REMAP: Using Rule Mining and Multi-objective Search for Dynamic Test Case Prioritization

Conference Paper · April 2018

DOI: 10.1109/ICST.2018.00015

CITATIONS
8

READS
169

5 authors, including:

Dipesh Pradhan
Simula Research Laboratory
15 PUBLICATIONS **107** CITATIONS
SEE PROFILE

Shuai Wang
Simula Research Laboratory
47 PUBLICATIONS **566** CITATIONS
SEE PROFILE

Tao Yue
Simula Research Laboratory
113 PUBLICATIONS **1,341** CITATIONS
SEE PROFILE

Marius Liaaen
Cisco Systems Norway
28 PUBLICATIONS **309** CITATIONS
SEE PROFILE

REMAP: Using Rule Mining and Multi-Objective Search for Dynamic Test Case Prioritization

Dipesh Pradhan[1,2], Shuai Wang[1], Shaukat Ali[1], Tao Yue[1,2], Marius Liaaen[3]
[1]Simula Research Laboratory, Oslo, Norway
[2]University of Oslo, Oslo, Norway, [3]Cisco Systems, Oslo, Norway
{dipesh, shuai, shaukat, tao}@simula.no, marliaae@cisco.com

***Abstract*— Test case prioritization (TP) prioritizes test cases into an optimal order for achieving specific criteria (e.g., higher fault detection capability) as early as possible. However, the existing TP techniques usually only produce a static test case order before the execution without taking runtime test case execution results into account. In this paper, we propose an approach for black-box dynamic TP using rule mining and multi-objective search (named as REMAP). REMAP has three key components: 1) *Rule Miner*, which mines execution relations among test cases from historical execution data; 2) *Static Prioritizer*, which defines two objectives (i.e., fault detection capability (*FDC*) and test case reliance score (*TRS*)) and applies multi-objective search to prioritize test cases statically; and 3) *Dynamic Executor and Prioritizer*, which executes statically-prioritized test cases and dynamically updates the test case order based on the runtime test case execution results. We empirically evaluated REMAP with random search, greedy based on *FDC*, greedy based on *FDC* and *TRS*, static search-based prioritization, and rule-based prioritization using two industrial and three open source case studies. Results showed that REMAP significantly outperformed the other approaches for 96% of the case studies and managed to achieve on average 18% higher Average Percentage of Faults Detected (*APFD*).**

Keywords*— multi-objective optimization; rule-mining; dynamic test case prioritization; search; black-box regression testing.

I. INTRODUCTION

While developing modern software systems in industry, regression testing is frequently applied to re-test the modified software to ensure that no new faults are introduced [1-3]. However, regression testing is an expensive maintenance process that can consume up to 80% of the overall testing budgets [4, 5]. Test case prioritization (TP) is one of the most widely used approaches to improve regression testing with the aim to prioritize a set of test cases into an optimal order for achieving certain criteria (e.g., fault detection capability) as early as possible [6-9].

However, the existing TP techniques [7, 10-14] usually produce a list of static prioritized test cases for execution, i.e., the order of prioritized test cases will not be updated based on the runtime test case execution results. Based on our collaboration with Cisco Systems [15, 16] focusing on cost-effectively testing video conferencing systems (VCSs), we noticed that there may exist underlying relations among the executions of test cases, which test engineers are not aware of when developing these test cases. For example, when the test case (T_1) that tests the amount of free memory left in VCS after pair to pair communication for a certain time (e.g., 5 minutes) *fails*, the test case (T_2) verifying that the speed of fan in VCS locks near the speed set by the user always *fails* as well. This is in spite of the fact that all test cases are supposed to be executed independently of one another. Moreover, we noticed that when T_2 is executed as *pass*, another test case (T_3) that checks the CPU load measurement of VCS also always *passes*. Thus, we argue that mining such execution relations among test cases based on historical execution data can improve the effectiveness of TP. For instance, if T_1 is executed as *fail*, T_2 should be executed as early as possible (e.g., execute T_2 right after T_1) since T_2 has a high chance to *fail*, i.e., detect a fault. Additionally, if T_2 is executed as *pass*, T_3 should be executed later (i.e., the priority of T_3 should be decreased), since T_3 has a high possibility to *pass*. Therefore, it is essential to prioritize the test cases in a dynamic manner considering the runtime test case execution results, which have not been properly addressed [14, 17-19].

With such motivation in mind, this work proposes a black-box TP approach named as REMAP, which uses rule mining and search to prioritize test cases in the absence of source code. REMAP consists of three key components, i.e., *Rule Miner* (*RM*), *Static Prioritizer* (*SP*) and *Dynamic Executor and Prioritizer* (*DEP*). First, *RM* defines *fail rules* and *pass rules* for representing the execution relations among test cases and mines these rules from the historical execution data. Second, *SP* defines two objectives: *Fault Detection Capability* (FDC) and *Test Case Reliance Score* (TRS), and applies multi-objective search to statically prioritize test cases. FDC measures the rate of failed execution of a test case in a given time, while TRS measures the extent to which the result of a test case can be used to predict the result of other test cases (based on the mined rules from *RM*), e.g., the execution result of T_1 (as *fail*) can help to predict the execution result of T_2(as *fail*). Third, *DEP* executes the statically prioritized test cases obtained from the *SP* and dynamically updates the test case order based on the runtime test case execution results, and *fail rules* and *pass rules* from *RM*.

To empirically evaluate REMAP, we employed a total of five case studies (two industrial ones and three open source ones): 1) two data sets from Cisco related with VCS testing, 2) two open data sets from ABB Robotics for Paint Control [20] and IOF/ROL [20], and 3) one open Google Shared Dataset of Test Suite Results (GSDTSR) [21]. We compared REMAP with 1) three baseline TP approaches, i.e., random search (RS)

[10, 12, 22] and two prioritization approaches based on Greedy [6, 8, 23]: G-1 based on FDC and G-2 based on FDC and TRS, 2) search-based TP (SBTP) approach [24-26], and 3) rule-based TP (RBP) approach [27] (using the mined rules from *RM*, i.e., *fail rules* and *pass rules*).

To assess the quality of the prioritized test cases, we use the widely applied evaluation metric, i.e., Average Percentage of Faults Detected ($APFD$) [6, 7]. The results showed that the test cases prioritized by REMAP performed significantly better than 1) RS, G-2, SBTP, and RBP for all the five case studies, and 2) G-1 for four case studies. In general, as compared to the five approaches, REMAP managed to achieve on average 18% (i.e., 33.4%, 14.5%, 17.7%, 14.9%, and 9.8%) higher values of $APFD$ for the five case studies.

The key contributions of this paper include:

- We define two types of rules (*fail rule* and *pass rule*) for representing execution relations among test cases and apply a rule mining algorithm (i.e., Repeated Incremental Pruning to Produce Error Reduction [28]) to mine these rules;
- We define two objectives (i.e., FDC and TRS) and integrate these objectives into search to statically prioritize test cases;
- We propose a dynamic approach using the mined rules to update the statically-prioritized test cases based on the runtime test case execution results;
- We empirically evaluate REMAP with five existing TP approaches using five case studies.

The rest of the paper is organized as follows. Section II motivates our work followed by a formalization of the TP problem in Section III. Section IV presents REMAP in detail, and Section V details the experiment design followed by presenting the results in Section VI. Section VII presents the related work, and Section VIII concludes this paper.

II. Motivating Example

We have been collaborating with Cisco Systems, Norway for more than nine years with the aim to improve the quality of their video conferencing systems (VCSs) [15, 16, 25]. The VCSs are used to organize high-quality virtual meetings without gathering the participants to one location. These VCSs are used in many organizations such as IBM and Facebook. Each VCS has on average three million lines of embedded C code, and they are developed continuously. Testing needs to be performed each time the software is modified. However, it is not possible to execute all the test cases since there is only a limited time for execution and each test case requires much time for execution (e.g., 10 minutes). Thus, the test cases need to be prioritized to execute the important test cases (i.e., test cases most likely to fail) as soon as possible.

Through further investigation of VCS testing, we noticed that a certain number of test cases turned to *pass*/*fail* together, i.e., when a test case passes/fails some other test case(s) passed/failed almost all the time (with more than 90% probability). This is despite the fact that the test cases do not depend upon one another when implemented and are supposed to be executed independently. Moreover, the test engineers from our industrial partner are not aware of these execution relations (i.e., test case failing/passing together) when implementing and executing test cases. Note that we consider each failed test case is caused by a separate fault since the link between actual faults and executed test cases are not explicit in our context, and this has been assumed in other literature as well [9, 20, 27].

Fig. 1 depicts a running example to explain our approach with six real test cases from Cisco along with their execution results within seven test cycles. A test cycle is performed each time the software is modified where a set of test cases from the original test suite are executed. When a test case is executed, there exist two types of execution results, i.e., *pass* or *fail*. *Pass* implies that the test case did not find any fault(s) while *Fail* denotes that the test case managed to detect a fault(s). From Fig. 1, we can observe that the execution of test cases T_1, T_2, T_3, T_4, T_5, and T_6 failed four, two, three, three, one, and four times, respectively within the seven test cycles. Moreover, we can observe that there exist certain relations between the execution results of some test cases. For example, when T_1 is executed as *fail*/*pass*, T_3 is always executed as *fail*/*pass* and vice versa, when both of them were executed.

