ISSN:2348-2079

Volume-5 Issue-2

International Journal of Intellectual Advancements and Research in Engineering Computations

Priority based test case generation for Multi object

Kavitha S[1] Department of CSE Sasurie college of engineering
Skavisivasamy18@gmail.com
Rajasekaran P[2] Assistan professorDepartment of CSE
rajasekaranp91@gmail.com

ABSTRACT

While performing regression testing, an appropriate choice for test case ordering allows the tester to early discover faults in source code. To this end, test case prioritization techniques can be used. Several existing test case prioritization techniques leave out the execution cost of test cases and exploit a single objective function (e.g., code or requirements coverage). In this paper, we present a multi-objective test case prioritization technique that determines the ordering of test cases that maximize the number of discovered faults that are both technical and business critical. In other words, our new technique aims at both early discovering faults and reducing the execution cost of test cases. To this end, we automatically recover links among software artifacts (i.e., requirements specifications, test cases, and source code) and apply a metric-based approach to automatically identify critical and fault-prone portions of software artifacts, thus becoming able to **importance during test case prioritization. We experimentally evaluated our technique on 21 Java applications. The obtained results support our hypotheses on efficiency and effectiveness of our new technique and on the use of automatic artifacts analysis and weighting in test case prioritization.**

1 INTRODUCTION

The history of software testing connected with the history of software engineering helps to understand how the practice of testing has evolved. Testing focused on hardware, program checkout and debugging were seen as one and the same. In the late 1950s, software testing was distinguished from debugging and became regarded as detecting the faults in the software . Software Testing is an essential stage in software development. Software testing is a process used to identify the

correctness, completeness, and quality of developed software. It includes a set of activities conducted with the intent of finding errors in software, so that it could be corrected before the product is released to the end users. Software testing is an activity to check whether the actual results match the expected results and to ensure that the software system is defect free.

2 RELATED WORK

To prioritize and select test cases a number of techniques have been proposed and empirically investigated [8], [9],[10], [11], [12], [13], [14], [15], [16]. Yoo et al. [1] and Mohanty et al. [3] survey existing research work in these fields. Results suggest that existing techniques mostly use either structural or functional coverage criteria with respect to source code executed by test cases. This is one of the aspects that makes our proposal different from those in the literature.A number of approaches use code coverage and additional code coverage1 to prioritize test cases with respect to their capability of executing the source code of software under test (e.g., [14], [17]). Most of these approaches identify test case orderings based on a single objective function (e.g., code coverage). Only a few approaches based on multi-objective optimization exist (e.g., [8], [18]). These approaches mainly consider code coverage information and execution cost of test cases: (i) optimize test cases by means of a Pareto front using both code coverage and execution cost or (ii) reduce a multi-objective problem to a single objective by using an optimization function code coverage and execution cost are explicitly considered when conducting test case selection. The approach can be also directly applied to test case prioritization. This work presents some similarities with that we present in this paper, namely the objective formulation takes into account source code coverage as a measure of test adequacy and execution time as a measure for cost. The most remarkable differences between these two approaches can be summarized as follows: we also consider the coverage of application requirements, to link them with source code we applied an IR technique, and we apply a metric-based approach to automatically identify critical and fault-prone portions of software artifacts (both source code and requirements).Another multi-objective test case prioritization approach is proposed by Sun et al. [19] for ordering test cases in GUI-based

applications. In fact, code (statement) coverage is traditionally used to test case prioritize, while event coverage criteria are largely adopted for GUI applications testing [20]. Hence, Sun et al. propose a multi-objective test case prioritization approach that exploits both criteria: statement and event coverage.

3 TRACEABILITY RECOVERY

Requirements traceability regards the documentation of bidirectional links among various related requirements and associated software artifacts produced in the entire development process. In other words, requirements traceability refers to the ability to describe and follow the life of a requirement, from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases [29]. This allows a software engineer to understand relationships that exist within and across different kinds of software artifacts. For example, documentation of traceability links might be crucial to be aware about: (i) source code in charge of implementing a given application requirement; (ii) requirements implemented by a specific part of the source code; and (iii) source code exercised by a test case. Traceability links are very often not documented at all and if this information exists it might be not updated or not aligned with the current implementation and documentation (e.g., [30], [31], [32], [33]). Therefore, methods and selection of an appropriate value for k is an open issue. A value for k should be large enough to fit the real structure of text, but small enough so that we do not also fit the sampling error or unimportant details.

