DevOps Enhancement with Continuous Test Optimization

Dusica Marijan
Simula, Norway
dusica@simula.no

Sagar Sen
Simula, Norway
sagar@simula.no

***Abstract*—Growing evidence suggests the DevOps approach enables faster development and deployment, and easier maintenance of applications. Still, the efficiency of DevOps is constrained by long cycle times. This paper presents the approach for improving time-efficiency in DevOps, and in particular continuous integration testing, using continuous test optimization. The approach uses test redundancy analysis to discover test overlap with respect to feature interaction coverage, and based on detected redundancy to reduce the size of a test suite. Smaller-size test suites execute faster and enable shorter test cycles, which further enables shorter release cycles. The approach has been experimentally evaluated using an industrial case study, against three metrics: industry practice of test selection for continuous integration testing, retest-all approach, and random test selection. The results suggest that the proposed test redundancy detection and reduction efficiently reduces test cycles in CI compared to industry practice and retest-all approach, and improves fault-detection effectiveness compared to random test selection[1].**

***Index Terms*—DevOps, Continuous integration, Continuous integration testing, Test optimization, Test redundancy**

I. Introduction

DevOps is a growing development practice that promises to enable faster development and efficient deployment of applications without compromising on quality. The main principle underlaying DevOps is increased communication and collaboration between development, testing, and operations, which makes it possible to minimize the time between making a change and deploying the change into production. Further, this makes it possible to respond faster to new and rapidly changing requirements, and thus remain agile. However, increased communication and collaboration do not suffice in reaching this goal on their own. A key aspect is enabling an *efficient and optimized continuous integration practice*. Continuous integration is a software development model of frequent integration and testing of source code to detect defects early in development. This practice has been associated with benefits such as improved code quality, more frequent releases, improved development productivity, less-costly development, and easier code maintenance [2], [3], [4]. The major bottleneck however in making continuous integration efficient is long-running testing of code changes after integration. In particular, after a change is made to the code, a set of automated tests are run to verify the change. As changes in DevOps are made frequently, testing of changes before integration runs frequently, which amplifies the need for time-efficiency in testing. This implies that for each integration cycle test suites need to be optimized for short run-time and maximized fault-detection.

Existing techniques to improve the efficiency of software testing in continous integration include test selection and prioritization [7], [8], [9], which aim to find an optimal combination and order of tests for achieving faster test runs. However, techniques that optimize testing for low run-time often compromize the ability of tests to detect faults [11], [12], [13], [14]. We argue that for DevOps, this aspect becomes especially important. It is essential that testing in DevOps is time-efficient, at the same time as being able to detect the most critical faults. Another issue that creates challenges for efficient testing in DevOps is the size of test suites. In particular, iterative feature requests and frequent changes of requirements constantly drive a test suite size larger and larger, while its quality is not maintained simultaneously. When test suites grow in size, the containing test cases start overlapping, covering the same features (parts of functionality) multiple times in different test instances. This effect is known as test redundancy, and it negatively affects time-efficiency of testing in DevOps. Besides directly increasing test effort (if all related test cases are post for running), test redundancy also increases test maintenance costs.

In this paper we propose an approach for improving time-efficiency of DevOps by continuously optimizing long running test cycles based on test redundancy detection and reduction. Specifically, for each test cycle we analyze test overlap with respect to changes and feature interaction coverage in a test suite, and then detect and remove tests that do not contribute to increased unique feature interaction coverage. We validated the approach in one industrial case study, comparing with industry practice and with the state of the art techniques. The approach demonstrated the improvement in time-effectiveness and fault detection effectiveness of CI testing in DevOps compared to industry practice.

The paper is organized as follows. In Section II we provide background and related work, as well as some challenges of efficient testing in DevOps in the presence of test redundancy. In Section III, we describe our approach for improving time-efficiency of DevOps with continuous test optimization based on redundancy detection. In Section IV we give results of the experimental study evaluating the approach against industry

[1]DOI reference number: 10.18293/SEKE2018-168

practice. We draw the conclusion in Section V.

II. Background and Related Work

In the following we revisit the key concepts underlying our work on making DevOps practices more cost-effective. We also summarize some of the critical challenges for effective testing in continuous integration and DevOps.