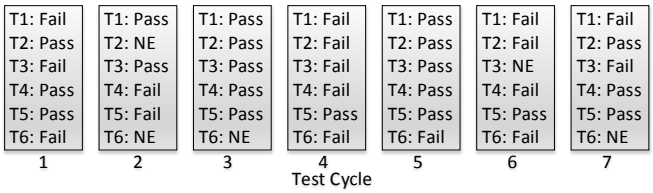

Fig. 1. Sample data showing the execution history for six test cases*.
*NE: Not Executed

Based on such an interesting observation, we can assume that internal relations for test case execution can be extracted from historical execution data, which can then be used to guide prioritizing test cases dynamically. More specifically, when a test case is executed as *fail* (e.g., T_1 in Fig. 1), the corresponding *fail* test cases (e.g., T_3 in Fig. 1) should be executed earlier as there is a high chance that the corresponding test case(s) is executed as *fail*. On the other hand, if a test case is executed as *pass* (e.g., T_4 in Fig. 1), the priority of the related *pass* test case(s) (T_2 in Fig. 1) need to be decreased (i.e., they need to be executed later). We argue that identifying such internal execution relations among test cases can help to facilitate prioritizing test cases dynamically, which is the key motivation of this work.

III. Basic Notations and Problem Representation

This section presents the basic notations (Section III.A) and problem representation (Section III.B).

A. Basic Notations

Notation 1. TS is an original test suite to be executed with n test cases, i.e., $TS = \{T_1, T_2, \dots, T_n\}$. For instance, in the running example (Fig. 1), TS consists of six test cases.

Notation 2. TC is a set of p test cycles that have been executed, i.e., $TC = \{tc_1, tc_2, \dots, tc_p\}$, such that one or more test cases are executed in each test cycle. For example, there are seven test cycles in the running example (Fig. 1), i.e., p=7.

Notation 3. For a test case T_i executed in a test cycle tc_f, T_i has two possible verdicts, i.e., $v_{ij} \in \{pass, fail\}$. For example in Fig. 1, the verdict for T_1 is *fail*, while the verdict

for T_2 is *pass* in tc_1. If a test case has not been executed in a test cycle, it has no verdict. For instance in Fig. 1, T_6 has no verdict (represented by NE) in tc_3. $V(T_i)$ is a function that returns the verdict of a test case T_i.

B. Problem Representation

Let S represents all potential solutions for prioritizing *TS* to execute, $S = \{s_1, s_2, \dots, s_q\}$, where $q = n!$. For instance in the running example with six test cases (Fig. 1), the total number of solutions $q = 720$. Each solution s_j is an order of test cases from *TS*, $s_j = \{T_{j1}, T_{j2}, \dots, T_{jn}\}$ where T_{ji} refers to a test case that will be executed in the i^{th} position for s_j.

Test Case Prioritization (TP) Problem: TP problem aims to find a solution $s_f = \{T_{f1}, T_{f2}, \dots, T_{fn}\}$ such that: $F(T_{fi}) \geq F(T_{fk}), where\ i < k \wedge 1 \leq i \leq n-1$ and $F(T_{fi})$ is an objective function that represents the fault detection capability of a test case in the i^{th} position for the solution s_f.

Dynamic TP Problem: The goal of dynamic TP is to dynamically prioritize test cases in the solution based on the runtime test case execution results to detect faults as soon as possible. For a solution $s_f = \{T_{f1}, T_{f2}, \dots, T_{fn}\}$, the dynamic TP problem aims to update the execution order of the test cases in s_f to obtain a solution $s_f{}' = \{T_{f1}{}', \dots, T_{fn}{}'\}$, such that $F(s_f{}') \geq F(s_f)$. $F(s_f{}')$ is a function that returns the value for the average percentage of fault detected for $s_f{}'$.

IV. Approach: REMAP

This section first presents an overview of our approach (named as REMAP) for dynamic TP that combines rule mining and search in Section IV.A followed by detailing its three key components (Section IV.B - Section IV.D).

A. Overview of REMAP

Fig. 2 presents an overview of REMAP consisting of three key components: *Rule Miner* (*RM*), *Static Prioritizer* (*SP*), and *Dynamic Executor and Prioritizer* (*DEP*). The core of *RM* is to mine the historical execution results of test cases and produce a set of execution relations (*Rules* depicted in Fig. 2) among test cases, e.g., if T_3 *fails* then T_1 *fails* in Section II. Afterwards, *SP* takes the mined rules and historical execution results of test cases as input to statically prioritize test cases for execution. Finally, *DEP* executes the statically prioritized test cases obtained from the *SP* one at a time and dynamically updates the order of the unexecuted test cases based on the runtime test case execution results (Fig. 2) in order to detect the fault(s) as soon as possible.

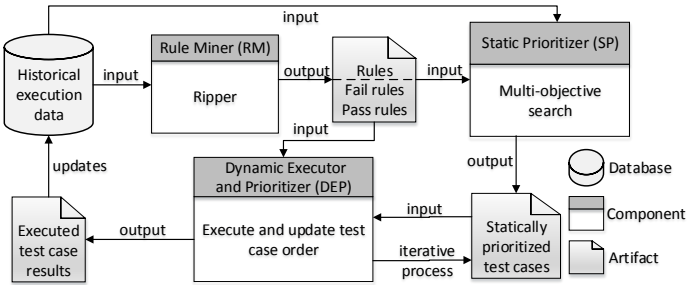

Fig. 2. Overview of REMAP.

B. Rule Miner (RM)

We first define two types of rules, i.e., *fail rule* and *pass rule* for representing the execution relations among test cases.

A fail rule specifies an execution relation between two or more test cases if a verdict of a test case(s) is linked to the *fail* verdict of another test case. The *fail rule* is formed as: $(V(T_i)\ AND, \dots, AND\ V(T_k)) \xrightarrow{fail} (V(T_j) = fail) \wedge T_j \notin \{T_i, \dots, T_k\}$, where $V(T_i)$ is a function that returns the verdict of a test case T_i (i.e., *pass* or *fail*). Note that for a *fail rule*, the execution relation must hold *true* for the specific test cases with more than 90% probability (i.e., confidence) as is often used in literatures [29-32]. For example in Fig. 1, there exists a *fail rule* between T_1 and T_3: $(V(T_1) = fail) \xrightarrow{fail} (V(T_3) = fail)$, i.e., when T_1 has a verdict *fail* (executed as *fail*), T_3 always has the verdict *fail* and this rule holds *true* with 100% probability in the historical execution data (Fig. 1). With *fail rules*, we aim to prioritize and execute the test cases that are more likely to *fail* as soon as possible. For instance, for the motivating example in Section II, T_3 should be executed as early as possible when T_1 is executed as *fail*.

A pass rule specifies an execution relation between two or more test cases if a verdict of a test case(s) is linked to the *pass* verdict of another test case. The *pass rule* is in the form: $(V(T_i)\ AND, \dots, AND\ V(T_k)) \xrightarrow{pass} (V(T_j) = pass) \wedge T_j \notin \{T_i, \dots, T_k\}$ where $V(T_i)$ is a function that returns the verdict of test case T_i (i.e., *pass* or *fail*). Similarly, for a *pass rule*, the execution relation must hold *true* with more than 90% probability (i.e., confidence) [29-32]. For instance in Fig. 1, there exists a *pass rule* between T_2 and T_4: $(V(T_4) = pass) \xrightarrow{pass} (V(T_2) = pass)$, i.e., when T_4 has a verdict *pass* (executed as *pass*), the verdict of T_2 is *pass* and this rule holds *true* with 100% probability based on the historical execution data (Fig. 1). With *pass rules*, we aim to deprioritize the test cases that are likely to *pass* and execute them as late as possible. For instance, for the motivating example, T_2 should be executed as late as possible when T_4 is executed as *pass*.

To mine *fail rules* and *pass rules*, our *RM* employs the Repeated Incremental Pruning to Produce Error Reduction (RIPPER) algorithm [28]. We used RIPPER since it is one of the most commonly applied rule-based algorithm [33], and it has performed well for addressing different problems (e.g., software defect prediction and feature subset selection) [32, 34-37]. More specifically, *RM* takes as input a test suite with *n* test cases, historical test case execution data (with a set of test cycles that lists the verdict of all the test cases). After that, *RM* produces a set of *fail rules* and *pass rules* as output. Fig. 3 demonstrates the overall process of *RM*.

Recall that there are two possible verdicts for each executed test case (Section III.A): *pass* or *fail*. For each test case in TS, *RM* uses the RIPPER algorithm to mine rules based on the historical execution data (i.e., TC) (lines 1-3 in Fig. 3). The detailed description of the RIPPER algorithm can be checked from [28, 38]. Afterwards, the mined rules are classified into a *fail rule* or *pass rule* one at a time (lines 4-8) until all of them are classified. If the verdict of the particular test case is *fail*, the rule related to this test case is classified as a *fail rule* (lines 5-6); otherwise, the rule is classified as a *pass rule* (lines 7 and 8 in Fig. 3). For instance, in Fig. 1, two rules

are mined for T_1 using RIPPER: $(V(T_3) = fail) \xrightarrow{fail} (V(T_1) = fail)$ as a *fail rule* and $(V(T_3) = pass) \xrightarrow{pass} (V(T_1) = pass)$ as a *pass rule*. This process of rule mining and classification is repeated for all the test cases in the test suite (lines 1-8 in Fig. 3), and finally, the mined *fail rule* and *pass rule* are returned (line 9 in Fig. 3).

