Improving Test Effectiveness Using Test Executions History: An Industrial Experience Report

Armin Najafi†, Weiyi Shang†, Peter C. Rigby‡
Department of Computer Science and Software Engineering
Concordia University, Montreal, Quebec, Canada
{a_ajaf, shang}@encs.concordia.ca†, peter.rigby@concordia.ca‡

***Abstract*—The cost of software testing has become a burden for software companies in the era of rapid release and continuous integration. Our industrial collaborator Ericsson also faces the challenges of expensive testing processes which are typically part of a complex and specialized testing environment. In order to assist Ericsson with improving the test effectiveness of one of its large subsystems, we adopt test selection and prioritization approaches based on test execution history from prior research. By adopting and simulating those approaches on six months of testing data from our subject system, we confirm the existence of valuable information in the test execution history. In particular, the association between test failures provide the most value to the test selection and prioritization processes. More importantly, during this exercise, we encountered various challenges that are unseen or undiscussed in prior research. We document the challenges, our solutions and the lessons learned as an experience report. Our experiences can be valuable for other software testing practitioners and researchers who would like to adopt existing test effectiveness improvement approaches into their work environment.**

***Index Terms*—Test effectiveness, Test prioritization, Test selection, Industrial experience report**

I. Introduction

Testing is an important, yet time consuming and costly process, especially for large software systems. Prior research estimates that testing consumes between 30% to 50% of the time in software development life-cycle [1]. For example, an industrial case study shows that it takes over two days to complete testing on a medium size video conferencing system [2].

The cost of testing has become a burden for software companies during rapid release [3], where continuous integration techniques are widely adopted in practice to receive feedback from testing as soon as changes are made to the source code [4]. With thousands of commits made to the source code every day, it is challenging to keep up with the speed of development. Google's version control repository receives over 16,000 commits every day [5], which results into a median of 27 test requests per minute [6]. Our industrial collaborator Ericsson has faced the same challenges for testing its large-scale software systems. Moreover, the changes in their testing processes may be even greater due to the complex testing infrastructure that is purposely designed for each software subsystem.

Test selection and test prioritization are proposed by prior research to improve the effectiveness of test executions, i.e., to find more failures in shorter duration time with a lower cost [7], [8], [9], [10], [6]. With test selection, tests are either executed or skipped on the fly. With test prioritization, tests are reordered such that more test failures can be captured earlier using limited testing resources. Therefore, we leverage the wisdom from prior research to assist Ericsson with improving the testing process of one its large-scale software systems. However, since prior research is typically evaluated on a particular industrial subject system (e.g., from Microsoft or Google), it is unclear to what extent the existing approaches can improve the test effectiveness in our subject system and whether there are challenges that are unseen by prior research.

In this paper, we share our industrial experience for adopting test selection and prioritization approaches to improve the test effectiveness of one of the large software systems in Ericsson. In particular, the adopted approaches are based on historical test failure frequency, test failure association, and the costs associated with the testing process. In order to evaluate the usefulness of these adopted approaches, we simulate applying these approaches to six months of testing data from a large-scale system in Ericsson. Our results show the importance of test execution history in enabling the test selection and prioritization approaches to assist with software testing in practice. Particularly, our results confirm the value of discovering associations among test failures as shown by recent research [6]. We encountered many engineering and design challenges for adopting and applying the approaches to the testing processes of Ericsson. In the end, we conquered the challenges and documented the challenges, our solutions and the lessons learned as an experience report. We believe that our experience in adopting existing test selection and prioritization approaches can help software practitioners and researchers who want to adopt software testing approaches into their work environment.

The major contributions of this paper are:

- We adopt and evaluate test selection and prioritization approaches with the goal of improving test effectiveness in a large industrial system with a complex testing infrastructure.
- We demonstrate the value of test execution history for improving test effectiveness in a large-scale industrial system in practice.
- We provide an industrial experience report that documents the challenges that are encountered and our lessons

learned during the adoption process of the test selection and prioritization approaches.

The remainder of this paper is organized as follows. Section II describes the background of the subject system and its testing process. Section III discusses the approaches that are adopted by our study to improve the test effectiveness of our subject system. Section IV presents the results of evaluating the adopted approaches. Section V discusses the challenges that we have encountered and the lessons learned during the experiments. Section VI discusses the threats to the validity of our findings. Section VII presents other related research in the literature. Finally, Section VIII concludes the paper.

II. Background and Subject System

In this section, we explain the background required for this paper, i.e., the subject system that we studied in Ericsson. Figure 1 presents an overview of the testing process of our studied subject system.

A. Subject system

The subject system in this study is a large-scale software system from Ericsson. The system has a large user base across the world and is currently being developed on a daily basis for new features and performing maintenance activities. The teams that develop this system consist of a large number of developers and testers across the globe. The system is developed using a modern typical programming language that is hosted in a typical version control repository. To ease the discussion about the subject system, we refer to it as *ELS* (Ericsson's Large-scale System) in the rest of this paper. Due to the criticality of the system, our study is conducted by simulation that is based on the test results from the past six months of the day of the study.

B. Testing process

The testing process in *ELS* is conducted on a special testing infrastructure which cannot be replicated easily. Therefore, in general, the changes to the source code of *ELS* have to wait in a queue in order to get tested in such a complex environment. In particular, the testing process consists of four steps. 1) Collecting commits. The commits that are made in the version control repository are collected and put in a queue to be tested as soon as the testing environment is available. 2) Testing the commits. Once the testing environment is free, commits from the queue are consolidated as a batch and are moved into the testing environment. There exists a large number of pre-designed tests which will be executed on the new commits. If all of the tests pass, the commits will be merged into the main trunk of the version control repository. If any of the tests fail, the commits will be sent back for a bi-section process [11]. 3) Bi-section. The bi-section process splits the commits of a failed batch in half by their time stamps. The first half of the commits are sent directly to the queue, with other commits that are already waiting in the queue to be tested. The second half of the commits will be waiting for the queue to be empty to enter the queue. 4) Manual check of the test results. Due to the complex testing infrastructure, not all test failures are due to software bugs. Therefore, once the test failures are located in one single commit (after multiple iterations of the bi-section process), a system expert will manually check the failures. A test failures is labeled as a false-positive if it is not caused by a software bug, but rather by an infrastructure issue in the testing environment. Or it can be labeled as a true-positive if the test failure is identified to be an actual bug in the main software system. Please note that the testing process that we described above does not include all of the details regarding the software quality assurance process in Ericsson. There may exist other approaches, infrastructure and test stages available in the testing flow. However, for the scope of this study, we only focus on the above-described testing process.

