Orchestration Strategies for Regression Test Suites

Renan Greca*†, Breno Miranda‡, Antonia Bertolino†
renan.greca@gssi.it, bafm@cin.ufpe.br, antonia.bertolino@isti.cnr.it
**Gran Sasso Science Institute*, L'Aquila, Italy, ‡*Federal University of Pernambuco*, Recife, Brazil, †*ISTI-CNR*, Pisa, Italy

***Abstract*—Regression testing is widely studied in the literature, although most research on the topic is concerned with improving specific sub-challenges of a wider goal. *Test suite orchestration* proposes a more comprehensive view of the challenge of regression testing, by merging and combining different techniques with a variety of objectives, including prioritizing, selecting, reducing and amplifying tests, detecting flaky tests and potentially more. This paper presents the key approaches and techniques that form test suite orchestration, along with common evaluation metrics, and discusses how they can be used together to ultimately provide an efficient and effective regression testing strategy. To illustrate the benefits of orchestration, we provide some examples of existing papers that take steps towards this goal, even if the specific terminology is not yet used. Orchestrated strategies utilizing existing regression testing techniques provide a pathway to practicality and real-world usage of the academic literature.**

***Index Terms*—software testing, regression testing, test case selection, test case prioritization, test suite reduction, test suite amplification, flaky test detection, test suite orchestration**

I. Introduction

It is important for companies and communities developing software to utilize methods to mitigate the possibility that faulty software will reach production. Today, most commercial and open-source software products are accompanied by a *test suite*, a series of automated tests that are used to provide a level of certainty that parts of a software, both in isolation and in conjunction, correctly perform the tasks to which they are assigned. One widely-adopted software testing technique is called *regression testing* (RT); its primary role is to execute the test suite with a certain frequency, in order to guarantee that recently introduced changes to the software do not affect previously-correct behavior. However, in large-scale software development (that is, with multiple developers and a large codebase), it is usually unfeasible to execute every test after every change, either because changes are too frequent, or because there are too many tests, or both. This is aggravated by the fact that most software is now developed in a *continuous* manner, i.e., following an iterative and cyclical process that allows for a short turnaround time between the design of a requirement, the development of a feature, and the delivery of an update to customers.

Our research covers *test suite orchestration strategies* for regression testing, which address the objective of managing and combining multiple RT techniques: test case prioritization (TCP), test case selection (TCS), test suite reduction (TSR), test suite amplification (TSA), and flaky test detection (FTD).

A framework for test suite orchestration strategies is introduced, illustrating how multiple techniques, each handling a distinct challenge, can be used in unison to improve the testing workflow. While the idea of combining RT techniques is not entirely novel, most RT research focuses on improving one or another specific technique without evidencing how different combinations of techniques might yield differing cost/effectiveness ratios [1]. The objective of this paper is to provide an overview of how differently RT techniques can be orchestrated, while providing common terminology for the advancement of this direction of research.

II. Background

Given the challenges associated with ever-expanding regression testing suites of continuously evolving software, we call *test suite orchestration* the art of generating, choosing, prioritizing and executing tests in order to maximize the effectiveness of testing while keeping costs within a desired budget. Our adoption of the term "orchestration" has been inspired by a concept in service-oriented architecture, where centralized logic manages the interoperability between members of a system [2]. Today, research on test orchestration is quite granular, with individual researchers mostly focusing on specific challenges within this topic. While this is important for the continuity and advancement of research, it fails in addressing the practical concerns of software developers, who desire a complete solution to aid the development cycle.

In general, test suite orchestration can be thought of as a broad challenge with the ultimate goal of improving regression testing in multiple aspects, composed of several sub-challenges, which are in turn addressed by groups of techniques [3] [4], namely the aforementioned TCP, TCS, TSR, TSA, and FTD, as well as others that could be considered as extensions.

Individually, each of these challenges can be its own field of research, and indeed many works have been published on them, introducing improvements that can provide substantial benefits. A specific configuration of techniques is referred to as an *orchestration strategy*; ideally, it should consider all RT challenges in unison, as solving each one alone is not sufficient to solve the problems faced by software developers in practice. Additional challenges and techniques remain relevant to the discussion of test suite orchestration and may be added to an overarching strategy, or some may be omitted due to specific constraints. In this section, common approaches and metrics for each sub-challenge are briefly summarized.

