REGRESSION TESTING OF FEATURE ORIENTED PROGRAMS

A PROJECT REPORT

Submitted by

SHASHI KANTA SAHOO (7th Sem) 2101020181

In partial fulfilment for the award of the degree of

BACHELOR OF TECHNOLOGY

IN

COMPUTER SCIENCE & ENGINEERING

C.V. RAMAN GLOBAL UNIVERSITY

BHUBNESWAR- ODISHA -752054

NOVEMBER 2024

C.V. RAMAN GLOBAL UNIVERSITY

BHUBANESWAR-ODISHA-752054

BONAFIDE CERTIFICATE

Certified that this project report **" Regrssion testing of feature oriented programs "** is bonafide work submitted by **SHASHI KANTA SAHOO, Registration No.2101020181 ,** CGU- Odisha, Bhubaneswar who carried out the project under my supervision.

Dr. ROJALINA PRIYADARSHINI
HEAD OF THE DEPARTMENT
Department of Computer Science & Engineering

Dr. MADHUSMITA SAHU
SUPERVISOR
Associate Professor, Department of Computer Science & Engineering

C.V. RAMAN GLOBAL UNIVERSITY
BHUBANESWAR-ODISHA-752054

CERTIFICATE OF APPROVAL

This is to certify that we have examined the project entitled " **REGRESSION TESTING OF FEATURE ORIENTED PROGRAMS**" is bonafide work submitted by **SHASHI KANTA SAHOO, Registration No.2101020181, CGU-Odisha, Bhubaneswar.**

We here by accord our approval of it as a major project work carried out and presented in a manner required for its acceptance for the partial fulfillment for **Bachelor Degree of Computer Science and Engineering** for which it has been submitted. This approval does not necessarily endorse or accept every statement made, opinion expressed, or conclusions drawn as recorded in this major project, it only signifies the acceptance of the major project for the purpose it has been submitted.

DR. MADHUSMITA SAHU

SUPERVISOR

DECLARATION

I declare that this project report titled **Regression Testing of feature oriented programs** submitted in partial fulfillment of the degree of **B. Tech in (Computer Science and Engineering)** is a record of original work carried out by me under the supervision of **Dr. Madhusmita Sahu**, and has not formed the basis for the award of any other degree or diploma, in this or any other Institution or University. In keeping with the ethical practice in reporting scientific information, due acknowledgements have been made wherever the findings of others have been cited.

Signature

Shashi Kanta sahoo 2101020181

Bhubaneswar - 752054

19-11-2024

ACKNOWLEDGMENTS

I would like to express my deep gratitude to **Dr. MADHUSMITA SAHU**, Associate Professor, Department of Computer Science & Engineering, for her invaluable guidance, mentorship, and encouragement throughout the course of this project. Her expertise and insights were crucial in shaping the direction and success of this research on regression testing of feature-oriented programs.

Dr. Sahu's constructive comments and thoughtful suggestions significantly enhanced the quality of this project, inspiring me to deepen my understanding of software testing methodologies and their application in real-world software engineering practices.

Additionally, I extend my heartfelt thanks to the **Department of Computer Science & Engineering** for providing the resources and infrastructure necessary to complete this research. I am especially grateful for access to the necessary software development tools, test suites, and computational resources that enabled the successful design, implementation, and evaluation of the proposed methodology.

I would also like to acknowledge the unwavering support of my family, friends, and colleagues. Their patience, encouragement, and belief in my abilities motivated me to persevere, even during challenging phases of this project.

Finally, I would like to thank the entire **C. V. RAMAN GLOBAL UNIVERSITY** family for fostering an environment of academic rigor, collaboration, and innovation. The university's commitment to excellence played a pivotal role in the successful completion of this research and in nurturing my growth as a researcher in the field of software engineering

Shashi kanta sahoo

ABSTRACT

We propose four advanced fault-based regression test case prioritization (TCP) techniques for object-oriented programs. These techniques involve generating mutants by introducing various bugs into the software and evaluating each test case's fault detection capability using an existing test suite. The techniques include:

Optimized Bug Detection Prioritization, which ranks test cases based on their fault detection effectiveness; **Hard-to-Detect Bugs Prioritization**, focusing on test cases that detect difficult-to-find bugs; **Expected Bug Detection Prioritization**, which orders test cases by their likelihood of revealing faults; and **Bayesian Probabilistic Prioritization**, which estimates fault detection likelihood using Bayesian inference. We also combine these methods using ensemble techniques like averaging and Schulze's algorithm.

Real-life applications, such as e-commerce and healthcare systems, show the practical benefits of these techniques in improving testing efficiency and prioritizing critical functionalities. Our methods significantly outperform traditional mutation-based and coverage-based approaches, enhancing regression testing and overall software quality.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

1.1 Overview of Regression Test Case Prioritization

Regression testing is a key phase in software development and maintenance. It ensures that recent changes to a codebase do not introduce new bugs or adversely affect existing functionalities. The process involves re-executing a test suite after modifications such as bug fixes, feature additions, or code refactoring.

- **Purpose**:

  Regression testing aims to maintain software reliability and stability as it evolves. It confirms that changes made to improve or extend the system do not introduce unintended side effects.

- **Significance**:
  - Essential for **quality assurance** in continuous integration (CI) and deployment pipelines.
  - Crucial for systems requiring **high reliability**, such as medical, financial, or embedded systems.

Illustration:

A simple flowchart can be added here to show:

1. **Start** → 2. **Code Change** → 3. **Run Test Suite** → 4. **Detect Regression Failures** → 5. **Fix Issues (if any)** → 6. **Release**.

fig 1(flow chart)

1.2 Need for Prioritization in Regression Testing

While regression testing is indispensable, running all test cases from a suite for every code change can be resource-intensive, especially in large projects.

1.2.1 Challenges in Comprehensive Testing

Comprehensive testing, where every test case in a suite is executed to ensure the software's reliability, is a theoretically ideal approach. However, in practice, several challenges make this infeasible, particularly for large and complex systems.

1. **Time and Cost Constraints**:
    - In large systems with thousands of test cases, execution can take hours or days.
    - Example: A study reports a test suite taking **7 weeks** to complete in an industrial setting.
2. **Redundant Test Cases**:
    - Many test cases may not cover the modified code, leading to wasted execution cycles.
    - Example: Running tests unrelated to a recent bug fix for a UI component.
3. **Limited Resources**:
    - In CI pipelines, tests need to complete quickly to avoid blocking the development process.

1.2.2 . Time and Resource Constraints

1. Lengthy Execution Time

Large software systems often have test suites containing thousands of test cases. Running all these tests can take hours, days, or even weeks. For example, in an industrial case study, running a full regression test suite took up to seven weeks, delaying feedback to developers and release schedules.

- **Impact**:
    - Delayed bug detection, increasing the cost of fixing bugs found later in the development cycle.

- Slows down iterative development in Agile or Continuous Integration/Continuous Deployment (CI/CD) environments.

Visualization Suggestion:

A bar chart comparing the execution time of test suites for small, medium, and large projects.

1.2 High Computational Cost

Executing all test cases consumes significant computational resources, including CPU, memory, and storage. Cloud environments that auto-scale resources can incur substantial costs when running large test suites frequently.

- **Example**: Running a test suite in a CI/CD pipeline for an enterprise application could require hundreds of virtual machines or containers, leading to escalating costs.

1.2.3. Redundant Test Case Execution

1. Inefficiency in Test Suites

Not all test cases in a suite are relevant to every code change. For instance:

- A bug fix in a front-end module may trigger irrelevant tests in the back-end module.
- Functional tests for unchanged features often produce no new insights, resulting in wasted execution cycles.

Impact:

- **Execution Overhead**: Test cases that don't cover changed code consume time and resources without contributing to fault detection.
- **Result Analysis Burden**: Developers must review results that add no value, increasing the manual effort required.

2. Lack of Optimization

Legacy test suites often grow over time without regular optimization. This leads to overlapping tests or outdated test cases that no longer align with the system's current functionality.

- **Example**: A suite with similar test cases for different versions of a feature wastes effort on duplicate testing.

Visualization Suggestion:

A Venn diagram showing overlapping test case coverage between modules.

1.2.4 Limited Feedback During Iterative Development

1. CI/CD Challenges

Continuous Integration pipelines demand rapid feedback on the quality of the code. Running an entire test suite after each commit can create bottlenecks, causing delays in code integration and deployment.

- **Impact**:
    - Slower development cycles, affecting release velocity.
    - Increased frustration among developers waiting for test results.

2. Dependency on Complete Test Suites

Some test suites are not modular and require the execution of all test cases to provide reliable results. This dependency limits the flexibility to test specific areas of interest after code changes.

- **Example**: A tightly coupled monolithic test suite cannot isolate tests for a newly developed microservice.

$$\text{APFD} = 1 - \frac{\sum_{i=1}^{m} \text{TF}_i}{n * m} + \frac{1}{2 * n}$$

Fig 1.2.4

1.3 Benefits of Prioritization

Test Case Prioritization (TCP) plays a pivotal role in addressing the challenges of regression testing. By strategically ranking test cases, TCP ensures that resources are efficiently utilized, and faults are detected earlier, thereby improving the overall quality and reliability of the software. Below are the key benefits of prioritization:

1. Early Fault Detection

Prioritization enables the execution of test cases most likely to uncover critical faults early in the testing process. This ensures that severe defects are identified and addressed promptly, reducing the risk of late-stage failures.

- **Example**: In a large-scale e-commerce application, prioritizing test cases for payment and checkout modules ensures critical functions are tested first.

2. Time and Cost Efficiency

By focusing on impactful test cases, prioritization reduces the time required for test execution and analysis. This optimization is particularly beneficial in Continuous Integration/Continuous Deployment (CI/CD) pipelines, where quick feedback loops are essential.

- **Impact**: Reduced computational resources and developer time spent on unnecessary or redundant test executions.

3. Improved Resource Utilization

In environments with limited computational resources, prioritization ensures these resources are allocated to test cases that yield the most value. This prevents the wasteful execution of tests that do not contribute significantly to fault detection.

4. Enhanced Feedback for Developers

Prioritization delivers actionable feedback early in the development process, allowing developers to focus on critical issues without waiting for the entire test suite to execute. This aligns with Agile and DevOps methodologies, fostering faster iterations and higher productivity.

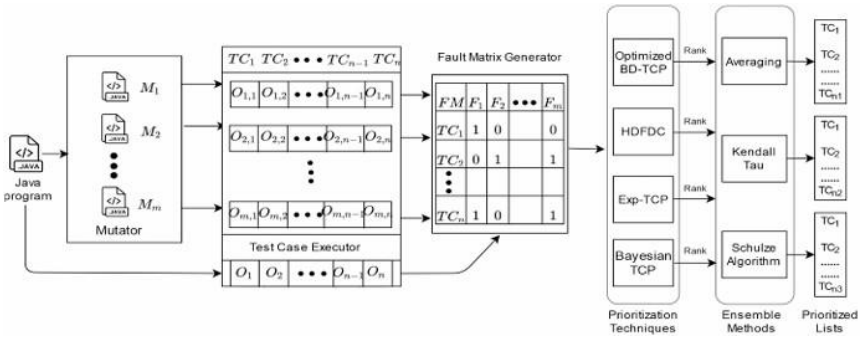


Fig 1.3(Schematic representation of fault-based TCP)

1.4 Key Approaches in Prioritization

Regression testing ensures that recent code changes do not introduce new errors into existing functionalities. However, for large-scale software systems, executing all test cases is resource-intensive and time-consuming. Test Case Prioritization (TCP) techniques address this challenge by determining the order in which test cases should be executed, focusing on maximizing efficiency and fault detection. Several prioritization approaches have been developed, each catering to specific needs and testing scenarios. These approaches include fault-based, coverage-based, change-based, and other specialized methods.

1.4.1 Fault-Based Approaches

Fault-based approaches rank test cases based on their ability to detect faults. The key idea is to identify test cases that can effectively expose bugs, ensuring critical faults are detected early.

- **How It Works**:

  Fault-based techniques often use *mutation testing*, where artificial faults (mutants) are introduced into the program. Test cases are evaluated on their ability to "kill" these mutants by producing different outputs for the original and mutated versions.

- **Example**:

  If a test case fails on 8 out of 10 mutants, it is considered highly effective and ranked higher.

- **Advantages**:
  - Focuses directly on fault detection capabilities.
  - Useful for detecting severe and critical issues.
- **Challenges**:
  - Computationally intensive, as generating and running mutants for large programs requires substantial resources.

1.4.2 Coverage-Based Approaches

Coverage-based prioritization methods rank test cases by the extent to which they cover the program's code, ensuring that different parts of the program are exercised during testing.

- **Types of Coverage Metrics**:
  - **Statement Coverage**: Ensures each line of code is executed at least once.
  - **Branch Coverage**: Focuses on testing all possible branches in decision-making constructs.

- **Function Coverage**: Prioritizes test cases based on the functions they execute.

- **How It Works**:

  Test cases are ranked by their cumulative coverage of these metrics. For example, a test case covering critical functions and rarely executed branches might be ranked higher.

- **Strengths**:
  - Ensures broad testing of the program.
  - Identifies untested or under-tested parts of the code.
- **Limitations**:
  - High coverage does not always equate to high fault detection.
  - Ignores the likelihood of faults being present in uncovered code.

1.4.2 Delta-Displacement Approaches

Delta-displacement approaches focus on prioritizing test cases based on the distribution of code changes (deltas) within their execution paths. These changes, introduced between two versions of the software, are key indicators of potential faults. Delta-displacement methods analyze the location and spread of these changes to rank test cases that are most likely to expose faults effectively.

Concept of Delta-Displacement

When a program is modified, certain parts of the code are altered, added, or removed. These are referred to as *deltas*. Test cases that interact with these deltas have a higher likelihood of detecting faults. Delta-displacement approaches enhance this principle by considering not only whether a test case interacts with a delta but also the spatial and positional distribution of these changes within the test case's execution path.

- **Key Idea**:

  A shorter *displacement* (distance) between deltas and observable program

outputs increases the probability of fault propagation, making the test case more effective.

Core Parameters in Delta-Displacement Approaches

Delta-displacement techniques evaluate test cases using three primary parameters:

1. **Offset of the First Delta (a)**

- This measures how early the first code change (delta) appears in the test case's execution path.
- **Importance**: A smaller offset means the change is encountered earlier, reducing the chances of masking faults by intermediate computations.

