantic understanding.
        ii. AST Representations: AST-based models provide a better balance between syntax and semantics, leading to more accurate predictions.
        iii. Graph-Based Models: These models excel in capturing complex relationships and dependencies, resulting in the highest prediction accuracy among the tested approaches.
    - Studies demonstrate that this advanced representation learning technique, particularly those based on ASTs and graph structures, can significantly enhance the ability of APR systems to predict patch correctness. This represents a substantial step forward in developing more reliable and effective automated program repair tools.

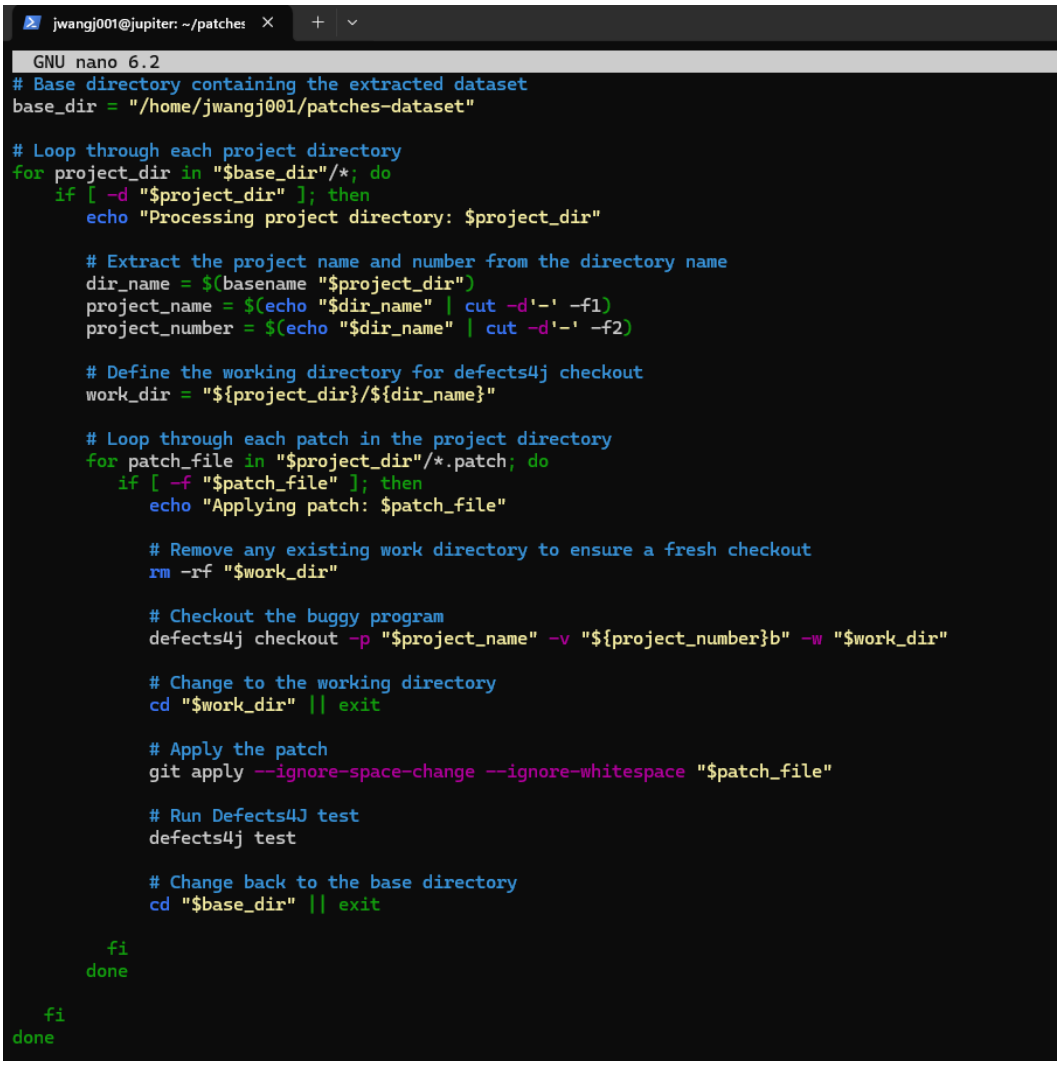
Methodology

My approach for this project involved a thorough evaluation of patches within the Defects4J database. I employed the following procedures to assess the effectiveness of the patches:

1. **Extracting Buggy Programs and Corresponding Patches**: I began by extracting the buggy programs and their associated patches from the Defects4J repository by checking them out from the defects4j library using the command shown below:

   defects4j checkout -p <project_id> -v <version_id> -w <work_dir>

2. **Automate Patch Application and Testing**: To automate the process of applying the patch and re-checking out the buggy program, I used a Bourne Shell script that iterates through every buggy program in the given directory and its corresponding patches. I then used git to apply a patch to the buggy program and used the "defects4j test" command to evaluate the effectiveness of the patch. Every time after applying a patch to the source code, I had to checkout that buggy program again to ensure that the program is in its original state before applying another patch to it. Attached below is the Bourne Shell Script that I developed to automate this task:



3. **Manual Inspection and Analysis of All Failed Patches**: For all the patches that failed their tests and/or cannot be applied to the source code, I manually inspected them and fixed them whenever possible.