A. Test Case Prioritization

One challenge of regression testing is to detect failing tests fast. The objective of *test case prioritization* (TCP) is to re-

2833-9061/23/$31.00 ©2023 IEEE
DOI 10.1109/AST58925.2023.00020

order test cases according to some definition of priority, in order to get faster feedback from the test execution [5]. Given a system-under-test (SUT) M and its test suite T, TCP can be described as a function $P(M, T)$ that provides a permutation T' of T, such that, given a metric function f, $f(T') > f(T)$. The optimal prioritization is one for which $f(T')$ is greater than or equal to any other possible permutation of T.

Among the criteria often used for TCP: *(1)* similarity-based attempts to diversify the execution of tests; *(2)* coverage-based aims at maximizing coverage with as few tests as possible; and *(3)* history-based prioritizes tests that have a history of failing or revealing faults [6].

Common metrics include: *(1)* average percentage of faults detected (APFD), which estimates how effective a prioritization is in detecting faults in fewer tests; *(2)* tests till first fault (TTFF), a count of how many tests were executed until one failed; or *(3)* developer feedback time, a measure of how long it takes for a developer to get a report if there is a failing test.

TCP is particularly useful in situations where the test suite is exceptionally large and detecting failures sooner allows for potential faults to be addressed quicker. It's also relevant in cases where the testing budget is limited but not consistent, so testing might stop at any time and only tests that failed until then can be added to a report.

A prioritized test suite still contains all test cases, so there is no loss of failures detection ability (assuming that test results are independent and the testing budget is sufficient) – what changes is the amount of time that it takes for one or more failures to be detected.

B. Test Case Selection

In regression testing, not all tests are relevant to a particular code change: if only a small part of one file was updated, it is unlikely that the entire project would be affected and the full regression test suite would have to be run. *Test case selection* (TCS)[1] addresses the challenge of selecting a subset of tests that is representative of the entire suite in a given situation [7] [8]. In other words, given subsequent versions of an SUT, M and M' and its test suite T, TCS can be described as a function $S(M, M', T)$ that selects a subset $T' \subseteq T$ to be used for testing M', considering the differences from M to M'.

We say that a TCS technique is *safe* if it guarantees that all tests whose outcome may be affected by a change are included in the selected subset [8]. That is, safe selection techniques output a subset T' while maintaining the output of a fault detection metric function $f(T') \geq f(T)$.

Examples of approaches for TCS are: *(1)* change-based, which executes tests that have some relation to modified files, classes, or methods; *(2)* model-based, which uses data extracted from models of the SUT to determine test execution; or *(3)* graph-based, which uses a graph representation of the SUT to detect control flow and select relevant tests [7].

Some metrics for TCS are: *(1)* selection count or percentage, which measures how many tests were executed in comparison to the original suite (e.g. $|T'| \leq |T|$); *(2)* testing time, or the time taken to execute the selected subset of tests; and *(3)* fault detection capability, used to determine the safeness of the proposed technique.

[1]Also referred to as Regression Test Selection (RTS) in the literature.

A potential drawback of TCS is that, depending on the size of the test suite and the execution time of individual tests, it may happen that the time needed to produce T' is greater than the savings provided by executing T' instead of T.

C. Test Case Reduction and Minimization

Without considering subsequent versions of the SUT, *test suite reduction* (TSR) aims to find a minimal subset of test cases such that the testing requirements are still met [9]. Thus, given an SUT M and its test suite T that satisfies a set of requirements $\{r_1, ..., r_n\}$, we describe TSR as a function $R(M, T, r)$ which outputs a test suite $T' \subseteq T$ such that each r_i is still satisfied.

There exists conceptual overlap between TCS and TSR, with the key differences being change-awareness and the objective of the result. While TCS uses a comparison between versions of the SUT and produces a set of tests meant to validate those changes, TSR can be performed on an isolated iteration of a program and is meant to detect tests that are no longer needed for full satisfaction of the requirements. TSR approaches are often coverage-based or requirements-based and evaluation metrics for TSR are often shared with TCS, since both are concerned with running fewer tests and reducing the overall testing time; however TSR must ensure that there is no loss in fault detection capability in the long-term evolution of the suite.