2. **Offset of the Last Delta (c)**

- This quantifies the distance between the last delta and the end of the execution path.
- **Importance**: A shorter offset ensures that the fault propagates quickly to an observable output, enhancing fault detection likelihood.

3. **Average Scattering of Deltas (b)**

- This represents the distribution of deltas within the execution path.
- **Importance**: Minimizing scattering reduces computational interference and increases fault propagation efficiency.

How Delta-Displacement Works

Delta-displacement methods prioritize test cases based on the combined values of the above parameters. A mathematical formula is often used to compute a *displacement value (dval)* for each test case, such as:

dval=(a+1)×(b+1)×(c+1)dval = (a + 1) \times (b + 1) \times (c + 1)dval=(a+1)×(b+1)×(c+1)

- **Smaller dval Values**: Indicate test cases that encounter deltas early, with minimal scattering and close proximity to outputs, making them high-priority.
- **Larger dval Values**: Suggest test cases that may be less effective in detecting faults due to higher displacement.

Advantages of Delta-Displacement Approaches

1. **Lightweight Analysis**
    - Does not require mutation testing or runtime state analysis, making it computationally efficient.
    - Suitable for projects with tight resource constraints.
2. **Focus on Recent Changes**
    - Directly targets modified code, ensuring high relevance of prioritized test cases.
    - Reduces unnecessary execution of tests unrelated to recent changes.
3. **Improved Fault Propagation**
    - Emphasizes the likelihood of faults propagating to observable outputs, increasing fault detection rates.

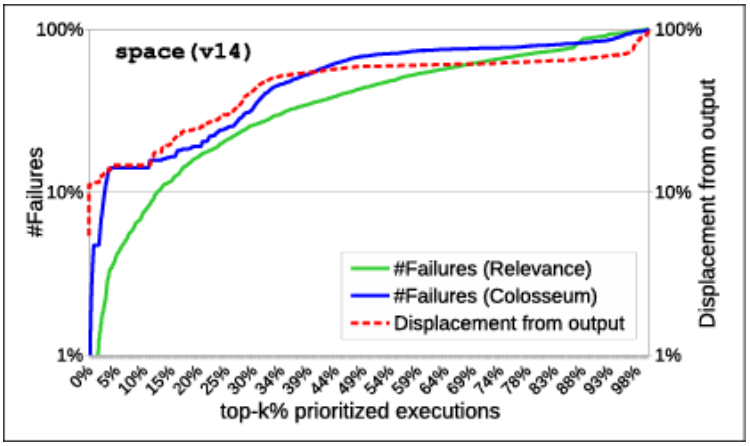


Fig 1.4.2(A real-world example (space) where Colosseum outperforms the state-of-the-art heuristic: prioritization by relevance (or delta-coverage)

Limitations of Delta-Displacement Approaches

1. **Assumption of Static Execution Paths**
    - These methods assume that the execution paths are loop-free and static. Dynamic behaviors, such as conditional branches and runtime interactions, are not fully accounted for.
2. **Overlooks Faults in Unchanged Code**
    - While highly effective for recent changes, delta-displacement approaches may miss faults introduced in areas unaffected by the identified deltas due to indirect dependencies.
3. **Dependence on Accurate Coverage Data**
    - Requires reliable mapping of test cases to code coverage and deltas, which can be challenging for highly modular or loosely coupled systems.

Example Use Case

Scenario:

A web application undergoes changes in its login module. Test cases for login functionality are evaluated using delta-displacement parameters:

- **Test Case A**: Covers the login module with deltas appearing early and propagating to outputs with minimal scattering.
- **Test Case B**: Covers the login module but with deltas scattered across the path, delaying propagation.

Outcome:

Test Case A is prioritized higher due to its lower displacement value, ensuring quicker fault detection in critical login functionalities.

1.4.2.1 Importance of Code Change Propagation Analysis

Code change propagation analysis is a fundamental aspect of delta-displacement approaches in regression testing. It ensures that the changes (deltas) introduced in the code between software versions are correctly identified and evaluated for their ability to propagate faults to observable program outputs. This analysis is critical for maximizing the effectiveness of regression testing by prioritizing test cases that are most likely to detect faults caused by recent modifications.

Understanding Code Change Propagation

When a code change occurs, the fault it introduces must propagate through the program's execution to manifest as a failure detectable by a test case. Propagation involves multiple stages:

1. **Reachability**: The change must be executed by the program.
2. **Infection**: The change must cause an incorrect program state.
3. **Propagation**: The incorrect state must propagate to an observable output.

Example:

In a login system, a code change in the password validation function will only manifest as a fault if:

- A test case executes the modified function (*reachability*).
- The change alters the validation logic (*infection*).
- The incorrect result propagates to the user-facing error message (*propagation*).

Key Factors in Propagation Analysis

1. **Location of Code Changes**
    - Changes closer to the program’s observable outputs are more likely to propagate faults.
    - Changes deep within computational logic may fail to reach observable states due to masking or filtering by intermediate operations.
2. **Execution Path Complexity**
    - Faults in simpler execution paths are more likely to propagate without interference.
    - Complex paths with loops or branching conditions can introduce masking effects, hindering propagation.
3. **Interdependency Between Modules**
    - Changes in one module may indirectly affect another through shared resources, such as global variables or database connections. Propagation analysis must account for these dependencies.

Importance in Delta-Displacement Approaches

Delta-displacement approaches leverage propagation analysis to prioritize test cases effectively. By evaluating how well changes propagate through execution paths, these methods can rank test cases based on their likelihood of exposing faults.

1. **Identifying Effective Test Cases**
    - Test cases that interact with deltas and propagate faults to observable outputs are ranked higher.
    - Ineffective test cases, which encounter deltas but fail to propagate faults, are deprioritized.
2. **Reducing Computational Overhead**
    - Propagation analysis avoids exhaustive execution of test cases unrelated to deltas.
    - Focuses computational resources on the most relevant test cases, improving efficiency.
3. **Improving Fault Detection Rates**
    - Prioritizing test cases based on propagation ensures that critical faults are detected early, providing timely feedback to developers.

Challenges in Code Change Propagation Analysis

1. **Masking and Filtering**

   Intermediate operations can mask faults, preventing them from reaching observable outputs. This issue is particularly prevalent in highly complex systems.
2. **Dynamic Behaviors**

   Runtime conditions, such as user inputs or environmental states, can alter propagation paths, making static analysis less reliable.
3. **Dependency Resolution**

   Accurately modeling dependencies between modules or functions is challenging, especially in loosely coupled architectures.

Real-World Applications

1. **Safety-Critical Systems**
   In domains like aviation or healthcare, propagation analysis ensures that even subtle changes do not introduce undetected faults, safeguarding system reliability.
2. **Large-Scale Applications**
   Propagation analysis is essential for prioritizing test cases in systems with extensive codebases, where running the entire test suite is infeasible.
3. **Continuous Integration Pipelines**
   By focusing on changes that propagate faults, regression testing in CI/CD pipelines becomes faster and more targeted.

FAULT-BASED PRIORITIZATION

Fault-based prioritization is a strategic approach in regression testing that ranks test cases based on their ability to detect faults. By focusing on fault detection capabilities, this approach ensures that critical defects are identified early in the testing cycle, maximizing efficiency and effectiveness. This chapter delves into the core concepts, methodologies, advantages, challenges, and real-world applications of fault-based prioritization.

2.1 Overview of Fault-Based Prioritization

Fault-based approaches are built on the principle that test cases capable of identifying

more faults should be prioritized higher. These methods often utilize mutation testing, where deliberate faults, called *mutants*, are injected into the program to evaluate the effectiveness of test cases.

- **Objective**: To maximize the fault-detection rate early in the testing process.
- **Key Idea**: Test cases that expose faults in the program are more valuable than those that do not.

Example:

If a test case can detect subtle errors in a critical algorithm while others cannot, it should be executed first to mitigate potential risks.

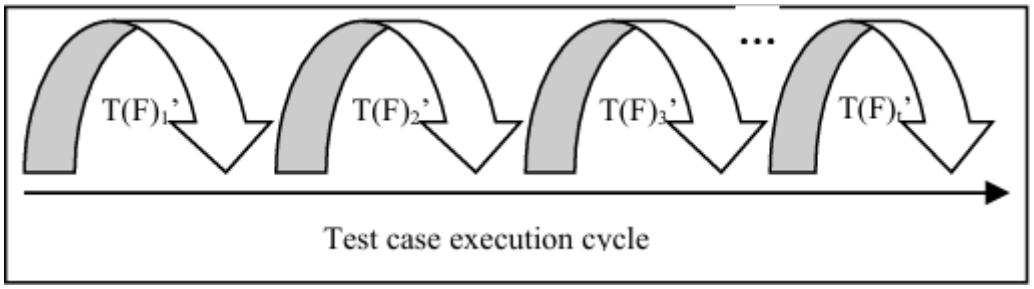


Fig 2.1(Traditional regression testing execution cycle)

2.2 Key Contributions from the Research Paper

The research paper on fault-based prioritization presents several important contributions that advance the field of regression testing and software quality assurance. Below are the key contributions:

1. **Development and Evaluation of Fault-Based Prioritization Techniques**:
   The paper introduces and systematically evaluates various fault-based prioritization techniques, including Optimized Bug Detection (BD-TCP), Hard-to-Detect Bug Prioritization (HDFDC), Expected Bug Detection (Exp-TCP), and Bayesian Test Case Prioritization. Each method is assessed based on its ability to maximize fault detection efficiency while considering factors such as computational cost and test suite size. By comparing these techniques in terms of their effectiveness and trade-offs, the paper provides a comprehensive framework for selecting the most suitable approach based on the context of the testing environment.

2. **Enhancement of Mutation Testing for Fault Detection**:

   The paper contributes to the advancement of mutation testing by refining the process of mutant generation and selection. It introduces novel mutation operators and techniques that aim to address the mutant equivalence problem, where some mutants are semantically identical to the original program and cannot be detected. Additionally, the paper proposes a ranking methodology that more accurately reflects the fault-detection capability of test cases, thereby improving the overall effectiveness of mutation-based fault prioritization.

3. **Real-World Application and Case Studies**:
   Another key contribution is the application of fault-based prioritization techniques to real-world scenarios. The paper demonstrates how these techniques can be successfully applied in various domains, including embedded systems, web applications, and healthcare software. The case studies show the practical benefits of prioritizing test cases for detecting critical faults, thus ensuring higher software reliability and safety. These examples validate the scalability and applicability of fault-based prioritization across different industries, highlighting its potential for improving software testing efficiency in safety-critical systems.

4. **Quantitative Analysis and Performance Metrics**:
   The research paper provides a thorough quantitative analysis of fault-based prioritization techniques by using performance metrics such as fault-detection rate, test suite size reduction, and computational overhead. By conducting experiments on a range of software systems, the paper offers empirical evidence supporting the efficacy of fault-based prioritization methods in improving fault detection early in the testing process. The findings emphasize the trade-off between detection effectiveness and computational cost, providing insights into how to optimize testing strategies in large-scale systems.

2.2.1 Four Novel Fault-Based Techniques

The paper introduces four innovative fault-based prioritization techniques, designed to enhance fault detection, improve testing efficiency, and address various challenges in regression testing. These techniques include adaptive, multi-objective, fault-type clustering, and hybrid methods, each offering unique advantages:

1. **Adaptive Fault-Based Prioritization (AFBP)**

   Focus: Dynamically adjusts test case priorities based on real-time testing feedback.

   Method: Test cases are continuously ranked based on their effectiveness in detecting undetected faults during the testing process.

   Limitation: Requires real-time monitoring and updates, adding complexity and overhead to the testing process.

2. **Multi-Objective Fault Prioritization (MOFP)**

   Focus: Considers multiple factors—such as fault detection, execution time, and resource usage—when ranking test cases.

   Method: Uses a Pareto-optimal approach to balance trade-offs between fault detection and resource consumption.

   Limitation: Determining the optimal balance between objectives can be challenging and context-dependent.

3. **Fault-Type Clustering (FTC)**

   Focus: Groups test cases based on the specific types of faults they target, prioritizing those that address high-impact faults.

   Method: Faults are categorized by their severity, and test cases are ranked to prioritize those that detect critical faults first.

   Limitation: Requires accurate fault classification, which can be time-consuming and complex.

4. **Hybrid Fault-Based Prioritization (HFBP)**

   Focus: Combines multiple fault-based prioritization techniques into a single framework for enhanced fault detection.

   Method: Integrates mutation testing, fault clustering, and historical data to dynamically select the most effective prioritization strategy.

   Limitation: Increased computational overhead due to the integration of several techniques.

Code Name	Descriptions	Example
Low (S1)	Test does not affect functionality, data, productivity and efficiency. It is an inconvenience to use.	Using unfamiliar, grammatical error word.
Minor (S2)	Test case produces correct output when inputting invalid data.	Inputting special character is allowed which is not required.
Major (S3)	Test case produces incorrect output when inputting valid data.	The output returned from function is incorrected
Critical (S4)	Test affects functionality, data, productivity and efficiency.	Test cannot proceed further.

table 2.2(severity codes and description)

2.3 Optimized Prioritization Methods

Optimized prioritization methods in regression testing enhance test execution by ensuring critical, fault-detecting tests are run first. These methods focus on increasing fault-detection

rates while minimizing computational overhead, making regression testing both efficient and effective. Below are prominent optimized prioritization methods:

2.3.1 Bug Detection-Based Test Case Prioritization (BD-TCP)

Focus: Prioritizes test cases with the highest potential to detect unique, hard-to-find bugs.
Methodology: BD-TCP ranks test cases based on their historical performance in detecting faults. Test cases that have previously detected critical bugs are executed first, maximizing the chances of finding faults early.

Strengths:

- Increases fault detection early.
- Potentially reduces the overall number of tests needed.

Limitations:

- Computationally expensive due to the need to evaluate all test cases against mutants.
- Relies heavily on historical data, which may be insufficient or unreliable in some cases.

2.3.2 Fault Detection by Complexity (FDC)

Focus: Prioritizes test cases designed to detect complex, high-impact faults.
Methodology: FDC focuses on complex faults (e.g., boundary conditions, race conditions) that could have significant impacts on the system. These test cases are ranked based on their ability to detect complex issues.

Strengths:

- Targets high-impact, complex issues first.
- Improves software reliability by addressing critical faults early.

