Regression Testing in Developer Environment for Absence of Code Coverage

M. Thillaikarasi1, K. Seetharaman2

1Department of Computer Science and Engineering, Annamalai University, Annamalai Nagar, India
2Computer Science Wing, Annamalai University, Annamalai Nagar, India
Email: m.thillaikarasi@yahoo.com

Received 17 April 2014; revised 15 May 2014; accepted 10 June 2014

Copyright © 2014 by authors and Scientific Research Publishing Inc.
This work is licensed under the Creative Commons Attribution International License (CC BY).
http://creativecommons.org/licenses/by/4.0/

Abstract

The techniques of test case prioritization schedule the execution order of test cases to attain respective target, such as enhanced level of forecasting the fault. The requirement of the prioritization can be viewed as the en-route for deriving an order of relation on a given set of test cases which results from regression testing. Alteration of programs between the versions can cause more test cases which may respond differently to following versions of software. In this, a fixed approach to prioritizing test cases avoids the preceding drawbacks. The JUnit test case prioritization techniques operating in the absence of coverage information, differs from existing dynamic coverage-based test case prioritization techniques. Further, the prioritization test cases relying on coverage information were projected from fixed structures relatively other than gathered instrumentation and execution.

Keywords

Software Testing, Regression Testing, Test Case, Prioritization, JUnit, Call Graph

1. Introduction

Regression testing concentrates on finding defects after a major code change has occurred. Specifically, it exposes software regressions or old bugs that have reappeared. It is an expensive testing process that has been estimated to account for almost half of the cost of software maintenance. To improve the regression testing process, test case prioritization techniques organizes the execution level of test cases. Further, it gives an improved rate of fault identification, when test suites cannot run to completion (Figure 1).

As stated earlier, it provides an effective evaluation of previous coverage data accuracy, and also creates an impact to the precision of prioritization.

http://dx.doi.org/10.4236/jsea.2014.78057
New test cases for which coverage data are collected cannot be easily added onto prioritized orders. Thus, test suites are increased to cover newly adjoined program or system components. In view of the above, it is not clear about associating their use into a regression testing method.

A fixed approach to prioritizing test cases and also JUnit test case prioritization techniques operating in the absence of coverage information (JUPTA) varies from dynamic coverage-based techniques. Here, it prioritizes test cases based on coverage information estimated from static structures rather than gathered instrumentation and performance.

2. Techniques Revisited

Hyunsook Do et al. [1] designed and performed a controlled experiment examining whether test case prioritization can be effective on Java programs tested under JUnit and evaluated that test case prioritization can significantly improve the rate of fault detection of JUnit test suites. In addition, [2]-[4] studied the cost factors under less expensive with prioritization.

P. K. Chittimalli et al. [5] studied the inaccuracies that can exist when an application—regression-test selection—uses estimated and obsolete coverage data and overhead incurred.

D. Jefferey et al. [6] presented a new approach for test suite reduction that attempts to use additional coverage information on test, whereas some additional test cases in the reduced suites that are redundant with respect to the testing criteria are used for suite minimization.

James A. Jones et al. [7] presented new algorithms for test-suite reduction and prioritization that can be tailored effectively with modified condition/decision coverage, MC/DC.

Zheng Li et al. [8] addressed the problems of choice of fitness metric, characterization of landscape modality, and determination of the most suitable search technique to apply.

S. Elbaum et al. [9] showed that each of the prioritization techniques considered can improve the rate of fault detection of test suites.

Moonzoo Kim [10] analyzed the strong and weak points of two different software model checking technologies in the viewpoint of real-world industrial application.

Zielińska [11] discussed the system requirements, design, architecture and modes of operation. It also contains a detailed comparison of the FEAT framework.

M. Shahid et al. [12] combined code based coverage and requirement based coverage to get hybrid coverage information to support regression testing.

S. Segura et al. [13] showed that mutation testing is an effective and affordable technique to measure the effectiveness of test mechanisms in OO systems.

3. Overview

Test case prioritization techniques schedule test cases so that those with the highest priority, according to some criterion, are executed earlier in the regression testing process than lower priority test cases.

Over the past few years, many test case prioritization techniques have been proposed in the literature. Most of these techniques require data on the dynamic execution in the form of code coverage information for test cases.
However, the collection of dynamic code coverage information on test cases has several associated drawbacks, including cost increases and reduction in prioritization precision. New test cases in which coverage data have not yet been collected cannot be easily added onto prioritized orders.

Regression Testing merely reorders test cases to optimize a score function. Based on the hypothesis, many test cases do not reveal faults. Regression testing techniques select a subset of the existing test cases for further execution. RTP provides a partial solution by allowing testers to execute as many test cases as they can implement. However, conventional RTP score functions do not consider time constraints, they assume that all of the test cases will be executed.

