2013 IEEE International Conference on Software Maintenance

Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study

Dusica Marijan, Arnaud Gotlieb, Sagar Sen
Certus Software V&V Centre
Simula Research Laboratory
Email: dusica, arnaud, sagar@simula.no

***Abstract*—Regression testing in continuous integration environment is bounded by tight time constraints. To satisfy time constraints and achieve testing goals, test cases must be efficiently ordered in execution. Prioritization techniques are commonly used to order test cases to reflect their importance according to one or more criteria. Reduced time to test or high fault detection rate are such important criteria. In this paper, we present a case study of a test prioritization approach ROCKET (Prioritization for Continuous Regression Testing) to improve the efficiency of continuous regression testing of industrial video conferencing software. ROCKET orders test cases based on historical failure data, test execution time and domain-specific heuristics. It uses a weighted function to compute test priority. The weights are higher if tests uncover regression faults in recent iterations of software testing and reduce time to detection of faults. The results of the study show that the test cases prioritized using ROCKET (1) provide faster fault detection, and (2) increase regression fault detection rate, revealing 30% more faults for 20% of the test suite executed, comparing to manually prioritized test cases.**

***Keywords*—software testing; continuous integration; regression testing; test case prioritization; history-based prioritization**

I. Introduction

One key aspect of efficient continuous integration software development is a short feedback loop from code commits to test execution reports. In a limited testing time, an important challenge to address is a trade-off between (a) selecting test cases that have high fault detection ability and (b) maximizing the number of test cases that can be executed in available time. This challenge stems from our experience in testing industrial video conferencing systems (VCS), in collaboration with our partner. The VCS is developed in a *continuous integration* environment. Changes made by developers are merged to the mainline VCS codebase on a daily basis. Testing is performed each time a new change is proposed to the VCS codebase. The software system is built and regression tested for every single change before it is committed to the mainline. Execution of a single test case for the VCS requires about 30 minutes on an average and a maximum of 45 minutes. Executing a complete set of 100 test cases for one VCS product would require at least 2 days. Therefore, we ask how can we select and prioritize a subset of test cases that can detect most of the regression faults, while being executed within a time limit? This is the question we address in this paper with the approach called Prioritization for Continuous Regression Testing (ROCKET). We demonstrate ROCKET to efficiently prioritize test cases for our industrial VCS case study.

As a major advantage in many industrial settings, ROCKET approach does not require source code. It takes four inputs: (1) a test suite as a set of test cases, (2) a failure status and (3) execution time for each test case for several last test executions, and (4) an upper bound for the total time of testing. The prioritization algorithm computes a weight for each test case based on the distance of its failures from the current execution and the test execution time. The algorithm embeds domain-specific heuristics pertaining to a specific industrial setting. Therefore, ROCKET prioritizes the set of test cases based on test historical information and time limitations, using domain-specific heuristics.

We validate ROCKET in regression testing of commercial VCS by comparing to manual ordering performed by test engineers and random test ordering, and demonstrate that ROCKET improves effectiveness and time to detection of regression faults. In particular, the results show that test cases prioritized using ROCKET detect 30% more regression faults with 20% of the test suite executed, comparing to manually prioritized test cases. The results further show that test cases prioritized using ROCKET at the same time decrease test execution time for 40% with 20% of the test suite executed, comparing to manually prioritized test cases.

The main contributions of the paper are:

- An algorithm with tool implementation for ordering test cases in regression testing based on historical failure data, testing time constraints and domain-specific heuristics, so that the test cases with shorter time to execute and that revealed regression faults execute earlier.
- A case study to validate the effectiveness of the approach in continuous regression testing of industrial video conferencing software.

The paper is organized as follows. Section II provides background, gives motivation and reviews related work on regression testing. In Section III we describe regression test prioritization for continuous integration and our algorithm ROCKET. Section IV describes a case study of ROCKET in regression testing of industrial video conferencing software and presents the results of the study. In Section V we draw conclusion and give directions of future work.

II. Background and Related Work

In this section we highlight some of the challenges in continuous regression testing in industrial setting. We also review prior work on prioritization for regression testing and relate it to our industrial partner's context.

