An Empirical Study of Regression Testing Techniques

**Ani Rahmani1, Joe Lian Min2 and Asri Maspupah3**

1,2,3Department of Computer Engineering and Informatics   
Bandung State Polytechnic

Jln. Gegerkalong Hilir Ds. Ciwaruga – Bandung 40012 - Indonesia

**Abstract.** Regression testing is a testing process for established software and developed ones. Adding features to the software should not cause errors in the previous software version. There are several methods for implementing regression testing. The most straightforward is to do retest-all, but this technique is costly and protracted because it has to execute all test cases in the test suite. This situation is the nature of regression testing research, i.e. determining how to execute only a few test cases to shorten the time, and the faults can be found as much as possible. Recently there are various algorithms for regression testing process. This paper explains an empirical study of several regression testing methods. We use some software under test (SUT), regression program, and automatic testing tools that available as an open-source. The empirical study shows that each technique has a specific characteristic, and also has effectiveness significantly to reduce execution time testing process.

1. **Introduction**

Regression testing (RT) is re-testing activity, to ensure that modifications to the system do not affect to other parts, and unrelated factors remain stable as before [1]. In general, there are three approaches in regression software testing: regression test minimization (RTM), regression test selection (RTS), and test case prioritization (TCP) [2]. The RTM technique will reduce or delete test cases that are considered redundant or nearly the same. RTS will select the test cases that meet the specific criteria. While TCP will order the test cases according to the specified criteria for prioritizing [3].

Recently regression testing research area has a challenge to get better effectiveness. For instance, RTS research has conducted in the study [1][2] and [3]. The study [4] [5] and [6] studied an RTM technique, and the last, study [7][8] and [9] in the TCP technique. Meanwhile, authors [10] and [11] studied regression testing in common.

H, Do [10] has reviewed 99 papers that discuss regression testing from some perspective: concepts, metrics, technology, scalability comparisons, and the trend approaches. Dini et al. [4] investigated the effect of two techniques, "fine-grained" and "coarse-grained" in selecting test cases based on five-level dependence. The study conducted trials on 800 commit processes for regression testing. His research results show that the average fine-grained RTS is more effective than the search-based test case generation. Meanwhile, Anderson [11] recommends a TCP technique based on elementary data. This study shows an increase in the mean of reduction in the test suite and testing execution time.

Other research has looked at regression testing from several points of view. Rosero [12] has surveyed 460 research papers and described 25 articles that discuss the application of 31 regression testing techniques. This study recommended the strategies and measurement of regression testing, including costs and the ability to detect efficient faults.

This paper is an initial attempt to understand regression testing techniques by investigating some existing algorithms for the regression testing technique. The observation aims to see how these techniques work and how effective they are. In the next sections, we explain the method (section2); the results and discussion in section 3, and finally (section 4) is the conclusion.

1. **Method**

There are several activities in this study. All of that explain in this section.

# *2.1 Regression Testing Techniques Selected*

Although there are many algorithms in each regression testing technique, this study only uses one algorithm for each regression testing technique. Each algorithm is studied from how it works, the input required, and the output results. The algorithms are:

1. BabelRTS for regression test selection (RTS) process; This study utilizes the BabelRTS source code available at Github [13].
2. Fast++ for regression rest minimization (RTM) process; Source code is available on Github, and for execution, we modified accordingly[14].
3. Greedy for test case prioritization (TCP) process. The source code is implemented based on the algorithm in the study [15].

2.2 Software under Test (SUT)

Software under test (SUT) in this study identifies by the line of code (LoC), the number of faults, and the number of test cases. Table 1 shows the lists of SUTs for test case selection or regression test selection (RTS) studies. OpenMRS and JodaTime are available on Github, while NumbertoWord and Palindorm were implemented ourself. On the RTM and TCP study, we use the print\_token and print\_token2 software as SUT. These applications area available in software-artefact infrastructure repository (SIR) [16]. Print\_token has eight versions consisting of one original version and seven mutant versions. In this study, we use the original version and the seventh version.

