STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXV**, Number 2, 2020
DOI: 10.24193/subbi.2020.2.06

DEFECT PREDICTION-BASED TEST CASE PRIORITIZATION

CRISTINA MARIA TIUTIN, MARC-TITUS TRIFAN, AND ANDREEA VESCAN

Abstract. Changes in the software necessitate confirmation testing and regression testing to be applied since new errors may be introduced with the modification. Test case prioritization is one method that could be applied to optimize which test cases should be executed first, involving how to schedule them in a certain order that detect faults as soon as possible.

The main aim of our paper is to propose a test case prioritization technique by considering defect prediction as a criteria for prioritization in addition to the standard approach which considers the number of discovered faults. We have performed several experiments, considering only faults and the defect prediction values for each class. We compare our approach with random test case execution (for a theoretical example) and with the fault-based approach (for the Mockito project). The results are encouraging, for several class changes we obtained better results with our proposed hybrid approach.

1. Introduction

In order to establish if a delivered software is reliable, appropriate verification practices have to be performed. A valuable and proper technique in this sense is provided by testing. In spite of its significant advantages, testing is, most of the time, a particularly expensive and demanding activity. The growth in software complexity is reflected in an exponential manner towards the cost of testing. As software evolves, the number of tests needed to preserve correct functionality increases as well, leading to a proportionate extension in the time

Received by the editors: 22 May 2020.
2010 *Mathematics Subject Classification.* 68M15,6804,68N30.
1998 *CR Categories and Descriptors.* D.2.8 **[Software engineering]**: Metrics – *Complexity measures* C.4 **[Performance of systems]** – *Performance attributes* D.2.4 **[Software engineering]**: Software/Program Verification – *Reliability* D.2.5 **Software engineering**: Testing and Debugging – *Testing tools* .
Key words and phrases. Test Case Prioritization, Regression Testing, Defect Prediction, Average Percentage of Faults Detected (APFD).

taken to execute the test suite. Thus, a solution which manages to lower the cost, while also preserving the verification quality is required.

The source code quality is a characteristic of any software project. The constant variation of this metric should be kept within certain bounds to ensure the proper functionality of an application, while also not neglecting its general cost. The quality of the code involves taking into consideration many factors, like the number of programmers writing the code or the change frequency as the entire bug-fixing process is performed. Increased consideration must be given to this process, as a module in which a found bug is fixed may be prone to future failures. Considering the overall quality of the source code (which will be later reflected in the quality of the product), a specific amount of attention should be involved in the bug fixes. One of the methods that help with this aspect is regression testing, through the various existing approaches.

Regression Testing (RT) [7] is "the process of validating modified software to detect whether new errors have been introduced into previously tested code and to provide confidence that modifications are correct". Similar other definitions may be found in [4] as "the retesting of the software that occurs when changes are made to ensure that the new version of the software has retained the capabilities of the old version and that no new defects have been introduced due to the changes", in [10] as "performing testing after making a functional improvement or repair to the program", and in [20] as "rechecking test cases passed by previous production versions".

Regression testing [1] is an essential part of any viable software development process and in practice is often incorporated into a continuous integration service. It is well known that if the regression tests do not finish in a timely manner, the development process is disrupted.

The challenging problem regarding regression testing refers to the fact that at a given point during the development, the test suites are so large that running all the test cases would take too much time. There are many approaches that investigate the regression testing problem, from test suite minimization to test case selection and test case prioritization. A review on various RT techniques is presented in paper [3]. A more detailed view about subtypes of RT is presented in [14].

The Test Case Prioritization (TCP) [3] technique helps to increase the rate of fault detection. It also increases in practice the effectiveness of test suites. To evaluate the regression-based test suite prioritization, the ordering is in general measured using the APFD (Average Percentage Faults Detected) metric [12]. More information regarding APFD metric is provided in Section 2.2.

Test Case Prioritization has been widely researched as a strategy for reducing the time needed to discover regressions in software. While many different

approaches have been developed and evaluated, previous experiments have focused on faults or code coverage used as prioritization techniques, and only few approaches [11] considered the defect prediction probability of a class. The probability associated to a class may determine the likelihood of it to contain software bugs. This metric could be used as a test case prioritization technique. We propose an investigation in this direction, i.e. test case prioritisation based on defect prediction.