A. DevOps

DevOps is defined as a set of software engineering practices that aim to build an agile relationship between development and operations. A key principle is constant communication and collaboration between development and operations, enabling benefits such as faster development and deployment of features into production, faster detection and correction of issues, and cost-effective running of dependable software with minimal risks. Existing approaches for improving testing in DevOps are still narrow. There is an approach reported for run-time monitoring and reporting to developers, referred to as the filling-the-gap tool, which enhances and automates the delivery of application performance information to the developer, with the goal of improving the quality of service or reducing maintenance cost [20]. However, more approaches addressing various other challenges in continuous integration testing for DevOps are missing, and such limited state of the art in this field gives even more motivation for our research.

B. Continuous Integration

Continuous integration (CI) is a practice deemed a key enabler for DevOps. CI is a technique that continuously integrates code changes from all team members, merges them with the mainline, and verifies the changes against regressed aspects of the modified code (for unintended effects) with automated tests. The CI practice prevents working on isolated branches for too long, which over time start diverging from each other, leading to high effort of integrating such multiple branches into the mainline. An important aspect of CI is enabling rapid automated regression testing of code changes, which will give quick feedback to developers on the correctness of their changes. Since CI runs frequently, if it takes long time, it introduces time-inefficiency in DevOps. To support rapid regression testing in CI, we explore the concept of test optimization. Similarly, to improve CI testing, one approach was proposed combining test selection in the pre-commit stage with test prioritization in the post-commit stage [7]. However, this approach does not investigate the effect of test redundancy on the time-effectiveness of CI testing, which is one key focus of our work.

C. Test Redundancy

Test redundancy can be defined with respect to coverage metrics, for example pairwise feature coverage, such that if two tests check interaction between the same pair of features, then one of these tests is redundant with respect to one another. Features represent smaller units of software-under-test functionality that are self-contained. Considering pairwise feature coverage in the example provided below, two tests $TestA$ and $TestB$ cover the identical feature set $\{b, c\}$. As the pairwise feature set covered by $TestA$ is a proper subset of the pairwise feature set covered by $TestB$, we say that $TestA$ is redundant with respect to $TestB$.

$$TestA = [b, c] \rightarrow \{b, c\}$$
$$TestB = [a, b, c] \rightarrow \{a, b\}, \{b, c\}, \{a, c\}$$

Test redundancy can be caused by multiple factors, such as test reuse in manual test specification, when existing tests are modified for testing new similar functionality, unintentionally leaving parts of already tested functionality. Other causes include incomplete requirements specification, redundancy of requirements, legacy, static test suites, parallel testing, or distributed testing [6]. In this work we are interested in analyzing test redundancy in integration tests, which test varying number of feature interactions to expose any faults in interaction between integrated individual code components.

D. Regression Test Optimization

DevOps promotes iterative development, where smaller self-contained changes are made to software frequently. Every change is regression tested to check whether it introduces any faults caused by the interaction of integrated components. Since speed is one of the key requirements of efficient DevOps, regression testing requires only a set of relevant test cases. This is especially important in fast-evolving systems where test suites used for validating the correctness of systems grow quickly. Previous studies have shown that test suite size has a large impact on the overall test cost in the software development lifecycle [12], [15], [16]. Therefore, finding such a set of relevant test cases is the goal of regression test optimization. Specifically, regression test optimization aims to find an optimized set and order of regression tests that satisfy predefined optimization objectives. This includes selecting a relevant set S' based on S, known as test selection [17], and finding the execution order of tests in S', known as test prioritization [19]. In this work we focus on test optimization guided by the analysis of test redundancy. i.e feature interaction overlap among different tests. Existing test optimization approaches have not been targeted towards DevOps and CI, and specifically have not been investigating the impact of test redundancy on the performance of CI testing in DevOps.

