SRRTA: Regression Testing Acceleration via State Reuse

Jinhao Dong
Yiling Lou
HCST, CS, Peking University
Beijing, China
jinhao.dong@stu.pku.edu.cn
yiling.lou@pku.edu.cn

Dan Hao*
HCST, CS, Peking University
Beijing, China
haodan@pku.edu.cn

ABSTRACT

Regression testing is widely recognized as an important but time-consuming process. To alleviate this cost issue, test selection, reduction, and prioritization have been widely studied, and they share the commonality that they improve regression testing by optimizing the execution of the whole test suite. In this paper, we attempt to accelerate regression testing from a totally new perspective, i.e., skipping some execution of a new program by reusing program states of an old program. Following this intuition, we propose a state-reuse based acceleration approach SRRTA, consisting of two components: state storage and loading. With the former, SRRTA collects some program states during the execution of an old version through three heuristic-based storage strategies; with the latter, SRRTA loads the stored program states with efficiency optimization strategies. Through the preliminary study on *commons-math*, SRRTA reduces 82.7% of the regression testing time.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Regression Testing, Regression Testing Acceleration, State Reuse

ACM Reference Format:
Jinhao Dong, Yiling Lou, and Dan Hao. 2020. SRRTA: Regression Testing Acceleration via State Reuse. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3324884.3418928

1 INTRODUCTION

Regression testing is to rerun tests on modified software so as to guarantee the quality of modified software. Nowadays regression testing plays a critical role in software development but it is very costly [8].

To alleviate the cost issue of regression testing, researchers have put dedicated efforts in the domain of test selection, reduction, and prioritization, and have proposed a large number of approaches accordingly [4–7, 9–20, 23–26, 28–30, 32, 33]. In particular, test selection aims to select and run the tests relevant to software modification, test reduction aims to reduce a test suite by removing redundant tests, and test prioritization aims to schedule the execution order of a test suite. The commonality of them is that they address the cost issue of regression testing in the same way, i.e., optimizing the given test suite.

In this paper, we attempt to address the cost issue of regression testing in a totally new direction, optimizing the execution process of an individual test. Given two programs P_0 (before modification) and P_1 (after modification), the execution of P_0 is similar to P_1 due to the similarity between P_0 and P_1. Based on this intuition, some of the program states of P_0 may be the same as P_1, where a program state refers to a set of variables' values during execution [27]. Therefore, we can reuse program states of P_0 to address the cost issue of regression testing.

In particular, we propose a regression testing acceleration approach SRRTA, which first collects program states during the execution of P_0 (i.e., state storage) and then reuses these program states during the execution of P_1 (i.e., state loading). As state storage and loading would introduce extra costs in regression testing time, it is essential for SRRTA to balance its performance and cost so as to achieve the final acceleration. To address the challenge, SRRTA designs storage strategies in attempt to record the program states before/after time-consuming code in the state storage phase, and reduces loading costs with efficiency optimization strategies in the state loading phase. We further conduct a preliminary study on *Apache Commons Math*, and find that SRRTA reduces 82.7% regression testing time of P_1 but incurs 25.2% extra testing time of P_0 and 95.3M space. Moreover, we investigate the influence of different storage strategies and find that skipping executions of loop structures (i.e., $Store_{loop}$ strategy) can achieve the largest acceleration at most cases compared to the other strategies.

The contributions of this paper are summarized as follows: (1) a new dimension to alleviate the cost concern of regression testing; (2) a new acceleration technique SRRTA based on program state reuse; (3) a preliminary study demonstrating the performance of SRRTA.

2 APPROACH

Given two programs P_0 (before modification) and P_1 (after modification) in regression testing, we propose a state-reuse based acceleration approach SRRTA following the intuition: the program

*Dan Hao is the corresponding author. HCST is short for Key Lab of High Confidence Software Technologies (Peking University), MoE, Beijing, China.