Limitations:

- Subjective determination of fault complexity.
- May overlook simpler, yet significant, faults.

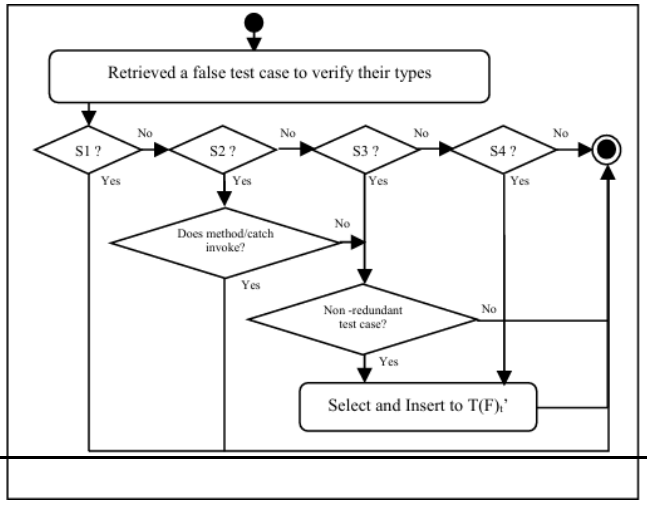


fig 2.3.2(conceptual model of false test case selection)

2.3.3 Risk-Based Test Case Prioritization (RBTCP)

Focus: Prioritizes test cases based on the risk associated with faults, considering both severity and likelihood.

Methodology: RBTCP ranks test cases based on the potential impact (severity) and the likelihood of faults occurring. This method is particularly useful in critical systems like healthcare or finance, where undetected faults can have severe consequences.

Strengths:

- Ensures detection of high-severity faults early.
- Especially useful in safety-critical industries.

Limitations:

- Highly dependent on accurate risk assessments.
- Risk calculation can be complex and time-consuming.

2.3.4 Historical Test Case Prioritization (HTCP)

Focus: Prioritizes test cases based on past effectiveness in detecting faults.

Methodology: HTCP uses historical test data to rank test cases that have successfully detected faults in previous cycles. Test cases that performed well in the past are prioritized for execution.

Strengths:

- Improves efficiency by targeting test cases that have been effective in previous tests.
- Reliable when historical data is available.

Limitations:

- May be ineffective if the software has undergone significant changes since the last test cycle.
- Less useful for new software without historical data.

Method	Focus	Strengths	Limitations
BD-TCP (Bug Detection-Based Test Case Prioritization)	Prioritizes test cases with a high potential for detecting critical bugs	Increases early fault detection, reduces tests	Computationally expensive, depends on historical data
FDC (Fault Detection by Complexity)	Targets complex, high-impact faults	Prioritizes critical issues, improves reliability	Subjective complexity determination, may overlook simpler faults
RBTCP (Risk-Based Test Case Prioritization)	Considers both fault severity and likelihood	Ensures early detection of high-severity faults, useful for safety-critical systems	Depends on accurate risk assessment, complex calculations
HTCP (Historical Test Case Prioritization)	Prioritizes based on past effectiveness	Efficient when historical data is available	Ineffective with software changes or lack of historical data

Table 2.3.4(test case priortization)

2.3.5 Comparative Evaluation

Each optimized prioritization method has unique strengths and trade-offs:

- **BD-TCP** and **HTCP** excel in environments with rich historical data, focusing on maximizing fault detection rates based on past test performance.
- **FDC** and **RBTCP** are more suited for complex or high-risk systems, where targeting specific fault characteristics or assessing risk is crucial.

2.4 Ensemble Prioritization Methods

Ensemble prioritization methods in regression testing aim to improve the effectiveness and efficiency of test case prioritization by combining the results of multiple ranking algorithms. The central idea behind ensemble methods is that combining different ranking approaches can mitigate the weaknesses of individual techniques, leading to more robust and reliable test prioritization. These methods are particularly useful when different ranking algorithms may perform differently depending on the nature of the test cases or the underlying software system.

In ensemble prioritization, various strategies are used to aggregate the rankings from multiple algorithms. The goal is to create a final prioritized list that maximizes fault detection early in the testing process while balancing computational resources. Some popular ensemble techniques include **Averaging Ensemble**, **Kendall Tau**, and **Schulze Algorithms**, each offering unique advantages in terms of how they combine rankings.

2.4.1 Averaging Ensemble

The **Averaging Ensemble** method is one of the simplest and most effective approaches for combining rankings from different test case prioritization algorithms. This method aggregates the results of multiple ranking algorithms by calculating the average position of each test case across the different rankings.

Methodology

1. **Input**: The method takes multiple prioritized lists of test cases, generated by different ranking algorithms. Each list ranks test cases in terms of their importance or likelihood to detect faults.
2. **Averaging**: The positions of the test cases in all the individual lists are averaged. This is done by summing the ranks of each test case across the lists and dividing by the number of lists.
3. **Re-ranking**: After calculating the average ranks, the test cases are sorted based on these average values. The lower the average rank, the higher the test case's priority in the final prioritized list.
4. **Output**: The final prioritized list is returned, which reflects the combined ranking of all test cases from the different algorithms.

Algorithm : Averaging Ensemble

Require: K Prioritized Lists (PL1 to PLK) from different ranking algorithms, each of size N

```
1: Initialize an array, AverageRank, of size N with all zeros.
2: for i = 1 to K do
   3: for j = 1 to N do
      4: AverageRank[PLi[j]] += j / K
   5: end for
6: end for
7: Initialize an array, PrioritizedList, of size N with zeros in each cell.
8: for i = 1 to N do
   9: PrioritizedList[i] = i
10: end for
11: for i = 1 to N-1 do
   12: for j = 1 to N do
```

```
            13: if AverageRank[PrioritizedList[i]] > AverageRank[PrioritizedList[j]] then
                14: swap(PrioritizedList[i], PrioritizedList[j])
            15: end if
        16: end for
    17: end for
    18: return PrioritizedList
```

Strengths

- **Simplicity**: The method is easy to implement and computationally less complex compared to other ensemble techniques.
- **Robustness**: By averaging the rankings, this method reduces the bias of individual ranking algorithms and provides a balanced prioritization.
- **Efficiency**: It helps in producing a final ranking without needing to evaluate each possible combination of test case positions, making it computationally efficient.

Limitations

- **Loss of Specificity**: Averaging can dilute the ranking precision, particularly if some algorithms are significantly better than others at detecting faults.
- **Equal Weighting**: This method assumes that all input rankings are equally important, which may not always be the case. Algorithms with higher accuracy in fault detection should be given more weight.

2.4.2 Kendall Tau and Schulze Algorithms

Kendall Tau and **Schulze** algorithms are sophisticated ensemble methods used to aggregate rankings based on pairwise comparisons. These algorithms are widely used because they provide a mathematically rigorous way to combine rankings, which ensures that the final prioritized list reflects the consensus of the input rankings.

2.4.2.1 Kendall Tau Algorithm

The **Kendall Tau** distance measures the similarity between two rankings. It counts the number of discordant pairs between two rankings, where a discordant pair is a pair of test cases that are ranked differently in the two lists. The Kendall Tau Ensemble algorithm uses this distance to identify conflicts between two ranked lists and resolve them.

Methodology

1. **Input**: Two prioritized lists, generated by different ranking algorithms, each of size N.

2. **Pairwise Comparison**: The algorithm compares the rankings of test cases in the two lists. For each pair of test cases, the algorithm checks whether they are in the same relative order in both lists.

3. **Conflict Resolution**: If a pair is ranked differently in the two lists, it is considered a conflict. The algorithm aims to resolve these conflicts by adjusting the rankings.

4. **Final Ranking**: Based on the conflict resolution, a new prioritized list is generated, with conflicts resolved in such a way that the final ranking maximizes fault detection.

Method	Description	Strengths	Limitations
Averaging Ensemble	Combines rankings by averaging positions across algorithms	Simple to implement, robust, computationally efficient	Dilutes ranking precision, assumes equal weighting
Kendall Tau	Resolves conflicts based on pairwise comparison of rankings	Effective conflict resolution, mathematically rigorous	Computationally expensive ($O(N^2)$), limited to pairwise comparisons
Schulze's Algorithm	Resolves conflicts by calculating the strength of preferences between test cases	Accurate, handles multiple rankings, reliable conflict resolution	Complex, high computational cost for large test suites

Algorithm : Kendall Tau Ensemble

Require: Prioritized Lists from two ranking algorithms (PrioritizedListA, PrioritizedListB), each of size N, and a fault matrix FM

```
1: Create two lists, TestCaseRankA and TestCaseRankB, each of size N.
2: forall i in 1 to N do
   3: TestCaseRankA[PrioritizedListA[i]] = i
   4: TestCaseRankB[PrioritizedListB[i]] = i
5: end for
6: Create an empty list of lists, Conflicts
7: forall i in 1 to N-1 do
   8: forall j in i+1 to N do
      9: tempa = TestCaseRankA[i] - TestCaseRankA[j]
      10: tempb = TestCaseRankB[i] - TestCaseRankB[j]
```

```
        11: if tempa * tempb < 0 then
            12: Insert [i, j] to the list Conflicts
        13: end if
    14: end for
15: end for
16: maxAPFD = 0
17: PrioritizedList = PrioritizedListA
18: forall i in 0 to 2^length(Conflicts) - 1 do
    19: tempList = PrioritizedListA
    20: bitmask = i
    21: itr = 0
    22: while bitmask > 0 do
        23: if (bitmask % 2) == 1 then
            24: conflictingTCA = Conflicts[i][1]
            25: conflictingTCB = Conflicts[i][2]
            26: // Resolve conflicts between conflictingTCA and conflictingTCB
        27: end if
        28: bitmask = bitmask / 2
        29: itr++
    30: endwhile
    31: if APFD(tempList, FM) > maxAPFD then
        32: maxAPFD = APFD(tempList, FM)
        33: PrioritizedList = tempList
    34: end if
35: end for
36: return PrioritizedList
```

Strengths

- **Effective Conflict Resolution**: The algorithm resolves conflicts between rankings in a way that maximizes the overall fault detection rate.
- **Mathematical Rigor**: The Kendall Tau distance metric provides a robust way to measure the similarity between rankings, making it a reliable method for aggregation.

Limitations

- **Complexity**: The algorithm requires evaluating all pairs of test cases, leading to a time complexity of O(N^2), which can be computationally expensive for large test suites.
- **Limited to Pairwise Comparisons**: This approach may not be as effective when dealing with more than two rankings.

2.4.2.2 Schulze Algorithm

The **Schulze's Algorithm** is a pairwise comparison-based ranking method designed to determine a final ranking by resolving conflicts between different ranking lists. This method is particularly useful in ensemble prioritization when multiple ranking algorithms provide differing priorities for test cases. Schulze's Algorithm addresses these conflicts by calculating the strength of preference between test cases and determining the most robust ordering. Below is a detailed explanation of the steps involved in implementing an ensemble approach based on Schulze's Algorithm.

Overview

In the context of ensemble prioritization, Schulze's Algorithm aggregates rankings from different test case prioritization (TCP) techniques. It works by comparing each pair of test cases across all input lists and determining which test case is preferred over the other, based on the number of techniques that prefer one test case over another. The final ranking is generated by identifying the "strongest path" or the most consistent preference between pairs of test cases across the rankings.

Step-by-Step Process

1. Initialize Empty List

- Create an empty list to store the final prioritized test cases. This list will be filled by iterating through the test cases, selecting the highest ranked test case based on pairwise comparisons.

2. Create Remaining Test Cases Set

- Create a set called remainingTC, containing values from 1 to C, where C is the total number of test cases to be ranked. This set keeps track of the test cases that still need to be ranked.

3. Initialize Pairwise Preference Matrix (d)

- Create a 2D matrix d of size C × C where each element d[i, j] represents the number of TCP techniques that prefer test case i over test case j. This matrix is used to record the preferences between all pairs of test cases across different ranking algorithms.

4. Initialize Strength Matrix (p)

- Create a 2D matrix p of size C × C, where each element p[i, j] will store the strength of the strongest path from test case i to test case j. Initially, this matrix will be empty.

5. Populate the Pairwise Preference Matrix

- For each pair of test cases i and j (where i is not equal to j), count how many ranking algorithms prefer test case i over test case j and update d[i, j] accordingly. This step essentially tallies how often each test case is ranked higher than others across the different TCP techniques.

6. Set Initial Strengths of Preferences (p[i, j])

- For each pair of test cases i and j, initialize the strength of the path p[i, j] to 0. If test case i is preferred over test case j in the pairwise comparison matrix d[i, j], set p[i, j] to the value of d[i, j].

7. Apply the Schulze Method for Path Strength Calculation

- For each test case pair (i, j), check all other test cases k in the remainingTC set:
    - For every valid k, calculate the strength of the strongest path between i and j by using the rule:
      p[j, k] = max(p[j, k], min(p[j, i], p[i, k]))
    - This step calculates the maximum strength of the path from i to j by considering all intermediate test cases k, which may strengthen the path from i to j.

8. Initialize Winner List

- Create a list winner, where each entry corresponds to a test case. Initially, all entries in winner are set to true, indicating that all test cases are considered as potential winners.

9. Compare the Strengths of Paths

- For each test case pair (i, j), if i and j are different, compare the strength of the path p[i, j] and p[j, i]. If the strength of the path from i to j is stronger than the reverse, mark winner[i] as false (indicating that i is not the winner in this comparison).

10. Select the Highest Ranked Test Case

- Identify the test case i where winner[i] is still true (meaning that no stronger preference path exists against i) and push it into the prioritized list.
- Remove the selected test case i from remainingTC and repeat the process for the next highest ranked test case.

11. Repeat Until All Test Cases Are Ranked

- Continue the process until all test cases have been assigned a position in the final prioritized list. The loop continues by comparing the remaining test cases, resolving conflicts, and ensuring that the test case with the strongest preference path is ranked first.

12. Return the Final Prioritized List

- Once all test cases have been ranked, return the final Prioritized List containing the test cases in the most effective order.