1063-6773/13 $26.00 © 2013 IEEE
DOI 10.1109/ICSM.2013.91

A. Continuous Regression Testing in Practice

The following scenario illustrates the challenges imposed on a VCS test engineer performing continuous regression testing. AB is a test engineer whose task is to regression test the latest release of VCS software product, before the software will be integrated to the mainline codebase. AB is given a collection of about 100 test cases, a test execution log containing failure history information and x-hour time frame to complete the task (time frame changes from task to task). The test cases are system-level, and testing does not require source code. This task imposes two challenges on AB. First, AB has to select test cases to detect as many regression faults in software modifications as possible. Second, testing is firmly restricted to a specific duration and AB has to select and execute tests to satisfy that constraint. His objective is to select as many shorter tests as possible, in order to increase the number of executed tests and possibly the number of detected faults.

B. Prioritization for Regression Testing

Test case prioritization has been extensively studied in literature. One class of the proposed prioritization techniques requires source code [1], [2] and such techniques were inapplicable to our needs. Sherriff presents the approach to prioritize tests by analysing effects of changes through singular value decomposition [3]. However, the approach does not consider time constraints common for regression testing in continuous integration environment. Srikanth proposes a technique that prioritizes tests based on the combination of fault detection rate over time and minimal test setup time, when switching configurations in execution [4]. This approach however does not consider time taken to execute tests. Bryce presents a metric for cost-based prioritization [5], using combinatorial interaction coverage as objective. Do presents a cost model for test prioritization and shows how different time constraints affect prioritization: time to setup testing, time to identify and repair obsolete tests, human time to inspect results [6], [7]. Our work aims at prioritization with time constraints, but only in test execution. Similarly, Walcott proposes a technique to reorder test suits in the presence of time constraints [8]. Still, this techniques considers that all tests have the same execution time. In addition to prioritization based on historical fault data and test execution cost, for our industrial partner it was important to incorporate domain-specific heuristics in the prioritization technique, due to specific industrial setting.

III. Prioritization for Continuous Regression Testing

At the basic level, the goal of regression test case prioritization is to find the execution order for the given set of test cases that optimizes a given objective function. The objective function in continuous integration environment of our industrial partner is (1) selecting test cases with highest consecutive failure rate for the given number of executions, and (2) maximizing the number of executed test cases while satisfying given testing time constraints.

In this context, we define our objective function g as follows. For the given set of test cases $S = \{S_1, S_2, ..., S_n\}$ and the failure status for each test case in S over the last m successive executions (if the test case has been executed before), $FS = \{\{fs_{1,1}, fs_{2,1}, ..., fs_{m,1}\}, \{fs_{1,2}, fs_{2,2}, ..., fs_{m,2}\}, ..., \{fs_{1,n}, fs_{2,n}, ..., fs_{m,n}\}\}$, we calculate the cumulative priority for each test case $P = \{p_{S_1}, p_{S_2}, ..., p_{S_n}\}$. Given the P and the test case execution time $Te = \{Te_1, Te_2, ..., Te_n\}$, we define the objective function as follows:

$$g = (maximize(p), minimize(Te))$$

Now, we define the problem of regression test case prioritization as finding the order of test cases from S, such that $(\forall S_i)(i = 1..n)g(S_i) \geq (g(S_{i+1})$.