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3418928

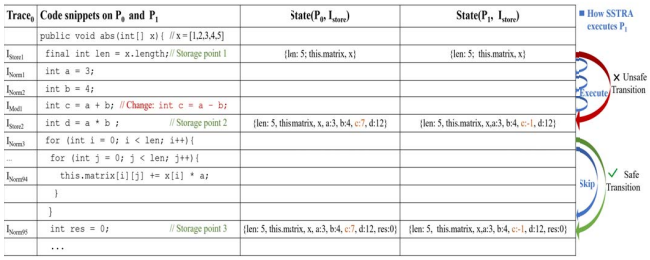

Figure 1: Illustration example

states unaffected by regression modifications can be directly reused instead of executing the relevant instructions. Following this intuition, the latter executions are skipped so that the regression testing process of P_1 can be accelerated. Figure 1 is used to illustrate SRRTA. The second column presents an old program P_0, while regression modification occurs on the sixth row.

2.1 Overview

Instead of executing all the instructions of P_1, SRRTA aims to only execute the instructions affected by regression modifications by reusing the stored program states of the old version. To facilitate state reuse, SRRTA consists of two components: state storage and state loading. In state storage (integrated in the testing process of P_0), SRRTA stores the program states at pre-defined code locations (i.e., *storage points*) during test execution, where a program state is a set of active variable values[1]. In state loading (integrated in the regression testing process of P_1), for instructions between any adjacent storage points, SRRTA skips their executions by loading program states of the next storage point if they are not affected by the modified code; otherwise, SRRTA executes these instructions.

2.2 State Storage

While executing P_0, SRRTA collects the program states at the pre-defined storage points on-the-fly by storing all active variables and their values when the instruction on each storage point is executed.

We use L_δ to denote the set of code elements modified between P_0 and P_1, and L_S to denote the set of code elements set as storage points by SRRTA. The execution trace of a test case on P_0 is represented as a sequence of instructions, i.e., $trace =< I_1, I_2, I_3, ..., I_n >$. Each instruction I is a unique dynamic instance of the static code element L which is mapped by the function $L = loc(I)$. For ease of representation, we classify each instruction I into three categories: (i) *modified instructions (i.e., I_{Mod})*, the instances of code elements modified between P_0 and P_1, i.e., $loc(I) \in L_\delta$; (ii) *storage instructions (i.e., I_{Store})*, the instances of code elements set as storage points, i.e., $loc(I) \in L_S$; (iii) *normal instructions (i.e., I_{Norm})*, the instructions which are not modified nor storage instructions. If a storage point happens to be modified from P_0 to P_1, its relevant instructions are regarded as modified instructions instead of storage instructions. Moreover, while executing each storage instruction (i.e., I_{Store}), SRRTA records the program state $State(P_0, I_{Store})$ including the active variables and their values in the memory.

As shown in the example, SRRTA sets three storage points (i.e., I_{Store1}, I_{Store2}, and I_{Store3}) on P_0, and therefore after the phase of state storage, SRRTA collects three sets of program states respectively (shown in Column "$State(P_0, I_{store})$").

Storage strategy. At each storage point, SRRTA records all active variables and their values regardless of their types, so as to guarantee the program state is reusable. For example, for variables with basic types (e.g., integer or float), SRRTA directly stores the identifiers and their values; for objects, SRRTA stores the values of all the domain variables. This exhaustive storage mechanism induces extra overheads (i.e., time/space) in the execution of P_0. The overheads are dependent on the number of storage points: recording program states of all instructions can be extremely expensive in time and space, while setting too sparse storage points might be insufficient to capture reusable program states. To balance the effectiveness and efficiency of SRRTA, storage points are supposed to (i) be set as few as possible in P_0 and (ii) help skipping the execution of instructions as many as possible in P_1. Since the intuition of SRRTA is to skip instruction executions by reusing the program states which are safe from regression modifications, an ideal storage strategy is to store the program states immediately before and after the modified code locations, however, which are unavailable before the release of the new version program P_1. Therefore, SRRTA is designed to guard time-consuming code locations with storage points. In particular, we propose three heuristic storage strategies by setting storage points in entries/exits of loops ($Store_{loop}$), method invocations ($Store_{method}$) and conditional statements ($Store_{branch}$), which are often involving quantities of instruction executions and can be quite time consuming for execution. In this paper, we use the combination of these heuristic storage strategies (i.e., $Store_{all}$) as the default setting, and investigate the impact of different strategies in Section 4.