The aim of this paper is to propose and investigate an algorithm that considers both the faults and the defect prediction probability as criteria for prioritization. We employ the use of defect prediction probability of a class as described in [11] where the aim was to study how to configure the Schwa tool [5] to maximize the likelihood of an accurate prediction. More information regarding the computation of the defect prediction probability of a class is provided in Section 2.2.

The reminder of the paper is organized as follows: Section 2 presents related work regarding the test case prioritization and also referring to defect prediction, Section 3 describes our approach regarding Test Case Prioritization that considers also the defect prediction probability. The experiments in Section 4 revealed that our approach finds better solutions for several class changes, and the last section outlines the concluding remarks and future work.

2. Background on Test Case Prioritization and Defect Prediction

Regression testing [7] is an important process that mostly every software systems will go through multiple time during its development and maintenance process. Being highly time consuming, performance improvements are a must, thus several studies are currently proposed to minimize the cost of the process and maximize its efficiency.

2.1. Definition of Test Case Prioritization.

Definition for TCP is given by Graves [7] in what follows.

Definition 1. ***Test case prioritization*** [7]*: Given a test suite, T, the set of permutations of T, PT; a function from PT to real numbers, f.*

Problem: to find $T \in PT$ such that

$$(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')] \tag{1}$$

The function f assigns a real value to a permutation of T according to the test adequacy of the particular permutation.

The ideal order would be the one that reveals faults soonest, the rate of which can be expressed in APFD (Average Percentage Faults Detected) metric

(described in the next subsection). The latest regression testing approaches use various code coverage criteria since fault detection is not known in advance. Several approaches are scrutinized by Rothermel et al. [15] using various coverage measurements, showing that coverage prioritisation can improve the rate of fault detection [19].

In what follows we present a short overview of various test case prioritization techniques and also studies related to defect prediction.

2.2. **Test case prioritization approaches.** This section details the APFD and defect probability metrics, alsor presenting various existing approaches for test case prioritization.

Average Percentage Faults Detected

APFD measures *the effectiveness of TCP as the rate of fault detection* achieved by the produced ordering of test cases [16]. APFD relates with earlier fault detection abilities.

Definition 2. *Average Percentage Faults Detected (APFD)* [12] *is defined as follows:*

$$APFD = 1 - \frac{Tf1 + Tf2 + ... + Tfm}{mn} + \frac{1}{2n} \tag{2}$$

Where n be the no. of test cases and m be the no. of faults. (Tf1,...,Tfm) are the position of first test T that exposes the fault.

Paterson et. al [11] proposes a test case prioritisation technique considering defect prediction, a strategy which analyses code features in order to predict the likelihood that a file or function inside a software system is faulty. The paper introduces a test case prioritisation approach, namely G-clef, that uses bug prediction data to reorder a test suite in such a way that it concentrates first on the classes that are prone to include faults. The paper not only presents this new test case prioritization strategy, but also compares the approach with other nine existing approaches using an empirical study on real faults.

Defect prediction probability of a class

Schwa [5] uses a ranked-based technique, Time-Weighted Risk (TWR) to estimate how reliable a Java class is, thus the function has its maximum value when a component was changed recently:

$$twr(t_i) = \frac{1}{1 + e^{-12t_i + w}} \tag{3}$$

Schwa [5] estimates the likelihood that a Java class c contains a bug using Equation 4, in which each of the three factors (i.e., revisions, authors, and fixes) is calculated and modified by a weight, where the sum of all weights

must be equal to 1. For each component the score is computed as provided in Equation 4. Intuitively, a Java class with higher defect value is less reliable, thus more likely to contain a bug, than those classes with low defect value.

$$\begin{aligned} score = revisions * revisions_{weight} \\ + fixes * fixes_{weight} \\ + authors * authors_{weight} \end{aligned} \tag{4}$$

Test case prioritisation approaches

Test Case Prioritisation (TCP) arranges test cases into an optimal order so that a specific criteria is met as early as possible. The work of [13] presents a black-box strategy called REMAP that incorporates three fundamental components: a rule miner, a static prioritiser, and a dynamic executor and prioritiser. The relations between test cases are defined using fails and pass rules being mined from the historical execution data. Multi-objective search is applied to statically prioritise test cases considering two objectives: fault detection capability and test case reliance score. The test case order is dynamically updated using the results of the test case execution and the fail and pass rules.

2.3. **Defect prediction approaches.** This section contains various existing approaches related to defect prediction approaches.