E. Challenges of Testing in CI

Testing in continuous integration and DevOps is amenable to a number of challenges. First, it is highly ***sensitive to long runtime***, since feedback on source code integration needs to be provided as rapidly as possible. Fast feedback enables faster test cycles, which further enable faster release cycles. Second, test effort needs to be steered towards achieving ***just the quality required for deployment to staging or production***. If more effort is put into testing, this may negatively affect time- and cost-efficiency of testing. Third, since testing is time-limited in CI and DevOps, testing process (and in particular test selection) should be ***continuously optimized***, guided by risk analysis, based on the type of code changes made and their impact. Risk-analysis would enable defining a dynamic regression scope for each build and test iteration, with multiple layers of tests to enable iterative, and faster feedback. Fourth, as test suites grow overtime, to cover new functionality added to the codebase, tests start to overlap, building ***test redundancy***. This creates the risk of increased test effort as many similar tests may seem relevant, and therefore selected for running. Therefore, the key challenge lies in identifying test redundancy and selecting test cases so that redundancy is minimized. This will help reduce long test runtime, and will enable reaching just the required level of quality. Furthermore, high levels of ***test automation*** are needed, in test selection and optimization (apart from test execution, where automation is considered a prerequisite for CI and DevOps).

III. Improving the efficiency of CI with Continuous Test Optimization

The approach for improving efficiency and effectiveness of CI that is proposed in this paper exploits the idea of *continuous test optimization based on test redundancy analysis*. As stated previously, one critical challenge of CI and DevOps is enabling short and effective test cycles given redundant test suites. A redundant test suite contains test cases that overlap, given a specific feature coverage criteria. In this context, cost-effective testing entails a trade-off between the size of a test suite and its comprehensiveness. Here, we refer to comprehensiveness as the ability of a test suite to detect faults caused by interactions between two features (pairwise coverage). For the sake of simplicity, in this paper we will restrict our approach to pairwise coverage only, while the proposed concept can be extended to any-wise feature coverage (consequently entailing some higher computational complexity). To better illustrate the complexity of CI testing in the presence of redundant test case with overlapped feature coverage, we present the following example.

A. Illustrating Example

A software system under test consists of a set of functionality modules, referred to as $features\ FS = \{f_1, f_2, ..., f_n\}$. Features are used to build a set of solution configurations for video conferencing. Some features are $f_1 = video\_resolution$, $f_2 = audio\_resolution$, $f_3 = audio\_protocol$, $f_4 = point\_to\_point\_calls$, $f_5 = multi\_party\ \_calls$. A test suite $TS = \{t_1, t_2, ..., t_m\}$ is developed for testing these solution configurations, where test cases partially cover the total set of features FS in different combinations. $Cov(t_i) = \{f_1, f_2, ..., f_n\}$ denotes a set of features tested by a test case t_i. As the system under test evolves incrementally through continuous development and testing, TS evolves continuously and grows larger. New test cases are added covering new functionality, but also interactions between new and old functionality. The same features become part of multiple tests, but because of large size of a test suite, the same combinations of features often become part of multiple tests. This in turn increases test effort, as the same interactions between system functionality are executed multiple times. In the example shown below, four test cases t_1, t_2, t_3, and t_4 cover a set of features $\{f_1, f_2, f_3, f_4, f_5\}$ in different combinations. Considering a pairwise interaction coverage as a criterion for test redundancy analysis, the covering set of features for tests t_1 and t_4 overlap with the covering set of features for tests t_2 and t_3 respectively. Since this overlap represents proper subsets i.e. $Cov(t_1) \subset Cov(t_2), Cov(t_4) \subset Cov(t_3)$, tests t_1 and t_4 are redundant with respect to tests t_2 and t_3.

$$Cov(t_1) = \{f_1, f_2\}$$
$$Cov(t_2) = \{\{f_1, f_2\}, \{f_1, f_3\}, \{f_2, f_3\}\}$$
$$Cov(t_3) = \{\{f_1, f_4\}, \{f_1, f_5\}, \{f_4, f_5\}\}$$
$$Cov(t_4) = \{f_4, f_5\}$$
$$Cov(t_1) \subset Cov(t_2),\ Cov(t_4) \subset Cov(t_3)$$

B. The Approach

Our approach to reducing test cycles with redundancy detection and reduction explores the concept of *total test redundancy* and *partial test redundancy*. To explain these concepts, we introduce the following definitions.