In Figure 1, storage points 2 and 3 are guarding a loop structure which spans over 91 instructions (i.e., from I_{Norm_3} to $I_{Norm_{94}}$) since the loop statements (e.g., `j < len`) are executed repeatedly.

2.3 State Loading

Algorithm 1 presents the state loading process of SRRTA. Given the execution trace *trace* on P_0 and the program states of each storage instruction (i.e., $State(P_0, I_{Store})$), for any two adjacent storage instructions (i.e., I_{Store_i} and $I_{Store_{i+1}}$), if the instructions between them are not affected by any modification, SRRTA skips their executions and directly loads the stored program state on $I_{Store_{i+1}}$, which is a *safe transition* from I_{Store_i} to $I_{Store_{i+1}}$ and the *trace* on P_1 between them is the same as P_0 (i.e., given in Line 19 - Line 21). Otherwise, SRRTA executes these instructions to avoid violating the validity of program P_1 and the *trace* on P_1 can be acquired through execution (i.e., given in Line 23 - Line 27).

Safe transition. A transition from I_{Store_i} to $I_{Store_{i+1}}$ is regarded as a safe transition, only if between I_{Store_i} and $I_{Store_{i+1}}$ (i) there is no modified instruction; and (ii) there is no instruction accessing the variables whose values are changed in the program states on I_{Store_i} (i.e., given in Line 7 - Line 17). For the latter, SRRTA (i) first compares the program states on I_{Store_i} of the old program P_0 and

[1] Active variables are the variables valid in run-time memory.

Algorithm 1: State loading on P_1 execution

Input: P_0; execution trace $trace_0$; program states $State(P_0, I_{Store})$
Output: P_1; execution trace $trace_1$; program states $State(P_1, I_{Store})$

$trace_1 \leftarrow []$ // Initialize trace in P_1
$I_{cur}, I_{next} \leftarrow Entry$ // Initialize instruction counter
$\mathbb{I}_s \leftarrow StoreIns(trace_0)$ // Initialize storage instruction list
$i \leftarrow 1$
repeat
  $I_{next} \leftarrow I_{storage_i}$
  /* A safe transition or not? */
  $Flag_{safe} \leftarrow True$
  /* Compute changed variables between P_0 and P_1 */
  $V_p \leftarrow Diff(State(P_0, I_{cur}), State(P_1, I_{cur}))$
  for I *in* $trace_0[I_{cur}, I_{next}]$ **do**
    if $I \in I_{Mod}$ **then**
      $Flag_{safe} \leftarrow False$ // Modified Instructions
      break;
    if V_p *accessed by* I **then**
      $Flag_{safe} \leftarrow False$ // Polluted variables accessed
      break;
  /* A safe transition from I_{cur} to I_{next} */
  if $Flag_{safe}$ **then**
    $trace_1.append(trace_0[I_{cur}, I_{next}])$
    // Update the stored variables in V_p
  /* A unsafe transition from I_{cur} to I_{next} */
  else
    $Loading(I_{cur})$ // Load current state
    $Ins \leftarrow Execute(P_1)$
    $trace_1.append(Ins)$
    // Update the stored states
  $I_{cur} \leftarrow I_{next}$
  $i \leftarrow i + 1$
until $i > |\mathbb{I}_s|$
return $trace_1, State(P_1, I_{Store})$

the new program P_1, and identifies the *polluted variables* (which refer to the variables whose values are different between versions and are denoted as V_p); (ii) then performs data dependency analysis to check whether any variable in V_p is accessed by the instructions between I_{Store_i} and $I_{Store_{i+1}}$. In particular, SRRTA only regards read operations as accessing operations, since writing a variable in memory overwrites its original value, which actually eliminates the propagation of the changed values.