When discussing a defect prediction solution, one approach taken into consideration is a multi-objective approach [2], the two conflicting objectives being the cost and effectiveness. The approach allows software engineers to choose between various predictors: predictors that identify a high number of defect-prone artifacts, predictors requiring a lower cost, and predictors achieving a cost-effectiveness compromise.

In paper [9], the search problem was enhanced, such that the defect prone classes are predicted using the Object-Oriented metrics design suite instead of static code metrics. An extensive comparison of eighteen machine learning techniques in the context of defect prediction was performed. Six releases of widely used Android application package were used.

A qualitative and quantitative study regarding defect prediction was done [17] to investigate what practitioners consider and expect in contrast to research findings. The study that was done through interviews and questionnaires. The results revealed that most respondents are willing to adopt defect prediction techniques, but there is a discrepancy between practitioners' perceptions and supported research evidence regarding defect density distribution. Also, the most preferred level of granularity of defect prediction by practitioners is at the feature level.

However, as mentioned before, there exists a gap regarding test case prioritization and defect prediction probability of classes. The approach we propose in this work addresses an empirical attempt of the method to consider defect prediction probability of a class as well.

3. Our Defect Prediction - based Test Case Prioritization

Our approach investigates the test case prioritization problem considering various criteria (faults and defect prediction probability) using a Greedy-based approach in order to select the test case ordering. We compare our approach that uses the defect probability with two other approaches: one approach that only considers the faults found by the test cases and the other that is a random execution of the test suite.

As we mentioned before, our approach is based on greedy strategy. We have tested 3 versions of greedy algorithm in order to obtain the ordering which is expected to retrieve the best results, as graphically depicted in Figure 1: a random prioritization, a faults-based prioritization order, and also an approach based on defect prediction.

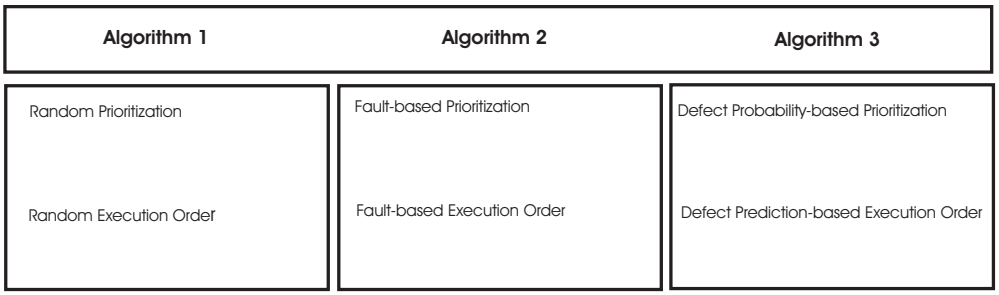

Figure 1. Overview of the three investigated Test Case Prioritization approaches.

We compare our approach with the basic Random Prioritization approach that is described in Algorithm 1. The approach just randomly orders the test cases that are relevant for the changed c class and afterwards adds the remaining test cases from the test suite. We performed our expriments considering that each class c of the project p is changed.

The second approach in our investigation uses the number of faults discovered by each test case that is relevant for the changed class c. The description of this approach is provided in Algorithm 2.

Random Prioritization Algorithm
Data:
list of all classes in a project with their corresponding tests;
list of all tests with corresponding number of bugs discovered for each test.
Result:
prioritized list of tests (and APFD) for each class in the project.
foreach *class c in project p* **do**
    get the list of tests relevant for c;
    randomly order the tests;
    add to the resulted list all the tests that are not relevant for the class;
    compute APFD for the obtained prioritized list of tests;
end

Algorithm 1: Random Prioritization Algorithm

Faults-Based Prioritization Algorithm
Data:
list of all classes in a project with their corresponding tests;
list of all tests with corresponding number of bugs discovered for each test.
Result:
prioritized list of tests (and APFD) for each class in the project.
foreach *class c in project p* **do**
    get the list of tests relevant for the class c;
    sort list descending based on the number of bugs discovered by each test;
    add randomly to the resulted list all the tests that are not relevant for the class;
    compute APFD for the obtained prioritized list of tests;
end

Algorithm 2: Faults-Based Prioritization Algorithm

Our new proposed approach uses the defect prediction probability for each class. The tests are order based on the maximum defect probability among all test cases that are relevant to the changed class c. The description of this approach is provided in Algorithm 3.