Regarding the terms reduction and minimization, both options are used nearly interchangeably in the literature. According to Yoo and Harman [4], the difference in terminology is subtle: while both remove tests from the suite, *minimization* implies this change is temporary, while *reduction* stands for permanent removal of tests. Generally speaking, the same techniques can be applied for both ends so, from the perspective of researchers, the two terms are not distinct.

D. Test Case Amplification and Augmentation

As a program evolves and grows in scope, so must its test suite to keep up with the additional features. Today, this is mostly done manually by the development teams, in some cases by the developers themselves; in some teams, developers test each other, or there can be designated testers whose job is to ensure other people's changes satisfy requirements and are error-free.

Unfortunately, tests do not add to the perceived value of a software product, since they do not provide direct functionality to end users. Thus, in a lot of cases, developers are encouraged to spend little time writing new tests or improving existing ones. As such, an automated solution can prove to be valuable to both reduce the manual work done by programmers and to improve the overall quality of new test cases. *Test case amplification* (TSA) is the technique to achieve that goal.

Given an SUT M and its test suite T that *partially* satisfies a set of requirements $\{r_1, ..., r_n\}$, TSA is described as a function $A(M,T,r)$ outputting a test suite $T' \supseteq T$ that satisfies more r_is than T. Much like TSR, coverage-based and requirements-based approaches are common, although the objective is naturally to increase coverage rather than maintain it. Metrics for measuring the output include the relative increase in coverage/requirements.

This problem is related to *test suite generation* (TSG) although, for the discussion of regression testing, TSA is a more useful concept. The difference is that TSA increases or enhances a pre-existing test suite, while TSG generates one from scratch. Since the latter does not presume an ongoing and evolving regression test suite, it was determined that it falls out of the scope of discussion of test suite orchestration. That said, from a purely technical standpoint, both challenges are strongly related and, indeed, TSA is sometimes referred to as *incremental generation* of test cases.

There is a nuanced distinction between the usage of amplification and augmentation, and the two are sometimes conflated in the literature. Generally, *amplification* refers to any improvement of the test suite, which may happen by adding new tests or enhancing existing ones. On the other hand, *augmentation* implies the creation of new tests without modifying previous ones. By this definition, *test suite augmentation* is a problem embedded within test suite amplification, so we can refer as amplification the combined challenge.

E. Test Flakiness

Flaky tests are test cases that can be observed to pass or fail non-deterministically when executed on unchanged code [10]. This is highly undesirable in regression testing, because the non-deterministic behavior of flaky tests result in confounding signals to the testers and developers. The impact of flakiness on regression testing is substantial, because understanding if a test is truly flaky requires a large debugging effort; additionally, false alarms can erode developers' trust in the process rigor and indirectly hinder testing efficacy [11], [12]. Thus, a *flaky test detection* (FTD) technique can provide great benefits to testers.

The straightforward approach to FTD consists of rerunning the failing test cases a number of times, e.g., 10 times [13], to ascertain that failures are not intermittent. However, such approach, which is now embedded in several testing infrastructures, is costly, and researchers have proposed various other approaches that can help reduce the costs of identifying flakiness, either by coupling test rerunning with code analysis techniques that can reduce the amount of test reruns [14], [15], or even before executing the tests, e.g., by using machine learning [16], [17]. FTD techniques are considered dynamic or static depending on whether they are based on test execution or not, respectively.

What is now clear is that any framework for regression testing must be made aware of flakiness and possibly include appropriate mechanisms that can automatically detect flake tests with sustainable efforts. One difficulty is that the causes and characteristics of flakiness can be heterogeneous [10], so possibly more than one technique will be needed. For instance, Lam et al. propose an enhanced regression testing approach that addresses test order dependency [18].

III. A Framework for Test Suite Orchestration

Software regression testing has undergone extensive research in the last several decades. The largest part of solutions, though, address separately one dimension of the problem at a time. We believe that, by orchestrating the differing RT techniques presented in the previous section into a combined execution, we can achieve the most from the restricted subset of test cases that can be executed at each new release.