From Figure 1, the transition from I_{Store_1} to I_{Store_2} is unsafe since there is a modified instruction between them (i.e., I_{Mod1}), and thus SRRTA executes all instructions between I_{Store_1} and I_{Store_2}. The transition from I_{Store_2} to I_{Store_3} is safe: although program states are different on I_{Store_2}, the polluted variable `c` (addressed by orange in the figure) is not accessed by any instruction between I_{Store_2} to I_{Store_3}. Therefore, SRRTA skips the execution of the entire loop by directly loading the program states on I_{Store_3}.

Loading efficiency optimization. Frequently loading program states can induce extra time costs in regression testing. To reduce the time costs, SRRTA merges successive safe transitions to reduce the times of state loading, and only reloads value-changed variables to reduce the number of loading variables each time. For example, when the transition from I_{Store_i} to $I_{Store_{i+1}}$ and the transition from $I_{Store_{i+1}}$ to $I_{Store_{i+2}}$ are both safe transitions, SRRTA directly loads program states on $I_{Store_{i+2}}$ (i.e., merging the two transitions). In addition, on each storage point $I_{Store_{i+1}}$, SRRTA only loads the variables that are overwritten by instructions between I_{Store_i} and $I_{Store_{i+1}}$. From Figure 1, when loading program states on I_{Store_3}, SRRTA only reloads the variable `this.matrix`, because it is the only variable whose value is modified in the loop.

3 IMPLEMENTATION

SRRTA is implemented with ASM [2], and two challenges exist in the implementation.

Collecting/loading variables values. During run-time, variables are stored in the *local variable table*, whose values cannot be obtained directly. ASM provides bytecode instructions which can place variables in the local variable table to the *operand stack*, where variables can be operated. Therefore, SRRTA first inserts bytecode instructions to transfer values from local variables table to the operand stack and then invokes the store and load functions we define. In this way, the variables and their values during run-time can be further logged into local files.

Transition between storage points. SRRTA realizes transitions between storage points by adding jump (e.g., `GOTO`) instructions between them. Whether a transition is safe and which storage point SRRTA should jump to can only be determined during run-time, but instrumentation has to be conducted before the program execution. To address this problem, when instrumenting P_1, SRRTA marks all the storage points with unique labels, encodes transitions through `GOTO` instructions with these labels, and inserts all the possible transitions on each storage point with conditional judgments. In this way, according to the run-time information and conditional judgments, SRRTA can determine to conduct transitions or not, and decide the next storage point to jump to on-the-fly.

4 PRELIMINARY STUDY

In this work, we study two research questions: (1) **RQ1**: how does SRRTA perform on reducing regression testing time? (2) **RQ2**: how do different storage strategies impact the performance of SRRTA?

4.1 Study Design

Subjects. We evaluate SRRTA on a real-world open-source project Apache Commons Math [1], based on one of whose snapshots [2] we construct 15 pairs of test cases and source code under test. For each method under test, we use the original code as the old program P_0, and construct a new program P_1 by randomly seeding syntactic modification on P_0. To facilitate modification seeding, we adopt popular mutation operators widely used in previous studies [21, 22, 31] (e.g., arithmetic operator replacement or literal value replacement), and randomly decide the locations and number of mutants[3].

Independent variables. Besides the default setting of SRRTA, i.e., the combination of the three storage strategies, we investigate the effectiveness of each individual strategy as well as the random strategy (i.e., $Store_{random}$), which randomly sets the same number of storage points as the best strategy among the three above.