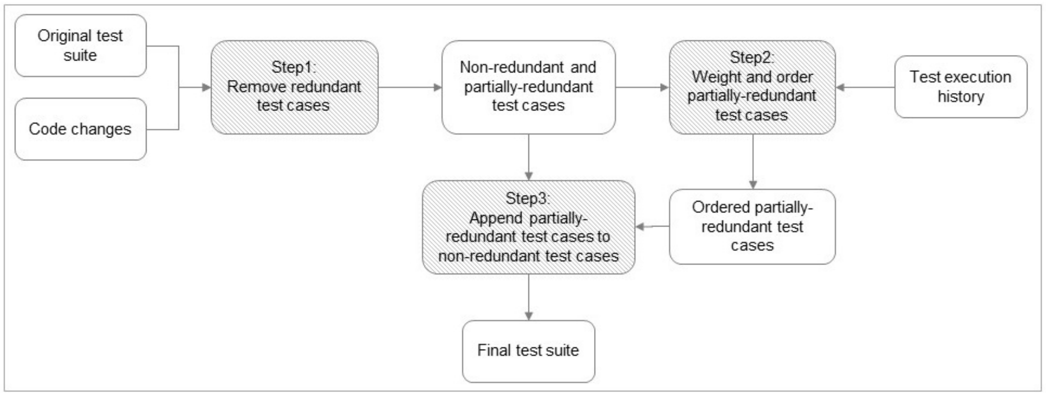
Def 1: A test case t_1 is totally redundant of a test case t_2, if $Cov(t_1) \subseteq Cov(t_2)$.

Def 2: A test case t_1 is partially redundant of a test case t_2, if $Cov(t_1) \neq Cov(t_2)$ and $Cov(t_1) \cap Cov(t_2) \neq 0$.

In the first step of the approach, we address total redundancy, by detecting test cases whose covering set of features is fully covered by another test case. After such test cases have been identified, we remove them from a test suite. In the second step of the approach we address partial redundancy by combining interaction coverage metrics with historical fault detection effectiveness of tests obtained from test logs. The underlying idea is that partially redundant test cases which have historically exhibited good fault revealing performance can be classified as non-redundant, and otherwise as redundant. This idea is supported by studies showing that test execution history can help improve cost-effectiveness of testing [7], [8], [9]. [21].

Step 1: Given the system under test and the changes to its source code, as well as an existing test suite, we first find a set of tests impacted by the changes. This is performed automatically, using association links between test cases and features (software functionality) covered by the test cases. Next, for the obtained test suite, we analyze test overlap between test cases to find those test cases whose covering feature interactions are completely covered by other test cases. We remove such test cases from a test suite, obtaining a test suite with only non-redundant and partially redundant test cases.

Step 2: Next, we look into partially redundant test cases contained within the test suite. These are the test cases whose covering feature interactions are partially covered by other test cases. We obtain execution history for these test cases



Original test suite
Code changes
Step1: Remove redundant test cases
Non-redundant and partially-redundant test cases
Step2: Weight and order partially-redundant test cases
Test execution history
Step3: Append partially-redundant test cases to non-redundant test cases
Ordered partially-redundant test cases
Final test suite

Fig. 1. Redundancy detection and reduction methodology.

(for those that have been executed before) and analyze their fault detection effectiveness in past executions. In particular, we assign different weights to test cases chronologically, based on how recently they detected faults. Those tests that detected faults more recently have higher weight than those that detected faults earlier. If n is the number of historical test execution iterations, then the weight ω is calculated as follows:

$$\omega = \begin{cases} \frac{1}{n}, & \textit{for test cases that have been executed before} \\ 1, & \textit{for newly added test cases} \end{cases}$$

Next we sort the test cases based on ω value, such that those test cases with higher fault detection effectiveness are ordered higher in the list of partially redundant test cases. The rationale for sorting is that CI is bound by tight time constraints. Therefore, we want to ensure that the most important test cases are run first, in case that available time budget for testing is smaller than the time required for running the whole test suite.

Step 3: Finally, the resulting test suite consists of test cases that have been identified as non-redundant in *Step 1*, which come first in order, followed by a set of partially redundant test cases, ordered based on their fault detection effectiveness. The approach is schematically presented in Figure 1.

IV. Experimental Case Study

We conducted a set of experiments to evaluate the effectiveness of the proposed approach for test redundancy reduction in CI and DevOps. The approach was applied to an industrial case study of optimizing CI development and testing of video conferencing telepresence systems developed by Cisco Norway. These systems enable full "in-person" meeting experience with high-end video communication and collaboration between multiple parties using cameras with eye-tracking features and directional microphones which allow the transmission of high definition video and audio, as well as wireless sharing of presentations and other documents.