III. Adopted Approaches

Software testing and regression testing improvement have been largely studied in the software engineering literature. Kazmi et al. [12] and Suleiman et al. [13] provide extensive reviews on the recent studies in the literature for test selection and prioritization. In this section, we present the approaches that we have adopted in order to improve the test effectiveness of *ELS* in Ericsson. In particular, we adopted approaches that perform test selection and test prioritization.

A. Test Selection

We adopt test selection approaches that are based on prior test failure frequency, association, and the costs of the testing process. Test selection approaches are applied before the start of each batch, for which all of the test execution results before the day of testing are used as learning data for our analysis. We do not learn from the test execution data of the same day of the test executions, as test failure results need to be manually labeled as true-positives or false-positives by the testers by the end of each day. Therefore, we only obtain the latest labels at the end of each day.

Based on Failure Frequency

Intuitively, tests that previously failed frequently are more likely to fail again later [7], [6]. Therefore, the frequency of past test failures can be used as an indicator for suggesting test selection opportunities.

***Microsoft's FreqSelect*:** Anderson et al. [7] propose an approach that calculates the frequency of test failures using the test executions history. Afterward, the tests that failed more frequently in the past are recommended to be selected again later.

***FreqSelect*:** Our adopted approach is closely related to the test selection approaches proposed by Anderson et al. [7], where only the tests that have prior failures are selected. In addition, to consider the cases where test failures can be false-positives in our testing process, we only consider the tests that have prior true-positive test failures as opposed to considering all of the test failures.

Based on Failure Association

As the prior study shows, there exists a large number of co-failures in test executions [6]. Hence, associations can be effectively leveraged for improving test effectiveness [6].

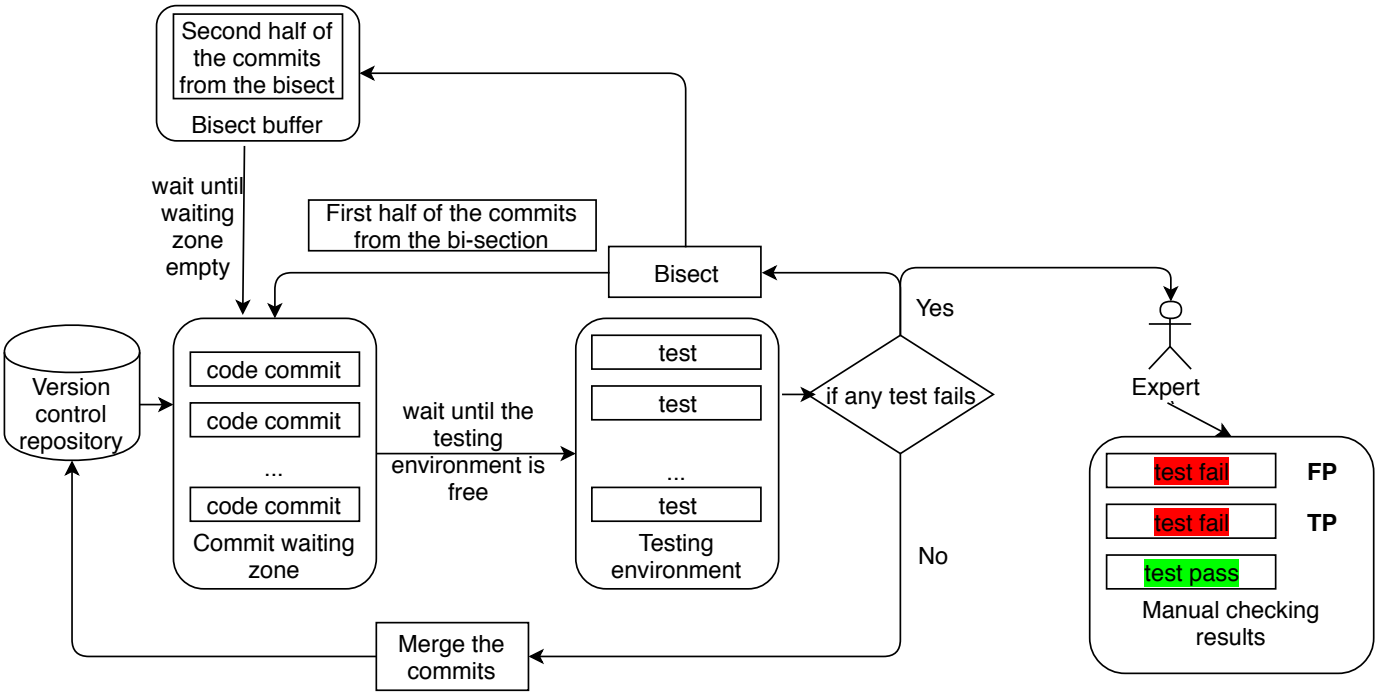


Fig. 1: An overview of the testing process of *ELS*.

***Microsoft's AssocSelect*:** Anderson et al. [7] perform association analysis on the test failures. In particular, failures in certain subsets of tests are used to determine other subsets that are likely to fail. With the identified association rules, Anderson et al.'s approach selects the tests that are more likely to fail again.