Defect-Prediction Prioritization Algorithm
Data:
list of all classes in a project with their corresponding tests;
list of all tests with corresponding number of bugs discovered for each test.
Result:
prioritized list of tests (and APFD) for each class in the project.
foreach *class c in project p* **do**
  get the list of tests relevant for the class;
  foreach *test t in tests for class c* **do**
    get the list of all classes tested by the test t;
    access the maximum defect probability from all the classes;
  end
  sort list descending based on the maximum defect probability;
  add randomly to the resulted list all the tests that are not relevant for the class;
  compute APFD for the obtained prioritized list of tests;
end

Algorithm 3: Defect-Prediction Prioritization Algorithm

We will exemplify our approach by using a theoretical example provided in Table 1 and Table 2.

	t_1	t_2	t_3	t_4	t_5	t_6	defectProb
c_1	1	0	0	0	0	0	0.25
c_2	0	0	0	1	0	0	0.65
c_3	1	0	0	1	0	0	0.70
c_4	0	0	0	0	1	0	0.55
c_5	0	0	0	0	0	1	0.45
c_6	1	0	0	0	0	0	0.85
c_7	1	0	0	0	1	1	0.80
c_8	0	0	0	0	0	0	0
c_9	1	1	1	0	0	1	0.82
c_{10}	0	0	0	0	1	0	0.72

TABLE 1. Class to tests cases matrix

	f_1	f_2	f_3	f_4	f_5
t_1	1	0	0	0	0
t_2	0	0	0	1	0
t_3	0	0	0	0	0
t_4	1	0	0	0	1
t_5	0	1	0	0	0
t_6	0	0	1	0	0

TABLE 2. Test cases to bugs matrix

We consider the provided order of the test cases as

$$\text{TC}=[t_1, t_2, t_3, t_4, t_5, t_6].$$

As we mention earlier, test case prioritization is applied in the context of regression testing, thus when a change is done in the source code due to the fixing of a fault. In that follows next, we present each approach using this theoretical case study and considering that the changed class is c_3.

The random algorithm starts from an existing order of the tests (t_1, t_2, ..., t_n) and after generates a new random execution order of the test suite.

First, the tests corresponding for class c_3 are selected. Those tests are (t_1, t_4). The algorithm randomly chooses an order for the list and then appends the rest of the tests (that do not test the class c_3). A possible order of the tests may be TC3=[t_1, t_4, t_5, t_6, t_2, t_3].

The most used criteria regarding the test case prioritization is the one regarding the faults the test cases discover, thus ordering the test cases to be executed using the discovered number of faults. For the theoretical example this ordering is: C3=[t_4, t_1, t_5, t_6, t_2, t_3].

The tests t_1 and t_4 are ordered descending based on the number of bugs discovered. Test t_1 discovers 1 bug, while t_4 discovers 2 bugs, such that the final order of the tests may be TC3=[t_4, t_1, t_5, t_6, t_2, t_3], or any random order between the third and the last test.

Our proposed approach considers also the defect probability defined by Peterson et. all in paper [11].

The probability is defined as the likelihood that a Java class contains a bug. This estimation is computed based on three weights, corresponding to the importance given to revisions, fixes and authors, applied to the Time-Weighted Risk of a class, a metric which uses different features to estimate how reliable a Java class is. The sum of the weights must be equal to 1. For the proposed approach, the weights were defined as 0.25 for revisions, 0.5 for fixes and 0.25 for authors. The computation uses an upper limit for the number of commits taken into consideration when estimating the defect probability. Two different values were chosen as an upper limit, 50 and 1000 commits respectively.

Our defect prediction-based prioritization approach is performed in three major steps: the first one selects first the test cases that are related to the class that was changed and order them based on the number of discovered faults, the second one orders the remaining test cases from this set (if there are test cases that did not discover any faults) based on the defect probability value, and the last step adds randomly the test cases that did not take part of the changed class, thus completing the test suite with all the test cases.

Using the defect probability prioritization algorithm, the ordering of the tests t_1 and t_4 is determined by the maximum defect probability for the classes that are tested for each test. For test t_1, the classes verified by it are c_1, c_3, c_6, c_7 ,c_9, the maximum probability being 0.85, for c_6. In a similar matter,

for test t_4 the verified classes are c_2 and c_3, with a maximum probability of 0.70, corresponding to c_3. Based on this information, the test order will be t_4, followed by t_1. A possible test ordering for class c_3 is TC3=[t_4, t_1, t_5, t_6, t_2, t_3] (or the last 4 tests in any order).