Tools Learned in this Research Intern

During this internship, I developed proficiency in several crucial tools that were integral to the project's success. Each tool offered specific functionalities that supported various aspects of the work, including automation, version control, build management, and testing. The following is a summary of the key tools and their contributions:

1. Linux

- In this project, Linux was used for script automation, environment setup, and running various tools and commands. The command-line interface in Linux offers powerful utilities for automation and system management, making it an ideal choice for this type of project.

2. Git
    - Git is a distributed version control system designed to handle large projects with speed and efficiency. It allows multiple developers to work on the same project simultaneously, track changes, and collaborate efficiently. Git's branching model facilitates experimentation, as branches can be easily created, merged, and deleted.
    - In this project, Git was used to apply patches, ensuring smooth integration of patches into the codebase.
3. Maven
    - Maven is a build automation tool used primarily for Java projects. It describes how the software is built and its dependencies. Maven provides a uniform build system, project information, and guidelines for best practices development, which makes it easier to manage large projects.
    - In this project, Maven serves as an environment to run PITest (PIT) on the projects.
4. PITest (PIT)
    - PIT is a state-of-the-art mutation testing tool for Java. It runs on top of existing unit tests and uses mutation operators to generate mutants. It measures the effectiveness of test cases by checking how many mutants are detected and eliminated.
    - The purpose of this tool is to help identify the blind spots in the test suite, which helps enhance the thoroughness in testing and quality of the software.

Challenges

Throughout the project, several significant challenges were encountered, which highlighted the complexities and limitations of automated patch correctness assessment:

1. **Patch Application Issues**
   A common issue was patches that failed to apply to the source files. This problem often stemmed from syntax errors or inconsistencies in the .patch files. These issues required manual intervention to resolve. The process involved:
    a. Creating a backup of the original source file:

```
cp source_file.java source_file_original.java
```

b. Manually editing the source file to incorporate the intended changes from the patch:

```
nano source_file.java
```

c. Generating a new patch file to capture the differences:

```
diff -u source_file_original.java source_file.java > example.patch
```

d. Applying the corrected patch to the original file:

```
patch source_file_original.java < example.patch
```

e. Verifying the accuracy of the applied patch:

```
cat example.patch
```

2. **Incomplete Fixes Leading to New Bugs**
   Another challenge involved patches that, while successfully addressing the specific bug, inadvertently introduced new issues, causing other tests to fail. This phenomenon, known as patch overfitting, highlights the difficulty in creating comprehensive test cases that cover all possible scenarios. The presence of these incomplete fixes indicated that the patches were not holistic solutions and required further refinement to avoid new defects.

3. **Fundamental Patch Failures**
   Some patches failed to resolve the initial bug entirely, as indicated by the persistence of failed tests. These patches were deemed fundamentally incorrect and needed to be discarded. This issue emphasized the challenge of distinguishing between correct and incorrect patches in an automated environment, especially when initial test cases are insufficiently comprehensive.

These issues emphasize the need for advancements in the automation process. Future research should focus on developing and testing novel algorithms that enhance automated program repair (APR) process. Specifically, efforts should aim to reduce or eliminate the occurrence of incorrect patches and improve the overall effectiveness of automated patch generation and evaluation.

Future Work

Due to the time constraints, I was only able to work up to the point of analyzing and fixing the failed patches. If given more time to work on this project, I would apply PIT Mutation Testing to investigate why patches that were labelled as “INCORRECT” still passed all its tests. By determining the number of mutants killed and the number of mutants that survived, I can gain

deeper insights into the specific behaviors and edge cases that the test suite is currently handling or missing. On top of that, I can also find ways to enhance the comprehensiveness of the tests, such as coming up with edge cases or other important cases that the original test suites did not consider. If a high number of mutants survive, it suggests that the test suite might be missing important checks and needs enhancement.

To improve this project as a whole, we should focus on enhancing automated testing methodologies and expanding the scope of analysis to include a wider array of software projects. One key aspect is to integrate the suggestions from the reviewed articles, especially those related to enhancing the Defects4J framework by increasing the number of bugs in Defects4J. Doing so would provide a richer database full of bugs, thus offering a more comprehensive testbed for evaluating patch correctness.

Acknowledgements

This work was supported by the Distributed Research Experiences for Undergraduates (DREU) program. The guidance and mentorship of Dr. Ghanbari were invaluable in this research. All errors and omissions are my responsibility.

References

[1] René Just, Darioush Jalali, and Michael D. Ernst. "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs." Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA), 2014, pp. 437-440. ACM.

[2] Ali Ghanbari, and Andrian Marcus. "Patch Correctness Assessment in Automated Program Repair Based on the Impact of Patches on Production and Test Code." Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea. ACM, 2022, pp. 237-248. https://doi.org/10.1145/3533767.3534368.

[3] Mike Papadakis, Michail Kintis, Jie Zhang, Yue Jia, and Mark Harman. "Mutation Testing Advances: An Analysis and Survey" Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), 2019, pp. 1165-1174. ACM.

[4] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. "Classification of Overfitting Patches with Automatic Feature Learning." Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), ACM, 2021. PDF file.

[5] Haoye Tian, Kiu Liu, Abdoul Kader Kabore Anil Koyuncu, Li Li, Jacques Klein, and Tegawende F. Bissyande. "Evaluating Representation Learning of Code Changes for Predicting Patch Correctness." Proceedings of the 42nd International Conference on Software Engineering (ICSE), ACM, 2020. PDF file.