**Table 1**. List of SUT

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| No | SUT Name | Description | Regression  technique | Source code Language | *a* | *b* | *c* |
| 1 | OpenMRS | Medical Record System Open Source | RTS | Phyton | 90 | 7 | 20 |
| 2 | NumbertoWord | Convert the number to the word | RTS | Phyton | 70 | 5 | 10 |
| 3 | Palindrom | Reverse the word | RTS | Phyton | 51 | 3 | 10 |
| 4 | JodaTime | Jona Time Transformer | RTS | Phyton | 71657 | 11 | 24 |
| 5 | Print\_token | Identify and print a token | RTM | C | 563 | 11 | 1107 |
| 6 | Print\_token2 | Identify and print a token | TCP | C | 570 | 9 | 1100 |

*Notes: a. Line of Code (LoC); b. Number of Fault; c. Number of test case*

2.3 Automatic Testing Tools

This empirical study uses an automatic testing tools BabelRTS (for RTS process) [13]*,* and the ***MTS*** tools that available in the SIR repository[16] for RT*M* and TCP processes. Both tools are the open source software.

2.4 Effectiveness Metric

This study uses one metric for each technique, adjusted for specific their characteristics. RTS and RTM will reduce the number of test cases from the test suite. Thus, it will undoubtedly affect the execution time of the testing process. Based on this situation, the RTS and RTM techniques use the execution time metric. Meanwhile, since the TCP will not subtract the test cases from the test suite, like the most TCP researchers, we use average percentage fault detection (APFD) metric to measure the effectiveness technique.

1. **Result and Discussion**

This section explains the result of this study.

3.1 BableRTS for Regression Test Selection

BabelRTS is an algorithm as well as a tool built using the Python language. The source code is openly available on Github[13]. BabelRTS has a function to run the test case selection process. This function selects a test case that must be tested again when the SUT is modified. In general, babelRTS performs the following processes:

1. Hashing all files in the SUT used
2. Comparing the hashing of the previous version of SUT and the latest version of SUT.
3. Mark all files differently with the status change. A different file filled refers to the file changed, and the file that was recently added.
4. Looking for the dependency of the file given the status change and changing its status to change.
5. Save the hashing of the latest version of SUT and save the process results in a file with JSON format (Table 2).

Table 2. Output Files List

|  |  |  |
| --- | --- | --- |
| No. | File name | Description |
| 1 | selected.json | A JSON file that contains a list of test cases that must be retested as a result of modifications |
| 2 | dependency.json | A JSON file that contains a list of dependencies for all files on SUT. |
| 3 | change.json | JSON file that contains a list of change files in the latest version |
| 4 | hashes.json | JSON file which contains hashing of all SUTs. |

3.2 FAST++ Algorithm for Regression Test Minimizing

The FAST ++ algorithm has two inputs: test suite and a fault matrix. Format for the fault matrix is ​.txt, and contains a list of test cases that found the faults. Globally, the FAST ++ algorithm has 3 (three) stages: (i) Preparation, (ii) Reduction, and (iii) Completion. The FAST++ algorithm shows below.

|  |  |
| --- | --- |
| **Input: Test Suite T, Fault Matrix F**  **Output: Reduce Test Suite File f**   1. P 🡨 RandomProjection(T) 2. S 🡨 FirstSelection(P) 3. R 🡨 List(S) 4. D 🡨 Distance() 5. D(S) 🡨 0 6. **while** (Size(R) < B) **do** 7. **for** **all** t ∈ P **do** 8. **if** d(P(t)), P(s)2 < D(t) then 9. D(t) 🡨 d(P(t), P(s))2 10. S 🡨 ProportionalSample(P,D) 11. R 🡨 Append(R,S) 12. D(s) 🡨 0 13. **end while** 14. **fd1** 🡨 FDL(R, F) 15. **fd** 🡨 OpenFile(‘reduceTS.txt’) 16. **for** **all** tc ∈ R **do** 17. **write** (f, tc) 18. CloseFile(f) 19. **return** f | In the preparation stage, FAST++ will map the test cases in the test suite to a vector-based on the vector-space model and reduce the test cases' dimensions using random projection. This stage is a pre-processing to enter the reduction stage. In the reduction stage, the vector is entered into the k-means algorithm to calculate its relationship. In the FAST ++ algorithm, the cluster value for the k-means is the number of test cases that took in one test. In this study, we define this value as 20% of the number of test cases. The last step is “completed”. At this stage, the reduced test case is inserted into a file with a text extension (.txt). The output is the selected test case number. Therefore, steps are needed to return the chosen test case into the same format as the input test suite. |

3.3 Greedy Algorithm for Test Case Prioritization Technique

The Greedy algorithm works on the principle that the test case with the highest weight will first take, and follows with the second weight, and so on. The weight is calculated based on the number of faults that the test case detects.