4. Experiments and results

Choosing the method for our research investigation was based on the book of Yin [18] that supported us in identifying the case study method and revealed how to do the research design.

Generalization from a case study to theory is an important issue, Yin [18] stating that the analytic generalization should be used for case studies: multiple cases resemble multiple experiments, thus the mode of generalization being analytic. Replication [18] may be claimed if two or more cases support the same theory: some replications seek to duplicate the exact conditions of the original experiment, others change some experimental conditions.

4.1. **Experiments Design.** The replication strategy that we used considered various number of classes, test cases and faults, those numbers being changed from one experiment to the others. The first experiment considers the theoretical example with a small number of classes, test cases and faults. The next two experiments are based on a open source project and even if the same number of classes, test cases and faults are used, they have different number of classes with the defect probabilities greater than 0 as described next.

Experiment 1: 10 classes, 6 test cases, 5 faults.

Experiment 2: 365 classes, 116 test cases, 38 faults, 199 classes with defect probabilities > 0 (considering 50 commits).

Experiment 3: 365 classes, 116 test cases, 38 faults, 336 classes with defect probabilities >0 (considering 1000 commits).

It is worth mentioning that we performed the above experiments considering that each class was changed, thus applying test case prioritization for each scenario.

4.2. **Case studies used.** The experiments are based on a theoretical project and on an open project, Mockito from the Defect4J database [8]. We have used the defect probability defined by Peterson et. all in paper [11] as we mentioned earlier in Section 3.

The Defect4J database [8] contains a set of software projects that contain reproducible bugs. The fact that the projects were originally obtained from different version control systems allows the possibility of collecting information regarding the faults of the program from versioning perspectives. Each project

has associated a list of faults with the corresponding test that discover a particular fault. In addition, a list of classes that are verified during the execution of a particular test is also kept.

Those files represented the base for the data pre-processing. Two comma separated value files were created, representing different matrices useful for further computations. One of the matrices represented a binary bug-to-tests correspondence, where 1 denotes that a bug is discovered by a certain test. The form of a row is class name and k values of 1 or 0, where k is the number of tests. The second matrix represents the tests relevant for a particular class. A row-structure contains the class name and k values of 1 or 0, where k is the number of tests, where 1 represents if a particular test is relevant for the given class.

Another tool that was used for computing the defect probability for the classes of a certain project is Schwa [6]. The output contains a json file with the defect probability for each class, while also mentioning a defect probability value for each method that is part of a class. The information was processed in a csv file, which is a mapping between the class names and the defect probabilities obtained. Two separate files were obtained, one by taking into account the last 50 commits of the Mockito project, obtained from the Defect4J library, and other file made by analyzing the last 1000 commits of the same project.

The effectiveness of the proposed prioritization technique, thus the ordering of the test cases is assessed using the rate of faults detected using the APFD metric as described in the above sections.

4.3. **Experiment 1.** The first case study considers the theoretical example that we provided in Section 3. The case study contains: 5 classes, 7 test cases and 4 faults, along with the defect probabilities being provided. The class that was modified is c_3.

We consider the provided order of the test cases as

$$\text{TC}=[t_1, t_2, t_3, t_4, t_5, t_6].$$

The test cases that are involved in the c_3 class that is changed are: t1 and t4.

The execution of the random algorithm found the following result: [t1, t4, t3, t2, t6, t5]. The obtained APFD values is: 0.48.

The execution of the algorithm that considers only the number of discovered faults by the test cases found the following result: [t4, t1],[t5, t6, t2, t3]. The APFD value is: 0.62.

The execution of our proposed algorithm found the following result: [t1, t4],[t5, t6, t2, t3]. The APFD value is: 0.58.

For this example, the proposed algorithm does find better solution than the random approach (APFD is greater, $0.58 > 0.48$), but did not find better solution than the faults-based approach ($0.58 > 0.62$).

As mentioned earlier, we have also executed the algorithm considering that each class is changed, that performing test case prioritization for each class modification. The results in Figure 2 still do not find better solution for our approach. We executed the algorithm such that each class in the project is changed thus being required to apply test case prioritization in each scenario.