The objective of the experiments was to evaluate the ***test execution time*** and ***fault-detection effectiveness*** of CI testing when using our approach for test redundancy reduction. We compared the approach to *existing industry practice of CI testing*, *retest-all* approach, and *random test selection*. Specifically, we are interested in answering the following research questions:

RQ1: What effect does the redundancy reduction approach (RED) have on the duration of test cycles in CI compared to industry testing practice (IP)?

RQ2: What effect does the redundancy reduction approach have on the duration of test cycles in CI compared to retest-all (RA) approach?

RQ3: What effect does the redundancy reduction approach have on fault detection effectiveness of test suites compared to random test selection (RS)?

A. Industrial System

The industrial system used in the experimental case study represents a video conferencing system that is developed following a DevOps practice. The system is highly complex, implementing around one hundred features, which are used in different combinations across different solution configurations for video conferencing. As the system is developed in CI, code updates are made frequently. Each update is followed by integration testing cycle, to verify the correctness of newly added functionality and to ensure that no regressions are introduced in existing functionality. The test suite is developed incrementally and is large in size, covering system features in various combinations of interactions. Extensive test suites for integration and configuration testing, together with the requirements for short test cycles make cost-effective testing of video conferencing systems in DevOps challenging. The test data used in the experiments consists of 400 test cases. Test execution history for these test cases is available for 6 last test execution runs, which gives 2400 test case executions. Historical data includes test execution result (pass/fail) and test execution duration.

B. Experiment Setup and Methodology

We address the posed three research questions RQ1-RQ3 in experiments E1, E2, and E3, respectively.

In experiment E1, we compare RED with IP, in terms of total test execution time. First, for the features affected by changes, we select a set of impacted test cases. These tests represent the initial test suite. Then we obtain execution history for these test cases, and apply our redundancy reduction approach to the initial test suite. We analyze test overlap in terms of pairwise feature coverage, and remove test cases whose covering set of pairwise feature interactions is completely contained within another test case(s). At this point, the test suite contains non-redundant test cases and partially redundant test cases. Next, for all partially redundant test cases, we analyze which of them have shown good fault-detection performance in the past. Based on this information, we eliminate test cases that have not contributed to increased fault detection effectiveness. Next, we calculate weights for the rest of partially redundant test cases, rewarding more recent fault detection higher. We sort these test cases according to their weight, and append them to the previously identified non-redundant test cases. For the resulting set of test cases, we measure the percentage reduction of the test suite size compared to the size of the original test suite (used by industry practitioners for testing the changes). We run experiment E1 5 times, for all available historical test execution data.

In experiment E2, we compare RED with RA, in terms of total test execution time. We chose RA as a comparison metric because this is a commonly used approach to regression testing in practice. Specifically, we modified the RA in a way to include only the tests affected by changes, versus retesting the entire test suite. Therefore, for the features affected by changes, we select a set of impacted test cases. These tests represent the initial test suite. Then we apply our redundancy reduction approach to the initial test suite, and examine test overlap in terms of pairwise feature coverage. We remove redundant test cases base on this criterion, which gives a test suite containing non-redundant test cases and partially redundant test cases. Then we analyze execution history for the partially redundant test cases, and eliminate those which historically have not showed to increase fault detection. Next, we order remaining partially redundant test cases based on their recent fault detection performance and append them to the previously identified non-redundant test cases. This gives us the final test suite. For this test suite, we measure the percentage reduction of the test suite size compared to the size of the initial test suite.

In experiment E3, we compare RED with RS, in terms of fault-detection effectiveness. We compare with RS because this is a commonly used alternative to automated guided test selection and reduction, primarily driven by low cost and low complexity. First, for the features affected by changes, we select a set of impacted test cases, which represent the initial test suite. Then we apply the same setup as in E1 to obtain the set of non-redundant test cases followed by an ordered set of partially redundant test cases, as the final reduced test suite. Next we measure the total test execution time for the final test suite (time limit), based on historical test execution time of each test case. Afterwards, we start randomly selecting test cases from the initial test suite, accumulating execution time of each selected test case. We repeat this process until the time limit is reached. The resulting test suite obtained in this process is the randomly selected test suite. Now we measure the loss of fault detection of the randomly selected test suite compared to the fault-detection of the final test suite. The measured value is the percentage of faults that were detected by the final test suite and not by the randomly selected test suite. Because of the randomness in RS approach, we repeat the experiment 100 times.