***AssocFreqSelect*:** We adopted *Microsoft's AssocSelect* with the goal of minimizing the redundancy among tests by only selecting the cheapest test to run if multiple tests are associated with each other. In particular, we leverage test failures in each batch (see Section II) to perform association rules mining using the Apriori algorithm [14]. We obtain a list of association rules pointing to the test cases that tend to co-fail with each other. Some rules may have low confidence and should not be used to perform the selection. Therefore, we only consider the association rules that have over 0.8 confidence.

Afterward, we use these rules to dynamically select the test cases that are most effective and at the same time have the least amount of redundancy among each other. We first rank the test cases based on their probability of finding true-positive test failures as extracted from the test executions history (c.f. the *FreqSelect* approach). Then we iterate through the list from the top one by one and check if there exists any association rule between the current test and any of the previously analyzed ones. If there exists such a rule, we only keep the test that has a shorter average execution time on the list. In this way, we remove the redundancy among the tests while shortening the tests duration time and maintaining the effectiveness of the tests in finding true-positive failures.

The goal of our approach (*AssocFreqSelect*) is different from that of Anderson et al. (*Microsoft's AssocSelect*). *Microsoft's AssocSelect* aims to select the tests that are more likely to fail using failure associations. However, *AssocFreqSelect* aims the opposite, i.e., associated test failures may be redundant and they should not be executed. *AssocFreqSelect* removes tests based on the existence of association to avoid redundancy among the test failures.

Based on Cost

Test execution comes with a cost, especially for the complex testing infrastructure that is used by *ELS*. With such high testing costs, tests can be skipped if skipping them is more cost-effective than executing them [8].

***Microsoft's Theo*:** Herzig et al. [8] propose a tool named THEO that stands for Test Effectiveness Optimization using Historic data. THEO is designed based on a cost model that dynamically skips tests when the expected cost of running one exceeds the expected cost of skipping it. In particular, *Microsoft's Theo* calculates two values for every test execution, cost of execution and cost of skip. It then decides which action is more economical. Similarly, in our *TheoSelect* approach we select test cases based on an online cost analysis for every test case execution. Two costs are calculated for every test case execution: cost of execution and cost of skip, calculated as follows:

$$
\begin{aligned}
cost_{exec} &= cost_{machine} + (P_{FP} * cost_{inspect}) \\
cost_{skip} &= P_{TP} * cost_{escaped} * time_{delay} * \#engineers
\end{aligned}
\tag{1}
$$

In the equation, $cost_{machine}$ is an estimate of the costs associated with running a test in the test environment for a certain amount of time. $cost_{inspect}$ is an estimate for the time

delay and the costs associated with the engineers that will perform the inspections after a test failure. This parameter mainly affects the costs associated with running a test case when a false-positive needs to be inspected. $cost_{escaped}$ represents the average cost of an escaped defect. #engineers is the number of engineers that will get involved when a new slip-through is introduced into the tests flow. $time_{delay}$ is the amount of time delay that a new slip-through will impose on the tests flow. Moreover, P_{FP} and P_{TP} are the probabilities for finding a true-positive or a false-positive for every test case. These two values are calculated for every test case based on total number of past executions in history and the number of manually identified true positives and false positives.

Consequently, after evaluating the equations, if the cost of running a certain test case is shown to be higher than skipping it, the test case execution will be skipped or vice versa.

***TheoSelect*:** We adopted the approach proposed by Herzig et al. [8]. We call this approach *Microsoft's Theo* and is identical to our *TheoSelect* with different parameters. *Microsoft's Theo* is a generic test selection tool that selects test cases based on a cost analysis for each execution on the fly. We cannot unveil the exact values that we have chosen for the parameters due to confidentiality reasons. However, as the ratio of the parameters matter in our use case, we set $cost_{machine}$ to 1 unit, $cost_{escaped}$ to 13.1 unit and $cost_{inspection}$ to 30 units as the ones used by Herzig et al. [8].

B. Test Prioritization

***Kim's Prio*:**. Kim et al. [10] propose a test prioritization technique based on test execution history. The prioritization technique gives priority scores to each test for their execution. They propose three approaches for calculating the scores. First, the tests that have not been executed recently are given a higher score. Second, the tests that have revealed more faults recently will get a higher score. Finally, a higher score is given to the tests that cover functions which are infrequently covered in the past testing sessions.

***Google's Prio*:** Elbaum et al. [9] propose another test selection and prioritization technique. Their technique prioritizes tests that have recently found failures in a specified time window based on the previous execution history.

***Prio*:** Our prioritization technique is similar to *Kim's Prio* and *Google's Prio*. In our approach, we give a higher priority to tests that have found more failures and have a shorter average execution time. We rank the test cases based on the ranking values calculated as follows:

$$priority\_value = \frac{\#total\_failures}{total\_executions\_duration} \quad (2)$$

where the $\#total\_failures$ is the true-positive failures detected by the tests and the $total\_executions\_duration$ is all of the time that is spent on executing the tests. The total duration of test execution has been incorporated into the equation in order to normalize the effectiveness of the test cases. In this way, we give a higher priority to tests that find more failures with shorter execution time, i.e., less effort.

Our approach relies only on the test executions history, such as the number of true-positive failures and test executions duration to calculate the priority values. We do not use test coverage information since all too often, it is not available in the test process of *ELS* in Ericsson. Similar to test selection approaches, we learn on all of the test execution results until one day before of every test.

IV. Results of the Adopted Approaches

In this section, we present the simulation results of our adopted approaches (see Section III). The results are based on simulating the scenarios for each test selection or test prioritization approach using six months of test results obtained from the test environment of *ELS*.

A. Test Selection

We use three metrics to evaluate the adopted approaches for test selection.

Total test execution time reduction. Our first metric demonstrates how much test execution time is reduced by performing each of our test selection techniques. In particular, we first calculate the total time that is needed to execute all of the tests. Then using each test reduction technique, we calculate the required time to only execute the tests that are selected (i.e., not removed) by each technique. Finally, we calculate the reduction percentage based on the total time needed for running all of the tests and running only the selected tests by each selection approach.