3.1 Latent Semantic Indexing

LSI assumes that there is some underlying or latent structure in word usage that is partially obscured by variability in word choice, and uses statistical techniques to estimate this latent structure. LSI uses information about co-occurrence of terms (latent structure) to automatically discover synonymy between two or more terms. The latent structure of the content is obtained by

applying a Singular→ı Value Decomposition (SVD) to a m n matrix C (also named term-by-document matrix), where m is the number of terms and n is the number of documents (artifacts in our case). By applying SVD, each term and each artifact could be represented by a vector in the k space (i.e., the dimensionality reduction of the latent structure) of underlying concepts.Indeed, we use SVD to construct a low-rank approximation Ck to the term-document matrix, for a value of k that is far smaller than original rank of C.Thus, we map each row/column to a k−dimensional space, which is defined by k principal eigenvectors (corresponding to the largest

eigenvalues) of CCT →and CTC. The matrix Ck is itself still an m n matrix, irrespective of k.

3.2 IR-Based Traceability Recovery

In a typical text retrieval problem, a software engineer writes a textual query and retrieves documents that are similar to that query. In IR-based traceability recovery a set of source artifacts (used as the query) are compared with set of target artifacts (even overlapping). Hence, the number of queries is equal to the number of source artifacts. To compute similarities between vectors, we use the new k-dimensional space as we did the original representation. Similarity between vectors can be computed by different measures (e.g., Euclidean distance) [41]. In traceability recovery, the widely used measure is cosine similarity [36] between each pair of source and target software artifacts. The larger the cosine similarity value, the more similar the source artifact to the target one is. Source artifacts are normalized in the same way as target ones (i.e., the corpus). Different set of techniques could be used (e.g., stop word removal and/or stemming). In our case, normalization is performed by removing non-textual tokens, splitting terms composed of two or more words, and eliminating all the terms from a stop word list and with a length less than three characters. Finally, a Porter stemmer [41] is applied on lexemes to reduce them to their root form. All possible pairs (candidate traceability links) are reported in a ranked list. Irrelevant pairs of artifacts can be removed using a threshold that selects only a subset of top links, i.e., retrieved links. Well known strategies for threshold selection are [36]: Constant Threshold, a constant threshold is chosen; Scale Threshold, a threshold is computed as percentage of best similarity value between two vectors; Variable Threshold, all links among those candidate are retrieved links whether their similarity values are in a fixed interval. In this work, we use the Constant Threshold strategy to limit possibility of loosing links by considering a large number of link candidates. IR-based traceability recovery approaches retrieve also links between source code and target artifacts that do not coincide with correct ones: some are correct and others not. This is why these approaches are semi-automatic and requirehuman intervention to remove erroneously recovered traceability links. To reduce possible biases in test case prioritization results due to human factors/decisions, we do not perform any further analysis to remove erroneously recovered traceability links. It is worth

mentioning that a traceability recovery process could be executed (e.g., in background) every time a tester want or requirements and/or source code are modified in accordance to maintenance tasks. In our case, this choice reduces the impact of the overhead computational cost for the recovery of traceability links on the execution of our test case prioritization approach.

4 RELEVANT CODE AND REQUIREMENTS

In the following, we present metrics and algorithms proposed to identify portions of application code and requirements that are potentially critical and fault-prone.

4.1 Metrics

Code. Fault detection capability of a test suite cannot be known before executing test cases. Therefore, we have to resort to potential fault detection capability of a test suite. It can be estimated considering the amount of code covered by test cases in a test suite at run-time [12]. A test case that covers a larger set of code statements has a higher potential fault detection capability (i.e., potentially more faults should be revealed) than one test case that covers a smaller set of statements. We define CCov(t) as the amount of code statements exercised during the execution of a given JUnit test case t. A variant of this code coverage measure is WCCov(t). For a given test case, it is defined as a weighted source code coverage measure in which the coverage of source code. the set of source code statements. CodeCovered is the set of statements covered by the execution of the test case t, while s is a code statement of an application and ws ($0 < ws < 1$) is a predefined weight associated to each code statement. The higher the ws value, the greater the relevance a tester gives to statements is. In our previous work [4], we left the tester to manually specify such a weight for different parts (e.g., Java classes and packages) of code. In fact, this weight ws is expected to be useful to customize the measurement of code coverage according to testing needs. For example, a class implementing a critical service for an application needs to be tested more than other classes. In our approach, we exploit a metric-based approach to automatically identify such a weight for each Java class of the application under test by considering code characteristics. Code metrics allow ordering application classes according to their estimated fault-proneness when computing artifact coverage. Given a test suite S and an ordering OrdS for test cases

in this suite:cum

$$CCov(ti) = CCov(tj)$$

where ti is a test case in the suite. The cumulative code coverage for ti is computed by summing single code coverage (i.e., the code covered only by the test case) of all those test cases from t0 to ti−1. Requirements.