Input: Set of test cycles $TC = \{tc_1, tc_2, \dots, tc_p\}$, set of test cases $TS = \{T_1, T_2, \dots, T_n\}$
Output: A set of *fail rules* and *pass rules*
Begin:
1 **for** ($T \in TS$) **do** // for all the test cases
2 $RuleSet \leftarrow \emptyset$
3 $RuleSet \leftarrow RIPPER(T, TC)$ // mine rules for T using TC
4 **for** ($rule\ in\ RuleSet$) **do** // for each mined rule
5 **if** ($V(T) = fail$) **then** // classify as *fail rule*
6 $fail\_rule \leftarrow fail\_rule \cup rule$
7 **else if** ($V(T) = pass$) **then** // classify as *pass rule*
8 $pass\_rule \leftarrow pass\_rule \cup rule$
9 return $(fail_{rule}, pass_{rule})$
End

Fig. 3. Overview of rule miner.

In this way, a set of *fail rules* and *pass rules* can be obtained from RM. For instance, in the motivating example (Section II), we can obtain four *fail rules* and three *pass rules* for the six test cases from the seven test cycles (Fig. 1) as shown in TABLE I.

TABLE I. FAIL RULE AND PASS RULE FROM THE MOTIVATING EXAMPLE

Fail Rule	***Pass Rule***
1. $(V(T_1) = fail) \xrightarrow{fail} (V(T_3) = fail)$	5. $(V(T_1) = pass) \xrightarrow{pass} (V(T_3) = pass)$
2. $(V(T_2) = fail) \xrightarrow{fail} (V(T_4) = fail)$	6. $(V(T_3) = pass) \xrightarrow{pass} (V(T_1) = pass)$
3. $(V(T_3) = fail) \xrightarrow{fail} (V(T_1) = fail)$	7. $(V(T_4) = pass) \xrightarrow{pass} (V(T_2) = pass)$
4. $(V(T_4) = fail) \xrightarrow{fail} (V(T_2) = fail)$	

C. Static Prioritizer (SP)

History-based TP techniques have been widely applied to prioritize the test cases based on their historical failure data, i.e., test cases that failed more often should be given higher priorities [12, 13, 20, 39, 40]. For example, T_1 and T_6 failed the highest number of times (i.e., four times) in Fig. 1, and therefore they should be given the highest priorities for execution out of the six test cases. Moreover, we argue that the execution relations among the test cases should also be taken into account for TP. The test cases whose results can be used to predict the results for more test cases (using *fail rules* and *pass rules*) need to be given higher priorities since their execution result can help predict the result of other test cases. For example in Fig. 1, the execution result of T_1 is related to one test case T_3, while the execution result of T_5 is not related to any test case as shown in TABLE I. Thus, it is ideal to execute T_1 earlier than T_5 since based on the execution result of T_1, the related test case (i.e., T_3) can be prioritized (if T_1 *fails*) or deprioritized (if T_1 *passes*).

Thus, our *Static Prioritizer* (*SP*) defines two objectives: Fault Detection Capability (FDC) and Test Case Reliance Score (TRS), and uses multi-objective search to statically prioritize test cases before execution (as shown in Fig. 2). We formally define two objectives to guide the search toward finding optimal solutions in detail below.

Fault Detection Capability (FDC): FDC for a test case is defined as the rate of failed execution of a test case in a given time period, and it has been frequently applied in the literature [13, 25]. The FDC for a test case is calculated as: $FDC_{Ti} = \frac{Number\ of\ times\ that\ T_i\ found\ a\ fault}{Number\ of\ times\ that\ T_i\ was\ executed}$. FDC of T_i is calculated based on the historical execution information of T_i. For instance, in Fig. 1, the FDC for T_4 is 0.43 since it found fault three times out of seven executions. The FDC for a solution s_j can be calculated as: $FDC_{sj} = \frac{\sum_{i=1}^{n} FDC_{Ti} \times \frac{n-i+1}{n}}{mfdc}$, where, s_j represents any solution j (e.g., $\{T_1, T_4, T_3, T_6, T_2, T_5\}$ in Fig. 1), $mfdc$ represents the sum of FDC of all the test cases in s_j, and n is the total number of test cases in s_j. Notice that a higher value of FDC implies a better solution. The goal is to maximize the FDC of a solution since we aim to execute the test cases that fail (i.e., detect faults) as early as possible.

Test Case Reliance Score (TRS): The TRS for a solution s_j can be calculated as: $TRS_{sj} = \frac{\sum_{i=1}^{n} TRS_{Ti} \times \frac{n-i+1}{n}}{otrps}$, where $otrps$ represents the sum of TRS of all the test cases in s_j, n is the total number of test cases in s_j, and TRS_{Ti} is the test case reliance score (TRS) for the test case T_i. The TRS for a test case T_i is defined as the number of unique test cases whose results can be predicted by executing T_i using the defined *fail rules* and *pass rules* extracted from *RM* (Section IV.B). For instance in Fig. 1, TRS for T_1 is 1 since the execution of T_1 can only be used to predict the execution result of T_3 based on the rule number 1 and 5 (TABLE I) while TRS for T_5 is 0 as there is no test case that can be predicted based on the execution result of T_5. The goal is to maximize the TRS of a solution since we aim to execute the test cases that can predict the execution results of other test cases as early as possible.

We further integrated these two objectives into the most widely used multi-objective search algorithm Non-dominated Sorting Genetic Algorithm II (NSGA-II) [41] that has been widely applied in different contexts [42-45]. Moreover, we chose the widely-used *tournament selection operator* [41] to select individual solutions with the best fitness for inclusion into the next generation. The *single point crossover operator* is used to produce the offspring solutions from the parent solutions by swapping some test cases in the parent solutions. *Swap mutation operator* is used to randomly change the order of test cases in the solutions based on the pre-defined mutation probability, e.g., $\frac{1}{n(t)}$, where $n(t)$ represents the size of the test suite. Note that *SP* produces a set of non-dominated solutions based on Pareto optimality [46-48] with respect to the defined objectives. The Pareto optimality theory states that a solution s_a dominates another solution s_b if s_a is better than s_b in at least one objective and for all other objectives s_a is not worse than s_b [42]. Note that we randomly choose one solution from the generated non-dominated solutions as input for the component *Dynamic Executor and Prioritizer* (*DEP*) since all the solutions produced by *SP* have equivalent quality.

D. Dynamic Executor and Prioritizer (DEP)

The core of *DEP* is to execute the statically prioritized test cases obtained from *SP* (Section IV.C) and dynamically

update the test case order using the *fail rules* and *pass rules* mined from *RM* (Section IV.B) as shown in Fig. 2. Thus, the input for *DEP* is a prioritization solution (i.e., static prioritized test cases) from *SP* and a set of mined rules (i.e., *fail rules* and *pass rules*) from *RM*. The overall process of *DEP* is presented in Fig. 4, and it is explained using the motivating example (Section II) in Fig. 5.

Input: Solution $s_f = \{T_{f1}, \dots, T_{fn}\}$, *pass rules* PR, *fail rules* FR
Output: Dynamically prioritized test cases
Begin:
1 $k \leftarrow 1$
2 $AF \leftarrow AlwaysFailing(s_f)$ // set of test cases that always *fail*
3 $AP \leftarrow AlwaysPassing(s_f)$ // set of test cases that always *pass*
4 **while** ($k \leq n$) and (termination_conditions_not_satisfied) **do**
5 $TE \leftarrow Get-test-case-execution(s_f, AF, AP)$
6 $verdict \leftarrow execute(TE)$ // execute the test case
7 **if** ($verdict$ = '*pass*') **then**
8 **if** ($TE \in PR$) **then** // if the test case has a *pass rule*
9 move the related test cases backward to the end of solution
10 **else if** ($verdict$ = '*fail*') **then**
11 **if** ($TE \in FR$) **then** // if the test case has a *fail rule*
12 move the related test cases forward to execute next
13 $s_{final} \leftarrow s_{final} \cup TE$ // dynamically prioritized solution
14 $k \leftarrow k + 1$
15 remove TE from s_f // remove executed test case from s_f
16 return s_{final}
End

Fig. 4. Overview of dynamic executor and prioritizer.

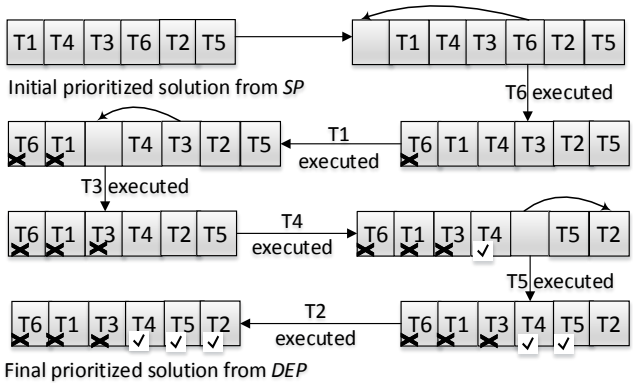

Fig. 5. Example of dynamic test case execution and prioritization.