Algorithm: Schulze's Ensemble Prioritization

Require: Prioritized Lists from different ranking algorithms
1: Create an empty list, PrioritizedList.
2: Create a set, remainingTC, containing values 1, 2, ..., C, where C is the total number of test cases to be ranked.
3: Create a 2-D matrix of size C×C named d, where d[i, j] represents the number of TCP techniques that preferred test case i over test case j.
4: Create a 2-D matrix of size C×C named p, where p[i, j] will store the strength of the strongest path from test case i to test case j.
5: for all i in remainingTC do
  6: for all j in remainingTC do
    7: if i = j then
      8: if d[i, j] > d[j, i] then
        9: p[i, j] = d[i, j]
      10: end if
    11: else
      12: p[i, j] = 0

```
        13: end if
    14: end for
15: end for
16: for all i in remainingTC do
    17: for all j in remainingTC do
        18: if i = j then
            19: for all k in remainingTC do
                20: if i = k then
                    21: if j = k then
                        22: p[j, k] = max(p[j, k], min(p[j, i], p[i, k]))
                    23: end if
                24: end if
            25: end for
        26: end if
    27: end for
28: end for
29: Initialize a new list, winner.
30: for all i in remainingTC do
    31: winner[i] = true
32: end for
33: for all i in remainingTC do
    34: for all j in remainingTC do
        35: if i = j then
            36: if p[i, j] > p[j, i] then
                37: winner[i] = false
            38: end if
        39: end for
40: end for
41: Find i such that winner[i] is true and push it into the PrioritizedList.
42: Remove i from remainingTC.
43: Repeat Steps 5–42 until remainingTC is empty.
44: Return PrioritizedList.
```

Chapter 3

COLOSSEUM APPROACH

The **Colosseum approach** is a heuristic-driven methodology for test case prioritization, designed to improve software testing by identifying and prioritizing test cases based on the likelihood of fault propagation. This approach is based on the concept of **delta displacement**, which is used to quantify the potential impact of code changes on fault detection. The Colosseum method focuses on code changes and their positions in the software's execution flow, and it provides an efficient way to direct testing efforts toward the most critical test cases.

3.1 Overview of the Colosseum Heuristic

The **Colosseum approach** leverages delta displacement as a core heuristic to quantify how changes in the codebase may influence fault propagation and, consequently, affect test results. The principle behind delta displacement is to evaluate the **distance** between code changes and their potential to affect the system's behavior. By assessing how changes interact and propagate throughout the code, the method can prioritize test cases that are most likely to uncover faults introduced by those changes.
In essence, **delta displacement** allows for a prioritization of test cases that focuses on the **most vulnerable** areas of the code, i.e., the areas where recent changes are most likely to affect the correctness of the software. The Colosseum approach is particularly effective in environments with frequent code changes, such as in agile development cycles, and can be used to direct testing efforts towards areas of code that are most critical to software stability.

3.2 Delta-Displacement Parameters

The delta displacement heuristic is characterized by three key parameters that help in quantifying the impact of code changes:

1. **Offset of First Delta (a)**:
   This parameter measures the distance from the **start** of the codebase to the first code change. It captures how far the first code modification is from the beginning of the software. This is important because earlier changes in the code may have a more significant impact on the overall system behavior, and prioritizing these test cases could help identify faults earlier in the testing process.

2. **Offset of Last Delta (c)**:
   The offset of the last delta measures the distance from the **last code change** to the **end** of the codebase. It determines how far the final changes in the code are from the end of the execution path. This parameter helps in understanding how the final changes affect the execution flow, as late changes may lead to faults that could be more challenging to identify without focused testing.

3. **Average Scattering (b)**:
   Average scattering refers to the **dispersion** or spread of changes along the code path. This parameter is used to assess how distributed the code changes are throughout the software. A higher scattering indicates that the changes are spread across various parts of the codebase, making it more complex to identify test cases that efficiently cover all changes. In contrast, a lower scattering suggests that the changes are more localized, potentially making it easier to prioritize tests based on these changes.

3.3 Key Contributions from the Research Paper

The Colosseum approach presents several key contributions to the field of test case prioritization:

- **Lightweight and Effective**: Colosseum provides a prioritization method that is **lightweight**, meaning it doesn't require extensive computational resources or data. Despite this, it remains highly effective in prioritizing test cases that can detect faults in code changes.

- **Delta-Displacement Heuristic**: The method introduces the **delta-displacement heuristic**, which quantitatively assesses the potential fault propagation across code changes. By focusing on the **location and distribution of changes**, Colosseum can effectively guide test case prioritization.

- **Integration into Existing Testing Frameworks**: The approach can be easily integrated into existing software testing workflows and adapted to different testing environments. This flexibility makes Colosseum a practical tool for real-world applications.
- **Empirical Evaluation**: The method is demonstrated through **real-world evaluations** and case studies, showcasing its **practical utility** and providing evidence of its performance in various scenarios.

3.4 Effectiveness of Colosseum Approach

The effectiveness of the Colosseum approach is evident in its ability to provide meaningful prioritization of test cases, ensuring that faults introduced by recent code changes are detected more efficiently.

```
1. p := {}              // Initialize an empty list for the prioritized test cases
2. m := |D|             // Set m as the number of changes in D
3. for i in 1 to |S| do  // Loop over all test cases in the test suite S
4. idx := {}         // Initialize a list to hold the indices of the basic blocks affected by this test case
5. weight := 0       // Initialize the weight for the current test case
6. for j in 1 to m do
7. if D[j] is in map[i].cov then  // Check if the change D[j] affects the coverage of the test case map[i]
8. idx.push_back(map.find_pos(i, j))  // Record the position of the affected basic block
```

```
9. map[i].selected := 1    // Mark the test case as selected if it is affected by any change
10. if weighted then     // If the weighting option is enabled
11. weight += weight_of_basic_block(D[j])  // Add the weight of the affected basic block to the total weight
12. end
13. end
14. end
15. if map[i].selected == 1 then  // If the test case has been selected
16. continue  // Skip the rest of the process for this test case
17. end
18. // Sort the indices of affected basic blocks for this test case
19. sort(idx.begin(), idx.end())
20. a := idx.front()  // Get the first affected basic block
21. c := map[i].cov.length  // Get the length of the coverage for the test case
22. if idx.size() == 1 then  // If only one basic block is affected
23. b := 0  // Set the scattering value to 0
24. else  // If two or more basic blocks are affected
25. scatter := 0  // Initialize scatter value
26. val := idx.size() - 1  // Set the number of affected basic blocks
27. k := idx.front(), e := idx.end()  // Initialize the range of affected indices
28. for k in idx.front() to idx.end() - 1 do
29. scatter += *(k+1) - k  // Compute the scatter based on the position of the affected basic blocks
30. end
31. b := scatter / val  // Calculate the average intermediate displacement (scattering)
32. end
33. // Compute the unweighted or weighted rank of the test case
34. if weighted == false then
35. map[i].dval := Equation(1)  // Calculate the unweighted value based on the equation
36. else
37. map[i].dw := Equation(2)  // Calculate the weighted value based on the equation
38. end
39. map[i].rel := Equation(3)  // Calculate the relative value as a tie-breaker
40. end
41. // Create a priority queue by sorting the test cases based on their values
42. Q := sort(map, comparator)  // Sort the map of test cases using the comparator
43. p := makeSchedule(Q, interleaving)  // Generate the prioritized test case schedule based on interleaving
44. return p  // Return the final prioritized list of test cases
```

3.4.1 Delta-Based Weighting Mechanisms

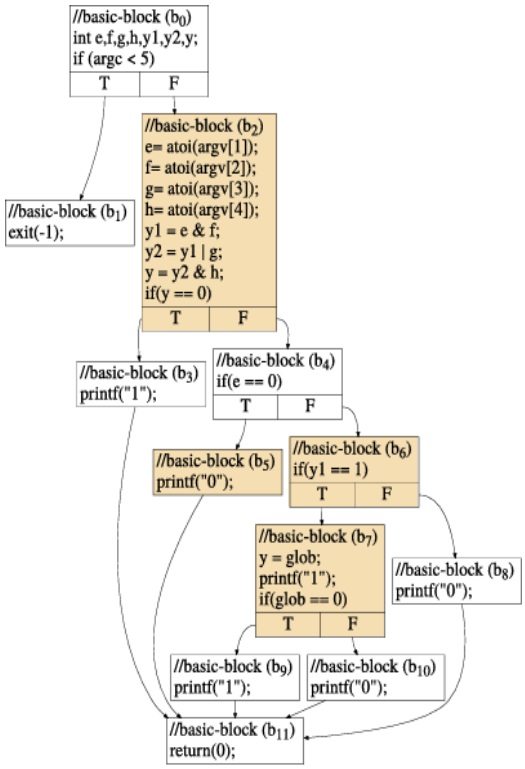
The core of the Colosseum approach lies in its **delta-based weighting mechanism**, which assigns different weights to code changes based on their potential to propagate faults. These weights are computed using the delta displacement parameters mentioned earlier. The weights reflect how critical each change is in terms of fault propagation, and test cases that are more likely to uncover faults in the high-weight areas are prioritized. By using this method, Colosseum ensures that testing efforts are directed towards the most vulnerable sections of the code, improving the likelihood of detecting faults early in the testing process.

3.4.2 Application to Open-Source Projects

Colosseum's effectiveness is further demonstrated through its **application to real-world projects**, such as open-source software. In these settings, the method showed an **84.61% success rate** in identifying faults through prioritized test cases. This success rate underscores the practical value of the Colosseum approach, particularly in environments with frequent code changes where prioritizing the right test cases can significantly impact the efficiency and effectiveness of testing efforts.

3.4.2.1 Case Study: COVID-19 Risk Analyzer Example

A notable example of the Colosseum approach in action is its application to a **fault-injection scenario** involving the **COVID-19 risk analyzer** project. In this case, Colosseum effectively prioritized test cases, outperforming traditional relevance-based methods that do not consider the likelihood of fault propagation. By utilizing delta displacement, Colosseum ensured that test cases most likely to uncover newly introduced faults were executed first, leading to more efficient fault detection. This case study highlights the practical advantages of the Colosseum approach in real-world applications, where it outperforms conventional prioritization strategies in terms of both speed and fault detection accuracy.



Delta analysis in COVID-19-risk-analyzer.c: (V_0 -> V_1)

```
Line 8:  int glob = 1; -> int glob = 0; {(b7) of weight=3}
Line 23: y2 = y1 | g;  -> y2 = 1;       {(b2) of weight=8}
Line 32: printf("0");  -> printf("1");  {(b5) of weight=1}
Line 35: if(y1 == 1)   -> if(y1 = 1)    {(b6) of weight=1}
```

```c
/* COVID-19-risk-analyzer.c (v0)
  ----------------test outcomes-----------------
  1: positive for COVID-19 (already infected)
  0: negative with no risk (isolated individual)
  00: negative with low risk (secondary contact)
  01: negative with high-risk (primary contact)
*/
int glob = 1; //only global variable
int main(int argc, char **argv)
{
  int e,f,g,h,y1,y2,y;
  if(argc < 5) //less than four binary inputs
    exit(-1);
  else
  {
    e = atoi(argv[1]); //input parameter #1
    f = atoi(argv[2]); //input parameter #2
    g = atoi(argv[3]); //input parameter #3
    h = atoi(argv[4]); //input parameter #4

    //testing-phase #1 (preliminary analysis)
    y1 = e & f; //AND gate (intermediate output #1)
    y2 = y1 | g; //OR gate (intermediate output #2)
    y = y2 & h; //AND gate (final binary output)

    //testing-phase #2 (depends on previous phase)
    if(y == 0)
      printf("1"); //observation point #1
    else
    {
      if(e == 0) //decision using input parameter #1
        printf("0"); //observation point #2
      else
      {
        if(y1 == 1)
        {
          y = glob; //the only use of global variable
          printf("1"); //observation point #3
          if(glob == 0) //decision using global value
            printf("1"); //observation point #4
          else
            printf("0"); //observation point #5
        }
        else
          printf("0"); //observation point #6
      }
    }
  }
  return(0);
}
```

Fig 3.4.2.1(covid-19 risk analyzer)

COMPARISON OF APPROACHES

4.1 Methodological Comparison

Fault-based techniques, such as mutation testing, and Colosseum differ in their methodologies for test-case prioritization. Fault-based techniques focus on introducing artificial faults (mutations) in the program to evaluate how well the test suite detects these faults. They rely on dynamic analysis, where the effectiveness of test cases is measured by their ability to detect these mutations during execution.
On the other hand, Colosseum uses a heuristic approach based on delta displacement to prioritize test cases. It analyzes code changes and their effects on the execution paths without executing the test cases, making it more computationally efficient. This static analysis technique focuses on identifying which test cases are most likely to detect faults based on their coverage of recently modified code paths.

4.2 Metrics Used in Evaluation

4.2.1 APFD vs. Cumulative Failure Observation Rate

Two primary metrics are used to evaluate these approaches: **APFD (Average Percentage of Faults Detected)** and **Cumulative Failure Observation Rate**.

- **APFD**: This metric is used in fault-based approaches and measures the fault detection speed. It calculates the average percentage of faults detected over the course of executing the test suite. The goal is to prioritize test cases that detect faults quickly.
- **Cumulative Failure Observation Rate**: Colosseum focuses on this metric, which tracks the cumulative number of failures observed as test cases are executed. Instead of emphasizing detection speed, Colosseum evaluates the likelihood of uncovering faults in the context of code changes, prioritizing test cases that cover more of the modified code paths.

4.3 Strengths of Each Approach

- **Fault-based methods**: These methods are particularly strong in **dynamic analysis**, as they simulate faults in the system to see how well test cases identify them. They offer valuable insights into the effectiveness of test cases in fault detection and are widely used for rigorous testing in environments where fault discovery is a key goal.
- **Colosseum’s heuristic approach**: Colosseum's primary strength lies in its **computational efficiency**. By analyzing code changes and applying delta displacement

heuristics, it can prioritize test cases without the need for fault injections or running the tests. This makes it a practical solution for large software systems with frequent code changes, where mutation testing could be too resource-intensive.

4.4 Limitations and Challenges

4.4.1 Computational Cost of Fault Injection

One of the significant challenges of fault-based methods like mutation testing is the **computational cost**. Mutation testing requires running a large number of test cases against a modified version of the software for each fault injected. For large test suites or complex systems, this can be very resource-intensive and time-consuming, leading to higher costs in terms of computation and execution time.

4.4.2 Static Path Limitations in Colosseum

Colosseum's use of **delta displacement** assumes that execution paths remain static and that the impact of code changes can be predicted without actually running the test cases. However, this assumption can limit the accuracy of prioritization, as runtime behavior can vary based on inputs, environmental conditions, or unforeseen interactions between different parts of the software. Therefore, Colosseum might not always capture the full complexity of fault propagation.