Number of slip-through test failures. Our second metric, reveals the percentage of the true-positive test failures that may have been missed due to a test not being selected. The value zero for this metric means that our test selection approach does not remove any test that would have been a true-positive test failure. Intuitively the lower the value of this metric, the better the approach is. This metric is particularly important since the slip-through true-positive test failures may have a direct impact on the end users.

Total cost reduction. This metric is a measure for showing the impact of the test reductions and their side effects in terms of a concrete cost value. For this metric we calculate the costs of running each test case or encountering a missed true-positive using the parameters given in Section III. In particular, cost of running each batch can be calculated as follows:

$$\begin{aligned} cost_{batch} = & \sum_{test\_cases} test\_case_{execution\_time} * cost_{machine} \\ & + \#FP * cost_{inspection} \\ & + \#slip\_throughs * cost_{escaped} * time_{delay} * \#engineers \end{aligned} \quad (3)$$

where the total cost is the sum of the costs of the test infrastructure during the test execution, the effort by the practitioners to inspect any false-positive test failures and the cost of the slip-through test failures if an important test is not selected. Afterward, we calculate the total cost by assuming that all of the tests are selected, i.e., there exist no slip-through

TABLE I: Overall comparison of the three adopted test selection approaches.

	Reduction in execution time	Slip-throughs	Reduction in cost
FreqSelect	0.01%	10.22%	-0.73%
AssocFreqSelect	13.91%	11.36%	13.08%
TheoSelect	41.78%	34.65%	39.23%

test failures. Finally, the total cost reduction is calculated based on the cost of using each test selection approach and the total cost of selecting all tests as follows.

$$reduction\_percentage\_in\_cost = 100 * (1 - \frac{simulated\_total\_cost}{total\_cost}) \quad (4)$$

Table I shows the results of our evaluations for each of our approaches. The results show that by only considering the frequency of past test failures, reducing the test execution time is not trivial. In fact, by only saving 0.01% of the test execution time, the approach let 10.22% of the true positive test failures untested. Such missed true-positive failures result in an increase in cost. On the other hand, the *AssocFreqSelect* approach shows significant improvement over the *FreqSelect* approach. The *AssocFreqSelect* approach is able to maintain a similar slip-through rate but reducing both the time of test executions (13.91% reduction) and the costs (13.08% reduction). Our results confirm the findings from recent research by Zhu et al. [6] where the association between the test failures are found to be effective for re-ordering the tests.

Association between tests is a valuable source of information for improving test effectiveness.

The cost analysis approach, i.e. *TheoSelect*, is designed to optimize the cost of testing, where the results demonstrate much higher (almost three times) cost reduction in testing compared to *AssocFreqSelect*. On the other hand, the slip-through test failures are also almost three times of that compared to *AssocFreqSelect*. Therefore, the practitioners can choose to lower the total costs and prefer to tolerate the fact that some failures may be missed by the testing process. This can be especially tolerable if it can be proved that there are other mechanisms in the flow that can catch these missed failures (c.f., Section V). Otherwise, having too many slip-throughs, despite reducing the costs can have a significant impact on the end users.

Even by taking advantage of a cost-based test selection approach, practitioners may still face the trade-off between the slip-through test failures and the cost of testing.

B. Test Prioritization

There are generally two criteria for evaluating test prioritization; Reaching the first test failure, or reaching all test failures as early as possible [6]. In order to evaluate the adopted approach, we consider both of the two metrics for our evaluations.

Reduced execution time until the first failure. Our first prioritization specific metric simply accounts for how fast a given order of running tests can find the first test failure. For each batch, we first reorder all of the tests using our test prioritization approach. Based on the prioritized order, we calculate the total execution time until we encounter the first test failure. We compare the test execution time to the actual test execution time that was spent to reach the first test failure in the test execution history from *ELS*. In order to minimize the bias of the original order in the test execution history of *ELS*, we also compare the test execution time with prioritizing the tests in random order. We repeat the random prioritization 1,000 times for each batch and calculate the average execution time needed to reach the first test failure.

Area under the curve of the cumulative lift chart of the test failures. The second metric compares the order given by our adopted approach with the optimal order. The optimal order gives priority to the tests that would fail in each batch and have a shorter execution time. In particular, we use cumulative lift charts [15]. Figure 2 shows an example of the cumulative lift chart of the test failures of a batch, where the y-axis shows the number of test failures and the x-axis shows the percentage of the spent test execution time so far. For each batch, we generate such a chart for both the optimal order and the order given by our adopted approach. The effectiveness of the prioritization technique is evaluated by the size of the area under the curve in each chart. Hence, for each batch, we calculate the size of the area under the curve for the optimal order and the given order by our adopted approach. We calculate the percentage (P_{opt}) of the area under the curve of the optimal order, that is covered by the order from our adopted approach. Therefore, P_{opt} has a range from 0 to 1. The closer P_{opt} is to 1, the better our prioritization algorithm is, i.e., closer it is to the optimal prioritization of the tests.

Our test prioritization approach can be used after the test selection approaches or as a standalone approach (as practitioners may not want to skip any tests). Figure 3 shows the evaluation of our approaches for reaching the first failure, i.e., the reduced execution time until the first failure happens. The results show that our *Prio* approach can effectively reduce the test executions without having any impact on the product quality. Using our prioritization algorithm after our selection techniques leads to further reductions in total test executions. However, this comes with the cost of introducing missed true positives into the product line and a worse test ordering. Figure 3 (a) compares our prioritization techniques with the current default order obtained from the execution history of our subject system in Ericsson. This figure shows that all of our approaches demonstrate significant improvement over the default order in required test execution time for reaching the first failure. Particularly, our best approach, standalone *Prio* reduces the time for finding the first failure, with a median of 60.05%. Figure 3 (b) shows similar evaluations for comparing the results of our approaches to the results