Let us assume a prioritization solution is obtained as $\{T_1, T_4, T_3, T_6, T_2, T_5\}$ (Fig. 5) from *SP* for the test cases in the motivating example (Section II). Moreover, seven *fail rules* and *pass rules* are extracted from *RM* as shown in TABLE I. Initially, *DEP* checks if there are any test case(s) that always had the verdict *fail* or *pass* in the historical execution data. If there exists any such test case(s), *DEP* adds them to the set *AF* (if the verdict is always *fail*) or the set *AP* (if the verdict is always *pass*) (lines 2 and 3 in Fig. 4). For example in Fig. 1, T_6 always had the verdict *fail* in historical execution data, and thus T_6 is added to AF.

Afterwards, *DEP* identifies the test cases to execute using the algorithm *Get-test-case-execution* (Algorithm 1). More specifically, *Get-test-case-execution* uses the statically prioritized solution, a set of always *fail* test cases AF, and a set of always *pass* test cases AP, to find the test case to execute. If there exists any test case in AF (line 1 in Algorithm 1), it is selected for execution (line 2 in Algorithm 1), removed from AF (line 3 in Algorithm 1), and returned (line 12 in Algorithm 1) to *DEP* (Fig. 4). For instance, initially, T_6 is selected for execution from AF in Fig. 5. Afterward, the selected test case (i.e., T_6) is executed (line 6 in Fig. 4), and based on the verdict of the executed test case, the *pass rule* or *fail rule* is employed (lines 7-12 in Fig. 4). Then, the executed test case is added to the final solution (line 13 in Fig. 4) and removed from the statically prioritized solution for execution (line 15 in Fig. 4). If there exists no related test case(s), the next test case is selected for execution (line 5 in Fig. 4).

Algorithm 1: Get-test-case-execution

Input: Solution $s_f = \{T_{f1}, \dots, T_{fn}\}$, set of always *fail* test cases AF, set of always *pass* test cases AP
Output: A test case for execution
Begin:
1 **if** ($|AF| > 0$) **then**
2 $TE \leftarrow F(AF_0)$ // get the first *fail* test case
3 remove AF_0 from AF // remove the *fail* test case
4 **else** $TE \leftarrow F(s_{f0})$ // if there are not any test cases in AF
5 $move\_T \leftarrow True$ // initially test cases can be moved
6 $first\_T \leftarrow TE$
7 **while** (TE in AP and $move\_T$) **do**
8 move s_{f0} to end of s_f // move test case to the end
9 $TE \leftarrow F(s_{f0})$
10 **if** ($first\_T = TE$) **then**
11 $move\_T \leftarrow False$ // do not move test case anymore
12 return TE
End

To select the next test cases, Algorithm 1 is employed again. If there exists no more test case in AF, the first test case from the statically prioritized solution is selected for execution if it is not present in the set of *pass* test cases AP (lines 4-7 in Algorithm 1). However, if a test case is present in AP, the next test case is selected from the static prioritized solution (lines 7-11 in Algorithm 1). For example, T_1 is executed after T_6 in Fig. 5 since it is the first test case from the statically prioritized solution obtained from *SP*, and it is not included in AP. Based on the verdict of the test case (line 6 in Fig. 4), *pass rule* or *fail rule* is employed once more (lines 7-12 in Fig. 4). If the test case has the verdict *fail* (line 10), it is checked if it is a part of any *fail rule* (line 11 in Fig. 4). If it is indeed a part of a *fail rule*, the related *fail* test case(s) is moved to the front of the solution s_f (line 12), e.g., T_3 is moved in front of T_4 in Fig. 5 since there exists a *fail rule* between T_1 and T_3.

Alternatively, if the test case has a verdict *pass*, it is checked if it is a part of any *pass rules* (line 7 in Fig. 4). If so, the related test case(s) is moved at the end of the solution. For example, T_2 is moved after T_5 in Fig. 5 since T_4 *passed* and there is a *pass rule* between T_4 and T_2 as shown in TABLE I. This process of executing and moving the test case is repeated until all the test cases are executed (e.g., all six test cases are executed shown in Fig. 5) or the termination criteria (e.g., a predefined time budget) for the algorithm is met. The final solution lists the optimal order of the test cases that were executed. For example, the final execution order of these six test cases in Fig. 5 is: $\{T_6, T_1, T_3, T_4, T_5, T_2\}$ with three *fail* test cases: T_6, T_1, and T_3. Finally, at the end of the test cycle, the

historical execution results of test cases are updated based on the verdicts of the executed test cases as shown in Fig. 2. Note that REMAP updates the mined rules after each test cycle rather than after executing each test case since the mined rules will not be changed largely if updated after executing each test case and rule mining is computationally expensive [35, 49].

Note that REMAP includes *Static Prioritizer* (i.e., *SP* in Section IV.C) before *Dynamic Executor and Prioritizer* (i.e., *DEP* in Section IV.D) since we observed that not all the test cases can be associated with *fail rules* and *pass rules* (Section IV.B). For instance, in the motivating example (Section II), the test case T_5 can be associated with no *fail rules* and *pass rules* (Section IV.B). Thus, as compared with randomly choosing a test case for execution when there is no mined rule to rely on, *SP* can help to improve the effectiveness of TP.

V. Empirical Evaluation Design

In this section, we present the experiment design (TABLE II) with research questions, case studies, evaluation metrics, and statistical tests (Section V.A - Section V.D).

TABLE II. Overview of the Experiment Design

RQ	Comparison with	Case Study	Evaluation Metric	Statistical Test
1	RS, Greedy	CS_1 – CS_5	*APFD*	Vargha and Delaney $\hat{A}_{12}$, Mann-Whitney U Test
2	SSBP			
3	RBP			

A. Research Questions

To evaluate the effectiveness of REMAP, we will address the following research questions.

RQ1. Sanity check: Is REMAP better than random search (RS) and Greedy for the five case studies?

This RQ is defined to check if our TP problem is non-trivial to solve. We used RS and two types of Greedy algorithms: 1) G-1, which prioritizes the test cases based on the objective FDC (Section IV.C) such that the test case with high FDC are given high priority [50] and 2) G-2, which prioritizes the test cases based on two objectives: FDC and TRS, where the two objectives are combined such that each objective is given equal weight.

RQ2. Is REMAP better than static search-based TP approach (SSBP), i.e., static prioritization solutions obtained from *SP* (Section IV.C)?

This RQ is defined to evaluate if dynamic prioritization (*DEP* in Section IV.D) can indeed help to improve the effectiveness of TP as compared with TP approaches without considering runtime test case execution results.

RQ3. Is REMAP better than rule-based TP approach (RBP), i.e., TP only based on the *fail rules* and *pass rules* from *RM* (Section IV.B)?

Answering this RQ can help us to know if *SP* together with *DEP* can help improve the effectiveness of TP as compared with rule-based TP approach. Based on [27], we designed rule-based TP approach **(**RBP) by applying the *RM* and *DEP* components. Note the difference between RBP and our approach REMAP is that RBP uses the solutions produced by RS (from RQ1) as input rather than a statically prioritized solution from *SP*.

B. Case Studies

To evaluate REMAP, we selected a total five case studies: two case studies for two different VCS products from Cisco (i.e., CS_1 and CS_2), two open source case studies from ABB Robotics for Paint Control (CS_3) [20] and IOF/ROL (CS_4) [20], and Google Shared Dataset of Test Suite Results (GSDTSR) (CS_5) [21]. For each case study, test case execution result is linked to a particular test cycle such that each test cycle is considered as occurrence of regression testing. More specifically, each case study contains historical execution data of the test cases for more than 300 test cycles as shown in TABLE III. The historical execution data of the test cases are used to mine execution relations using the *RM* (Section IV.B) and calculate the FDC for each test case (Section IV.C). Note that the open source case studies for CS_3 - CS_5 are publicly available at [51].

TABLE III. Overview of the Case Studies used for Mining Rules

CS	Data Set	#Test Cases	#Test Cycles	#Verdicts
CS_1	Cisco Data Set1	60	8,322	296,042
CS_2	Cisco Data Set2	624	6,302	149,039
CS_3	ABB Paint Control	89	351	25,497
CS_4	ABB IOF/ROL	1,941	315	32,162
CS_5	Google GSDTSR	5,555	335	1,253,464

Note that if a test case has been executed more than once in a test cycle, we consider its latest verdict in the test cycle as its actual verdict when mining the rules using *RM* (Section IV.B). This is because based on our industrial collaboration, we noticed that a test case could be executed more than once in a test cycle due to various reasons, e.g., device problems (e.g., need to restart), network problems and test engineers only take the final verdict of the test case into account when debugging or reporting.

C. Experiment Tasks and Evaluation Metric

For the industrial case studies (i.e., CS_1 and CS_2), we dynamically prioritize and execute the test cases in VCSs for the next test cycle (that has not been executed yet, e.g., test cycle 6,303 for CS_2) and evaluate the six approaches: RS, G-1, G-2, SSBP, RBP, and REMAP. However, for the open source case studies (i.e., CS_3 - CS_5), we do not have access to the actual test cases to execute them as mentioned in Section V.B. Thus, we use the historical test execution results without the latest test cycle for prioritizing the test cases and compare the performance of different approaches based on the actual test case execution results from the latest test cycle. For instance, for CS_5, which has execution results for 336 test cycles [51], we used historical results from 335 test cycles to prioritize test cases for the next test cycle (i.e., 336). More specifically, we look at the execution result of each test case from the latest test cycle to know if the verdict of a test case is *fail* or *pass*, and accordingly, we apply the *fail rules* or *pass rules*. Note that such a way of comparison has been applied in the literature when it is difficult to execute the test cases at runtime in practice [20, 27].