One idea behind our objective function is that given a short test execution time during regression testing, an effective prioritization criterion should select test cases that proved to fail in the previous test executions following chronological manner. The highest failure weight corresponds to the failure exposed in the $(current - 1)$ execution and the failure in every precedent execution is weighted lower than the failure in its successive execution. While re-executing the test cases that failed in precedent intermediate execution is mandatory in regression testing, very often faults detected in the second or third last execution and corrected reoccur later, due to "quick bug fixes" or masking effects that prevent fault detection. Similarly, a successful precedent intermediate execution for a test case lowers its failure impact (i.e. priority). Another idea behind our objective function is that given a short test execution time during regression testing, in addition to selecting test cases that failed previously, an effective prioritization criterion should select test cases that execute quickly, so to increase the number of executed test cases, as every test case can potentially detect faults. If a test case takes a lot of time to execute, even if it revealed failure in the precedent intermediate execution, executing this test case will prevent other failing test cases with the same failure impact (weight) but with shorter execution time to execute. The following example justifies this prioritization objective. There are five test cases (T1 to T5) shown in Table 1, with the distribution of regression faults in five consecutive test executions (E1 to E5). For the sake of simplicity, a test case is considered to reveal the same fault across the executions. Each test case has a time unit (test execution time). One possible order of test cases that maximizes the coverage of regression faults from previous executions is $Po1 = [T1, T4, T3, T5, T2]$. Consider the case where the upper bound for executing tests is 7 time units. In this case only $T1$ will be executed, checking against only one fault. Now, if we order test cases with respect to historical fault data and execution time, e.g. $Po2 = [T5, T3, T4, T2, T1]$, four test cases can execute in given 7 time units, $T5$, $T3$, $T2$ and $T4$, checking against four faults. If higher priority is given to shorter test cases with the same high failure impact, the prioritization improves the fault detection effectiveness of testing.

TABLE I. Test case prioritization for continuous regression testing.

Test case	E1	E2	E3	E4	E5	Exe time
T1	×	×			×	5
T2			×			2
T3		×			×	1
T4			×	×		3
T5	×	×				1

A. ROCKET Algorithm

The algorithm input is fourfold: a set of n test cases to prioritize $S = \{S_1, S_2, ..., S_n\}$, the execution time for each test $Te = \{Te_1, Te_2, ..., Te_n\}$, a failure status for each test case in S over the last m successive executions $FS = \{FS_1, FS_2, ..., FS_n\}$, where $FS_i = \{fs_1, fs_2, ..., fs_m\}$, and available test execution time. The algorithm starts by assigning $zero$ priority to all test cases. Next, based on the FS, failure matrix MF of size $m \times n$ is created and filled such that

$$MF[i,j] = \begin{cases} 1, & \text{if } S_j \text{ passed in } (current - i) \text{ execution} \\ -1, & \text{if } S_j \text{ failed in } (current - i) \text{ execution} \end{cases}$$

The failure matrix is shown in Figure 1. Field values in the matrix specify for each test case from a test suite of size n whether the test case passed or failed during the last m successive executions. If the test case failed, $MF[i,j]$ is -1, otherwise $MF[i,j]$ is 1. Each row i in the matrix represents the distance of a test-suite execution from the current execution ($i = 0$). Distances are assigned weights ω_i that reflect the impact of the failure occurred in the test execution that is i steps far from the current test execution. The weight is assigned based on the following domain-specific heuristic:

$$\omega_i = \begin{cases} 0.7, & \text{if } i = 1 \\ 0.2, & \text{if } i = 2 \\ 0.1, & \text{if } i \geq 3 \end{cases}$$

$$MF_{m,n} = \begin{pmatrix} -1 & 1 & \cdots & 1 \\ 1 & 1 & \cdots & -1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & -1 \end{pmatrix}$$

Fig. 1. Test suite failure matrix.

Given the failure matrix MF and the test-suite execution distance weighting heuristic, the algorithm calculates the cumulative priority for each test case $P_S = \{p_{S_1}, p_{S_2}, ..., p_{S_n}\}$, as follows:

$$p_{Si} = \sum_{i=1..m} MF[i,j] * \omega_i$$

Next, the algorithm creates a matrix $MP_{s \times t}$ that classifies test cases from S into t classes, based on the calculated priority values. s is the size of the largest test case class (in terms of the number of test cases). All test cases in the same class have the same priority value, which increases by 1 for the successive class. While all test cases from the same class are equally relevant for the current test-suite execution, they differ by their execution time. Given the matrix $MP_{s \times t}$ and the test cases execution time $Te = \{Te_1, Te_2, ..., Te_n\}$, the algorithm checks for each test case if its execution time exceeds overall available testing time. If it does, the test case is assigned the priority value $t + 1$. Otherwise, the algorithm increases the priority of a test case p_{Si} by its execution time value normalized to $[0, 1]$, with respect to the maximal test case execution time T_{max} in S:

$$p_{Si} = \begin{cases} t + 1, & Te_i \geq T_{max} \\ p_{Si} + \dfrac{Te_i}{T_{max}}, & otherwise \end{cases}$$

ROCKET has been implemented in a tool developed in collaboration with our industrial partner's test department. The tool is aimed for automating test selection and scheduling in continuous integration.