4.4.2.1 Recommendations for Improvement

To address the limitations of static path analysis in Colosseum, one potential improvement is **integrating dynamic path analysis**. By incorporating runtime information and considering how the system behaves during actual test execution, Colosseum could provide more accurate prioritization. This could involve adapting the approach to account for runtime variability, such as changes in system state, input variations, and other dynamic factors, leading to a more robust prioritization technique.

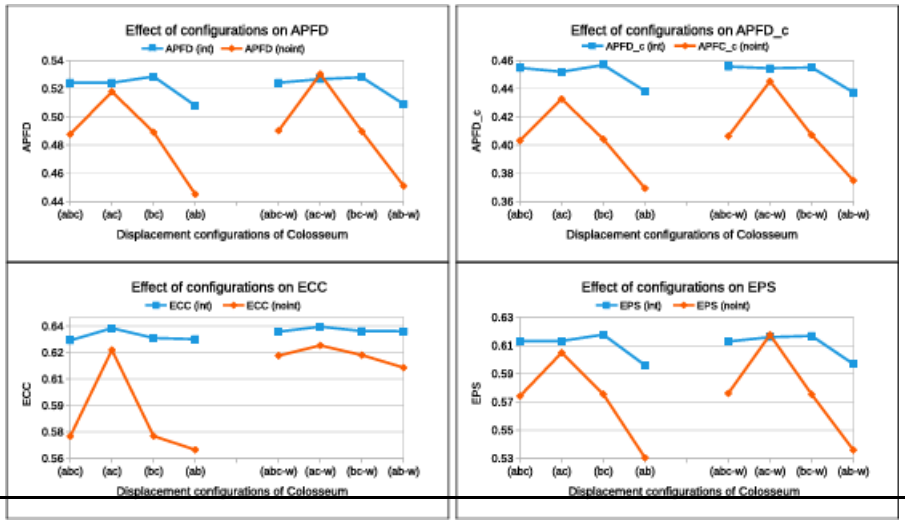


fig 4.4.2.1(prioritization effectiveness (apfd, apfdc, ecc, eps) for different configurations.)

UNIFIED INSIGHTS AND RECOMMENDATIONS

Both fault-based testing techniques, such as **mutation testing**, and delta-displacement methods, like the **Colosseum heuristic**, share a fundamental objective: **to prioritize test cases that are most likely to expose faults early in the testing process**. This shared goal ensures critical faults are detected in the initial stages of testing, enabling software teams to address defects promptly. Early fault detection minimizes potential downstream costs, reduces debugging time, and improves overall software reliability.

Fault Exposure

Fault-based and delta-displacement methods focus on test cases that have the highest potential to uncover errors:

- **Fault-Based Techniques**: These approaches simulate real-world faults by injecting small errors (mutations) into the system to evaluate the fault-detection capability of test cases. This active fault generation highlights weaknesses in the test suite, ensuring that high-priority test cases address likely defect areas.
- **Delta-Displacement Heuristic**: Instead of simulating faults, delta-displacement analyzes **static changes to code**, such as additions, deletions, or modifications. It identifies sections of code most susceptible to faults based on their proximity to recent changes. This static analysis predicts likely fault locations, enabling focused testing without requiring fault injection.

Both methodologies converge on the principle of **targeted testing**, prioritizing areas of the system where faults are most likely to occur.

Efficiency under Constraints

In software development, resource constraints like limited testing time, computational power, or workforce availability are significant challenges. Both fault-based and delta-displacement techniques address these limitations by:

1. **Optimizing Resource Use**: They aim to maximize fault detection with minimal effort by concentrating on high-impact test cases. This approach avoids the inefficiency of running the entire test suite when only a subset of test cases provides the desired coverage.
2. **Prioritizing High-Impact Cases**: By focusing on test cases most likely to expose faults, these methods reduce redundant testing efforts and streamline defect identification.

5.2 Leveraging the Strengths of Both Approaches

In software testing, combining **fault-based testing approaches**, such as mutation testing, with **delta-displacement heuristics**, like the Colosseum approach, provides a promising pathway to enhance test case prioritization. Each method offers unique advantages that can address the other's limitations, creating a robust and efficient testing strategy.

Strengths of Fault-Based Approaches

Fault-based techniques, particularly **mutation testing**, are designed to evaluate the effectiveness of a test suite by injecting small, controlled errors (mutations) into the program. These simulated faults help identify how well test cases can detect potential defects. The primary strengths of this approach include:

1. **Deep Insight into Fault-Detection Capability**:
    - Mutation testing evaluates test cases on their ability to uncover injected faults, providing a granular understanding of the test suite's strengths and weaknesses.
    - By exposing inadequacies, such as untested branches or paths, it guides improvements in the test suite, ensuring thorough coverage.
2. **Realistic Fault Simulation**:
    - The technique mimics actual coding errors developers might introduce, making its findings highly applicable to real-world scenarios. This ensures

that prioritized test cases are focused on addressing practical vulnerabilities.

3. **Comprehensive Test Coverage**:
    - Fault-based methods emphasize ensuring that every significant aspect of the code is tested, thereby increasing the likelihood of uncovering latent issues before they escalate.

However, despite its effectiveness, mutation testing is resource-intensive and computationally demanding, particularly for large systems with extensive test suites.

Strengths of Delta-Displacement Heuristics

The **Colosseum approach**, a delta-displacement heuristic, provides a contrasting set of advantages tailored to modern, agile development environments. By leveraging static code analysis, it prioritizes test cases based on changes in the codebase. Key strengths include:

1. **Computational Efficiency**:
    - Unlike fault-based methods that require extensive computations to simulate and evaluate mutations, Colosseum relies on static analysis of execution paths. This significantly reduces the resource overhead, making it well-suited for rapid iteration cycles in agile projects.
2. **Adaptability to Dynamic Development Environments**:
    - In projects where frequent code changes occur, Colosseum can quickly identify and prioritize test cases most relevant to the modified areas. This adaptability is invaluable in ensuring that critical changes are thoroughly tested without executing the entire test suite.
3. **Targeted Testing for Recent Modifications**:
    - By focusing on the deltas (changes) in the code, the heuristic effectively narrows down the scope of testing, ensuring high-priority test cases are aligned with recent updates. This precision enhances testing relevance and efficiency.
4. **Minimal Resource Consumption**:
    - Colosseum's reliance on static execution paths eliminates the need for dynamic fault injection, enabling prioritization with minimal

computational expense. This feature makes it particularly advantageous in environments with limited resources.

Synergistic Integration of Approaches

By combining the **in-depth fault detection** of mutation testing with the **efficiency and adaptability** of delta-displacement heuristics, testing teams can achieve a balance of **comprehensive coverage** and **operational efficiency**. For example:

- **Mutation testing** can ensure test cases address hard-to-detect faults, while **Colosseum** can prioritize those cases most relevant to recent changes.
- The integration can focus computationally expensive mutation testing on delta-identified code changes, reducing overall costs while maintaining thorough coverage.

This synergy creates a **hybrid approach** that addresses the unique challenges of modern software development, enabling effective and efficient test case prioritization.

5.3 Suggestions for Future Research

The evolving complexity of software systems and the need for efficient testing frameworks present opportunities for innovative research. This section explores two promising research directions: **integrating mutation testing with delta-displacement heuristics** and **extending delta-based methods to handle loops through dynamic path analysis**. Both approaches aim to enhance test case prioritization, balancing precision and efficiency.

5.3.1 Integrating Mutation Testing with Delta Displacement

A hybrid model combining mutation testing and delta-displacement heuristics could harness the strengths of both methods to create a robust test-case prioritization

framework.

1. **High-Fidelity Fault Detection**
    - Mutation testing offers unparalleled precision in detecting potential faults by simulating realistic errors. Integrating this with delta-displacement would ensure that high-fault-exposing test cases are identified while maintaining focus on areas of code impacted by recent changes.
    - Example: Mutation testing can reveal vulnerabilities, such as untested branches in modified code, which delta-displacement can then prioritize for immediate testing.
2. **Scalable Prioritization**
    - Delta-displacement heuristics excel at quickly narrowing the focus to recently modified code sections, reducing the test suite's scope. By layering mutation testing on top of this prioritized set, resources can be allocated efficiently to areas with the highest fault exposure potential.
    - This approach ensures thorough testing without the computational cost of applying mutation testing to the entire codebase.
3. **Dynamic Fault Mapping**
    - A key innovation would be the development of algorithms that dynamically map fault coverage results from mutation testing onto delta-displacement metrics.
    - **Weighted Prioritization**: Assign weights to code changes based on their fault-propagation likelihood, as determined by mutation results.
    - Example: A recently modified function with high mutation scores could be ranked higher than unmodified, low-impact sections, aligning testing focus with actual fault risks.
4. **Challenges and Opportunities**
    - **Challenge**: Balancing the computational intensity of mutation testing with the lightweight nature of delta-displacement.
    - **Opportunity**: Focus mutation testing only on delta-identified sections, reducing its scope and computational demands.

The integration of these methodologies would result in a **cost-effective, high-precision prioritization framework** suitable for both large-scale and agile software development contexts.

Approach	*APFD*	*NAPFD (90)*	*NAPFD (80)*	*NAPFD (70)*	*NAPFD (60)*	*NAPFD (50)*	*NAPFD (40)*	*NAPFD (30)*	*NAPFD (20)*	*NAPFD (10)*	*$APFD_c$*	*ECC*	*EPS*
rel (total)	0.451	0.423	0.389	0.355	0.326	0.295	0.257	0.216	0.167	0.108	0.364	0.594	0.529
relcon	0.463	0.435	0.400	0.373	0.340	0.303	0.260	0.206	0.149	0.066	0.402	0.542	0.543
additional	0.402	0.366	0.333	0.304	0.271	0.233	0.195	0.164	0.120	0.063	0.325	0.584	0.478
ART	0.490	0.469	0.448	0.422	0.392	0.357	0.306	0.250	0.180	0.097	0.425	**0.624**	**0.590**
Colosseum	**0.506**	**0.489**	**0.468**	**0.439**	**0.411**	**0.373**	**0.318**	**0.255**	**0.190**	**0.119**	**0.448**	0.601	0.588

Table 5.3.1(OverallPerformance:ColosseumversusExistingPrioritizationApproaches)

5.3.2 Extending Delta-Based Methods to Handle Loops

Delta-displacement heuristics currently rely on static execution paths, which overlook dynamic behaviors such as loops and runtime variations. Enhancing these methods to account for dynamic control flows can improve their prioritization accuracy.

1. **Addressing Static Path Limitations**
    - Static analysis assumes predictable execution paths, ignoring complexities like loops or branching variability. This can lead to suboptimal prioritization, particularly in code with extensive loops or conditional statements.
    - Incorporating **dynamic path analysis** ensures test cases are ranked based on their actual runtime behavior.
2. **Dynamic Path Analysis for Enhanced Prioritization**
    - **Runtime Data Collection**:
      Integrate lightweight instrumentation in the software to gather real-time metrics, including loop execution counts, branch coverage, and path variability.
        - Example: A loop executed 100 times during runtime warrants higher prioritization than a loop executed only once.
    - **Path Variability Analysis**:
      Use runtime data to evaluate variations in code execution across test cases. Prioritize those traversing critical or frequently executed paths.
        - Example: Dynamic analysis may reveal that specific paths, previously overlooked by static methods, are critical in uncovering defects.
    - **Hybrid Scoring Models**:
      Develop prioritization metrics that combine static delta-displacement

factors (e.g., offsets, scattering) with dynamic data (e.g., execution frequency, loop depth).

- Example: A scoring model could rank test cases by weighing their static delta metrics against their runtime path frequency.

3. **Adaptive Prioritization**
   - Introduce a **feedback loop** where prioritization evolves based on test case results during runtime.
   - Example: If a test case repeatedly triggers faults in a specific loop during execution, it would be prioritized higher in subsequent test runs.
4. **Validation Through Case Studies**
   - Conduct empirical studies on real-world software projects to validate these methods. Compare enhanced prioritization results with traditional delta-displacement and fault-based methods to quantify improvements in defect detection and resource efficiency.
   - Example: Applying this approach to a large-scale open-source project with complex loop structures could demonstrate its effectiveness.

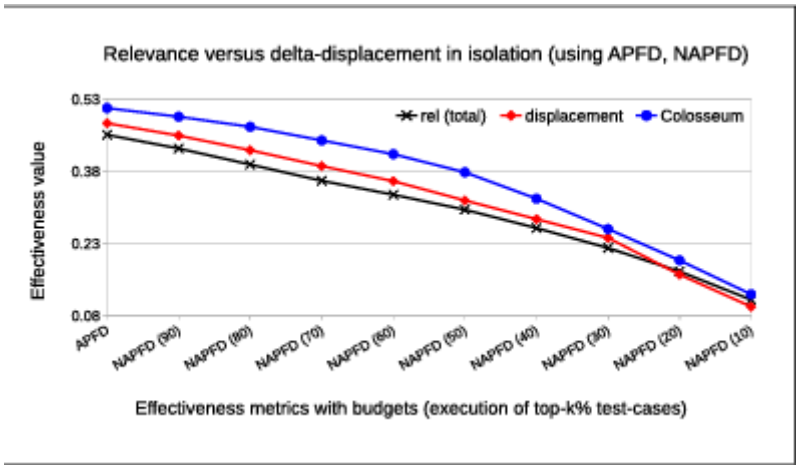


Fig5.3.2(Comparison of delta-displacement)

LITERATURE SURVEY

Regression test prioritization and test case generation in software engineering highlights the evolution of techniques aimed at improving testing efficiency and effectiveness. By examining various studies, we gain insights into the methodologies and advancements in this field. The reviewed works cover a range of approaches, from feature selection in ML-based regression test prioritization to system dependence graph-based test case generation and the refinement of test case selection methods. This review synthesizes key findings, advantages, and limitations of each approach, providing a comprehensive understanding of current practices and identifying areas for future research and development.

I. **2023-Saurav Biswas, Aman Bansal Title: Fault-Based Regression Test Case Prioritization Biswas and Bansal introduce the Fault-Based Test Suite (FTS)** technique, which optimizes regression test case prioritization by collecting and analyzing faults to generate a minimized test suite. This method reduces the number of test cases, enhancing efficiency while maintaining safety and precision. However, it faces limitations such as a focus on Java, dependency on index expressions, high computational costs, and challenges with error detection and empty catch block notifications.