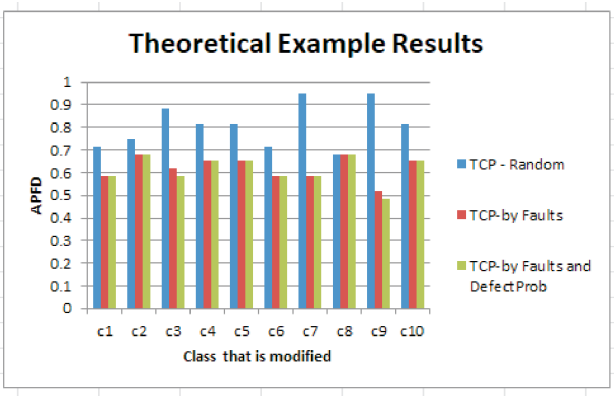

FIGURE 2. Theoretical example results

4.4. **Experiment 2.** Our next experiment considers the Mockito project from the Defect4J dataset [8].

The Mockito case study contains: 365 classes, 116 test cases and 38 faults, along with the defect probabilities being provided. The defect probabilities were computed using the last 50 available commits, which resulted in values for 190 classes. The rest of the classes had associated a defect probability of 0.

For example, for class "org.mockito.internal.invocation.serializablemockitomethod" the APFD values found for each approach are: TCP-Random APFD=0.72, for TCP-byFaults APFD=0.72, and for TCP-byFaultsAndDefectProb APFD=0.72.

Comparing our approach with the byFaults approach we obtained the results in Figure 3: for 26 classes the APFD values are higher. If we consider APFD highest or equal then for 132 number of classes we obtained better or equal APFD results.

4.5. **Experiment 3.** The last conducted experiment considered the same Mockito case study with different defect probabilities computed.

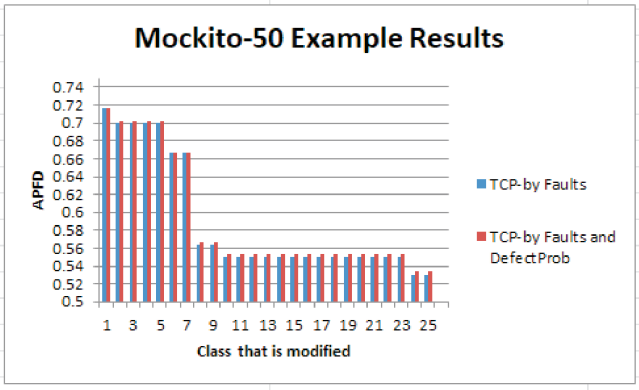

FIGURE 3. Mockito-50 Results: For out of 190 classes, we obtained better or equal APFD values for 132 classes, and obtained higher APFD values for 26 classes.

The Mockito case study contains: 365 classes, 116 test cases and 38 faults, along with the defect probabilities being provided. The defect probabilities were computed using the last 1000 available commits, which resulted in values for 336 classes. The rest of the classes had associated a defect probability of 0.

The results for this experiments revealed that for 114 classes we obtained higher or equal APFD when comparing TCP-byFaults approach with the TCP-byFaultsAndByDefectPred. Figure 4 contains these results.

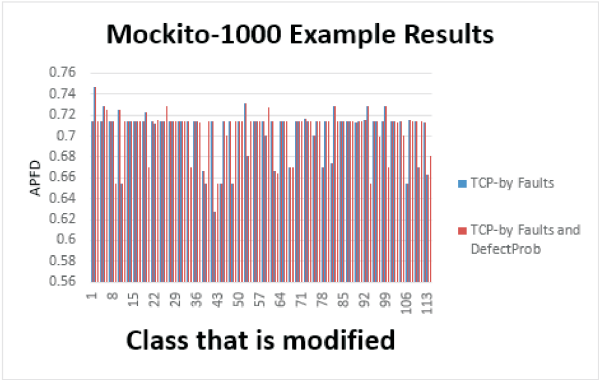

FIGURE 4. Mockito-1000 Results: For out of 336 classes, we obtained better or equal APFD values for 114 classes.

4.6. **Discussions.** Following the replication approach to multiple experiments [18], each individual experiment was finalized by an individual report (conclusion) that will be next considered to be part of a summary report, i.e. a cross-case conclusions.

In our case the results obtained for each experiment are reported in the above sections and in this section conclusions about the potential used of defect prediction probabilities for the test case prioritization problem are drawn: augmenting the standard test case prioritization criterion, i.e. number of faults discovered by the test cases, with the defect prediction probability of each class may lead to better results regarding which test cases should be first executed in the context of regression testing.

