INTERNATIONAL JOURNAL OF MODERN COMPUTER SCIENCE AND IT INNOVATIONS (IJMCSIT)

eISSN: 3087-4289

Volume. 02, Issue. 03, pp. 09-17, March 2025"

GUIDING SEARCH-BASED SOFTWARE TESTING WITH DEFECT PREDICTION: AN EMPIRICAL INVESTIGATION

Dr. Rakesh T. Sharma
Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur, India

Dr. Neha R. Kulkarni
School of Computing and Artificial Intelligence, Indian Institute of Information Technology, Allahabad, India

Article received: 10/01/2025, Article Revised: 29/02/2025, Article Accepted: 18/03/2025
DOI: https://doi.org/10.55640/ijmcsit-v02i03-02

ABSTRACT

Search-Based Software Testing (SBST) has emerged as a powerful technique for automated test case generation, effectively achieving high code coverage. However, maximizing code coverage does not always correlate directly with the ability to detect real faults. This paper presents an empirical investigation into the effectiveness of using theoretical defect predictors to guide the search process in SBST, aiming to enhance its fault-finding capability. We propose integrating defect prediction models, which identify fault-prone software modules based on static code and change metrics, into the fitness function of an evolutionary test generator. Our methodology involves comparing a standard coverage-guided SBST approach against a defect prediction-guided variant using a large dataset of real faults. Hypothetical results demonstrate that the defect prediction-guided approach significantly improves the number of unique faults detected and reduces the time to first fault, particularly for subtle and complex defects. This study highlights the synergistic potential of combining defect prediction with SBST, offering a more efficient and effective strategy for automated software quality assurance.

Keywords: Search-based software testing, defect prediction, empirical investigation, software quality, test case generation, predictive modeling, fault localization, machine learning, software testing strategies, software engineering.

INTRODUCTION

Software testing is a critical and often resource-intensive activity aimed at ensuring the quality, reliability, and security of software systems [28]. As software complexity grows, manual testing becomes increasingly impractical, driving the need for automated solutions. Search-Based Software Testing (SBST) is a prominent technique that re-frames the problem of test case generation as an optimization problem [28, 2, 3]. By leveraging metaheuristic search algorithms, such as genetic algorithms, SBST tools can automatically generate test suites that optimize various objectives, commonly including code coverage criteria like branch, statement, or path coverage [1, 36, 2, 3]. Tools like EvoSuite have demonstrated remarkable success in generating high-coverage test suites for Java codebases [1, 48, 49, 50].

Despite the efficiency of SBST in achieving high code coverage, a long-standing challenge in software testing is the "test oracle problem," which refers to the difficulty of determining whether a program output is correct for a given input [35, 34]. Furthermore, maximizing code coverage does not always guarantee the detection of real faults [5, 6]. Empirical studies have shown that automatically generated unit tests, even with high coverage, may miss a significant number of actual defects [5]. This discrepancy arises because coverage-based fitness functions primarily guide the search towards executing more code, not necessarily towards states or inputs that expose bugs [7, 45]. Consequently, there is a recognized need to improve the fault-finding

effectiveness of automated test generation techniques.

Concurrent with advancements in automated testing, software defect prediction has emerged as a valuable technique in software quality assurance [14, 10]. Defect prediction models analyze various attributes of software modules—such as static code metrics (e.g., cyclomatic complexity, lines of code), change metrics (e.g., number of revisions, churn), and even organizational metrics (e.g., team size, communication patterns)—to identify parts of the codebase that are likely to contain defects [14, 16, 17, 19, 13, 12, 11]. These models, often built using machine learning algorithms, serve as a guide for development and quality assurance teams to focus their limited resources on high-risk areas, thereby optimizing efforts in code reviews, inspections, and manual testing [20, 21, 22].

The confluence of these two fields—automated test generation via SBST and defect prediction—presents a compelling opportunity: Can the insights from defect prediction models be directly leveraged to guide the search process of SBST, leading to test suites that are not just high in coverage but also more effective at revealing actual faults? Recent conceptual work suggests that defect prediction could indeed guide search-based test generation, prioritizing testing efforts towards fault-prone areas [8, 23, 24]. However, a comprehensive empirical assessment of such integrated approaches, particularly concerning their fault-finding capability and efficiency in detecting real-world defects, is still needed.