4.2 Automatic Weighting

Our metric-based approach automatically weights both code ws and requirements wr of the application under test. In particular, we apply code metrics to measure a Maintainability Index for each Java class (MIclass). This index estimates the fault-proneness of each class. We use such an estimation for defining an order of the application classes. To prioritize all the requirements according to how they are implemented, we also compute a Maintainability Index for each of these requirement (MIreq).

Our automatic weighting approach is composed of the following steps:

1) Recovering traceability links. Links among software artifacts (i.e., source code and requirements) are recovered by applying LSI;

2) Computing metrics. For each class, we measure a set of metrics such as: size, complexity, coupling, and cohesion. For each requirement, a set of metrics is also computed to measure properties characterizing requirements: size, complexity, coupling, cohesion, scattering, and tangling degree.

3) Estimating maintainability indexes. The computed metrics are used in a software quality model to compute the maintainability index for each class and each requirement based on their actual implementation in the source code. Classes and requirements are ordered by ranking according to their maintainability index.

4.3 Identifying Cumulative Test Orderings

For each test case ti of a given test ordering OrdS, the measures cumCCov(ti), cumRCov(ti) and InverseCost(ti) are computed considering the position of ti in OrdS. Then, we computed the area of the curves obtained by plotting the values of the metric (on X axes) with respect to the test cases in OrdS (Y axes) in a Cartesian plan.To get a numerical approximation of that area, we used the Trapezoidal rule [49]. It computes the area of a curve as the area of a linear function that approximates that curve. For OrdS and each cumulative measure, the area (Area Under the Curve, AUC, from here on) estimates the code

coverage AUCcode(OrdS), the requirements coverage AUCreq(OrdS), and the execution cost AUCcost(OrdS), respectively. The area indicates how fast the test ordering OrdS converges. The larger AUC, the faster this test case ordering converges.

5 A PROPOSED SYSTEM FOR MULTI OBJECT PRIORITATION

NSGA-II uses a set of genetic operators (i.e., crossover,mutation, and selection) to iteratively evolve an initial population of candidate solutions. In our case, candidate solutions are test cases orderings. Evolution is guided by an objective function (i.e., the fitness function) that evaluates each candidate solution along considered dimensions. In each iteration, the Pareto front of best alternative solutions is generated from evolved population. The front contains the set of non-dominated solutions, i.e., those solutions that are not inferior (dominated) to any other solution in all considered dimensions. Population evolution is iterated until a (predefined) maximum number of iterations is reached. In our case, a Pareto front represents the optimal trade-off between the three dimensions determined by NSGA-II. The tester can then inspect a Pareto front to find the best compromise between having a test case ordering that balances code coverage, requirements coverage, and execution cost or alternatively having a test case ordering that maximizes one/two dimension/s penalizing the remaining one/s.

5.1 PRIORITIZATION

We improve the solution highlighted before by leveraging the capability of automatically identifying fault-prone portions of software artifacts, according to some characteristics of the source code of a given application (e.g., McCabe Cyclomatic Complexity) and its requirements (e.g., the number of classes that implement a requirement). Summarizing our approach provides the following new research contributions: (i) a novel multi-objective test case prioritization technique; (ii) the definition of a metric-based approach to automatically identify potential critical and fault-prone portions of application code and requirements; and (iii) a large experimental evaluation. we propose in our work the use of a meta heuristic algorithm to prioritize test cases according to the three considered dimensions.

5.2 Particle Swarm Optimization (PSO)

In comparison with genetic search, the particle swarm optimization is a relatively recent optimization technique of the swarm intelligence paradigm. It was first introduced in 1995 by Kennedy and Eberhart [10, 2]. Inspired by social metaphors of behavior and swarm theory, simple methods were developed for efficiently optimizing non-linear mathematical functions. PSO simulates swarms such as herds of animals, flocks of birds or schools of fish. Similar to genetic search, the system is initialized with a population of random solutions, called particles. Each particle maintains its own current position, its present velocity and its personal best position explored so far. The swarm is also aware of the global best position achieved by all its members. The iterative appliance of update rules leads to a stochastic manipulation of velocities and flying courses. During the process of optimization the particles explore the D-dimensional space, whereas their trajectories can probably depend both on their personal experiences, on those of their neighbors and the whole swarm, respectively. This leads to further explorations of regions that turned out to be profitable. The best previous position of particle i is denoted by pbesti, the best previous position of the entire population is called gbest. Figure 2 shows the general workflow of a PSO-algorithm. The termination criterion can be either a specific fitness value, the achievement of a maximum number of iterations or the general convergence of the swarm itself. Since its first presentation, many improvements and extensions have been worked out to improve the algorithm in various ways and have provided promising results for the optimization.