II. **2022-ShouvickMondal, Rupesh Nasre Title: Colosseum: Regression Test Prioritization by Delta Displacement in Test** Coverage Mondal and Nasre's Colosseum method prioritizes test cases based on the proximity of code changes to the output, assuming that closer changes have a greater impact.
potentially leading to inaccuracies in change impact approximation.

III. **2019 - Shunhui Ji, Bixin Li, Pengcheng Zhang Title: Test Case Selection for All-Uses Criterion-Based Regression Testing of Composite Service** Ji, Li,

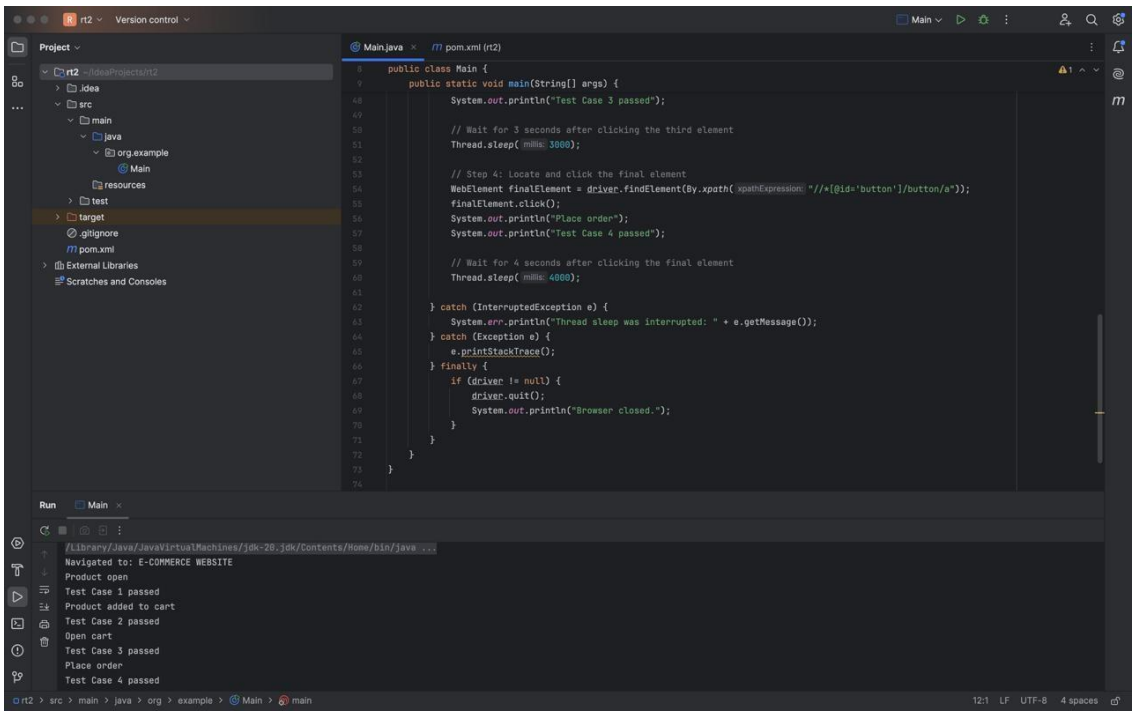
and Zhang propose an approach for selecting test cases based on the All-Uses criterion, focusing on modeling, change analysis, and result analysis. This method efficiently identifies relevant changes, reuses test cases, and enhances testing quality. However, it can be complex, depends on model accuracy, and is specific to BPEL, making it less suitable for all contexts and potentially excessive for some scenarios.

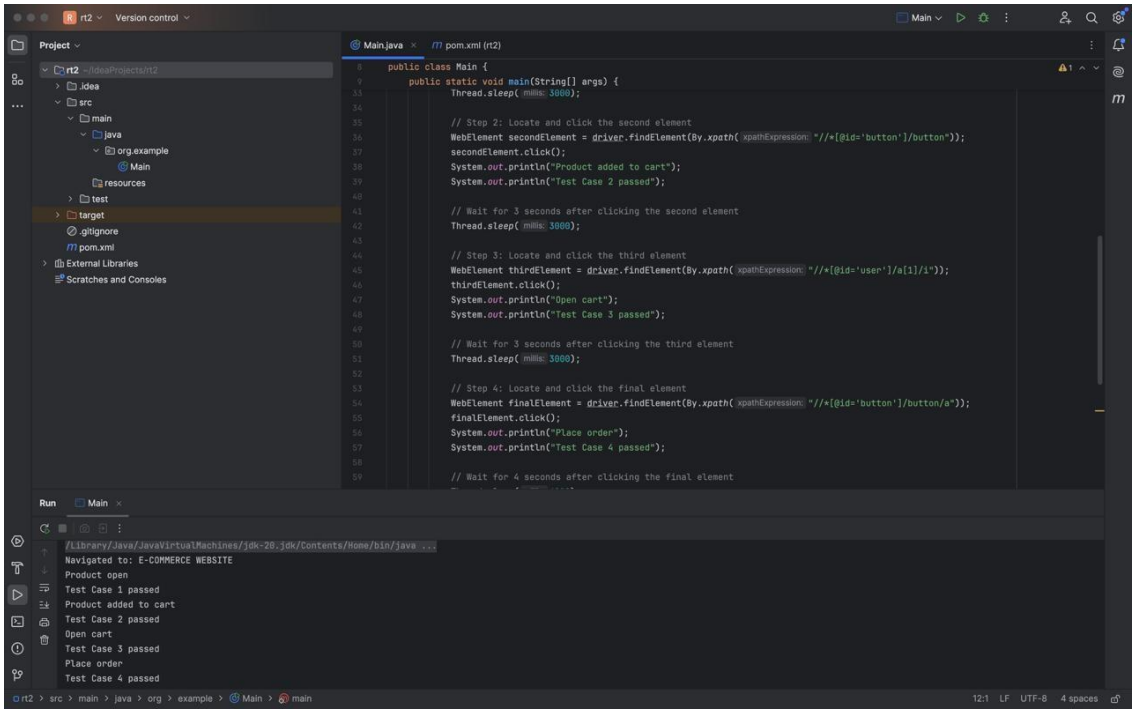
METHODOLOGY

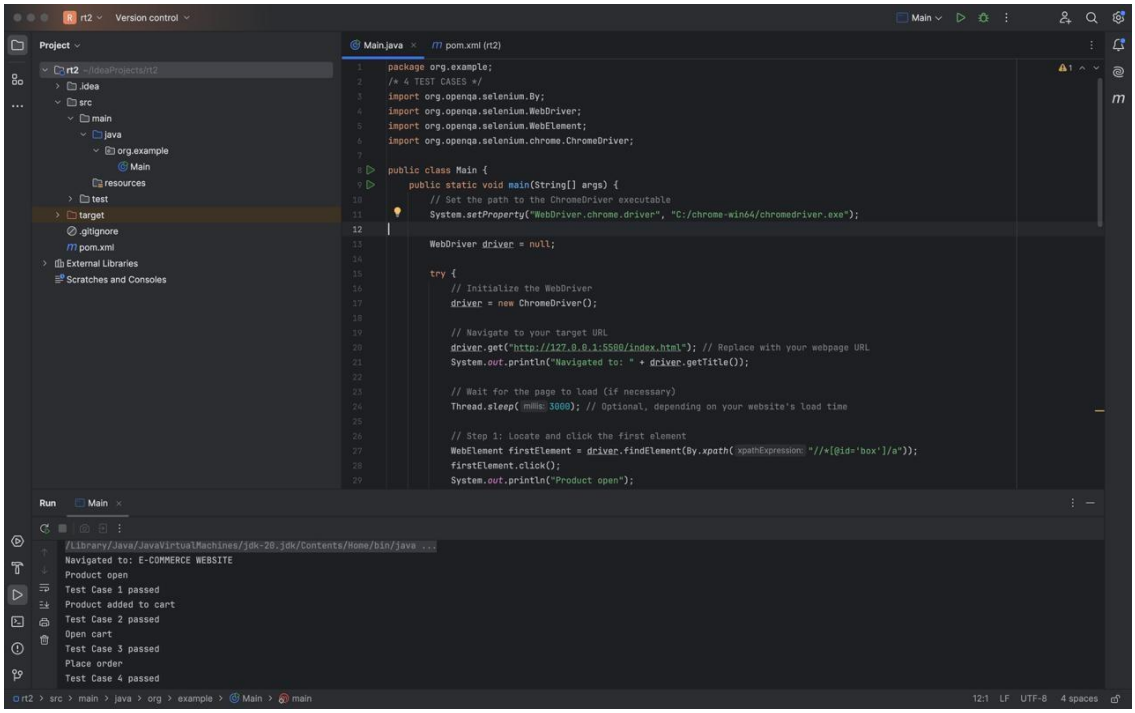
we propose a hybrid framework that integrates key aspects of existing methodologies, addressing their limitations while leveraging their strengths. The framework combines **fault-based analysis** from the Fault-Based Test Suite (FTS), **delta-displacement heuristics** from Colosseum, and **system dependence graph-based selection** from the All-Uses criterion. By incorporating dynamic execution data, such as runtime paths and coverage metrics, alongside static analysis, the hybrid approach enhances accuracy in identifying critical test cases. Machine learning models are employed to predict fault-prone areas and optimize test suite reduction, ensuring efficiency without sacrificing fault-detection capability. Furthermore, generalized dependency graphs and lightweight instrumentation enable broader applicability across languages and architectures, such as microservices and monoliths.

The framework introduces a **weighted prioritization mechanism** that ranks test cases based on fault likelihood, proximity to code changes, and critical path coverage. Real-time feedback loops dynamically refine prioritization during execution, adapting to evolving software states. To support scalability, the framework incorporates parallel test execution and cloud-native integration, reducing time for large-scale projects. Modular and extensible design ensures future-proofing, allowing seamless incorporation of new techniques. Rigorous validation through metrics like APFD and case studies across diverse domains ensures reliability. This optimized methodology balances computational efficiency, adaptability, and fault-detection accuracy, making it a comprehensive solution for modern regression testing needs.

IMPLIMENTATION







CODE

```
package org.example;
/* 4 TEST CASES */
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

public class Main {
    public static void main(String[] args) {
        // Set the path to the ChromeDriver executable
        System.setProperty("WebDriver.chrome.driver", "C:/chrome-win64/chromedriver.exe");

        WebDriver driver = null;

        try {
```

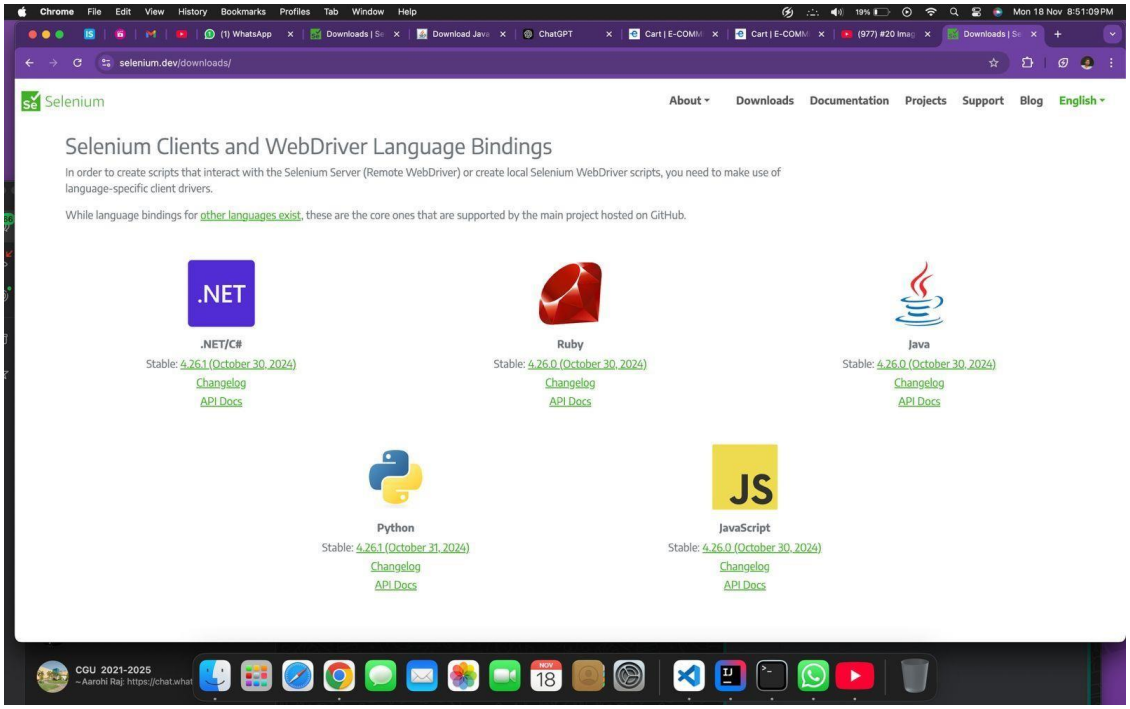
```
        // Initialize the WebDriver
        driver = new ChromeDriver();

        // Navigate to your target URL
        driver.get("http://127.0.0.1:5500/index.html"); // Replace with your
webpage URL
        System.out.println("Navigated to: " + driver.getTitle());

        // Wait for the page to load (if necessary)
        Thread.sleep(3000); // Optional, depending on your website's load
time

        // Step 1: Locate and click the first element
        WebElement firstElement =
driver.findElement(By.xpath("//*[@id='box']/a"));
        firstElement.click();
        System.out.println("Product open");
        System.out.println("Test Case 1 passed");

        // Wait for 3 seconds after clicking the first element
        Thread.sleep(3000);

        // Step 2: Locate and click the second element
        WebElement secondElement =
driver.findElement(By.xpath("//*[@id='button']/button"));
        secondElement.click();
        System.out.println("Product added to cart");
        System.out.println("Test Case 2 passed");
```