IV. Industrial Case Study

In this section we present a case study to validate the effectiveness of ROCKET in continuous regression testing of industrial software. We designed the experiments to answer the following research questions:

- RQ1: Can ROCKET-prioritized test suite increase the number of test cases executed in limited testing time, compared to manually prioritized one?
- RQ2: Can ROCKET-prioritized test suite at the same time increase the number of detected regression faults when only part of the test suite has been executed, compared with manually ordered test suite?

A. Software Studied

The studied VCS belongs to a class of configurable systems (10 products) that share many commonalities in structure and functionality and differ in features used to customize the software for specific user needs. The VCS testing team consists of 10 engineers that are responsible for testing final product releases with monthly frequency, and regression testing of feature updates at a daily base. Updates made to software features that are hardware dependent affect more products in a class and need to be tested for all affected products.

B. Methodology

In the experiments we used the data collected during 5 consecutive test case executions for the VCS ($E1$ to $E5$). Data include: a test-suite (the same test suite was used in all 5 executions), failing test cases for each execution and execution time for each test case. We analysed the data and built a matrix that shows how are failing test cases distributed over five consecutive regression test executions. For example, the test case T1 from Table 1 failed in test execution $E1$ and $E2$. Afterwards, the fault was fixed and T1 passed in execution $E3$ and $E4$, but due to regressions T1 failed again in execution $E5$. Using the failure information from the previous 5 test executions we ordered the test cases using ROCKET. To answer RQ1, we measured how much time will take to execute partial test suites for ROCKET-prioritized and manually-prioritized suites. To answer RQ2, we measured regression fault detection capability, in terms of the number of faults detected by partial test suits, for both prioritization approaches.

Additionally, we compared prioritization effectiveness of ROCKET with manual prioritization and random ordering of test cases using the Cost-cognizant weighted Average Percentage of Faults Detected measure (APFDC) [9]. APFDC rewards test case orders proportionally to their rate of "units-of-fault-severity-detected-per-unit-test-cost". In our case, the cost of test cases is determined by their execution time. We calculated and compared APFDC for the test cases prioritized using ROCKET, manually prioritized and randomly ordered test sets (we used a median value for 100 sets of randomly ordered test cases).

C. Results

Figure 2a) compares the execution time for the partial manually-prioritized and ROCKET-prioritized test suits, starting at 20% of the initial test suite size, with increments of 20%. In other words, the experiment analyses how many test cases will execute in a limited testing time for these two approaches. For the first 20% of the test suite prioritized using ROCKET, execution time is 40% less than for the 20% of manually prioritized test suite. For the first 40% of the test suite prioritized using ROCKET, execution time is still 28% less than the time required to execute 40% of manually prioritized suite. These results indirectly show that ROCKET maximises the number of test cases that can run in available testing time, compared to manual approach, positively answering RQ1. As the size of a test suite grows, the difference in test execution time for the two approaches decreases and, as expected, equals zero after the whole test suite has been executed.

Figure 2b) compares the number of detected faults for the partial manually-prioritized and ROCKET-prioritized test suits, starting at 20% of the suite total size, with increments of 20%. For 20% of the test suite executed, the test cases prioritized by ROCKET are able to detect 30% more regression faults comparing to manually prioritized test cases; 13 faults detected by ROCKET-prioritized test cases versus 10 faults detected by manually-prioritized test cases. These results show that ROCKET has higher regression fault detection rate compared to manual prioritization, positively answering RQ2.