Dependent variables. To measure the performance of SRRTA on regression testing acceleration, we collect the time of regression testing on P_1 without/with SRRTA respectively (i.e., *original time/online time*). Considering the time cost on instrumentation and its impact on P_0 execution, we collect both the instrumentation

[2] https://asm.ow2.io/
[3] Considering the scale of the project, no more than 3 mutants are used in this study.

Strategy	Execution Time on P_1			Extra Overhead			#Success
	Original	Online	Reduced	Collection Time	Ins Time	Space	
$Store_{all}$	13,713.0	2,379.1	11,333.8/82.7%	3,461.6/25.2%	1,469.2	97,593.8	15/15
$Store_{loop}$		2,755.6	10,957.4/79.9%	3,923.7/28.6%	1,538.3	114,100.9	10/10
$Store_{branch}$		6,800.9	6,912.1/50.4%	549.5/4.0%	933.6	5.3	5/10
$Store_{method}$		12,774.7	938.3/6.8%	1,606.1/11.7%	685.7	32,235.4	0/10
$Store_{random}$		8,563.8	5,149.2/37.5%	2,540.1/18.5%	1,056.8	74,156.0	5/15

Table 1: Results of SRRTA with various storage strategies (time is measured in milliseconds and space is measured in kilobytes)

time on P_0 and P_1, and the extra time cost in P_0 execution (compared to no instrumentation), and regard them as *offline* time cost of SRRTA. Moreover, we also record the space cost in storing program states, which is the extra *space overhead* of SRRTA. To eliminate randomness in time collection, we repeat all the experiments 10 times and adopt the average as the final results.

Threats to validity. The internal threat to validity lies in the implementations of our approach. To reduce this threat, two authors have carefully checked the code. The external threat to validity mainly lies in the subjects, including code modifications and test cases. We plan to extend the experiments on more real-world projects in the future. The construction threat to validity mainly lies in the simple statistics used in result analysis. We listed the complete results on the website of this project.

4.2 Result Analysis

Table 1 presents the results of SRRTA with various storage strategies. The former six columns present the average results, while the last column #*Success* presents the number of cases that SRRTA succeeds accelerating regression testing among the test cases that SRRTA can be applied on. Columns "Original " and "Online" show the regression testing time on the new program P_1 without/with SRRTA respectively. Column "Collection Time" shows the extra regression time caused by the instrumented P_0, Column "Ins Time" shows the instrumentation time, and Column "Space" shows the space cost, all of which can be regarded as the time and space cost of SRRTA. For ease of understanding, Column "Reduced" shows the reduced regression testing time achieved by SRRTA and the percentages in the fourth and fifth Columns show the corresponding ratio on the original regression testing time. The complete results and analysis are given in the website: **https://github.com/SRRTA/NIER-SRRTA**.

RQ1: acceleration. Shown by the second row, SRRTA with the default setting successfully accelerates all the 15 regression-testing cases with 82.7. On the other hand, we observe that SRRTA consumes non-trivial time and space during the testing process of the old program P_0, indicating the necessity to alleviate the cost issue of SRRTA in the future.

RQ2: storage strategies. As a storage strategy may not be applicable to all the 15 cases in this study, we show the total number of applicable cases after the "/" in the last column. For example, in a program without loop structures, $Store_{loop}$ strategy is not applicable. From this table, the storage strategy indeed affects the performance of SRRTA. For example, among the three proposed strategies, $Store_{loop}$ achieves the largest number of successful cases, and its online time is the smallest, confirming that loop structure may be usually time consuming in regression testing. We also observe that even $Store_{random}$ successfully accelerates 5 cases, indicating that reusing program states can indeed save a part of execution time; moreover, $Store_{random}$ performs worse than most of the storage strategies, indicating the effectiveness of the proposed heuristic storage strategies.