We employ the widely used Average Percentage of Fault Detected ($APFD$) metric as the evaluation metric to compare the performance of different approaches as is widely used in the literature. The $APFD$ [6, 7] metric is widely used to

TABLE IV. COMPARISON OF *APFD* FOR THE FIVE CASE STUDIES USING THE VARGHA AND DELANEY STATISTICS AND MANN-WHITNEY U TEST

RQ	Comparison	CS_1		CS_2		CS_3		CS_4		CS_5	
		$\hat{A}_{12}$	*p*-value	$\hat{A}_{12}$	*p*-value	$\hat{A}_{12}$	*p*-value	$\hat{A}_{12}$	*p*-value	$\hat{A}_{12}$	*p*-value
1	REMAP vs. RS	0.986	**<0.0001**	0.999	**<0.0001**	0.869	**<0.0001**	0.999	**<0.0001**	1	**<0.0001**
	REMAP vs. G-1	0.993	**<0.0001**	0.921	**<0.0001**	1	**<0.0001**	0.917	**<0.0001**	1	**<0.0001**
	REMAP vs. G-2	0.993	**<0.0001**	1	**<0.0001**	1	**<0.0001**	0.257	**<0.0001**	1	**<0.0001**
2	REMAP vs. SSBP	0.99	**<0.0001**	0.951	**<0.0001**	1	**<0.0001**	0.967	**<0.0001**	0.966	**<0.0001**
3	REMAP vs. RBP	0.881	**<0.0001**	0.921	**<0.0001**	0.784	**<0.0001**	0.917	**<0.0001**	0.943	**<0.0001**

measure the effectiveness of prioritized test cases in terms of detecting the faults for a particular test cycle [8, 18, 19, 23]. $APFD$ for a solution s_j can be calculated as: $APFD_{sj} = 1 - \frac{\sum_{i=1}^{m} TF_i}{n \times m} + \frac{1}{2n}$, where n denotes the number of test cases in the test suite TS, m denotes the number of faults detected by TS, and TF_i represents the first test case in s_j that detects the fault i. Note that the value of $APFD$ metric is between 0 and 1 (0-100%), and higher $APFD$ implies higher fault detection rates.

D. Statistical Tests and Experiment Settings

1) Statistical Tests: Using the guidelines from [52], the Vargha and Delaney $\hat{A}_{12}$ statistics [53] and Mann-Whitney U test [54] are used to statistically evaluate the results for the three research questions (i.e., RQ1- RQ3) as depicted in TABLE II. The Vargha and Delaney $\hat{A}_{12}$ statistics is a non-parametric effect size measure and evaluates the probability of yielding higher values for the evaluation metric for two approaches A and B. Mann-Whitney U test (*p*-value) is used to indicate whether the median values of the two samples are statistically significant. For two approaches A and B, A has a significantly better performance than B if $\hat{A}_{12}$ is higher than 0.5 and the p-value is less than 0.05. In our context, we use the Vargha and Delaney comparison without transformation [55] since we are interested in any improvement in the performance of the approach (e.g., REMAP) [42, 56].

2) Experiment Settings: We used the WEKA data mining software [57] to implement the component RM (Section IV.B). The input data format to be used by our RM is a file composed of instances that contain test case verdicts such that each instance in the file represents a test cycle. We employed a widely-used Java framework jMetal [58] to implement the component *SP* (Section IV.C) and the standard settings were applied to configure the search algorithm (i.e., NSGA-II) [58] as are usually recommended [52]. More specifically, the population size is set as 100, the crossover rate is 0.9, the mutation rate is 1/(Total number of test cases), and the maximum number of fitness evaluation (i.e., termination criteria of the algorithm) is set as 50,000. For the case studies, we encoded each test suite to be prioritized as an abstract format (e.g., JSON file), which contains the key information of the test cases for prioritization, e.g., test case id, FDC. Afterwards, NSGA-II is employed to produce the prioritized test suite that is formed as the same abstract format (e.g., JSON file), which consists of a list of prioritized test cases. Finally, the prioritized test cases are selected from the original test suite using the test case *id* and put for execution.

Each approach (except G-1 and G-2) was executed 30 times for each case study to account for the random variation of search algorithms [52], such that each run produced 100 solutions (population size for *SP*). Therefore, a total of 3,000 solutions were compared for each approach (except G-1 and G-2) in each case study. Since G-1 and G-2 always produce the same solution [8, 59, 60], they have one value per case study. The experiments were executed on a Mac OS Sierra with a processor Intel Core i7 2.8GHz and 16 GB of RAM.

VI. RESULTS, ANALYSIS, AND OVERALL DISCUSSION

A. RQ1. Sanity Check (REMAP vs. RS, G-1, and G-2)

Recall that RQ1 aims to assess the effectiveness of REMAP as compared to RS, G-1, and G-2 using $APFD$ (TABLE II). TABLE IV reports the results of comparing REMAP with RS, G-1, and G-2 in terms of $APFD$. Based on the results from TABLE IV, it can be observed that REMAP managed to significantly outperform both RS and G-2 for all the five case studies, and G-1 for four case studies since $\hat{A}_{12}$ values (the Vargha and Delaney statistics) are higher than 0.75 and *p*-values are less than 0.0001.

Moreover, Fig. 6 presents the boxplot of $APFD$ produced by REMAP and RS, such that there are 3,000 values for each case study. Note that G-1 and G-2 are not presented in Fig. 6 since they produce only one value for each case study as mentioned in Section V.D. TABLE V presents the average $APFD$ produced by REMAP and RS and actual $APFD$ for G-1 and G-2. Based on Fig. 6 and TABLE V, we can observe that the median $APFD$ produced by REMAP are much higher than RS and G-2 for all the five case studies, and G-1 for four case studies. Moreover, we can observe from TABLE V that on average REMAP achieved a better $APFD$ of 33.4%, 14.5%, and 17.7% as compared to RS, G-1, and G-2, respectively.

TABLE V. *APFD* FOR DIFFERENT APPROACHES FOR FIVE CASE STUDIES

Approach	CS_1	CS_2	CS_3	CS_4	CS_5
REMAP	76.86%	76.07%	73.35%	93.24%	94.02%
RS	48.54%	49.82%	48.66%	49.69%	49.7%
G-1	68.5%	42.1%	46.1%	95.5%	88.7%
G-2	62.5%	64.9%	22.47%	86.69%	88.5%
SSBP	65.38%	67.19%	41.11%	79.54%	85.8%
RBP	60.77%	65.56%	62.18%	83.21%	92.85%

Thus, we can answer RQ1 as REMAP can significantly outperform RS, G-1, and G-2 for 93.3% of the case studies (i.e., 14 out of 15) defined in Section V.A. Overall, on average REMAP achieved a better $APFD$ of 21.9% (i.e., 33.4%, 14.5%, and 17.7%) compared to RS, G-1, and G-2.

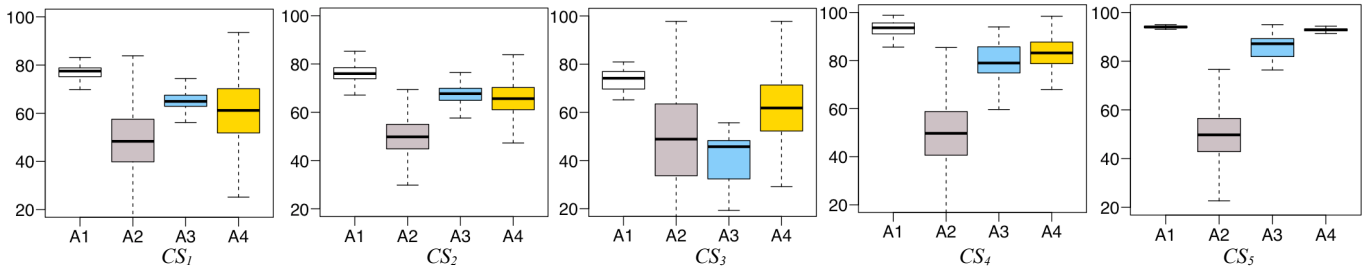

Fig. 6. Boxplot of *APFD* produced by different approaches for the five case studies*.
*A1: REMAP, A2: RS, A3: SSBP, A4: RBP

B. RQ2. (REMAP vs. SSBP)

This research question aims to assess the effectiveness of REMAP as compared to SSBP using the evaluation metric $APFD$. Based on the results from TABLE IV, we can observe that REMAP is able to perform significantly better than SSBP for all the five case studies. Additionally, Fig. 6 shows that the median $APFD$ values produced by REMAP are much higher than SSBP for all the five case studies. Moreover, as compared to SSBP, REMAP achieved a better $APFD$ of 14.9% across the five case studies as shown in TABLE V.

Therefore, we can answer RQ2, as REMAP can significantly outperform SSBP for all the five case studies, and on average REMAP had a better $APFD$ of 14.9% as compared to SSBP.