Before ordering the test-cases, it is necessary to know the number of faults that each test-case can detects. It aims to provide the weight on each test-case. Table 2 illustrates the ordered test-cases in the Greedy algorithm, and the faults detected.

Table 2. The Example of the Test Case and Fault Detected

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Fault  Test Cases | fault1 | fault2 | fault3 | fault4 | fault5 | fault6 | fault7 |
| tase-case-1 | X |  |  |  |  |  |  |
| tase-case-2 |  | X | X | X | X |  |  |
| tase-case-3 | X | X | X |  |  |  |  |
| tase-case-4 |  |  |  |  | X | X | X |
| tase-case-5 | X | X |  |  |  |  |  |
| tase-case-6 |  |  |  | X | X | X | X |
| tase-case-7 |  | X | X | X | X | X | X |

Test-case1 detects faults1, test-cases2 detects fault2, fault3, fault4 and fault5; test-case3 detect faults1, fault2, and fault3, and so on, up to the test-case7. Next, the test case and fault are converted into a two-dimensional array so that it can be sorted using the greedy algorithm (Table 3). While Table 4 shows the weight of each test case.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Table 3 Array of *Test Case* and *Fault*   |  |  | | --- | --- | | Test Case | Fault | | tase-case-1 | {f1} | | tase-case-2 | {f2, f3, f4, f5} | | tase-case-3 | {f1, f2, f3} | | tase-case-4 | {f5, f6, f7} | | tase-case-5 | {f1, f2} | | tase-case-6 | {f4, f5, f6, f7} | | tase-case-7 | {f2, f3, f4, f5, f6, f7} |   *Note: fi: faulti* | |  |  | | --- | --- | | Table 4 The Test Cases Weight | | | Test Case | Weight | | tase-case-1 | 1 | | tase-case-2 | 4 | | tase-case-3 | 3 | | tase-case-4 | 3 | | tase-case-5 | 2 | | tase-case-6 | 4 | | tase-case-7 | 6 | |

The Greedy algorithm will sort the test cases with the descending ordering. Therefore, the results of the test case sequencing are test-case7, test-case2, test-case6, test-case3, test-case4, test-case5, and test-case1.

3.4 Effectiveness Measure

According to the method explain above, the RTS and RTM techniques use the execution time metric to see its effectiveness, while TCP technique uses an APFD metric.

3.4.1 RTS dan RTM Effectiveness

We measure the effectiveness of RTS and TRM technique utilising the execution time. In this context, the formulation shows in formula 1. The empirical study results on four programs for RTS technique shows in Table5.

Effectiveness (RTS/RTM) = (ExecutionTime(RetestAll) - ExecutionTime(RTS/RTM))/ ExecutionTime(RetestAll)  X 100%. (1)

Table 5. RTS Empirical Results

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| No | SUT | Execution Time (second) | | Effectiveness (%) |
| Retest-All | BabelRTS |
| 1 | OpenMRS | 51.414 | 45.467 | 12% |
| 2 | NumberToWord | 3.573 | 2.432 | 32% |
| 3 | Palindrome | 3.644 | 2.518 | 31% |
| 4 | JodaTime | 24.973 | 19.803 | 21% |
| Average | | | | 23% |

In the empirical study of the *R*TM technique, we used ten iterations. Since print\_token has multiple test cases, and for one subtraction, it generates about 200 test cases, it has variable execution times. We decided to execute in 10 iterations to find 2000 test cases, which is twice the test cases in print\_token. *Table 6 shows the result of the RTM technique*.

Table 6. RTM Empirical Results

|  |  |  |  |
| --- | --- | --- | --- |
| Iteration | Retest All (second) | RTM (second) | Effectiveness (%) |
| 1 | 0,242 | 0,01 | 96% |
| 2 | 0,227 | 0,111 | 51% |
| 3 | 0,23 | 0,006 | 97% |
| 4 | 0,228 | 0,005 | 98% |
| 5 | 0,237 | 0,008 | 97% |
| 6 | 0,253 | 0,009 | 96% |
| 7 | 0,231 | 0,007 | 97% |
| 8 | 0,24 | 0,007 | 97% |
| 9 | 0,248 | 0,008 | 97% |
| 10 | 0,23 | 0,008 | 97% |

3.4.2 TCP Effectiveness

To measure the effectiveness of TCP, most researchers use average percentage fault detection (APFD). The APFD calculation is to take the average percentage of errors detected during the implementation of the test suite. The APFD value ranges from 0 to 100; a higher value implies a faster (better) error detection rate. Formula 2 is an APFD formulation [17].

|  |  |
| --- | --- |
|  | (2) |

**Notes**:

*T is a test-cases*

*n is a number of test cases,*

*m is the number of faults,*

*(Tf ..Tfm)is the first test position T which a fault found.*

We use print\_token2 as SUT to execute the TCP study. This program has nine faults and 1100 test cases. The result of without ordering is 82,5%, and the utilize Greedy algorithm has 99.712% APFD value. This value is considered quite useful in terms of effectiveness.