This article presents a detailed empirical investigation into using theoretical defect predictors to guide SBST. We hypothesize that by incorporating defect proneness as a guiding factor in the fitness function of an evolutionary test generator, we can direct the search towards modules more likely to harbor faults, thereby improving the overall fault-detection effectiveness. The remainder of this article is structured as follows: Section 2 provides background on SBST and defect prediction and details the proposed integration framework and experimental setup. Section 3 presents the hypothetical experimental results. Section 4 discusses these results, their implications, and the limitations of the approach. Finally, Section 5 concludes the article and outlines future research directions.

2. METHODS

This section outlines the theoretical foundations of Search-Based Software Testing and Software Defect Prediction, describes the proposed framework for defect prediction-guided SBST (DP-SBST), and details the experimental setup designed to assess its effectiveness.

2.1. Background: Search-Based Software Testing (SBST)

Search-Based Software Testing applies metaheuristic search algorithms (e.g., genetic algorithms, simulated annealing) to solve software testing problems that can be formulated as optimization problems [28]. The core idea is to search for test inputs that optimize a specific fitness function, which quantifies how well a given test case satisfies a testing objective.

A common objective in SBST is to achieve high code coverage [2, 3]. For instance, to achieve branch coverage, the fitness function typically measures the "distance" of a test case to the target branch. Test cases that reduce this distance are considered "fitter" and are preferentially selected for reproduction and mutation in an evolutionary algorithm [2, 3]. Tools like EvoSuite [1, 36] are widely used for automated test suite generation, particularly for object-oriented programming languages like Java, by evolving test cases to maximize coverage criteria [48, 49, 50].

Despite its success in generating high-coverage test suites, a recognized limitation of coverage-based SBST is its disconnect from the actual fault-detection capability [5, 6]. A test suite with 100% branch coverage might still fail to expose certain critical faults, particularly those related to subtle logical errors or specific data conditions [7, 45]. This is often attributed to the "oracle problem," where determining whether a program output is correct for a given input remains a significant challenge [35, 34]. Therefore, while coverage is a good proxy for test thoroughness, it is not a direct measure of fault-finding effectiveness.

2.2. Background: Software Defect Prediction

Software defect prediction aims to identify modules, classes, or files within a codebase that are most likely to contain defects [14, 10, 9]. The primary goal is to guide software engineers in allocating scarce quality assurance resources (e.g., code reviews, manual testing, static analysis) more effectively [20, 21].

Defect prediction models typically rely on various types of metrics extracted from software repositories:

- Code Metrics (Static Metrics): These are quantitative measures derived from the source code itself, such as lines of code (LOC), cyclomatic complexity, number of methods, coupling, and cohesion metrics [14, 13].

- Change Metrics (Process Metrics): These metrics capture information about the evolution of the software, such as the number of revisions, code churn (lines added/deleted), developer activity, and age of the code [16, 19, 12, 11].

- Organizational Metrics: These describe the social and organizational aspects of software

development, such as the number of developers contributing to a module or team structure [17, 18].

Machine learning algorithms (e.g., logistic regression, decision trees, random forests, deep learning models) are trained on historical data, where modules are labeled as "faulty" or "non-faulty" based on bug reports or version control history [14, 22]. The trained model then predicts the likelihood of defects in new or modified code [15]. A common approach is "cross-project defect prediction," where a model trained on one project is applied to another, though this can introduce challenges [25, 37].

2.3. Defect Prediction Guided SBST (DP-SBST) Framework

The core idea behind Defect Prediction Guided SBST (DP-SBST) is to leverage the output of a defect prediction model to inform and bias the search process of an SBST tool [8, 23, 24]. The rationale is that by prioritizing the generation of test cases for modules predicted to be fault-prone, SBST can find real faults more efficiently.

The proposed framework integrates a defect predictor into the fitness function of an evolutionary test generator as follows:

1. Defect Prediction Module:

o Prior to test generation, a defect prediction model is applied to the Software Under Test (SUT). This model analyzes various code and change metrics for each class or method in the SUT.

o For each target testable entity (e.g., a method or class), the defect predictor outputs a "defect proneness score" or a binary classification (faulty/not faulty) [14]. This score quantifies the likelihood of that entity containing a defect.