It sorts all the software elements based on ascending existing coverage. To break ties, it uses a descending priority measure of the elements. The requirements of the software elements identify the priority for their current iteration. The current coverage values are recognized to reflect the coverage of the selected test case and this is being added to the result set.

Advanced RTP approaches have been compared against coverage-based techniques. Two conventional coverage-based techniques are included in our experiments: method-level total coverage (TMC) and method-level additional coverage (AMC). TMC uses the total number of covered methods for ordering the test cases. AMC is the feedback-employing variations of TMC. A feedback-employing coverage-based technique orders test cases based on the number of code elements yet to be covered. Prominently, they have observed that a sizable performance gap exists between prioritization heuristics and optimal prioritization.

Depending on the technique chosen, a different set of actions needs to be activated. If the user prefers a coverage-based prioritization technique, a coverage analysis needs to be performed. An analysis can be done by using either a static or dynamic strategy. This coverage report gives the testers about the first evidence whether a coverage-based prioritization would be adequate or not in a particular project. The prioritization activity is also responsible for collecting any data needed for prioritization either a different type of technique is chosen (coverage nor change-based). This data can be collected directly from the project or from interactions with the user. The prioritization activity is responsible for running the prioritization algorithms. The most needed data were manipulated in the previous activities, the running of the prioritization algorithms should be a simple task. After the prioritization, a set of output artifacts must be generated (Generate Prioritized Order activity).

4. JUnit Test Case Prioritization

4.1. JUPTA

There are two components that can be resumed in the build of the specific JUPTA method:

JUnit has two levels of test cases, test method and test-class, and JUPTA can be implemented to schedule test cases at any level. JUPTA uses TA values to guide the test case prioritization process.

JUPTA can be used with or without feedback (using the “total” or “additional” strategies).

4.1.1. Process of JUPTA

The overall JUPTA approach can be summarized as follows:

1) Initially, the given test suite T contains all JUnit test cases which are to be prioritized and the prioritized test suite PS is empty.

2) On each repetition, JUPTA deletes one test case with the maximal testing ability (TA) from T and attaches it to PS.

3) The standard setup is continual until T to become clear out, then JUPTA stops the prioritization process and returns PS.

In the test case prioritization process, the (adjusted) TA values of all test cases may reach 0. But TS may not be empty yet. Although the test cases prioritized so far have achieved the maximum coverage, there are remaining test cases in the test suite to be prioritized and none can increase the current coverage. To handle this issue, additional JUPTA performs several prioritization cycles until all test cases have been prioritized. In each cycle, additional JUPTA takes the test cases to be prioritized as input, ignoring the test cases that have been prioritized in previous cycles. Moreover, it calculates the TA values of the input test cases and selects the test case with the largest TA. As soon as a test case is selected, additional JUPTA recalculates the TA values of the remaining test cases in the input and selects the test case with the largest TA.
4.1.2. Total TA Based Prioritization
Initially, TS contains all the test cases to be prioritized and PS is empty. JUPTA calculates initial TA of each test case in TS based on the call graph of the test case. Each time after JUPTA selects a test case with the highest TA in TS1, removes the test case from TS, JUPTA adds the test case into PS. JUPTA stops when the TS is empty. JUPTAT is analogous to the total technique based on method coverage, which prioritizes test cases, according to the number of methods covered by test cases.

4.1.3. Additional TA Based Prioritization
The usual process of this technique is similar to the first technique, however, each time after JUPTA selects a test case with the highest TA in TS, JUPTA adjusts TA values of the remaining test cases in TS via consideration of the test cases in Slide Set (which stores already prioritized test cases). In the adjustment, the methods covered by the call graphs of already selected test cases in Slide Set will be ignored in TA calculation, because it is more likely that they expose the same faults exposed by the prioritized test cases.

5. Proposed Work
5.1. Test Case Generation
A test case, in software engineering, is a set of conditions or variables under which a tester will determine whether an application, software, system or one of its features is working as it was originally established for it to do. The mechanism for determining whether a software program or system has been successful or unsuccessful is known as a test oracle. In some instance, an oracle may be a required or use case, while in others it could be a heuristic. It may take many test cases to determine whether a software program or system is considered sufficiently scrutinized to be released. Test cases are often referred to as test scripts, particularly when written they are usually collected into test suites (Figure 2).

5.2. Test Ability Generation
TA is a heuristic intended to predict the ability of a test case to reveal unexposed faults. Since its unknown, whether a test case will reveal faults before it has been run, TA is measured in terms of a test case’s potential to cover system components (in terms of total components covered or components newly covered) as a proxy to understand test case’s as a true fault-revealing potential.