On the other view, utilizing an APFD is more effective in cases where one test suite has one test case that may detect several faults. The overall idea behind APFD is to count the number of test cases (or parts of a test suite) run in a particular order and the number of fault data variations (fault unique) detected by test case execution. The maximum APFD is when the first test case detects all faults, and the minimum APFD is when the n-1 test case detects no defects.

1. **Conclusion**

According to this empirical study, regression testing techniques can significantly reduce the implementation time for the testing process. This situation will substantially support the software development process, which currently applies mostly iterative-incremental methods. On the other hand, this study still has limitations, because it only explores one approach in each technique, and only uses one metric to effectiveness measure.

However, empirical studies of this kind are useful for understanding existing techniques and developing new ones, with a broader approach and a wider variety of effectiveness metrics. Based on the existing approach, the opportunity to develop new techniques is still wide open.

1. **References**

[1] Chen Y, Probert R L and Sims D P 2002 Specification-based regression test selection with risk analysis *Proc. 2002 Conf. Cent. Adv. Stud. Collab. Res.* 1

[2] Bhandari P and Singh A 2017 Review of object-oriented coupling based test case selection in model based testing *Proc. 2017 Int. Conf. Intell. Comput. Control Syst. ICICCS 2017* **2018**-**January** 1161–5

[3] Banias O 2019 Test case selection-prioritization approach based on memoization dynamic programming algorithm *Inf. Softw. Technol.* **115** 119–30

[4] Cruciani E, Miranda B, Verdecchia R and Bertolino A 2019 Scalable Approaches for Test Suite Reduction *Proc. - Int. Conf. Softw. Eng.* **2019**-**May** 419–29

[5] Khan S U R, Lee S P, Ahmad R W, Akhunzada A and Chang V 2016 A survey on Test Suite Reduction frameworks and tools *Int. J. Inf. Manage.* **36** 963–75

[6] Taneja D, Singh R, Singh A and Malik H 2020 A Novel technique for test case minimization in object oriented testing *Procedia Comput. Sci.* **167** 2221–8

[7] Luo Q, Moran K, Zhang L and Poshyvanyk D 2019 How Do Static and Dynamic Test Case Prioritization Techniques Perform on Modern Software Systems? An Extensive Study on GitHub Projects *IEEE Trans. Softw. Eng.* **45** 1054–80

[8] Shin D, Yoo S, Papadakis M and Bae D H 2019 Empirical evaluation of mutation-based test case prioritization techniques *Softw. Test. Verif. Reliab.* **29** 1–28

[9] Anderson J, Azizi M, Salem S and Do H 2019 On the use of usage patterns from telemetry data for test case prioritization *Inf. Softw. Technol.* **113** 110–30

[10] Do H 2016 Recent Advances in Regression Testing Techniques *Adv. Comput.* **103** 53–77

[11] Anderson J, Azizi M, Salem S and Do H 2019 On the use of usage patterns from telemetry data for test case prioritization *Inf. Softw. Technol.* **113** 110–30

[12] Rosero R H, Gómez O S and Rodríguez G 2016 15 Years of Software Regression Testing Techniques - A Survey *Int. J. Softw. Eng. Knowl. Eng.* **26** 675–89

[13] Maurina BabelRTS, ﻿https://github.com/ICSE19-FAST-R/FAST-R, Accessed Okt-2020

[14] Anon Fast-R, ﻿https://github.com/ICSE19-FAST-R/FAST-R. Accessed Okt-2020

[15] Li Z, Harman M and Hierons R M 2007 Search algorithms for regression test case prioritization *IEEE Trans. Softw. Eng.* **33** 225–37

[16] Anon National Science Foundation, ﻿https://sir.csc.ncsu.edu/portal/index.php, Accessed Okt-2020

[17] Elbaum S, Malishevsky A G and Rothermel G 2002 Test case prioritization: A family of empirical studies *IEEE Trans. Softw. Eng.* **28** 159–82