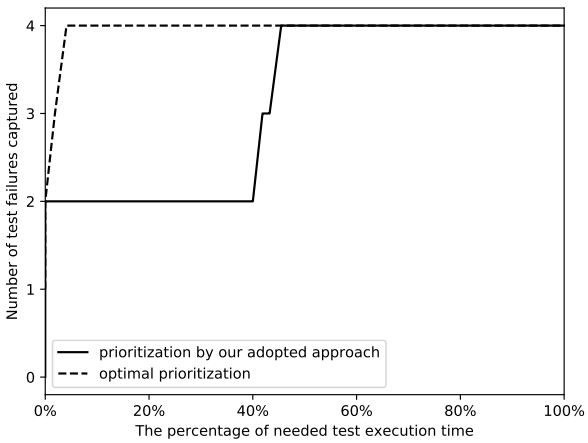
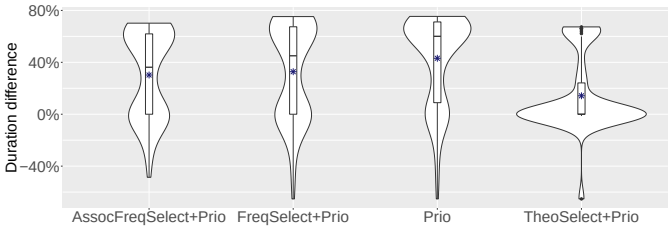
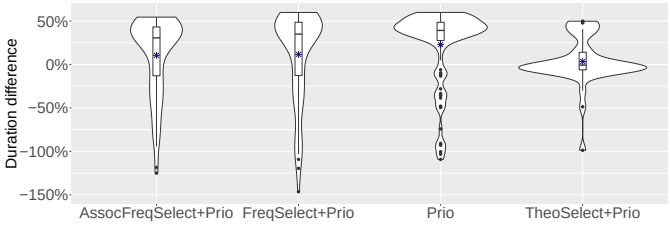


Fig. 2: An example of the cumulative lift chart of test failures. The dashed line is for the optimal order and the other line is for the order given by our adopted test prioritization approach.



(a) Comparing with the original order.



(b) Comparing with the random order.

Fig. 3: Violin plots of test execution time reduction by comparing our prioritization with the original order and the random order

obtained from averaging 1,000 iterations of random order. Although prior studies show that prioritizing tests in random order can be surprisingly effective [6], [10], our standalone *Prio* approach, demonstrates significant improvement over the averaged random order, with a median of 38.01%.

Figure 4 presents the results of our evaluations using the metric related to the area under the curve of the commutative lift chart of the test failures (P_{opt}). The median value of P_{opt} for each batch is almost 100% for all of our adopted approaches. Such results show that for almost half of the batches, the order provided by our approaches are either exactly or very close to the optimal order. The results show that a similar P_{opt} performance is demonstrated when our *Prio*

Fig. 4: Violin plots of distribution of the P_{opt} values of our prioritization results.

approach is used standalone or combined with *FreqSelect* or *AssocFreqSelect*. On the other hand, although combining *Prio* with *TheoSelect* shows a better prioritization, it is considered to be due to the removal of too many tests by *TheoSelect*, leading to a rather easier ordering for each batch.

Test prioritization by simply considering the past effectiveness of the tests can significantly help with reducing the time to reach the first failure as well as providing a close-to-optimal order for running the tests.

V. Challenges and Lessons Learned

In this section, we discuss the challenges that we faced during the implementation of our test selection and prioritization techniques in the industry and the lessons that we learned from our experiences. The challenges and lessons can shed more light on the path for other researchers and practitioners who would like to incorporate test selection and prioritization techniques in their software testing processes.

Challenge 1: Being reluctant to skipping unnecessary tests

Description: We started off our research by suggesting tests to be completely skipped or even removed from the tests flow. However, our experience shows that the test managers and developers are reluctant to removing tests from the tests flow in practice because they are afraid of leaving some product features untested. However, such a conservative approach is exactly one of the reasons for the ever-growing costs of software testing in the industry. Our experience reveals that industry experts are reluctant to removing tests from the tests flow unless they find it absolutely necessary. There is always a rather conservative approach for test executions by having a preference for increasing the level of quality assurance of the software with the price of spending more time and resources. As a result, even though our initial analysis revealed that some tests can be removed due to their low effectiveness, the preference of the test managers was to keep them anyway. Therefore, we put more emphasis on our test prioritization technique as in prioritization, no tests will be removed and only the order of running them will be changed.

Solution: We leveraged two approaches to address this challenge. First of all, we focused our techniques on test prioritization rather than test removal, i.e., instead of removing the tests, we adopted techniques that assign a low priority

to running them. Therefore, the practitioners can still execute those tests if it is necessary and the required extra resources are available. Second, instead of only providing historical evidence, we aimed to provide an explanation on why those tests can be removed. For example, we can explain that Test A can be removed since the exercised APIs are obsolete, as opposed to only showing that Test A has never detected a bug in the system.

Lessons learned: Due to the important role of testing in practice, developers and testers are often wary of any changes in their testing processes. Therefore, they would prefer a solution that is less invasive and is less likely to do any harm to their processes.

> *Lesson 1: To help with the engagement of our research, we find that a less intrusive approach is much easier to be accepted.*

After having discussions with the practitioners, we find that some of the tests are implemented with a special purpose. For example, a test may be particularly designed to capture a very rare but critical issue. In such cases, testers should not skip those tests since no other tests aim to detect the bugs for those scenarios.

> *Lesson 2: Tests that are shown to be ineffective may be particularly designed for a special purpose as opposed to being obsolete. Test selection and prioritization techniques should consider these special cases.*

Challenge 2: Coping with false-positive test failures

Description: Failures in test results may not always be associated with software issues. In the testing processes of our industrial partner, we often observe test failures that are false-positives. For example, due to the complicated hardware and test infrastructure involved in testing the code in Ericsson, the test failures may just be associated with a hardware failure or a noise in the system as opposed to an actual problem in the main product code. Such false-positives introduce challenges to our test selection and prioritization techniques. While learning the history of test executions, we need to take the false-positives into consideration. In addition, our approach should prioritize the tests that give fewer false-positive failures.