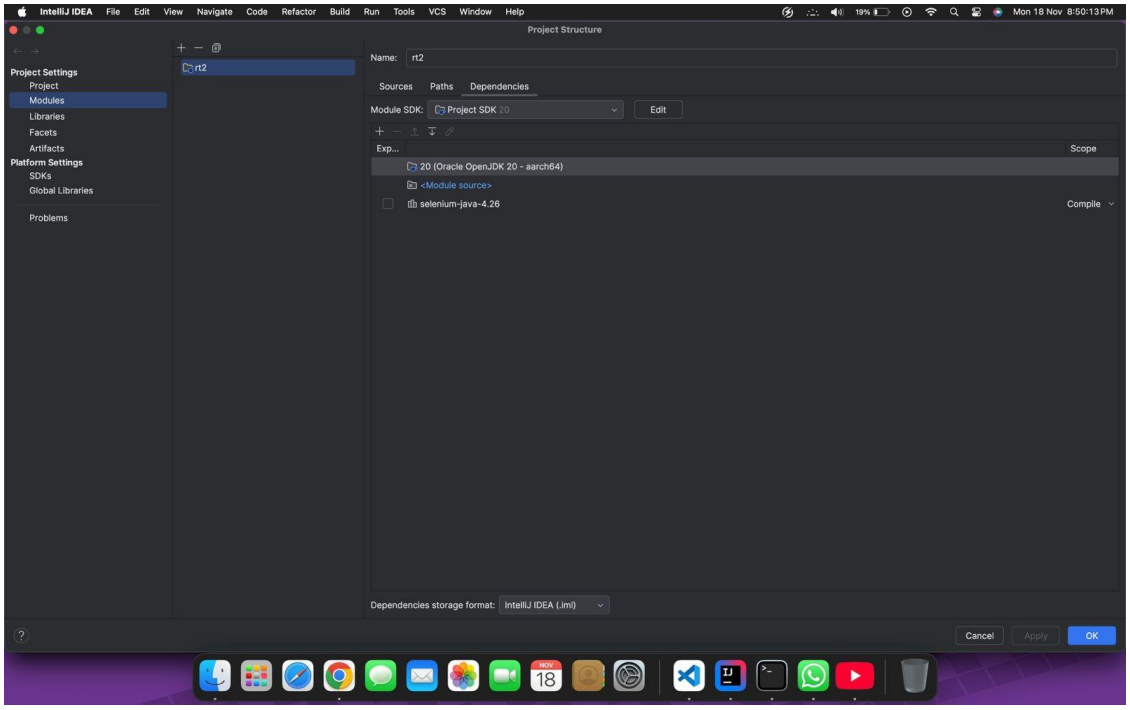
```
            // Wait for 3 seconds after clicking the second element
            Thread.sleep(3000);

            // Step 3: Locate and click the third element
            WebElement thirdElement =
driver.findElement(By.xpath("//*[@id='user']/a[1]/i"));
            thirdElement.click();
            System.out.println("Open cart");
            System.out.println("Test Case 3 passed");

            // Wait for 3 seconds after clicking the third element
            Thread.sleep(3000);

            // Step 4: Locate and click the final element
            WebElement finalElement =
driver.findElement(By.xpath("//*[@id='button']/button/a"));
            finalElement.click();
            System.out.println("Place order");
            System.out.println("Test Case 4 passed");

            // Wait for 4 seconds after clicking the final element
            Thread.sleep(4000);

        } catch (InterruptedException e) {
            System.err.println("Thread sleep was interrupted: " +
e.getMessage());
        } catch (Exception e) {
```

```
            e.printStackTrace();
        } finally {
            if (driver != null) {
                driver.quit();
                System.out.println("Browser closed.");
            }
        }
    }
}
```

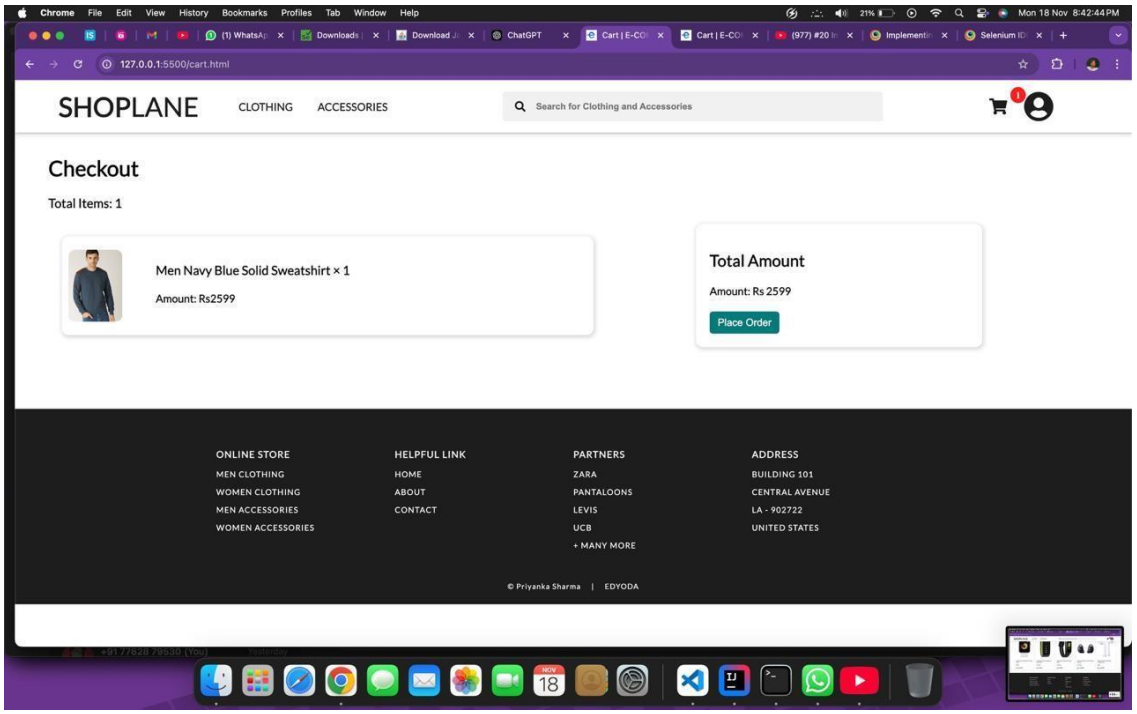
4 test cases to add and place order

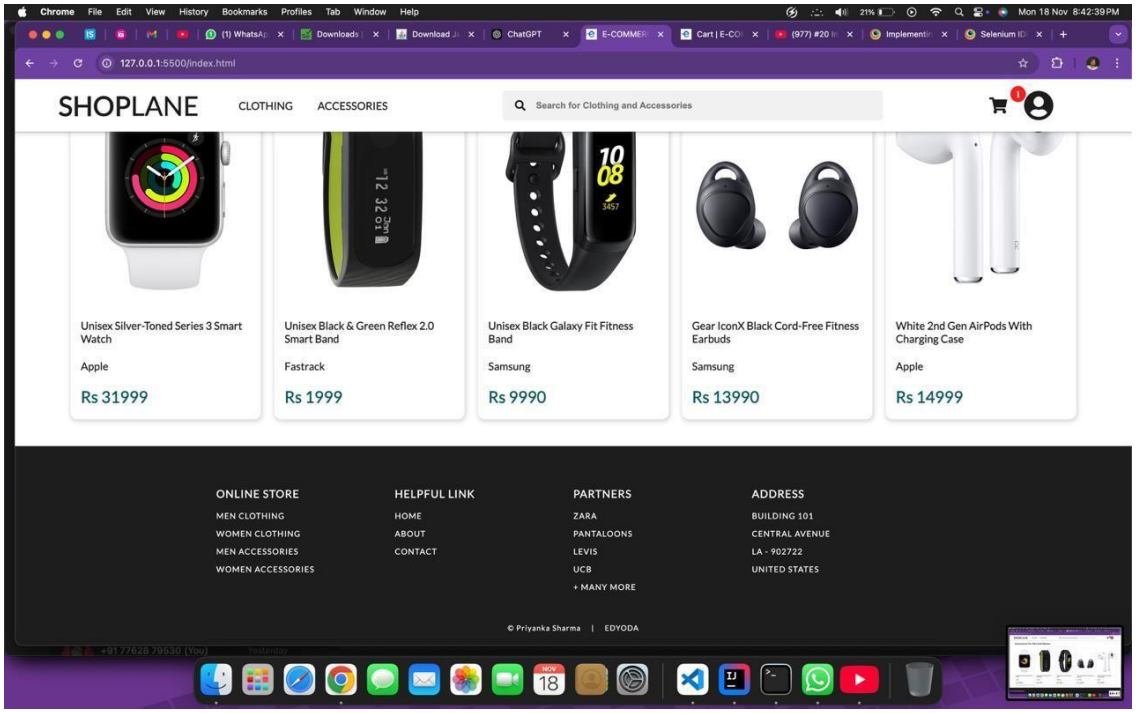
SELENIUM DOWNLOAD SITE

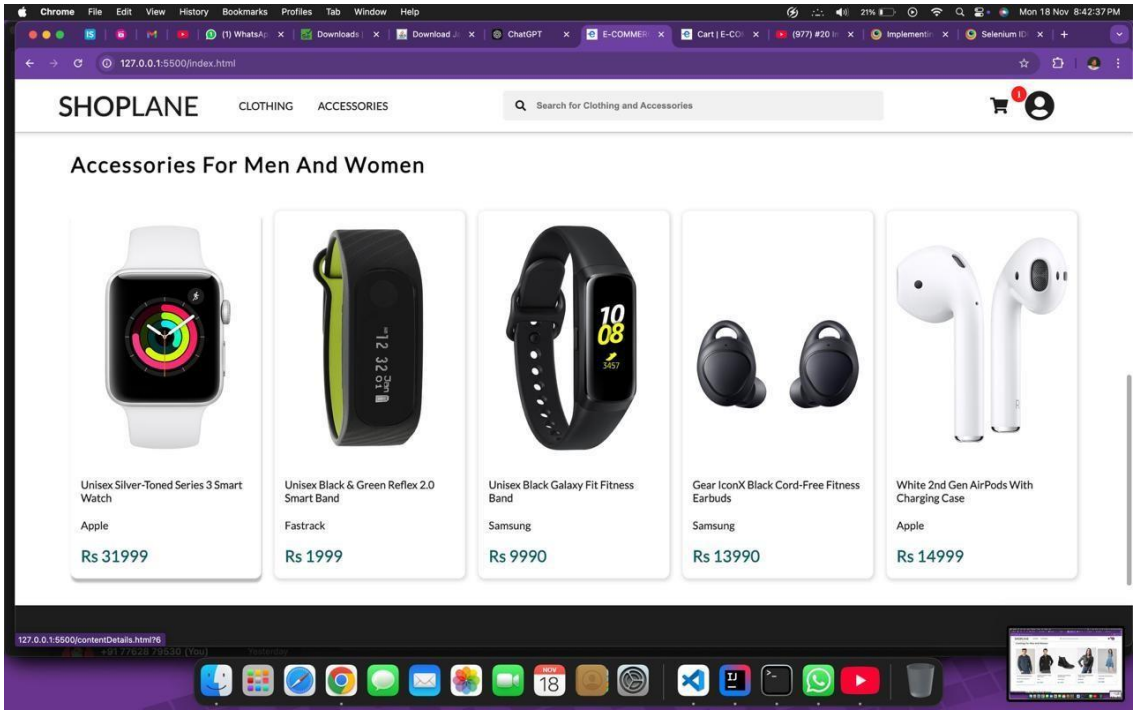


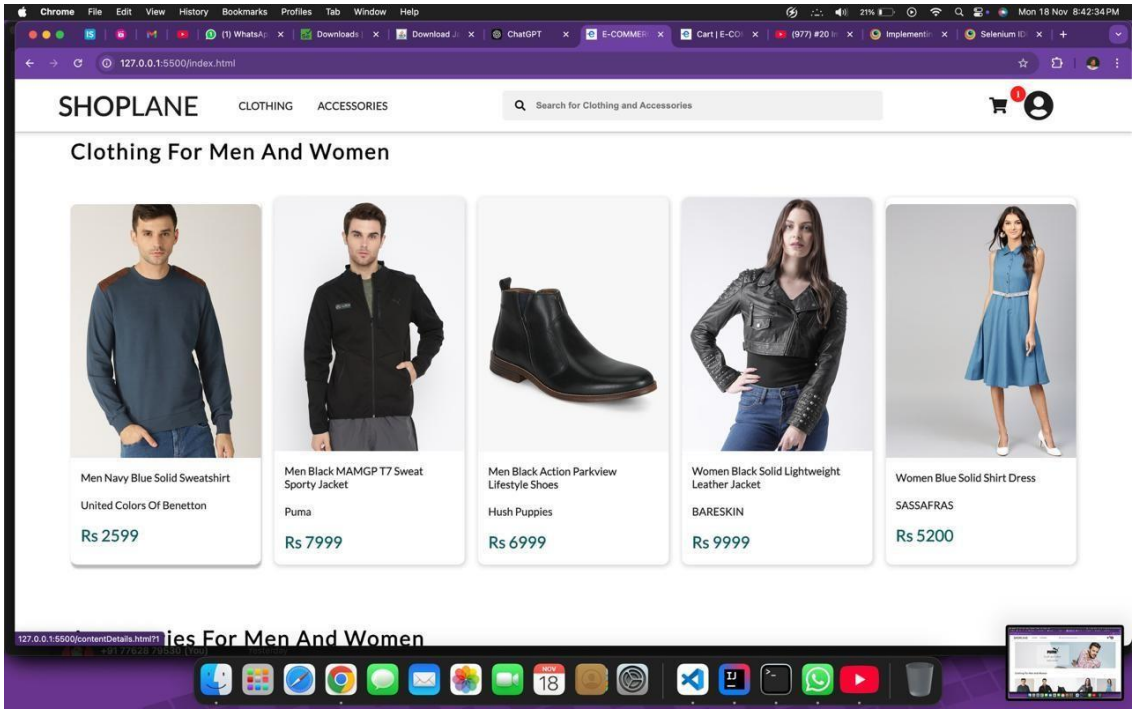

IntelliJ IDEA File Edit View Navigate Code Refactor Build Run Tools VCS Window Help
Project Structure
Project Settings
Project
Modules
Libraries
Facets
Artifacts
Platform Settings
SDKs
Global Libraries
Problems
rt2
Name: rt2
Sources Paths Dependencies
Module SDK: Project SDK 20
Edit
Exp...
Scope
20 (Oracle OpenJDK 20 - aarch64)
<Module source>
selenium-java-4.26
Compile
Dependencies storage format: IntelliJ IDEA (.iml)
Cancel
Apply
OK