C. Results and Analysis

In this section we present the results of the experiments E1, E2, and E3, addressing research questions RQ1, RQ2, and RQ3, respectively. The results are graphically presented in Figure 2, Figure 3, and Figure 4, respectively.

1) Time-effectiveness Compared with Industry Practice: In the first experiment addressing RQ1, we compared RED and IP in terms of test suite execution time. The results show that RED was able to reduce test cycle by 30% on average compared to IP. The results are shown in Figure 2. Y axis corresponds to the percentage reduction of test execution time of the reduced test suite compared to the test suite used by practitioners.

2) Time-effectiveness Compared with Retest-All: In the second experiment aimed to answer RQ2, we compared RED and RA in terms of test suite execution time. In this experiment, RA showed to reduce total test suite execution time by 35% compared to RA. The results are shown in Figure 3. Y axis corresponds to the percentage reduction of test execution time of the reduced test suite compared to retest-all approach.

3) Fault-detection Effectiveness Compared with Random Selection: In the third experiment addressing RQ3, we compared RED with RS in terms of fault detection effectiveness, for the same (given) test budget. The results demonstrate that RED can achieve up to 70% better fault detection compared to randomly selected test cases, and 40% on average better fault detection compared to randomly selected test cases, for the test suites used in the experiment. The results are shown in Figure 4. Y axis shows the distribution of the percentage of fault detection effectiveness gain of the reduced test suite compared to randomly selected test suite.

In summary, the results of the experiments E1, E2, and E3 demonstrate that the proposed approach can effectively reduce test cycles in CI compared to industry practice of CI testing by 35% on average, and compared to retest-all approach by 30% on average. The results further demonstrate that the proposed approach can improve fault detection effectiveness of a test suite compared to random test selection up to 70%, for the same test time budget.

D. Threats to Validity

A threat to external validity of the results is the choice of the industrial case study of continuous integration testing and the test dataset. While the used industrial context is an example of good industry practice, we cannot say that it is representative,

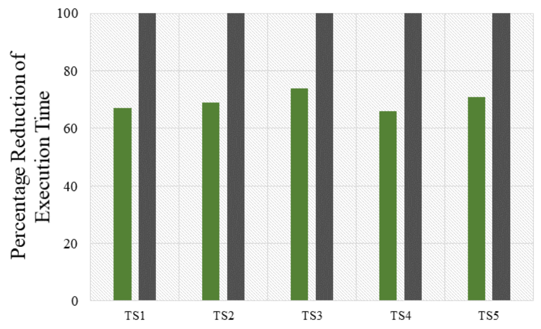


Fig. 2. Comparison of the proposed approach with $industry\_practice$ approach in terms test suite execution time.

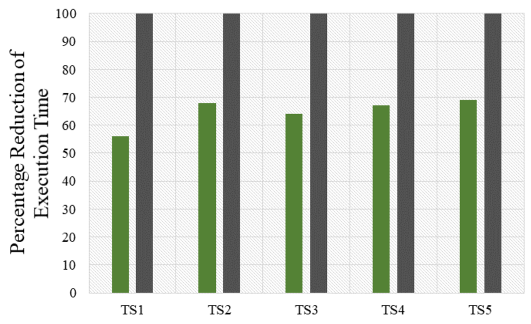


Fig. 3. Comparison of the proposed approach with $retest\_all$ approach in terms of test suite execution time.

and continuous integration testing can be applied differently in different companies. More studies are needed to verify whether our results generalize to other practices of CI and DevOps. This is part of our future work. A threat to internal validity could be potential faults in our implementations of the optimization algorithms. We have thoroughly tested the code to ensure that these threats are minimized.