C. RQ3. (REMAP vs. RBP)

Recall that this research question aims to compare the performance of REMAP with RBP for the five case studies using the evaluation metric $APFD$. It can be observed from TABLE IV that REMAP managed to significantly outperform RBP for all the five case studies. Additionally, from Fig. 6, we can observe that the median values for $APFD$ produced by REMAP are always higher than RBP. On average, REMAP managed to achieve a higher mean $APFD$ of 9.8% for the five case studies as compared to RBP based on the results from TABLE V.

Thus, we can answer RQ3, as REMAP manages to outperform RBP for all the five case studies significantly, and on average REMAP achieved a better $APFD$ of 9.8% as compared to RBP.

D. Discussion

For RQ1, REMAP significantly outperformed RS for all the five case studies, which implies that the TP problems are not trivial to solve and require an efficient TP approach. Moreover, REMAP performed better than G-1 for four case studies and G-2 for all the five case studies, which can be explained by the fact that G-1 and G-2 greedily prioritize the best test case one at a time in terms of the defined objectives until the termination conditions are met. However, G-1 and G-2 may get stuck in local search space and result in sub-optimal solutions [59]. Moreover, REMAP dynamically prioritizes the test cases based on the execution result of the test case using the mined rules defined in Section IV.B, which helps REMAP to prioritize and execute the faulty test cases as soon as possible while executing the test cases less likely to find fault later. On contrary, G-1 performed better than REMAP for CS_4 because all the test cases that failed in CS_4 had a high value of FDC (>0.9), and thus G-1 is more efficient in finding them since it prioritizes the test cases based only on FDC. However, note that higher FDC does not always imply better $APFD$ as shown in TABLE V, where G-1 has worse performance than even RS for the case studies CS_2 and CS_3.

For RQ2, REMAP manages to significantly outperform static search-based prioritization (SSBP) since it is essential to consider the runtime test case execution results in addition to the historical execution data when addressing TP problem. More specifically, the mined *fail rules* (Section IV.B) help to prioritize the related test cases likely to fail (to execute earlier) while the mined *pass rules* (Section IV.B) assist in deprioritizing the test cases likely to pass (to execute later).

For RQ3, REMAP significantly outperforms rule-based prioritization (RBP). This can be explained by the fact that RBP has no heuristics to prioritize the initial set of test cases for execution and certain randomness is introduced when there are no mined rules that can be used to choose the next test cases for execution. As compared with RBP, REMAP uses *SP* (Section IV.C) to prioritize the test cases and take them as input for *Dynamic Executor and Prioritizer* (*DEP*). Therefore, the randomness of selecting the test cases for execution can be reduced when no mined rules can be applied.

Moreover, *SP* component (Section IV.C) employed multi-objective search to statically prioritize test cases, which took on average 1.8 seconds, 19.6 seconds, 2.3 seconds, 9 minutes, and 25.1 minutes for CS_1, CS_2, CS_3, CS_4, and CS_5, respectively. The large difference in the running time for the five case studies is due to the wide range of test case number across the five case studies (i.e., 60 – 5,555 as shown in TABLE III). Based on the knowledge of VCS testing (Section II), the running time in seconds in acceptable when applying in practice and thus REMAP is applicable in terms of addressing test case prioritization problem at Cisco.

Furthermore, Fig. 7 shows the percentage of faults detected when executing the test suite for the five case studies using the four approaches: REMAP, G-1, G-2, SSBP, and RBP. As shown in Fig. 7, REMAP manages to detect the faults faster than the other approaches for almost all the case studies. On an average, REMAP only required 21% of the test cases to detect 81% of the faults for the five case studies while G-1, G-2, SBP, and RBP took 41%, 37%, 43%, and 36% of the test cases, respectively to detect the same amount of faults as compared with REMAP.

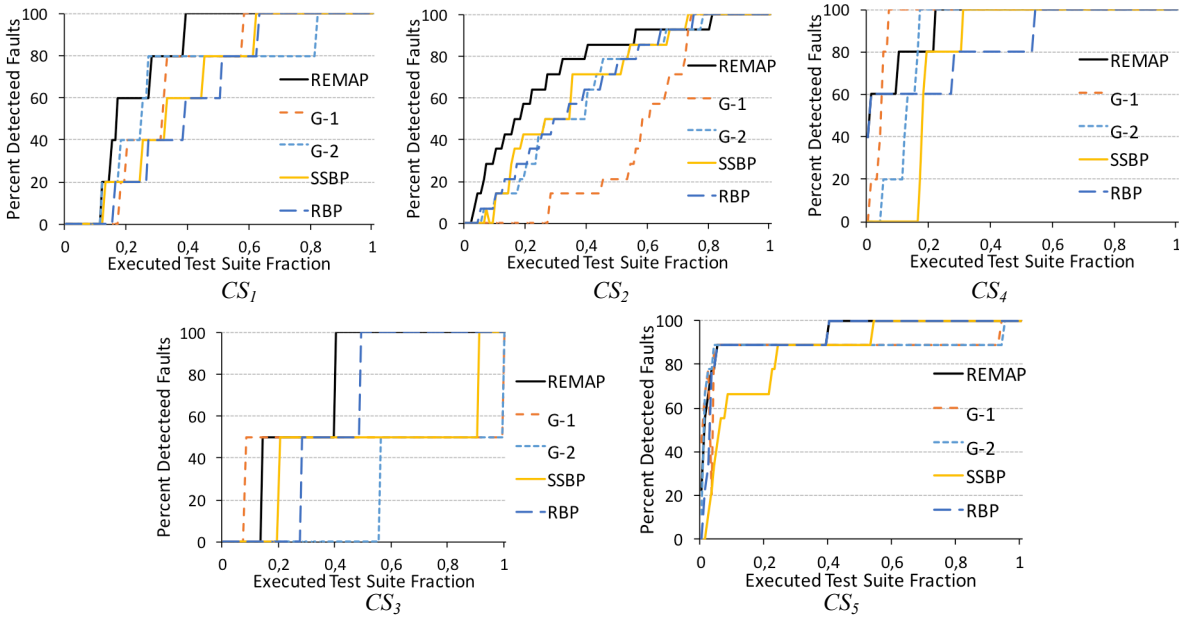

Fig. 7. Percentage of total faults detected on executing the test suite for the five case studies using the five approaches.

For CS_2, the mean $APFD$ produced by G-1 is worse than RS (TABLE V) since the test cases that failed in CS_3 had a very low value for FDC (<0.2), i.e., the test cases did not fail many times in the past executions. Additionally, for CS_3, the mean $APFD$ obtained by G-1, G-2, and SSBP is worse than RS (as shown in TABLE V) due to the fact that the test cases that failed in CS_3 had also a low FDC (<0.5), and one test case that failed in CS_3 had the FDC value as 0. Thus, SSBP, G-1, and G-2 are not able to prioritize that *fail* test case (with FDC 0) and it is executed very late (i.e., after executing more than 85% of the test suite) as shown in Fig. 7. However, REMAP and RBP still managed to find that *fail* test case by executing less than half the test suite. This is due to the fact that the mined rules help to deprioritize the test cases likely to *pass*, and thus, they are able to execute the test case for execution faster. Therefore, we can conclude that REMAP has the best effectiveness regarding detecting more faults by executing fewer test cases.

E. Threats to Validity

The threats to *internal validity* consider the internal parameters (e.g., algorithm parameters) that might influence the obtained results [61]. In our experiments, *internal validity* threats might arise due to experiments with only one set of configuration settings for the algorithm parameters [62]. However, note that these settings are from the literature [63]. Moreover, to mitigate the *internal validity* threat due to parameter settings of the rule mining algorithm (i.e., RIPPER), we used the default parameter settings that have performed well in the state-of-the-art [34, 36, 57, 64].

Threats to *construct validity* arise when the measurement metrics do not sufficiently cover the concepts they are supposed to measure [8]. To mitigate this threat, we compared the different approaches using the Average Percentage of Fault Detected ($APFD$) metric that has been widely employed in the literature [8, 18, 19, 23]. Another threat to *construct validity* is the assumption that each failed test case indicates a different failure in the system under test. However, the mapping between fault and test case is not easily identifiable for both the two industrial case studies and three open source ones, and thus we assumed that each failed test case is assumed to cause a separate fault. Note that such assumption is held in many literatures [9, 20, 27]. Moreover, we did not execute the test cases from the three open source case studies since we do not have access to the actual test cases. Therefore, we looked at the latest test cycle to obtain the test case results, which have been done in the existing literature when it is challenging to execute the test cases [20, 27].

The threats to *conclusion validity* relate to the factors that influence the conclusion drawn from the experiments [65]. In our context, *conclusion validity* threat arises due to the use of randomized algorithms that is responsible for the random variation in the produced results. To mitigate this threat, we repeated each experiment 30 times for each case study to reduce the probability that the results were obtained by chance. Moreover, following the guidelines of reporting results for randomized algorithms [52], we employed the Vargha and Delaney statistics test statistics as the effect size measure to determine the probability of yielding higher performance by different algorithms and Mann-Whitney U test for determining the statistical significance of results.