When combining multiple techniques into a cohesive orchestration strategy, the first and perhaps most important aspect to consider is the sequence of operations. For instance, there are two ways of using TCS and TCP together: we can either select a set of test cases and then prioritize these, or prioritize the entire test suite and run the selected tests in that given order. Including more techniques in the orchestration inevitably leads to more possible sequences.

By adding TSR to the orchestration, this operation could be performed before or after the selection and prioritization. By using it before, we already restrict the number of test cases the other techniques must deal with; doing it afterwards, the results of the reduction will only be used in the next execution of the test suite.

The combination becomes more interesting when adding TSA to the strategy. TSA could be the first technique to run, updating or adding test cases that will then serve as input for selection, prioritization and reduction. Or, it could be placed in between selection and prioritization, modifying the suite only according to the results of the selection. This could be desired if the TSA process is costly and running it with fewer targets greatly reduces the time it consumes.

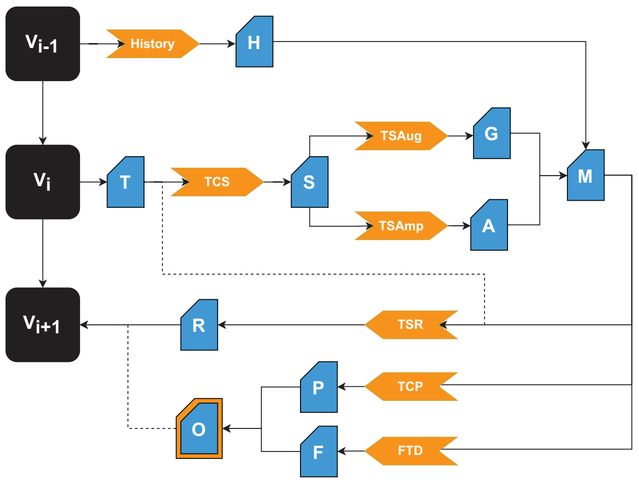
Continuing this line of thought, Figure 1 shows a diagram of a fully orchestrated test strategy. In it, we consider three subsequent versions of the SUT ($\mathbf{v_{i-1}}$, $\mathbf{v_i}$, $\mathbf{v_{i+1}}$). The chevron boxes represent some process being applied to the tests, while the cut rectangles represent variations of the test suite (e.g., a list of test cases).

The target of the orchestration is **T**, which is the test suite corresponding to version $\mathbf{v_i}$ of the SUT. The first technique to be applied is TCS, generating a subset of tests **S**. Additionally, from previous test execution logs, historical data, such as test that have recently failed, can be extracted, forming the set **H**.

This subset is then used as input for TSA techniques, in this example displayed separately as augmentation and amplification. The results are one set of *newly generated* additional tests **G** and one set **A** containing the amplified versions of the tests. At this point, information from **H**, **G** and **A** is merged into a list of tests **M**.

M is then used as input for three different techniques. On one side, TSR is used, using information from **M** and **T** to eliminate excessive redundancies in the suite and produces a tighter suite **R** that can be used as a starting point for the next cycle of orchestration (when it is time for version $\mathbf{v_{i+1}}$ of the SUT to be tested). On the other, TCP prioritizes the test cases to **P** and a static FTD technique provides a list **F** of potentially unreliable tests, which should be handled differently during execution.

Finally, the orchestrated test suite **O** is produced, which can be used to test the SUT version $\mathbf{v_i}$. Considering the cyclical nature of testing evolving systems, information produced by the execution of **O** can be stored and used as historical data available for future testing cycles, as well as for dynamic FTD.



Legend: $\mathbf{v_{i-1}}$, $\mathbf{v_i}$, $\mathbf{v_{i+1}}$: previous, current and next version of the SUT; **H**: output of history-based criteria; **T**: the test suite as of version $\mathbf{v_i}$; **S**: the selected test suite; **G**, **A**: the outputs of test suite augmentation and amplification, respectively; **M**: a *selected and enhanced* test suite combining the outputs of the previous steps; **R**: the reduced test suite; **P**: the prioritized test suite; **F**: a list of potentially flaky tests; **O**: the *orchestrated test suite* that should be executed for $\mathbf{v_i}$, the output of which becomes historical data for $\mathbf{v_{i+1}}$.