JUPTA takes the test cases to be prioritized as input, ignoring the test cases that have been prioritized in previous cycles. Calculates the TA values of the input test cases and selects the test case with the largest TA. Once a test case is selected, additional JUPTA recalculates the TA values of the remaining test cases in the input and selects the test case with the largest TA. The set of selected test cases recalculates the TA values of the test cases in TS, as soon as a test case is selected and removed from TS. This recalculation causes the TA values for all test cases remaining in TS to reach 0. The test cases prioritized so far have achieved maximal coverage, and no remaining test cases can increase coverage.

![Figure 2. Test case generation.](image-url)
5.3. Call Graph Generation

JUPTA analyzes the static call graphs of JUnit test cases and uses the sales of each JUnit test case as a basis for measuring the TA value of each JUnit test case. JUPTA uses TA values to guide the test case prioritization process, although TA is anticipated based on the “relevant” relation derived from static call graphs.

Average Percentage Faults Detected (APFD) Metric

\[
APFD = 1 - \frac{\sum_{i=1}^{num_f} TF_i}{num_i \times num_f} + \frac{1}{2 \times num_i}
\]

In the formula, \(num_i\) denotes the total number of test cases, \(num_f\) denotes the total number of detected faults, and \(TF_i\) (\(1 \leq i \leq num_f\)) denotes the smallest number of test cases in sequence that need to be run in order to expose fault “I”. The APFD values range is \([0, 1]\). For any given test suite, its \(nut\) and \(nymph\) are fixed so that higher APFD values signal that the average value of \(TF_i\) (variable \(I\) ranges from 1 to \(num_f\)) is lower and thus imply a higher fault-detection rate.

5.4. Fault Detection

The dynamic coverage-based techniques use actual coverage information, whereas JUPTA uses estimated coverage information. The former is instinctively better than the latter in terms of fault-detection effectiveness. Techniques using feedback tend is more effective than their corresponding techniques which does not use feedback. Then the techniques using feedback at the test-method level tend to be the most effective. Techniques using feedback retain the advantage over techniques without feedback because the latter may postpone the detection of faults in rarely covered statements. According to the results of the empirical study, dynamic coverage-based techniques can be more effective than JUPTA techniques for detecting seeded faults.

5.5. Results and Analysis

Java programs tested in the JUnit framework, which is widely used in developing Java software. A JUnit test case is a piece of executable source code containing testing content, we are using a sample Java application which contains a large number of test suites, we execute the test cases in different testing criteria (TMC, AMC, TSC, ASC, finally TCC) are useful for identifying prioritized test cases.

Select the Java source code path and choose the techniques Total method coverage, Additional method coverage, Total statement coverage, Additional statement coverage, finally Total class coverage. Coverage report shows that the % of method, statement, Class, covered in the test suites

- 0% = of instrumentation
- 50% = of coverage analysis
- 75% = of prioritization.

Is covered in the test suites we are using. Code tree shows that count of the method and statements called in the Test suites, now we show prioritized order for uncovering code coverage. Number of failed test cases 8, number of faults 8, Average Percentage for Fault Detection value is \(APFD = 60.12\). This technique experiment the importance of uncovered codes priority (Figure 3).

6. Simulation Results

TEST CASE GENERATION (Figures 4-10).

7. Conclusion and Future Enhancement

As a result, it is suggested that prioritization of the test cases is achieved only if the appropriate test strategy is employed along with a sufficiently high fault prediction accuracy. However, sufficient data are available to fit a predictive model and to develop good fault prediction accuracy, also the appropriate test strategy can significantly reduce the necessary level of test effort while still maintaining the same level of fault detection or by providing a higher level of fault detection with the same test effort. Further, by avoiding the need to instrument code and execute test cases, our approach may be more relevant than dynamic coverage-based approaches if cases gather coverage information which is inappropriate, nor cost effective. This technique can be improved
Figure 3. Block diagram of proposed work.

Figure 4. Test case generation.

Figure 5. Application location.
Figure 6. Coverage report.

Figure 7. Code tree.

Figure 8. Prioritized order.
using prioritized order of previous versions and customer requirements. It can be compared between executing test cases and uncovered a test case with the help of developer feedback if any.

References


Scientific Research Publishing (SCIRP) is one of the largest Open Access journal publishers. It is currently publishing more than 200 open access, online, peer-reviewed journals covering a wide range of academic disciplines. SCIRP serves the worldwide academic communities and contributes to the progress and application of science with its publication.

Other selected journals from SCIRP are listed as below. Submit your manuscript to us via either submit@scirp.org or Online Submission Portal.