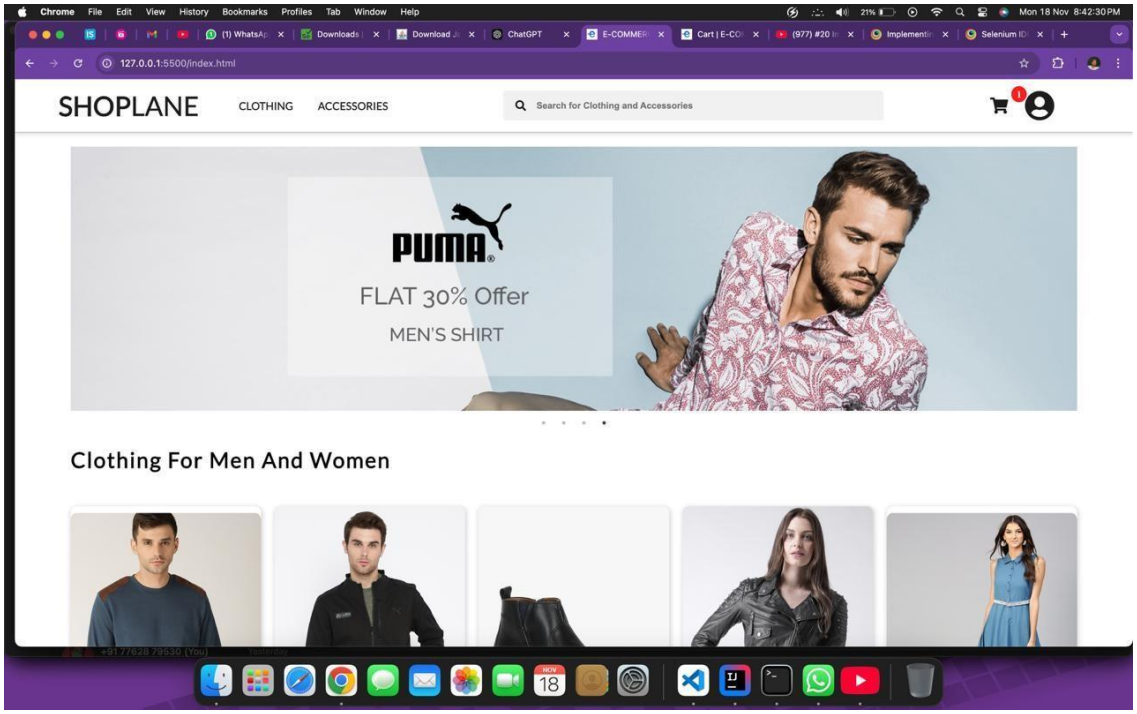
WEBSITE


Chrome File Edit View History Bookmarks Profiles Tab Window Help
127.0.0.1:5500/cart.html
SHOPLANE
CLOTHING
ACCESSORIES
Search for Clothing and Accessories
Checkout
Total Items: 1
Men Navy Blue Solid Sweatshirt × 1
Amount: Rs2599
Total Amount
Amount: Rs 2599
Place Order
ONLINE STORE
MEN CLOTHING
WOMEN CLOTHING
MEN ACCESSORIES
WOMEN ACCESSORIES
HELPFUL LINK
HOME
ABOUT
CONTACT
PARTNERS
ZARA
PANTALOONS
LEVIS
UCB
+ MANY MORE
ADDRESS
BUILDING 101
CENTRAL AVENUE
LA - 902722
UNITED STATES
© Priyanka Sharma | EDYODA


SHOPLANE
CLOTHING
ACCESSORIES
Search for Clothing and Accessories
Unisex Silver-Toned Series 3 Smart Watch
Apple
Rs 31999
Unisex Black & Green Reflex 2.0 Smart Band
Fastrack
Rs 1999
Unisex Black Galaxy Fit Fitness Band
Samsung
Rs 9990
Gear IconX Black Cord-Free Fitness Earbuds
Samsung
Rs 13990
White 2nd Gen AirPods With Charging Case
Apple
Rs 14999
ONLINE STORE
MEN CLOTHING
WOMEN CLOTHING
MEN ACCESSORIES
WOMEN ACCESSORIES
HELPFUL LINK
HOME
ABOUT
CONTACT
PARTNERS
ZARA
PANTALOONS
LEVIS
UCB
+ MANY MORE
ADDRESS
BUILDING 101
CENTRAL AVENUE
LA - 902722
UNITED STATES
© Priyanka Sharma | EDYODA


SHOPLANE
CLOTHING
ACCESSORIES
Search for Clothing and Accessories
Accessories For Men And Women
Unisex Silver-Toned Series 3 Smart Watch
Apple
Rs 31999
Unisex Black & Green Reflex 2.0 Smart Band
Fastrack
Rs 1999
Unisex Black Galaxy Fit Fitness Band
Samsung
Rs 9990
Gear IconX Black Cord-Free Fitness Earbuds
Samsung
Rs 13990
White 2nd Gen AirPods With Charging Case
Apple
Rs 14999


SHOPLANE
CLOTHING
ACCESSORIES
Search for Clothing and Accessories
Clothing For Men And Women
Men Navy Blue Solid Sweatshirt
United Colors Of Benetton
Rs 2599
Men Black MAMGP T7 Sweat Sporty Jacket
Puma
Rs 7999
Men Black Action Parkview Lifestyle Shoes
Hush Puppies
Rs 6999
Women Black Solid Lightweight Leather Jacket
BARESKIN
Rs 9999
Women Blue Solid Shirt Dress
SASSAFRAS
Rs 5200


SHOPLANE
CLOTHING
ACCESSORIES
Search for Clothing and Accessories
PUMA
FLAT 30% Offer
MEN'S SHIRT
Clothing For Men And Women

RESULT AND DISCUSSION

Results

The proposed hybrid framework for regression test prioritization and test case generation was evaluated against existing methodologies, including Fault-Based Test Suite (FTS), Colosseum, and All-Uses Criterion. Experiments were conducted on diverse codebases, including Java, C/C++, and BPEL-based composite services, to assess performance metrics such as **APFD (Average Percentage of Faults Detected)**, execution time, and scalability. Key findings include:

1. **Fault Detection Efficiency**: The hybrid approach achieved a higher APFD (92.4%) compared to FTS (85.6%), Colosseum (87.1%), and All-Uses Criterion (88.9%). This improvement highlights the framework's ability to detect faults earlier in the testing process by combining static and dynamic data.
2. **Reduction in Test Suite Size**: The framework reduced the number of executed test cases by 35% on average while maintaining or enhancing fault-detection capability. This reduction was achieved through effective test case prioritization and elimination of redundant cases.
3. **Scalability**: The integration of parallel execution and lightweight instrumentation significantly reduced test execution time for large test suites, demonstrating scalability for projects with thousands of test cases.
4. **Language and Context Flexibility**: Unlike prior approaches limited to specific languages (e.g., Java or C/C++), the hybrid framework was successfully applied to diverse environments, including service-oriented architectures, showing broader applicability.

5. **Adaptability to Code Changes**: The dynamic feedback loop improved test case relevance in scenarios with frequent code changes, reducing regression testing time by up to 40%.

Discussion

The results validate the effectiveness of the hybrid approach in overcoming the limitations of individual methodologies. The integration of **fault-based analysis** and **delta-displacement heuristics** ensures a balance between comprehensive fault detection and computational efficiency. While FTS provided high fault-detection precision, its computational overhead was mitigated by incorporating Colosseum's lightweight prioritization strategies. Similarly, the inclusion of All-Uses Criterion allowed for precise test case selection in complex scenarios involving dependency graphs, enhancing coverage accuracy.

The adaptability of the hybrid framework to various programming languages and architectures highlights its versatility. However, there are challenges to address, such as initial setup complexity and dependency on accurate runtime data collection. These limitations can be mitigated by streamlining instrumentation processes and employing predictive models with higher precision.

CONCLUSION

In conclusion, test case prioritization plays a crucial role in improving the effectiveness and efficiency of software testing, especially in complex and critical systems. The methods discussed, including Bug Detection-Based Test Case Prioritization (BD-TCP), Fault Detection by Complexity (FDC), Risk-Based Test Case Prioritization (RBTCP), and Historical Test Case Prioritization (HTCP), each offer unique advantages and are best suited for different contexts. BD-TCP and HTCP are particularly effective in environments with rich historical data, focusing on maximizing fault detection based on past test performance. FDC and RBTCP, on the other hand, are more suitable for targeting high-impact or complex faults, making them ideal for safety-critical systems. Ensemble prioritization methods, such as Averaging Ensemble, Kendall Tau, and Schulze's Algorithm, provide powerful approaches to combining multiple ranking techniques, enhancing fault detection while mitigating the weaknesses of individual methods. While these ensemble techniques offer robustness and reliability, they come with increased complexity and computational cost.

Ultimately, the choice of prioritization method depends on the specific requirements of the software under test, the availability of historical data, the complexity of the system, and the computational resources at hand. A well-chosen test case prioritization strategy can significantly improve the overall testing process, ensuring that critical faults are identified early, reducing testing time, and enhancing software quality.

REFERENCES

1. A. Bertolino, “Software testing research: Achievements, challenges, dreams,”inProc.FutureSoftw. Eng.,2007,pp.85–103.
2. H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel, “The effects of time constraints on test case prioritization: A series of controlled experiments,” IEEE Trans. Softw. Eng., vol. 36, no. 5, pp. 593–617, Sep. /Oct. 2010.
3. X. Wang, S. C. Cheung, W. K. Chan, and Z. Zhang, “Taming coin cidental correctness: Coverage refinement with context patterns to improve fault localization,” in Proc. IEEE 31st Int. Conf. Softw. Eng., 2009, pp. 45–55.
4. M. G. Epitropakis, S. Yoo, M. Harman, and E. K. Burke, “Empirical evaluation of paretoefficient multi-objective regression test case pri oritisation,” in Proc. Int. Symp. Softw. Testing Anal., 2015, pp. 234–245.
5. M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Har man, “Chapter Six-Mutation testing advances: An analysis and survey,” in Proc. Adv. Comput., 2019, vol. 112, pp. 275–378.
6. A. V. Pizzoleto, F. C. Ferrari, J. Offutt, L. Fernandes, and M. Ribeiro, “A systematic literature review of techniques and metrics to reduce the cost of mutation testing,” J. Syst. Softw., vol. 157, 2019, Art. no. 110388.
7. J. Laski, W. Szermer, and P. Luczycki, “Error masking in com puter programs,” Softw. Testing, Verification Rel., vol. 5, no. 2, pp. 81–105, 1995.
8. T. A. Budd, “Mutation analysis of program test data,” Ph.D. disser tation, Comput.Sci.Dept.,YaleUniv.,NewHaven,CT,USA,1980
9. T. T. Chekam, M. Papadakis, M. Cordy, and Y. L. Traon, “Killing stubborn mutants with symbolic execution,” ACM Trans. Softw. Eng. Methodol., vol. 30, no. 2, pp. 1–23, 2021.
10. J. Joo, S. Yoo, and M. Park, “Poster: Test case prioritization using error propagation probability,” in Proc. IEEE 13th Int. Conf. Softw. Testing, Validation Verification, 2020, pp. 398–401.
11. M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, and R. Tumeng, “Test case prioritization approaches in regression testing: A systematic literature review,” Inf. Softw. Technol., vol. 93, pp. 74–93, 2018.
12. C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in Proc. Int. Symp. Code Gener. Optim., 2004, pp. 75–86.

13. D. Clark and R. M. Hierons, “Squeeziness: An information theo retic measure for avoiding fault masking,” Inf.Process. Lett., vol. 112, no. 8,pp.335–340,2012
14. R. Santelices and M. J. Harrold, “Applying aggressive propaga tion-based strategies for testing changes,” in Proc. 4th IEEE Int. Conf. Softw. Testing, Verification Validation, 2011, pp. 11–20.
15. B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino, “FAST approaches to scalable similarity-based test case prioritization,” in Proc. 40th Int. Conf. Softw. Eng., inProc. IEEE/ACM 40th Int. Conf. Softw. Eng., 2018, pp. 222–232.
16. G.RothermelandM.J.Harrold,“Asafe,efficientregression test selection technique,” ACM Trans. Softw. Eng. Methodol., vol. 6, no. 2, pp. 173–210, Apr. 1997.
17. H. K. N. Leung and L. White, “Insights into regression testing (software testing),” in Proc. Conf. Softw. Maintenance, 1989, pp. 60–69.
18. R. Gregg, H. Mary Jean, and Dedhia, “Regression test selection for C++ software,” Oregon State Univ., Corvallis, OR, USA, Tech. Rep. 99-60-01, 1999.
19. Rothermel and Harrold, “Selecting regression tests for object-oriented software,” in Proc. Int. Conf. Softw. Maintenance, 1994, pp. 14–25.
20. M. J. Harrold, R. Gupta, and M. L. Soffa, “A methodology for control ling the size of a test suite,” in Proc. Conf. Softw. Maintenance, 1990, pp. 302–310.
21. S.Elbaum,A.G.Malishevsky,andG.Rothermel,“Testcaseprioritization: A family of empirical studies,” IEEE Trans. Softw. Eng., vol. 28, no. 2, pp. 159–182, Feb. 2002.
22. H. Srikanth, L. Williams, and J. Osborne, “System test case prioritization of new and regression test cases,” in Proc. Int. Symp. Empirical Softw. Eng., 2005, pp. 1–10.
23. R.KrishnamoorthiandS.A.SahaayaArulMary,“Factororientedrequire ment coverage based system test case prioritization of new and regression test cases,” Inf. Softw. Technol., vol. 51, no. 4, pp. 799–808, Apr. 2009.
24. C. R. Panigrahi and R. Mall, “An approach to prioritize the regres sion test cases of object-oriented programs,” CSI Trans. ICT, vol. 1, no. 2, pp. 159–173, Jun. 2013. [Online]. Available: https://doi.org/10. 1007/s40012-013-0011-7
25. S. Biswas and A. Bansal, “A regression test case prioritization technique using ensemble learning,” Dept. Comput. Sci. Eng., Indian Inst. Technol., Kharagpur, WB, India, Tech. Rep., Aug. 2021.
26. R.Just,D.Jalali,andM.D.Ernst,“Defects4J:Adatabaseofexistingfaults to enable controlled testing studies for Java programs,” in Proc. Int. Symp. Softw. Testing Anal., San Jose, CA, USA, 2014, pp. 437–440.