As for the cases that SRRTA fails to save online regression time, e.g., the online time of SRRTA on some tests increases 103.4ms and 966.4ms respectively, we manually investigate these cases and find the reasons may be that (i) SRRTA has not skipped execution of any instructions (i.e., no safe transition can be conducted) or (ii) the original execution time of the skipped tests is less than the extra costs introduced by SRRTA in run-time. In other words, for the cases that SRRTA significantly reduces the execution time, SRRTA must perform state loading at least once and the skipped instructions are supposed to be quite time consuming; and for the cases with no state loading, SRRTA does not change the original execution behavior and thus brings no improvement in execution efficiency.

5 RELATED WORK

In the literature, test selection, reduction, and prioritization have been proposed to alleviate the cost issue of regression testing [32]. Although a large number of techniques have been proposed in these domains, they address the cost issue of regression testing from the same perspective, optimizing the execution of a test suite, instead of individual test cases, which makes them different from our work.

Besides, there are some techniques (e.g., compiler optimizations) proposed in the literature to accelerate individual executions, but in different scenarios. Our work is inspired by the mutation testing acceleration approach AccMut, which uses only one process to represent each equivalence statements of mutants [31]. The most related work is the VMVM [3], which executes all the unit test cases in one JVM to save the initialization time of each test.

6 CONCLUSION AND FUTURE WORK

To alleviate the regression-testing cost issue, we present SRRTA through state reuse and it is demonstrated to be very promising. In the future, we will improve our work through three aspects.

SRRTA is applicable to consecutive scenario, where SRRTA updates the states if the states are changed when testing the other versions instead of P_0, given by Lines 21 and 27 in Algorithm 1. In the future we plan to investigate the performance of SRRTA in the consecutive scenario.

The extra overhead induced by storage is the main drawback of SRRTA, which may be alleviated by (1) removing unnecessary variables (i.e., no longer used later) from state storage, (2) sharing common states across tests, and (3) improving the storage strategy on time-consuming snippets.

Besides variables' values, code execution may also have other side effects (e.g., I/O operations), which may lead to safety issues of SRRTA and may be hidden behind call chains and is not easily identified. Therefore, in the future we plan to improve SRRTA by addressing these safety issues.

ACKNOWLEDGMENTS

This work was partially supported by National Natural Science Foundation of China under Grant Nos. 61872008 and 61861130363.

REFERENCES

[1] Apache. 2020. Commons Math. https://commons.apache.org/proper/commons-math/. Accessed May-12-2020.

[2] Apache. 2020. Commons Math. https://github.com/apache/commons-math/tree/bbfe7e4ea526e39ba0a79f0258200bc0d898f0de. Accessed May-12-2020.

[3] Jonathan Bell and Gail Kaiser. 2014. Unit test virtualization with VMVM. In *Proceedings of the 36th International Conference on Software Engineering*. 550–561.

[4] Lionel C Briand, Yvan Labiche, and George Soccar. 2002. Automating impact analysis and regression test selection based on UML designs. In *International Conference on Software Maintenance, 2002. Proceedings.* IEEE, 252–261.

[5] J Chen, Y Bai, D Hao, Y Xiong, H Zhang, L Zhang, and B Xie. 2016. A text-vector based approach to test case prioritization. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 266–277.

[6] Junjie Chen, Yanwei Bai, Dan Hao, Lingming Zhang, Lu Zhang, and Bing Xie. 2017. How do assertions impact coverage-based test-suite reduction?. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 418–423.

[7] Junjie Chen, Yiling Lou, Lingming Zhang, Jianyi Zhou, Xiaoleng Wang, Dan Hao, and Lu Zhang. 2018. Optimizing test prioritization via test distribution analysis. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 656–667.

[8] Pavan Kumar Chittimalli and Mary Jean Harrold. 2009. Recomputing Coverage Information to Assist Regression Testing. *IEEE Transactions on Software Engineering* 35, 4 (2009), 452–469. https://doi.org/10.1109/TSE.2009.4

[9] Daniel Di Nardo, Nadia Alshahwan, Lionel Briand, and Yvan Labiche. 2015. Coverage-based regression test case selection, minimization and prioritization: A case study on an industrial system. *Software Testing, Verification and Reliability* 25, 4 (2015), 371–396.