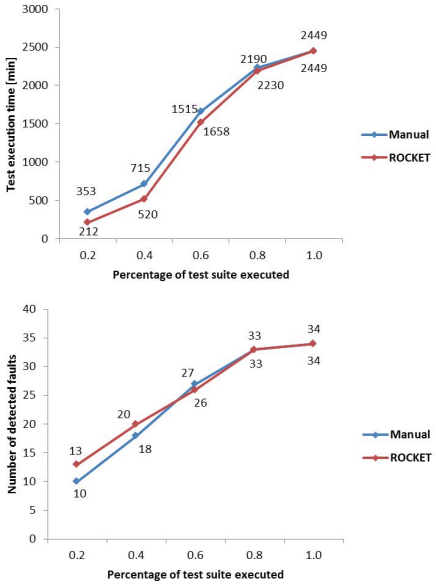

Fig. 2. a) Test execution time and b) Fault detection for manually-prioritized and ROCKET-prioritized test suite.

When using APFDC measure to compare the effectiveness in prioritizing regression tests for ROCKET approach versus manual prioritization and random ordering of test cases, we found that ROCKET outperforms other two prioritization techniques. In particular, $APFDC_{ROCKET} = 17.09$, $APFD_{manual} = 15.81$, and $APFD_{median\_random} = 13.85$. These results show that ROCKET has higher rate of regression fault detection per unit of test case cost (test execution time).

In summary, the results of the case study show that ROCKET efficiently prioritizes test cases for faster and more efficient regression fault detection, maximizing the number of executed test cases in limited period of time. This means that in cases where time required to execute complete test suite exceeds available testing time, using ROCKET will assure that as much test cases that test regression faults will execute.

D. Threats to Validity

In the study we used only one VCS software product to evaluate ROCKET with one test team, being a threat to external validity of our findings. However, the used VCS belongs to a class of configurable systems that are characterized by a high level of commonality between products. Therefore, we assumed that the VCS we used is representative of all VCS products and that regression testing process in other test teams does not significantly differ.

V. Conclusion

In this paper we presented the approach that prioritizes test cases for continuous regression testing, ROCKET, so that the tests that take shorter time to execute and have higher regression fault detection capability execute earlier. When applied to industrial continuous regression testing of video conferencing software, ROCKET showed to improve the efficiency and time to detection of regression faults, compared with manual and random test ordering. In future work we will perform more thorough evaluation of ROCKET with another testing team and on more VCS products, and especially for a longer period of usage, after the tool featuring ROCKET has been completely adopted and internalized with our partner. Later, we will extend the objective function to multiple prioritization criteria, such as the cost to fix the failure and the cost of switching test cases in execution that require manual intervention.

Acknowledgment

The authors thank to Marius Liaaen from Cisco. This work is supported by the Research Council of Norway.

References

[1] A. Srivastava and J. Thiagarajan, "Effectively prioritizing tests in development environment," in *Int. symp. on Soft. testing and analysis*, 2002.

[2] A. Orso, T. Apiwattanapong, and M. J. Harrold, "Leveraging field data for impact analysis and regression testing," in *ESEC*, 2003, pp. 128–137.

[3] M. Sherriff, M. Lake, and L. Williams, "Prioritization of regression tests using singular value decomposition with empirical change records," in *Proc. of the The 18th IEEE Int. Symposium on Software Reliability*, 2007.

[4] H. Srikanth, M. B. Cohen, and X. Qu, "Reducing field failures in system configurable software: cost-based prioritization," in *ISRE*, 2009.

[5] R. C. Bryce, S. Sampath, J. B. Pedersen, and S. Manchester, "Test suite prioritization by cost-based combinatorial interaction coverage," *Int. Journal Systems Assurance Eng. and Management*, pp. 126–134, 2011.

[6] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel, "An empirical study of the effect of time constraints on the cost-benefits of regression testing," in *Proc. of the 16th Int. Symp. on Foundations of software eng.*, 2008.

[7] H. Do and G. Rothermel, "Using sensitivity analysis to create simplified economic models for regression testing," in *ISTTA*, 2008.

[8] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Timeaware test suite prioritization," in *ISSTA*, 2006.

[9] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *Proceedings of the 23rd Int. Conference on Software Engineering*, 2001.