Fig. 1: Diagram showing a fully orchestrated strategy to the test suite execution and evolution.

IV. Examples of Test Suite Orchestration

In a previous publication, we presented a study directly comparing two recent practical and effective approaches to TCS and TCP, namely file-based selection (by Ekstazi [19]) and similarity-based prioritization (by FAST [20]) [21]. Our results show that Ekstazi generally outperforms FAST, although the effect size is negligible or small; however, their orchestration by Fastazi outperforms both with a non-negligible effect. Moreover, considering a limited test budget, Fastazi exposed a higher effectiveness in consistent way. After assessing the overhead imposed by each of the studied approaches, we can conclude that Fastazi is quite practical: if we parallelize the preparation steps, the additional cost of similarity-based prioritization of the test cases selected by Ekstazi is negligible.

Another example of a work addressing test suite orchestration (although the authors do not use this term) is the study by Shi et al. [22], who compare empirically TSR and TCS. The authors observe that both techniques aim at running only a subset of the test suite, but in orthogonal way. They thus ask which one is better considering the size of the reduced test suite, and the loss in terms of detection capability of change-related faults. For TCS their study adopts the Ekstazi tool above mentioned, whereas for reduction they remove redundant test cases using a greedy heuristic based on statement coverage. From the comparison they conclude that TCS on average returns a smaller test suite size, with no loss in change-related faults detection for safe TCS techniques. In comparison, TSR can miss a small percentage of change-related faults. They also evaluate a combination of the two techniques, specifically “selection of reduction” in which TCS is applied on the reduced suite obtained by TSR: this can further reduce the number of tests yielding the same loss in fault detection observed for TSR alone.

The above examples are just some initial studies towards test suite orchestration. This work paves the way to exploring a full range of potential strategies of combining differing criteria for the various RT techniques in different ways.

V. Conclusion

This paper presents a framework for test suite orchestration, an example built upon the goals of each RT technique which considers how they can be used to each others’ advantages. Validating such a model requires extensive experimentation, which unfortunately poses a technical challenge, as not every technique has an available and easily usable implementation. Even when the tools exist, the way each one handles inputs and outputs can be incompatible, so some alteration is needed.

The study of orchestration strategies might be perceived as an engineering effort that just consists of integrating existing tools and hence as such it does not bring new knowledge. We firmly contrast such view and believe that at the current status of RT research, we have enough choice of RT techniques, and the challenge in practice is which and how techniques would be more usefully combined.

Indeed, several questions remain unanswered regarding a fully orchestrated strategy. The possibility of executing all RT techniques at each new version of the SUT largely depends on the intervals between versions; if new versions are committed frequently, there might not be enough time to execute the full process. In such cases, an additional point to consider is which techniques are important for frequent execution, and which ones can be used in less frequent (e.g. nightly) testing.

For facilitating the progress in RT orchestration, it is very important that tools supporting the proposed techniques are made available for use by other researchers, and that proper benchmarks are shared for comparison of results from differing combinations. With this aim, our cited review [1] is conceived as a live repository and we invite colleagues from the AST community to contribute actively to keep it up-to-date.

Acknowledgements

This research was partially supported by a FACEPE grant (APQ-0783-1.03/21).

References

[1] R. Greca, B. Miranda, and A. Bertolino, "State of practical applicability of regression testing research: A live systematic literature review," *ACM Computing Surveys*, Jan. 2023, DOI: 10.1145/3579851.

[2] T. Erl, *Service-oriented architecture: concepts, technology, and design*. Pearson, 2005.

[3] H. Do, "Recent Advances in Regression Testing Techniques," in *Advances in Computers*. Elsevier, 2016, vol. 103, pp. 53–77, DOI: 10.1016/bs.adcom.2016.04.004.

[4] 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, 2012, DOI: 10.1002/stvr.430.