The threats to *external validity* are related to the factors that affect the generalization of the results [61]. To mitigate this threat, we chose five case studies (two industrial ones and three open source ones) to evaluate REMAP empirically. It is also worth mentioning that such threats to *external validity* are common in empirical studies [42]. Another threat to *external validity* is due to the selection of a particular search algorithm. To reduce this threat, we selected the most widely used search algorithm (NSGA-II) that has been applied in different contexts [42-44].

VII. Related Work

There exists a large body of research on TP [7, 8, 18, 22, 24, 66, 67], and a broad view of the state-of-the-art on TP is presented on [17, 68]. The literatures have presented different prioritization techniques such as search-based [8, 24, 66] and linear programming based [23, 69, 70]. Since our approach REMAP is based on historical execution results and rule mining, we discuss the related work from these two angles.

A. History-Based TP

History-based prioritization techniques prioritize test cases based on their historical execution data with the aim to execute the test cases most likely to fail first [12]. History-based prioritization techniques can be classified into two categories: static prioritization and dynamic prioritization. Static prioritization produces a static list of test cases that are not changed while executing the test cases while dynamic prioritization changes the order of test cases at runtime.

Static TP: Most of the history-based prioritization techniques produce a static order of test cases [9, 12-14, 39, 71, 72]. For instance, Kim et al. [12] prioritized test cases based on the historical execution data where they consider the number of previous faults exposed by the test cases as the key prioritizing factor. Elbaum et al. [9] used time windows from the test case execution history to identify how recently test cases were executed and detected failures for prioritizing test cases. Park et al. [71] considered the execution cost of the test case and the fault severity of the detected faults from the test case execution history for TP. Wang et al. [16, 25] defined fault detection capability (FDC) as one of the objectives for TP while using multi-objective search. As compared to the above-mentioned works, REMAP poses at least three differences: 1) REMAP defines two types of rules: *fail rule* and *pass rule* (Section IV.B) and mines these rules from historical test case execution data with the aim to support TP; 2) REMAP defines a new objective (i.e., test case reliance score for the component *SP* in Section IV.C) to measure to what extent, executing a test case can be used to predict the execution results of other test cases, which is not the case in the existing literatures; and 3) REMAP proposes a dynamic method to update the test case order based on the runtime test case execution results.

Dynamic TP: Qu et al. [50] used historical execution data to statically prioritize test cases and after that, the runtime execution result of the test case(s) is used for dynamic prioritization using the relation among test cases. To obtain relation among the test cases, they [50] group together the test cases that detected the same fault in the historical execution data such that all the test cases in the group are related. Our work is different from [50] in at least three aspects: 1) REMAP uses rule-mining to mine two types of execution rules among test cases (Section IV.B), which is not the case in [50]; 2) sensitive constant needs to be set up manually to prioritize/deprioritize test cases in [50], however, REMAP does not require such setting; 3) REMAP uses both FDC and TRS (Section IV.C) for static prioritization unlike [50] where only FDC is considered for static prioritization.

B. Rule Mining for Regression Testing

There are only a few works that focus on applying rule mining techniques for TP [73, 74]. The authors in [73, 74] modeled the system using Unified Modeling Language (UML) and maintained a historical data store for the system. Whenever the system is changed, association rule mining is used to obtain the frequent pattern of affected nodes that are then used for TP. As compared with these studies [73, 74], REMAP is different in at least two ways: 1) REMAP uses the execution result of the test cases to obtain *fail rules* and *pass rules*; 2) REMAP dynamically updates the test order based on the test case execution results.

Another work [27] proposed a rule mining based technique for improving the effectiveness of the test suite for regression testing. More specifically, they use association rule mining to mine the execution relations between the smoke test failures (smoke tests refer to a small set of test cases that are executed before running the regression test cases) and test case failures using the historical execution data. When the smoke tests fail, the related test cases are executed. As compared to this approach, REMAP has at least three key differences: 1) we aim at addressing test case prioritization problem while the test case order is not considered in [27]; 2) REMAP uses a search-based test case prioritization component to obtain the static order of test case before execution, which is different than [27] that executes a set of smoke test cases to select the test cases for execution; 3) REMAP defines two sets of rules (i.e., *fail rule* and *pass rule*) while only *fail rule* is considered in [27].

VIII. Conclusion

This paper introduces a novel rule mining and search-based dynamic prioritization approach (named as REMAP) that has three key components (i.e., *Rule Miner*, *Static Prioritizer*, and *Dynamic Executor and Prioritizer*) with the aim to detect faults earlier. REMAP was evaluated using five case studies: two industrial ones and three open source ones. The results showed that REMAP achieved a higher Average Percentage of Fault Detected ($APFD$) of 18% (i.e., 33.4%, 14.5%, 17.7%, 14.9%, and 9.8%) as compared to random search, greedy based on fault detection capability (FDC), greedy based on FDC and test case reliance score (TRS), static search-based prioritization, and rule-based prioritization.

In the future, we plan to evaluate REMAP with more test case prioritization approaches (e.g., linear programming based [23, 69, 70]) with more case studies to further strengthen the applicability of REMAP. Moreover, we want to involve test engineers from our industrial partner to deploy and assess the effectiveness of REMAP in real industrial settings.

Acknowledgement

This research was supported by the Research Council of Norway (RCN) funded Certus SFI. Shuai Wang is also supported by RFF Hovedstaden funded MBE-CR project. Tao Yue and Shaukat Ali are also supported by RCN funded Zen-Configurator project, the EU Horizon 2020 project funded U-Test, RFF Hovedstaden funded MBE-CR project, and RCN funded MBT4CPS project.

REFERENCES

[1] A. K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma, "Regression testing in an industrial environment," *Communications of the ACM,* vol. 41, no. 5, pp. 81-86. 1998.

[2] E. Borjesson and R. Feldt, "Automated system testing using visual gui testing tools: A comparative study in industry," in *Proceedings of the Fifth International Conference on Software Testing, Verification and Validation (ICST)*, pp. 350-359. IEEE, 2012.

[3] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology (TOSEM),* vol. 6, no. 2, pp. 173-210. 1997.

[4] C. Kaner, "Improving the maintainability of automated test suites," *Software QA,* vol. 4, no. 4, 1997.

[5] P. K. Chittimalli and M. J. Harrold, "Recomputing coverage information to assist regression testing," *IEEE Transactions on Software Engineering,* vol. 35, no. 4, pp. 452-469. 2009.

[6] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering,* vol. 27, no. 10, pp. 929-948. 2001.

[7] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: An empirical study," in *Proceedings of the International Conference on Software Maintenance*, pp. 179-188. IEEE, 1999.

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

[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*, pp. 235-245. ACM, 2014.

[10] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Transactions on Software Engineering,* vol. 28, no. 2, pp. 159-182. 2002.

[11] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice," in *Proceedings of the Eighth International Symposium on Software Reliability Engineering* pp. 264-274. IEEE, 1997.

[12] 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 (ICSE)*, pp. 119-129. IEEE, 2002.

[13] A. Khalilian, M. A. Azgomi, and Y. Fazlalizadeh, "An improved method for test case prioritization by incorporating historical test case data," *Science of Computer Programming,* vol. 78, no. 1, pp. 93-116. 2012.

[14] T. B. Noor and H. Hemmati, "A similarity-based approach for test case prioritization using historical failure data," in *Proceedings of the 26th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 58-68. IEEE, 2015.

[15] S. Wang, S. Ali, and A. Gotlieb, "Minimizing test suites in software product lines using weight-based genetic algorithms," in *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*, pp. 1493-1500. ACM, 2013.

[16] S. Wang, D. Buchmann, S. Ali, A. Gotlieb, D. Pradhan, and M. Liaaen, "Multi-objective test prioritization in software product line testing: an industrial case study," in *Proceedings of the 18th International Software Product Line Conference*, pp. 32-41. ACM, 2014.

[17] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability,* vol. 22, no. 2, pp. 67-120. Wiley Online Library, 2012.

[18] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon, "Comparing white-box and black-box test prioritization," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pp. 523-534. IEEE, 2016.

[19] Y. Lu, Y. Lou, S. Cheng, L. Zhang, D. Hao, Y. Zhou*, et al.*, "How does regression test prioritization perform in real-world software evolution?," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pp. 535-546. ACM, 2016.

[20] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 12-22. ACM, 2017.

[21] S. Elbaum, A. Mclaughlin, and J. Penix. (2014). *The Google Dataset of Testing Results*. Available: https://code.google.com/p/google-shared-dataset-of-test-suite-results

[22] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *Proceedings of the 23rd International Conference on Software Engineering*, pp. 329-338. IEEE Computer Society, 2001.

[23] D. Hao, L. Zhang, L. Zang, Y. Wang, X. Wu, and T. Xie, "To be optimal or not in test-case prioritization," *IEEE Transactions on Software Engineering,* vol. 42, no. 5, pp. 490-505. 2016.

[24] Z. Li, Y. Bian, R. Zhao, and J. Cheng, "A fine-grained parallel multi-objective test case prioritization on GPU," in *Proceedings of the International Symposium on Search Based Software Engineering*, pp. 111-125. Springer, 2013.

[25] S. Wang, S. Ali, T. Yue, Ø. Bakkeli, and M. Liaaen, "Enhancing test case prioritization in an industrial setting with resource awareness and multi-objective search," in *Proceedings of the 38th International Conference on Software Engineering Companion*, pp. 182-191. 2016.