Our approach considered only one of the classes to be modified, thus computing the regression test suite only for this one modification. Future work will tackle these multiple changes in the classes.

5. Conclusions

Regression testing, with all existing strategies plays an important role in identifying and fixing faults after software changes are performed. Test case prioritization is one of the strategies that could be applied and that can provide important information about the system under test regarding the best test cases that may identify existing faults.

Our hybrid test case prioritization approach considers not only the number of faults discovered by the test cases but also the defect probability of each class.

The design of our experiments considered various combinations of number of classes, test cases, faults and classes with defect probability not zero. Our hybrid approach is compared with two other approaches: a random based approach and a faults-based approach. The results are encouraging, for several class changes we obtained better results with our proposed hybrid approach.

References

[1] Paul Ammann and Jeff Offutt. *Introduction to Software Testing,2nd edition*. Cambridge University Press, 2016.

[2] G. Canfora, A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella. Multi-objective cross-project defect prediction. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 252–261, 2013.

[3] Anjali Choudhary and Tarun Dalal. A review on regression testing techniques. *IJ of Emerging Trends and Technology in Computer Science*, 4(3):56–59, 2015.

[4] Jean-Francois Collard and Ilene Burnstein. *Practical Software Testing*. Springer-Verlag New York, Inc., 2002.

[5] A. Freitas. Software repository mining analytics to estimate software component reliability, Faculdade de Engenharia da Universidade do Porto, Master's thesis, 2015.

[6] Andre Freitas. *Software Repository Mining Analytics to Estimate Software Component Reliability*. PhD thesis, Faculdade de engenharia da Universidade do Porto, 6 2015.

[7] T. L. Graves, M. J. Harrold, J. Kim, A. Porters, and G. Rothermel. An empirical study of regression test selection techniques. In *Proceedings of the 20th International Conference on Software Engineering*, pages 188–197, 1998.

[8] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 437–440, New York, NY, USA, 2014. Association for Computing Machinery.

[9] Ruchika Malhotra and Rajeev Raje. An empirical comparison of machine learning techniques for software defect prediction. In *Proceedings of the 8th International Conference on Bioinspired Information and Communications Technologies*, BICT '14, page 320–327, Brussels, BEL, 2014. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[10] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[11] D. Paterson, J. Campos, R. Abreu, G. M. Kapfhammer, G. Fraser, and P. McMinn. An empirical study on the use of defect prediction for test case prioritization. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pages 346–357, 2019.

[12] R Pradeepa and K VimalDevi. Effectiveness of test case prioritization using apfd metric: Survey. In *International Conference on Research Trends in Computer Technologies (ICRTCT—2013). Proceedings published in International Journal of Computer Applications (IJCA), ISSN: 0975–8887*, pages 1–4, 2013.

[13] D. Pradhan, S. Wang, S. Ali, T. Yue, and M. Liaaen. Remap: Using rule mining and multi-objective search for dynamic test case prioritization. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 46–57, 2018.

[14] Mohammad Rava and Wan M.N. Wan-Kadir. A review on prioritization techniques in regression testing. *International Journal of Software Engineering and Its Applications*, 01(1):221–232, 2016.

[15] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization: An empirical study. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*, pages 179–188. IEEE, 1999.

[16] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization: an empirical study. In *Software Maintenance, 1999. (ICSM '99) Proceedings. IEEE International Conference on*, pages 179–188, 1999.

[17] Z. Wan, X. Xia, A. E. Hassan, D. Lo, J. Yin, and X. Yang. Perceptions, expectations, and challenges in defect prediction. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[18] Robert K. Yin. *Case Study Research: Design and Methods (Applied Social Research Methods)*. Sage Publications, fourth edition. edition, 2008.

[19] Shin Yoo and Mark Harman. Pareto efficient multi-objective test case selection. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, page 140–150, New York, NY, USA, 2007. Association for Computing Machinery.

[20] Michal Young and Mauro Pezze. *Software Testing and Analysis: Process, Principles and Techniques*. John Wiley and Sons, 2005.

Department of Computer Science, Faculty of Mathematics and Computer Science, Babeş-Bolyai University, 1 Kogălniceanu St., 400084 Cluj-Napoca, Romania

Email address: {tcie2430, tmie2434}@scs.ubbcluj.ro, avescan@cs.ubbcluj.ro