Solution: In order to address the false-positives in the test results, we first incorporated analysis of the test logs in order to assist with determining whether a test failure is due to the test infrastructure or a real product issue. After we identified the false-positives in our test results, we changed the existing test selection and prioritization approaches to consider the existence of false-positives. The original *Microsoft's FreqSelect* and *Microsoft's AssocSelect* consider all test failures. However, our adopted approaches *FreqSelect* and *AssocFreqSelect* only consider the test execution history with true-positive test failures. Hence our approaches favor selection and prioritization of the tests that provide a higher number of true-positive test failures.

Lessons learned: We observed a prevalence of false-positive test failures in our experiments. Particularly, in certain software systems, the field environment can be noisy and complicated. Therefore, the testing infrastructure and the environment may contribute to a significant amount of test failures as false-positives.

> *Lesson 3: Proactively addressing the false-positives in the test results can help practitioners in accepting our suggestions for test selection and prioritization techniques.*

Challenge 3: Trade-off between slip-through test failures and the tests costs

Description: Reducing the cost of the tests can lead to an increase in the chance of having slip-throughs, i.e. having more bugs in the system that are left undetected due to insufficient software testing. The general goal for test selection and prioritization is to minimize both the costs of tests and also the number of slip-throughs. However, since there is a trade-off between the two, it is challenging to decide how much slip-throughs are accepted when aiming for reducing the costs of tests.

Solution: Although we provide different approaches for test selection and prioritization, there is no gold standard to help us decide what is a good trade-off between the slip-through rate and the costs. First, we tried to leverage the approach proposed by Herzig et al. [8] in order to generalize both the slip-throughs and the test costs into one consolidated cost metric. Moreover, we conducted meetings with practitioners to seek policies that may especially have been put in place for making decisions about the slip-through trade-offs.

Lessons learned: We realized that practitioners do not always have a strong opinion against having slip-throughs, with the argument that some bugs will be certainly caught in the following test stages and it may be more economical to let certain bugs slip through certain test stages, and be caught later on in another stage.

> *Lesson 4: Slip-through test failures are not always a negative phenomenon. Practitioners may prefer to lower the costs of the tests at certain stages and use other testing strategies to catch the slip-through failures in other stages of the tests flow.*

Challenge 4: Coping with test dependencies

Description: Tests are observed to be frequently co-failing during our experiments. Our approach *AssocFreqSelect* aims to identify the tests that often co-fail in order to reduce the testing costs. Intuitively, the co-failure of multiple tests may be due to test dependencies. Test dependencies are often considered harmful in prior research [16]. For example, if two tests have a dependency on ordering (Test A needs to be executed before Test B), we may not be able to leverage our test selection and prioritization technique to reduce the testing costs.

Solution: We aim to identify and resolve those dependencies to improve the quality of the tests. However, in most of

the associations that were identified by our *AssocFreqSelect* approach, we could not find any functional dependencies among the tests.

By manually examining the tests and conducting deeper investigations with the practitioners, we find that the tests that are found to co-fail with each other, are often both dependent on some common external resource. For example, it can be the case that both tests depend on a common hardware component that is currently failing to load. This can lead both of the tests to fail at the same time.

Current research on test dependencies does not focus on the cases where external resources are the root causes of the test co-failures. Therefore, we had to manually label the tests with their external dependencies with the relevant hardware to help with maintaining the tests.

Lessons learned: There exist many tests in our study that co-fail with each other, even though they may be functionally independent or may be considered irrelevant to each other. We find research efforts may be allocated to provide more sophisticated static analysis techniques to automatically identify the root causes of associations between the tests, leading to a more optimized approach in test selection and prioritization.

> *Lesson 5: Tests that co-fail may not be functionally dependent on each other. Instead, they may co-fail due to their co-dependence on some external resources.*

Challenge 5: Deciding on the granularity of the tests

Description: There exist different granularity levels for testing software code. Unit tests are more focused and localized while end-to-end system tests have a larger scope and scale. It is sometimes challenging for the practitioners to decide the right granularity level of each testing process. On one hand, the focused unit tests are less costly to run and investigate. On the other hand, the end-to-end system tests may be more realistic and be more effective in catching bugs that are more likely to exist in real environments.

Solution: We analyzed the associations obtained from our co-failure analysis of our *AssocFreqSelect* approach, with the goal of examining whether there exist any associations among the lower level unit tests and the more expensive end-to-end system tests. If so, practitioners may consider executing only the unit tests in order to save the costs associated with the end-to-end system tests.

Lessons learned: We find that there exists an insignificant overlap among the unit tests and the end-to-end system tests in our studied subsystem. In particular, we find that out of all true-positive test failures that are detected in the test results, 80% of them are detected by the end-to-end system tests and only 34% of them are detected by the unit tests. Such results show that 1) The more expensive end-to-end system tests capture most of the true-positive test failures and 2) The two test categories do not overlap heavily since only 13% of the failures are detected by both of the categories.

> *Lesson 6: The test failures from the unit tests and the end-to-end system tests have little overlap. Removing the more expensive tests in this case in order to reduce costs may lead to major consequences in the overall product quality.*

VI. Threats to Validity

In this section, we discuss the threats to the validity of the findings of this paper.

External validity. Our study is only conducted on one area of the testing processes of one of the subsystems of Ericsson. Although *ELS* may carry many characteristics that are common in large-scale software systems, the findings and experiences may not be generalizable. For example, many of our findings are associated with the complex and expensive testing infrastructure that is specially designed for *ELS*. In addition, our results are only based on the simulation of six months testing results. However, adopting test selection and prioritization techniques in systems with shorter test execution history may result in different experiences. Finally, the parameters that are used for tuning our adopted approaches may be only suitable for *ELS*.