[5] Y. Lou, J. Chen, L. Zhang, and D. Hao, "A Survey on Regression Test-Case Prioritization," *Advances in Computers*, vol. 113, pp. 1–46, 2019, DOI: 10.1016/bs.adcom.2018.10.001.

[6] M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, and R. Tumeng, "Test case prioritization approaches in regression testing: A systematic literature review," *Information and Software Technology*, vol. 93, pp. 74–93, Jan. 2018, DOI: 10.1016/j.infsof.2017.08.014.

[7] R. Kazmi, D. N. A. Jawawi, R. Mohamad, and I. Ghani, "Effective Regression Test Case Selection: A Systematic Literature Review," *ACM Computing Surveys*, vol. 50, no. 2, pp. 1–32, Jun. 2017, DOI: 10.1145/3057269.

[8] G. Rothermel and M. J. Harrold, "A framework for evaluating regression test selection techniques," in *International Conference on Software Engineering*, 1994, pp. 201–210, DOI: 10.1109/ICSE.1994.296779.

[9] S. U. Rehman Khan, S. P. Lee, N. Javaid, and W. Abdul, "A Systematic Review on Test Suite Reduction: Approaches, Experiment's Quality Evaluation, and Guidelines," *IEEE Access*, vol. 6, pp. 11 816–11 841, 2018, DOI: 10.1109/ACCESS.2018.2809600.

[10] O. Parry, G. M. Kapfhammer, M. Hilton, and P. McMinn, "A survey of flaky tests," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 1, oct 2021, DOI: 10.1145/3476105.

[11] A. Memon, Z. Gao, B. Nguyen, S. Dhanda, E. Nickell, R. Siemborski, and J. Micco, "Taming google-scale continuous testing," in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP 17. IEEE Press, 2017, pp. 233–242, DOI: 10.1109/ICSE-SEIP.2017.16.

[12] M. Harman and P. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis," in *Proceedings of the 18th International Working Conference on Source Code Analysis and Manipulation (SCAM 18)*. IEEE, 2018, pp. 1–23, DOI: 10.1109/SCAM.2018.00009.

[13] J. Micco, "The State of Continuous Integration Testing @Google," 2017, accessed: 2023-01-25. [Online]. Available: https://ai.google/research/pubs/pub45880

[14] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically detecting flaky tests," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE 18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 433–444, DOI: 10.1145/3180155.3180164.

[15] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "idflakies: A framework for detecting and partially classifying flaky tests," in *2019 12th ieee conference on software testing, validation and verification (icst)*. IEEE, 2019, pp. 312–322, DOI: 10.1109/ICST.2019.00038.

[16] G. Pinto, B. Miranda, S. Dissanayake, M. d'Amorim, C. Treude, and A. Bertolino, "What is the vocabulary of flaky tests?" in *Proceedings of the 17th International Conference on Mining Software Repositories*, ser. MSR '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 492–502, DOI: 10.1145/3379597.3387482.

[17] R. Verdecchia, E. Cruciani, B. Miranda, and A. Bertolino, "Know you neighbor: Fast static prediction of test flakiness," *IEEE Access*, vol. 9, pp. 76 119–76 134, 2021, DOI: 10.1109/ACCESS.2021.3082424.

[18] W. Lam, A. Shi, R. Oei, S. Zhang, M. D. Ernst, and T. Xie, "Dependent-test-aware regression testing techniques," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 298–311, DOI: 10.1145/3395363.3397364.

[19] M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test selection with dynamic file dependencies," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 211–222, DOI: 10.1145/2771783.2771784.

[20] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino, "FAST approaches to scalable similarity-based test case prioritization," in *International Conference on Software Engineering*, 2018, pp. 222–232, DOI: 10.1145/3180155.3180210.

[21] R. Greca, B. Miranda, M. Gligoric, and A. Bertolino, "Comparing and combining file-based selection and similarity-based prioritization towards regression test orchestration," in *Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test*, 2022, pp. 115–125, DOI: 10.1145/3524481.3527223.

[22] A. Shi, T. Yung, A. Gyori, and D. Marinov, "Comparing and combining test-suite reduction and regression test selection," in *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, 2015, pp. 237–247, DOI: 10.1145/2786805.2786878.