[10] Hyunsook Do and Gregg Rothermel. 2006. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering* 32, 9 (2006), 733–752.

[11] Susumu Fujiwara, G v Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. 1991. Test selection based on finite state models. *IEEE Transactions on software engineering* 6 (1991), 591–603.

[12] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 211–222.

[13] Dan Hao, Lu Zhang, and Hong Mei. 2016. Test-case prioritization: achievements and challenges. *Frontiers of Computer Science* 10, 5 (2016), 769–777.

[14] Dan Hao, Lu Zhang, Xingxia Wu, Hong Mei, and Gregg Rothermel. 2012. On-demand test suite reduction. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 738–748.

[15] Dan Hao, Lu Zhang, Lei Zang, Yanbo Wang, Xingxia Wu, and Tao Xie. 2015. To be optimal or not in test-case prioritization. *IEEE Transactions on Software Engineering* 42, 5 (2015), 490–505.

[16] Dan Hao, Lingming Zhang, Lu Zhang, Gregg Rothermel, and Hong Mei. 2014. A unified test case prioritization approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 2 (2014), 1–31.

[17] Dan Hao, Xu Zhao, and Lu Zhang. 2013. Adaptive test-case prioritization guided by output inspection. In *2013 IEEE 37th Annual Computer Software and Applications Conference*. IEEE, 169–179.

[18] Mary Jean Harrold, James A Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S Alexander Spoon, and Ashish Gujarathi. 2001. Regression test selection for Java software. *ACM Sigplan Notices* 36, 11 (2001), 312–326.

[19] Hadi Hemmati and Lionel Briand. 2010. An industrial investigation of similarity measures for model-based test case selection. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*. IEEE, 141–150.

[20] James A Jones and Mary Jean Harrold. 2003. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Transactions on software Engineering* 29, 3 (2003), 195–209.

[21] René Just. 2014. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. San Jose, CA, USA, 433–436.

[22] René Just, Bob Kurtz, and Paul Ammann. 2017. Inferring Mutant Utility from Program Context. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. Santa Barbara, CA, USA, 284–294.

[23] Yiling Lou, Junjie Chen, Lingming Zhang, and Dan Hao. 2019. A survey on regression test-case prioritization. In *Advances in Computers*. Vol. 113. Elsevier, 1–46.

[24] Yiling Lou, Dan Hao, and Lu Zhang. 2015. Mutation-based test-case prioritization in software evolution. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 46–57.

[25] Yafeng Lu, Yiling Lou, Shiyang Cheng, Lingming Zhang, Dan Hao, Yangfan Zhou, and Lu Zhang. 2016. How does regression test prioritization perform in real-world software evolution?. In *Proceedings of the 38th International Conference on Software Engineering*. 535–546.

[26] Hong Mei, Dan Hao, Lingming Zhang, Lu Zhang, Ji Zhou, and Gregg Rothermel. 2012. A static approach to prioritizing junit test cases. *IEEE transactions on software engineering* 38, 6 (2012), 1258–1275.

[27] Stephen Prata. 2014. *C primer plus*. Pearson Education.

[28] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test case prioritization: An empirical study. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*. IEEE, 179–188.

[29] Ripon K Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E Perry. 2015. An information retrieval approach for regression test prioritization based on program changes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 268–279.

[30] August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. 2014. Balancing trade-offs in test-suite reduction. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 246–256.

[31] Bo Wang, Yingfei Xiong, Yangqingwei Shi, Lu Zhang, and Dan Hao. 2017. Faster Mutation Analysis via Equivalence modulo States. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. Santa Barbara, CA, USA, 295–306.

[32] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability* 22, 2 (2012), 67–120. https://doi.org/10.1002/stv.430

[33] Lingming Zhang, Dan Hao, Lu Zhang, Gregg Rothermel, and Hong Mei. 2013. Bridging the gap between the total and additional test-case prioritization strategies. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 192–201.