Construct validity. The evaluation of our adopted approaches is based on the simulation of the testing processes. Therefore, the actual quality of the system has not been evaluated by incorporating our approaches in practice. In addition, the goals of our test selection and prioritization approaches may not always align with the goals of the practitioners. For example, our association based test selection approach aims to reduce the redundancy among the tests while practitioners may opt to favor those redundancies to retrieve more information about the test failures. Further research may be focused on evaluating such adopted approaches considering different goals of the practitioners.

The parameters used in cost functions in the *TheoSelect* approach are either estimated by the practitioners or adopted from the *Microsoft's Theo* approach. These parameters may not always exactly match the reality. Sensitivity analysis on the parameters may complement our findings and experiences.

Our test evaluation is based on training on all data that is available prior to the testing day. However, the testing results that are closer to the testing day may have a better power to predict the testing results. Our future research will evaluate the best duration of training data in an industrial testing environment. Our test evaluation simulation has an assumption for independence among the tests, meaning tests can be re-ordered or skipped without impacting others. However, as observed by Zhang et al. [16] this assumption may not always hold. Future research by leveraging our approach in practice may consider the dependencies among the tests for better test selection and prioritization results.

Internal validity. Our approach relies on the labels that are provided by practitioners to decide whether a test failure is a false-positive. These labels may be inaccurate and inconsistent due to the level of experience and subjective bias of the

practitioners. Further validation of the labels may complement the results of our study.

The simulation of evaluating the adopted approaches does not consider the impact of the approaches on batches of commits which are being tested. For example, skipping certain tests may lead to a different testing process. However, our simulation cannot change the execution of tests in history. Future research may study such impacts by using the adopted approaches in practice to select and prioritize tests on the fly.

VII. Related Work

Tremendous research efforts have been dedicated to improving test effectiveness. The effectiveness of the tests is optimized typically by test case minimization, selection, and prioritization, as proposed by Laali et al. [17].

Static analysis and test coverage are leveraged as the major source of information to improve the effectiveness of the tests [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28]. For example, in a recent work, Saha et al. [29] propose to address the test prioritization problem by reducing it to a standard information retrieval problem. They assume that test cases and source code usually embody meaningful identifiers and comments which can be treated as natural language. Therefore, information retrieval techniques can be utilized on them to give priority values for running the tests. Nardo et al. [21] evaluate seven coverage-based test improvement techniques on a common carefully designed industrial system. Their findings show that finer grained coverage information can be leveraged to provide 79.5% savings in execution costs while maintaining a fault detection capability level above 70%. On the other hand, Koochakzadeh et al. claim that the test coverage information itself can be misleading for eliminating the tests redundancy and can result in weaker test cases [30], [31]. Despite prior research, static analysis and test coverage based approaches are not suitable for our subject system. Due to the complexity of our subject system, the source code is often not or only partially available. Therefore, in this paper, we adopted approaches that depend on historical test execution information.

Historical information about the test executions is a valuable source for improving test effectiveness. A recent work by Zhu et al. [6] demonstrates the use of learning co-failures in test execution history to assist with test prioritization. Their approach prioritizes the tests by using the co-failure history of the tests and information about the tests that just recently failed at any moment. Noor et al. [18] build logistic regression models from the test executions history in order to rank the effectiveness of test cases. Anderson et al. [32] propose a classification based approach for predicting the test failures based on the test executions history.

Research also leverages search-based techniques to optimize test effectiveness. Panichella et al. [33] propose an approach for test case selection using a customized genetic algorithm named Diversity based Genetic Algorithm (DIV-GA). Their approach shows to improve the current state of the art by diversifying the solutions (sub-sets of the test suites) generated during the search process. Multi-objective based approaches are used in prior research by Tyagi et al. [34] and Souza et al. [35] to assist with test prioritization and selection. Both pieces of research use the multi-objective particle swarm optimization technique. Their objective functions include covering more faults, maximizing the test coverage and minimizing the execution time. We do not consider the use of a search-based approach due to its limitation on scalability and the need for optimizing a large number of tests for the subject system.

Empirical studies are conducted on the testing practices. Labuschagne et al. [36] propose a study for cost measurement of regression tests in practice. They study 61 Java projects running on Travis CI and find that 18% of test suite executions fail and that 13% of these failures are flaky. Among the non-flaky failures, only 74% were true-positives and the remaining 26% were false-positives. Their study emphasizes the importance of the works like ours for improving test effectiveness of continuous integration flows in large software projects.

VIII. Conclusion

Software testing consumes a significant amount of resources in software projects. Therefore, it is of major importance for our industrial partner to improve its test effectiveness. In this paper, we adopt and customize multiple test selection and prioritization techniques from prior research to improve the effectiveness of testing processes for *ELS*. By simulating the use of our adopted approaches on six months of testing data, we demonstrated the effectiveness of our approaches in reducing the costs, and test execution time. Our results show that our standalone test prioritization approach significantly outperforms all of our selection approaches, the combination of our prioritization and selection approaches and also the random order. Therefore, we conclude that test prioritization is the most effective and the least invasive approach for saving costs in testing processes. However, test selection approaches can also be used in scenarios where a large number of slip-throughs are acceptable. More importantly, we documented our challenges and lessons learned as an experience report. Such information is valuable for practitioners who are willing to adopt test selection and prioritization techniques into their day-to-day workflow. Our findings highlight the opportunities and challenges involved in leveraging test execution history for improving test effectiveness in both research and practice.

Acknowledgment

We would like to thank Ericsson for providing access to their system used in our study. The findings and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of Ericsson and/or its subsidiaries and affiliates. Moreover, our results do not reflect the quality of Ericssons products.

References

[1] L. Padgham, Z. Zhang, J. Thangarajah, and T. Miller, "Model-based test oracle generation for automated unit testing of agent systems," *IEEE Transactions on Software Engineering*, vol. 39, no. 9, pp. 1230–1244, Sept 2013.

[2] D. Marijan, A. Gotlieb, and S. Sen, "Test case prioritization for continuous regression testing: An industrial case study," in *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ser. ICSM '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 540–543.