[26] J. A. Parejo, A. B. Sánchez, S. Segura, A. Ruiz-Cortés, R. E. Lopez-Herrejon, and A. Egyed, "Multi-objective test case prioritization in highly configurable systems: A case study," *Journal of Systems and Software,* vol. 122, pp. 287-310. 2016.

[27] 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*, pp. 142-151. ACM, 2014.

[28] W. W. Cohen, "Fast effective rule induction," in *Proceedings of the Twelfth International Conference on Machine Learning*, pp. 115-123. Elsevier, 1995.

[29] W. Lin, S. A. Alvarez, and C. Ruiz, "Efficient adaptive-support association rule mining for recommender systems," *Data mining and knowledge discovery,* vol. 6, no. 1, pp. 83-105. Springer, 2002.

[30] W. Lin, S. A. Alvarez, and C. Ruiz, "Collaborative recommendation via adaptive association rule mining," *Data Mining and Knowledge Discovery,* vol. 6, pp. 83-105. 2000.

[31] R. J. Bayardo Jr, "Brute-Force Mining of High-Confidence Classification Rules," in *KDD*, pp. 123-126. 1997.

[32] W. Shahzad, S. Asad, and M. A. Khan, "Feature subset selection using association rule mining and JRip classifier," *International Journal of Physical Sciences,* vol. 8, no. 18, pp. 885-896. 2013.

[33] B. Qin, Y. Xia, S. Prabhakar, and Y. Tu, "A rule-based classification algorithm for uncertain data," in *Proceedings of the 25th International Conference on Data Engineering (ICDE)*, pp. 1633-1640. IEEE, 2009.

[34] B. Ma, K. Dejaeger, J. Vanthienen, and B. Baesens, "Software defect prediction based on association rule classification," 2011.

[35] A. Cano, A. Zafra, and S. Ventura, "An interpretable classification rule mining algorithm," *Information Sciences,* vol. 240, pp. 1-20. 2013.

[36] S. Vijayarani and M. Divya, "An efficient algorithm for generating classification rules," *International Journal of Computer Science and Technology,* vol. 2, no. 4, 2011.

[37] B. Seerat and U. Qamar, "Rule induction using enhanced RIPPER algorithm for clinical decision support system," in *Proceedings of the Sixth International Conference on Intelligent Control and Information Processing (ICICIP)*, pp. 83-91. IEEE, 2015.

[38] P.-N. Tan, *Introduction to data mining*: Pearson Education India, 2006.

[39] D. Marijan, A. Gotlieb, and S. Sen, "Test case prioritization for continuous regression testing: An industrial case study," in *IEEE International Conference on Software Maintenance (ICSM)*, pp. 540-543. IEEE, 2013.

[40] D. Pradhan, S. Wang, S. Ali, T. Yue, and M. Liaaen, "STIPI: Using Search to Prioritize Test Cases Based on Multi-objectives Derived from Industrial Practice," in *Proceedings of the International Conference on Testing Software and Systems*, pp. 172-190. Springer, 2016.

[41] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation,* vol. 6, no. 2, pp. 182-197. IEEE, 2002.

[42] F. Sarro, A. Petrozziello, and M. Harman, "Multi-objective software effort estimation," in *Proceedings of the 38th International Conference on Software Engineering*, pp. 619-630. ACM, 2016.

[43] A. S. Sayyad, T. Menzies, and H. Ammar, "On the value of user preferences in search-based software engineering: a case study in software product lines," in *Proceedings of the 35th International Conference on Software Engineering*, pp. 492-501. IEEE, 2013.

[44] J. Sadeghi, S. Sadeghi, and S. T. A. Niaki, "A hybrid vendor managed inventory and redundancy allocation optimization problem in supply chain management: An NSGA-II with tuned parameters," *Computers & Operations Research,* vol. 41, pp. 53-64. 2014.

[45] D. Pradhan, S. Wang, S. Ali, and T. Yue, "Search-Based Cost-Effective Test Case Selection within a Time Budget: An Empirical Study," in *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 1085-1092. ACM, 2016.

[46] N. Srinvas and K. Deb, "Multi-objective function optimization using non-dominated sorting genetic algorithms," *Evolutionary Computation,* vol. 2, no. 3, pp. 221-248. MIT Press, 1994.

[47] E. Zitzler, K. Deb, and L. Thiele, "Comparison of multiobjective evolutionary algorithms: Empirical results," *Evolutionary Computation,* vol. 8, no. 2, pp. 173-195. MIT Press, 2000.

[48] A. S. Sayyad and H. Ammar, "Pareto-optimal search-based software engineering (POSBSE): A literature survey," in *Proceedings of the 2nd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, pp. 21-27. IEEE, 2013.

[49] A. Veloso, W. Meira Jr, and M. J. Zaki, "Lazy associative classification," in *Sixth International Conference on Data Mining (ICDM)*, pp. 645-654. IEEE, 2006.

[50] B. Qu, C. Nie, B. Xu, and X. Zhang, "Test case prioritization for black box testing," in *Proceedings of the 31st Annual Computer Software and Applications Conference (COMPSAC)*, pp. 465-474. IEEE, 2007.

[51] *Supplementary material*.
Available: https://remap-ICST.netlify.com

[52] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pp. 1-10. IEEE, 2011.

[53] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics,* vol. 25, no. 2, pp. 101-132. 2000.

[54] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The Annals of Mathematical Statistics,* pp. 50-60. JSTOR, 1947.

[55] G. Neumann, M. Harman, and S. Poulding, "Transformed vargha-delaney effect size," in *Proceedings of the International Symposium on Search Based Software Engineering*, pp. 318-324. Springer, 2015.

[56] W. Martin, F. Sarro, and M. Harman, "Causal impact analysis for app releases in google play," in *Proceedings of the 24th International Symposium on Foundations of Software Engineering*, pp. 435-446. ACM, 2016.

[57] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining: Practical machine learning tools and techniques*: Morgan Kaufmann, 2016.

[58] J. J. Durillo and A. J. Nebro, "jMetal: A Java framework for multi-objective optimization," *Advances in Engineering Software,* vol. 42, no. 10, pp. 760-771. Elsevier, 2011.

[59] S. Tallam and N. Gupta, "A concept analysis inspired greedy algorithm for test suite minimization," *ACM SIGSOFT Software Engineering Notes,* vol. 31, no. 1, pp. 35-42. 2006.

[60] P. Baker, M. Harman, K. Steinhofel, and A. Skaliotis, "Search based approaches to component selection and prioritization for the next release problem," in *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pp. 176-185. IEEE, 2006.

[61] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case study research in software engineering: Guidelines and examples*: John Wiley & Sons, 2012.

[62] M. de Oliveira Barros and A. Neto, "Threats to Validity in Search-based Software Engineering Empirical Studies," *UNIRIO-Universidade Federal do Estado do Rio de Janeiro, techreport,* vol. 6, 2011.

[63] A. Arcuri and G. Fraser, "On parameter tuning in search based software engineering," in *Proceedings of the 3rd International Symposium on Search Based Software Engineering*, pp. 33-47. Springer, 2011.

[64] K. Song and K. Lee, "Predictability-based collective class association rule mining," *Expert Systems with Applications,* vol. 79, pp. 1-7. 2017.

[65] C. Wohlin, P. Runeson, M. Host, M. Ohlsson, B. Regnell, and A. Wesslen, "Experimentation in software engineering: an introduction. 2000," ed: Kluwer Academic Publishers, 2000.

[66] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Timeaware test suite prioritization," in *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 1-12. ACM, 2006.

[67] D. Pradhan, S. Wang, S. Ali, T. Yue, and M. Liaaen, "CBGA-ES: A Cluster-Based Genetic Algorithm with Elitist Selection for Supporting Multi-Objective Test Optimization," in *IEEE International Conference on Software Testing, Verification and Validation*, pp. 367-378. 2017.

[68] C. Catal and D. Mishra, "Test case prioritization: a systematic mapping study," *Software Quality Journal,* vol. 21, no. 3, pp. 445-478. 2013.

[69] L. Zhang, S.-S. Hou, C. Guo, T. Xie, and H. Mei, "Time-aware test-case prioritization using integer linear programming," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, pp. 213-224. ACM, 2009.

[70] S. Mirarab, S. Akhlaghi, and L. Tahvildari, "Size-constrained regression test case selection using multicriteria optimization," *IEEE TSE,* vol. 38, no. 4, pp. 936-956. 2012.

[71] H. Park, H. Ryu, and J. Baik, "Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing," in *Proceedings of the Secure System Integration and Reliability Improvement*, pp. 39-46. IEEE, 2008.

[72] Y. Fazlalizadeh, A. Khalilian, M. Abdollahi Azgomi, and S. Parsa, "Incorporating historical test case performance data and resource constraints into test case prioritization," *Tests and Proofs,* pp. 43-57. 2009.

[73] P. Mahali, A. A. Acharya, and D. P. Mohapatra, "Test Case Prioritization Using Association Rule Mining and Business Criticality Test Value," in *Computational Intelligence in Data Mining—Volume 2*, ed: Springer, 2016, pp. 335-345.

[74] A. A. Acharya, P. Mahali, and D. P. Mohapatra, "Model based test case prioritization using association rule mining," in *Computational Intelligence in Data Mining-Volume 3*, ed: Springer, 2015, pp. 429-440.