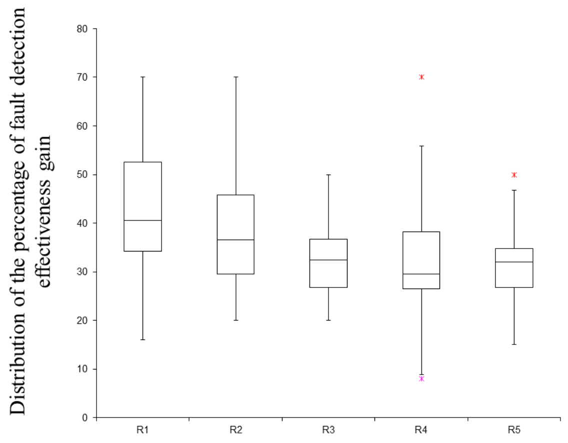


Fig. 4. Comparison of the proposed approach with $random\_test\_selection$ approach in terms fault detection effectiveness.

V. Conclusion

In this paper we proposed the approach for improving time-efficiency of DevOps using continuous test optimization. The approach is based on test redundancy analysis in terms of feature interaction coverage. By reducing test redundancy, it is possible to reduce CI test cycles and further release cycles in DevOps. The approach has been evaluated and has demonstrated improvement in time-efficiency compared to industry practice, retest-all approach, and random test selection.

Acknowledgment

This work is supported by The Research Council of Norway, through the Certus SFI project. We thank to Cisco Systems Norway for using their test data set and for fruitful discussions that contributed to this work.

References

[1] Economic Benefits of HP Future Smart Agile Transformation, Evidence and case studies (continuousdelivery.com).

[2] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, Quality and productivity outcomes relating to continuous integration in GitHub, International Symposium on the Foundations of Software Eng., 2015.

[3] A. Miller, A hundred days of continuous integration, AGILE, 2008.

[4] M. Leppanen, S. Makinen, M. Pagels, V. P. Eloranta, J. Itkonen, M. V. Mantyla, and T. Mannista, The highways and country roads to continuous deployment, IEEE Software, 2015.

[5] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, D. Dig, Trade-offs in continuous integration: assurance, security, and flexibility, 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), 2017.

[6] L. Bass, I. Weber, L. Zhu, DevOps: A Software Architect's Perspective, Addison-Wesley Professional, 2015.

[7] S. Elbaum, G. Rothermel, J. Penix, Techniques for Improving Regression Testing in Continuous Integration Development Environments, International Symposium on the Foundations of Software Engineering, 2014.

[8] D. Marijan, A. Gotlieb, S. Sen, Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study, IEEE International Conference on Software Maintenance (ICSM), 2013.

[9] D. Marijan, M. Liaaen, Effect of Time Window on the Performance of Continuous Regression Testing, ICSME, 2016.

[10] D. Marijan, M. Liaaen, Test Prioritization with Optimally Balanced Configuration Coverage, HASE, 2017.

[11] G. Rothermel, M. Harrold, J. Ostrin, and C. Hong, An empirical study of the effects of minimization on the fault detection capabilities of test suites, Int. Conference on Software Maintenance (ICSM), 1998.

[12] G. Rothermel, M. J. Harrold, J. Ronne, and C. Hong, Empirical studies of test-suite reduction, Software Testing Verification and Rel., 2002.

[13] W. Wong, J. Horgan, A. Mathur, and A. Pasquini, Test set size minimization and fault detection effectiveness: a case study in a space application, 21st Computer Software and App. Conference (COMPSAC), 1997.

[14] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness, ICSE, 1995.

[15] H.-Y. Hsu and A. Orso, Mints: A general framework and tool for supporting test-suite minimization, ICSE, 2009.

[16] Y. Yu, J. A. Jones, and M. J. Harrold, An empirical study of the effects of test-suite reduction on fault localization, ICSE, 2008.

[17] M. Grindal, B. Lindstrm, J. Offutt and S. F. Andler, An evaluation of combination strategies for test case selection, Empirical Software Engineering, vol. 11, 2006.

[18] A. Gotlieb, D. Marijan, FLOWER: optimal test suite reduction as a network maximum flow, Int. Symp. on Soft. Testing and Analysis, 2014

[19] Y.-C. Huang, K.-L. Peng and C.-Y. Huang, A history-based cost-cognizant test case prioritization technique in regression testing, Journal of Systems and Software, vol. 85, 2012

[20] J.F. Perez, W. Wang and G. Casale, Towards a DevOps Approach for Software Quality Engineering, Workshop on Challenges in Performance Methods for Software Development, 2015.

[21] J.M. Kim and A. Porter, Technique for Regression Testing in Resource Constrained Environments, International Conference on Software Engineering, 2001.