A. Using PSO to Complete the Construction of a Single Test In this subsection, we will show how to use PSO to complete the construction of a single test. During the process of our two different algorithms, they often generate some tests. In these tests, some factors are fixed to specific levels, while the other factors remain free to take any possible valid level. Here we use PSO to choose appropriate valid level for these unfixed factors with the aim of covering more new combinations. In Algorithm 1, we give a pseudo-code to complete the construction of a single test. The input of this algorithm is a factor set F, the particle number m in the swarm and a test with some factors fixed. The output of this algorithm is the test with all factors fixed. To choose the appropriate level of these free factors, we

use PSO. Here we make such a restriction that the manipulation over particles only updates the value of free factors. In Algorithm 1, the termination criteria adopted in this paper is that when iteration exceeds a maximum iteration number NCmax, the iteration process will terminate (in Line 7). The iteration process will terminate in advance when the fitness value of the particle reaches the maximum value (in Lines 14-16). Construction of a Single Test is as shown in Algorithm1.

B. Two PSO Based Approaches Based on Algorithm 1 used to complete the construction of a single test, in this paper we propose two different algorithms to guide the systematically generation process of pair wise test suites. One algorithm is based on one-test-at-a-time strategy. The other algorithm is based on IPO-like strategy.

1. Strategy A, One-test-at-a-time strategy: One-test-atatime strategy was firstly adopted by AETG approach and was further used by Bryce et al.. Using this strategy, it firstly generates an empty test suite TS, then generate a test t according to some strategies, remove the combinations covered by t and add t to the test suite TS. When all the combinations are covered, it terminates the loop and return combinatorial test suite TS. We modified this strategy in the single test generation phase. Here we randomly choose a pairwise combination from uncovered combination set Q and fix the corresponding factors according to the chosen M. LAKSHMI PRASAD, M. NIKHITHA, G. DIVYA, Y. KULASEKHAR REDDY International Journal of Scientific Engineering and Technology Research Volume.05, IssueNo.07, March-2016, Pages: 1358-1362 combination (in Lines 4-6). Then we will use Algorithm 1 to choose proper levels for other unfixed factors. The pseudo code is shown in the Algorithm 2 Strategy B, IPO-like strategy: IPO is firstly proposed by Tai and Yu [21]. Unlike one-test-at-a-time strategy, this strategy includes horizontal growth phase and vertical growth phase. But our approach has some difference compared to IPO. The pseudo-code is shown in Algorithm 3 From the description of these two approaches, we can find that all pairwise combinations are covered at least once by one test in the final generated test suite. This also demonstrates the correctness of our approach. The effectiveness and efficiency of our approach will be demonstrated results section.

6 CONCLUSION

We propose a multi-objective technique to identify test case orderings that are effective (in terms of capability in early discovering faults) and efficient (in terms of execution cost). To this end, our proposal takes into account the coverage of source code and application requirements and the cost to execute test cases. An IR-based traceability recovery approach has been applied to link software artifacts (i.e., requirements specifications) with source code and test cases. A test case ordering is then determined by using a multi-objective optimization, implemented in terms of NSGA-II. The proposed technique applies a metric-based approach to automatically identify critical and fault-prone portions of software artifacts, thus becoming able to give them more importance during test case prioritization. Our technique has been validated on 21 Java applications. The most important take-away result of our experimental evaluation is: our approach is able to identify test case orderings thatearly recover faults both technical and business relevant.

REFERENCES

[1] S. Yoo and M. Harman, “Regression testing minimization, selection and prioritization: a survey,” Software Testing, Verification and Reliability, vol. 22, no. 2, pp. 67–120, 2010.

[2] J. Karlsson and K. Ryan, “A cost-value approach for prioritizing requirements,” IEEE Software, vol. 14, pp. 67–74, 1997. [3] S. Mohanty, A. Acharya, and D. Mohapatra, “A survey on model based test case prioritization,” International Journal of Computer Science and Information Technologies, vol. 2, no. 3, pp. 1002 – 1040, 2011.

[4] M. Islam, A. Marchetto, A. Susi, and G. Scanniello, “A Multi- Objective Technique to Prioritize Test Cases Based on Latent Semantic Indexing,” in Proc. of European Conference on Software Maintenance and Reengineering. IEEE Computer Society, 2012, pp. 21–30.