[3] K. Beck, *Extreme Programming Explained: Embrace Change*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

[4] P. Duvall, S. M. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2007.

[5] R. Potvin and J. Levenberg, "Why google stores billions of lines of code in a single repository," *Commun. ACM*, vol. 59, no. 7, pp. 78–87, Jun. 2016.

[6] Y. C. Zhu, E. Shihab, and P. Rigby, "Test re-prioritization in continuous testing environments," in *Proceedings of the 34th International Conference on Software Maintenance and Evolution*, ser. ICSME, September.

[7] J. Anderson, S. Salem, and H. Do, "Improving the effectiveness of test suite through mining historical data," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 142–151.

[8] K. Herzig, M. Greiler, J. Czerwonka, and B. Murphy, "The art of testing less without sacrificing quality," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 483–493.

[9] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 235–245.

[10] J.-M. Kim and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: ACM, 2002, pp. 119–129.

[11] A. Zeller, "Yesterday, my program worked. today, it does not. why?" in *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-7. London, UK, UK: Springer-Verlag, 1999, pp. 253–267.

[12] R. Kazmi, D. N. A. Jawawi, R. Mohamad, and I. Ghani, "Effective regression test case selection: A systematic literature review," *ACM Comput. Surv.*, vol. 50, no. 2, pp. 29:1–29:32, May 2017.

[13] D. Suleiman, M. Alian, and A. Hudaib, "A survey on prioritization regression testing test case," in *2017 8th International Conference on Information Technology (ICIT)*, May 2017, pp. 854–862.

[14] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, A. I. Verkamo *et al.*, "Fast discovery of association rules." *Advances in knowledge discovery and data mining*, vol. 12, no. 1, pp. 307–328, 1996.

[15] T. Mende and R. Koschke, "Revisiting the evaluation of defect prediction models," in *PROMISE*, 2009.

[16] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin, "Empirically revisiting the test independence assumption," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 385–396.

[17] M. Laali, H. Liu, M. Hamilton, M. Spichkova, and H. W. Schmidt, "Test case prioritization using online fault detection information," in *Reliable Software Technologies – Ada-Europe 2016*, M. Bertogna, L. M. Pinho, and E. Quiñones, Eds. Cham: Springer International Publishing, 2016, pp. 78–93.

[18] T. B. Noor and H. Hemmati, "Studying test case failure prediction for test case prioritization," in *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, ser. PROMISE. New York, NY, USA: ACM, 2017, pp. 2–11.

[19] A. Shi, T. Yung, A. Gyori, and D. Marinov, "Comparing and combining test-suite reduction and regression test selection," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 237–247.

[20] S. Wang, J. Nam, and L. Tan, "Qtep: Quality-aware test case prioritization," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 523–534.

[21] D. D. Nardo, N. Alshahwan, L. Briand, and Y. Labiche, "Coverage-based regression test case selection, minimization and prioritization: a case study on an industrial system," *Software Testing, Verification and Reliability*, vol. 25, no. 4, pp. 371–396.

[22] M. Kumar, A. Sharma, and R. Kumar, "An empirical evaluation of a threetier conduit framework for multifaceted test case classification and selection using fuzzyant colony optimisation approach," *Software: Practice and Experience*, vol. 45, no. 7, pp. 949–971.

[23] P. Konsaard and L. Ramingwong, "Total coverage based regression test case prioritization using genetic algorithm," in *2015 12th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, June 2015, pp. 1–6.

[24] S. Singh and F. Sahib, "Optimized test case prioritization with multi criteria for regression testing," 2014.

[25] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, Nov 2012, pp. 11–20.

[26] N. Kukreja, W. G. J. Halfond, and M. Tambe, "Randomizing regression tests using game theory," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov 2013, pp. 616–621.

[27] C. Nguyen, P. Tonella, T. Vos, N. Condori, B. Mendelson, D. Citron, and O. Shehory, "Test prioritization based on change sensitivity: an industrial case study," 2014.

[28] X. Qu, M. B. Cohen, and G. Rothermel, "Configuration-aware regression testing: An empirical study of sampling and prioritization," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08. New York, NY, USA: ACM, 2008, pp. 75–86.

[29] R. K. Saha, L. Zhang, S. Khurshid, and D. E. Perry, "An information retrieval approach for regression test prioritization based on program changes," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 268–279.

[30] N. Koochakzadeh and V. Garousi, "A tester-assisted methodology for test redundancy detection," *Adv. Soft. Eng.*, vol. 2010, pp. 6:1–6:13, Jan. 2010.

[31] N. Koochakzadeh, V. Garousi, and F. Maurer, "Test redundancy measurement based on coverage information: Evaluations and lessons learned," in *2009 International Conference on Software Testing Verification and Validation*, April 2009, pp. 220–229.

[32] J. Anderson, S. Salem, and H. Do, "Striving for failure: An industrial case study about test failure prediction," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2, May 2015, pp. 49–58.

[33] A. Panichella, R. Oliveto, M. D. Penta, and A. De Lucia, "Improving multi-objective test case selection by injecting diversity in genetic algorithms." *IEEE Transactions on Software Engineering*, vol. 41, no. 4, pp. 358 – 383, 2015.

[34] M. Tyagi and S. Malhotra, "Test case prioritization using multi objective particle swarm optimizer," in *2014 International Conference on Signal Propagation and Computer Technology (ICSPCT 2014)*, July 2014, pp. 390–395.

[35] L. S. de Souza, R. B. C. Prudłncio, and F. d. A. Barros, "A comparison study of binary multi-objective particle swarm optimization approaches for test case selection," in *2014 IEEE Congress on Evolutionary Computation (CEC)*, July 2014, pp. 2164–2171.

[36] A. Labuschagne, L. Inozemtseva, and R. Holmes, "Measuring the cost of regression testing in practice: A study of java projects using continuous integration," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 821–830.