[5] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman, “Indexing by Latent Semantic Analysis,” Journal of the American Society of Information Science, vol. 41, no. 6, pp. 391–407, 1990.

[6] A. Marcus and J. I. Maletic, “Recovering documentation-to-sourcecode traceability links using latent semantic indexing,” in Proc. Of International Conference on Software Engineering. IEEE Computer Society, 2003, pp. 125–137.

[7] A. De Lucia, M. Di Penta, and R. Oliveto, “Improving source code lexicon via traceability and information retrieval,” IEEE Transactions on Software Engineering, vol. 37, no. 2, pp. 205 –227, march-april 2011.

[8] Z. Li, M. Harman, and R. Hierons, “Search algorithms for regression test case prioritization,” IEEE Transactions on Software Engineering, vol. 33, no. 4, pp. 225 –237, april 2007.

[9] M. Salehie, S. Li, L. Tahvildari, R. Dara, S. Li, and M. Moore, “Prioritizing requirements-based regression test cases: A goal-driven practice,” in Proc. of European Conference on Software Maintenance and Reengineering. IEEE Computer Society, 2011, pp. 329 –332.

[1S0] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, “Timeaware test suite prioritization,” in Proc. of International Symposium on Software Testing and Analysis. ACM, 2006, pp. 1–12.

[11] S. Yoo and M. Harman, “Using hybrid algorithm for pareto efficient multi-objective test suite minimisation,” Journal of Systems and Software, pp. 689–701, 2009. 0098-5589 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/ publications/rights/index.html for more information.

[12] V. R. Basili and R. W. Selby, “Comparing the effectiveness of software testing strategies,” IEEE Transactions on Software Engineering, vol. 13, pp. 1278–1296, December 1987.

[13] S. Elbaum, A. G. Malishevsky, and G. Rothermel, “Test case prioritization: A family of empirical studies,” IEEE Transactions on Software Engineering, vol. 28, pp. 159–182, February 2002.

[14] G. Rothermel, R. Untch, C. Chu, and M. Harrold, “Test case prioritization: an empirical study,” in Proc. of International Conference on Software Maintenance. IEEE Computer Society, 1999, pp. 179 –188.

[15] D. Marijan, A. Gotlieb, and S. Sen, “Test case prioritization for continuous regression testing: An industrial case study,” in Proc. of International Conference on Software Maintenance. IEEE Computer Society, 2013, pp. 540–543.

[16] D. Di Nardo, N. Alshahwan, L. Briand, and Y. Labiche, “Coveragebased test caseprioritisation: An industrial case study,” in

Proc. of International Conference on Software Testing, Verification and Validation. IEEE Computer Society, 2013, pp. 302–311.

[17] L. Zhang, D. Hao, L. Zhang, G.

Rothermel, and H. Mei, “Bridging the gap between the total and additional test-case prioritization strategies,” in Proc. of International Conference on Software Engineering. IEEE Computer Society, 2013, pp. 192–201.

[18] S. Yoo and M. Harman, “Pareto efficient multi-objective test case selection,” in Proc. of International Symposium on Software testing and Analysis. ACM, 2007, pp. 140–150.

[19] W. Sun, Z. Gao,W. Yang, C. Fang, and Z. Chen, “Multi-objective test case prioritization for gui applications,” in Proc. of ACM Symposium on Applied Computing. ACM, 2013, pp. 1074–1079.

[20] A. M. Memon, M. L. Soffa, and M. E. Pollack, “Coverage criteria for gui testing,” SIGSOFT Softw. Eng. Notes, vol. 26, no. 5, pp. 256–267, Sep. 2001.

[21] R. Kavitha, V. Kavitha, and N. Kumar, “Requirement based test case prioritization,” in Proc. of International Conference on Communication Control and Computing Technologies, 2010, pp. 826–829.

[22] M. Arafeen and H. Do, “Test case prioritization using requirementsbased clustering,” in Proc. of International Conference on Software Testing, Verification and Validation, March 2013, pp. 312–321.

[23] C. Nguyen, A. Marchetto, and P. Tonella, “Test case prioritization for audit testing of evolving web services using information retrieval techniques,” in Proc. of International Conference on Web Services. IEEE Computer Society, 2011, pp. 636 – 643.

[24] C. Fang, Z. Chen, K. Wu, and Z. Zhao, “Similarity-based test case prioritization using ordered sequences of program entities,” Software Quality Journal, vol. 22, no. 2, pp. 335–361, 2014.

[25] OMG, “Unified modeling language (OMG UML) specification, version 2.3,” Object Management Group, Tech. Rep., May 2010.