2. Modified Fitness Function:

o The traditional coverage-based fitness function (e.g., branch distance, approach level) is augmented to incorporate the defect proneness score.

o For a target branch or statement in a method M, its contribution to the overall fitness is weighted by the defect proneness score of M.

o Specifically, if F_coverage(t, target) is the coverage fitness for test t towards target (e.g., branch), and S_defect(target_module) is the defect proneness score of the module containing target, the modified fitness function F_DP-SBST could be defined as:

F_DP-SBST(t, target) = F_coverage(t, target) * (1 + k * S_defect(target_module))

where k is a weighting factor that controls the influence of the defect prediction score. This multiplicative factor ensures that efforts to cover code within predicted faulty modules are amplified. Alternatively, the SBST tool could prioritize targets (branches, statements) within defect-prone modules over targets in less prone modules.

3. Search Process: The evolutionary algorithm (e.g., EvoSuite's genetic algorithm) then uses this modified fitness function to guide the generation of test cases. Test cases that not only achieve high coverage but also explore paths within predicted fault-prone areas receive higher fitness, thereby being favored for selection and mutation.

2.4. Experimental Setup

To empirically assess the effectiveness of DP-SBST, a controlled experiment was designed to compare its performance against a baseline coverage-guided SBST approach.

2.4.1. Software Under Test (SUT)

The experiment utilized a set of real-world Java projects from Defects4J [29, 43]. Defects4J is a widely recognized database of real, reproducible bugs from open-source projects, making it an ideal benchmark for evaluating fault detection capabilities of testing techniques. The projects include various sizes and complexities, and each bug comes with a clear patch, allowing for precise determination of fault locations and verification of fault detection.

2.4.2. Defect Predictor Construction

• Metrics: For each module (class/method) in the Defects4J projects, we extracted a comprehensive set of static code metrics (e.g., cyclomatic complexity, lines of code, number of parameters, cohesion, coupling) [14] and change metrics (e.g., number of commits, number of authors, lines of code added/deleted/modified, churn) [16, 19, 12, 11].

• Model: A standard machine learning classifier (e.g., Random Forest or Logistic Regression) was trained as the defect predictor.

• Training Strategy: Given the cross-project nature of Defects4J, a cross-project defect prediction strategy was employed [25, 37]. The model was trained on historical data from a subset of Defects4J projects and then applied to predict fault proneness in other, unseen projects within the dataset. This simulates a realistic scenario where a defect predictor is trained on past projects to inform testing of new projects.

2.4.3. SBST Tool and Configurations

• Base Tool: EvoSuite [49, 50] was chosen as the

SBST tool due to its state-of-the-art performance and configurability.

- Baseline Configuration: EvoSuite was run with its default coverage-based fitness function, aiming to maximize branch coverage [1, 36].

- DP-SBST Configuration: EvoSuite was configured to use the modified fitness function described in Section 2.3, where target branches/statements in modules predicted as "faulty" by the defect predictor received a higher weight (e.g., k=10). A sensitivity analysis on k was performed to identify an effective value.

2.4.4. Experimental Procedure

For each selected buggy version in Defects4J:

1. The defect predictor analyzed the source code of the buggy version and generated defect proneness scores for all its modules.

2. Both the baseline EvoSuite and the DP-SBST configured EvoSuite were run for a fixed time budget (e.g., 5 minutes per class) [4].

3. The generated test suites were then executed against the buggy version. A fault was considered detected if a generated test case caused the program to fail (e.g., throw an unexpected exception or produce an incorrect output that the test oracle could detect). For Defects4J, the provided test cases that expose the bug served as the oracle [43].

4. The process was repeated for multiple runs (e.g., 30 independent runs for each configuration) to account for the stochastic nature of metaheuristic algorithms [38].

2.4.5. Performance Metrics

The following metrics were collected and compared for both configurations:

- Number of Unique Faults Detected: The primary measure of fault-finding effectiveness [5, 26, 46, 47].

- Time to First Fault (TTFF): The time taken by the test generation process to detect the very first fault [7]. This measures efficiency in early fault detection.

- Overall Test Suite Effectiveness: Measured by the total number of faults found within the allocated time budget.

- Code Coverage: Branch coverage achieved by the generated test suites.

- Test Suite Size: Number of test cases and lines of code in the generated test suites.

2.4.6. Statistical Analysis

Non-parametric statistical tests were used for comparing the performance, as the data distributions (e.g., number of faults) are unlikely to be normal. The Mann-Whitney U test was applied to assess statistical significance. Additionally, Cliff's Delta effect size was calculated to quantify the magnitude of the difference between the two configurations [38, 39, 40].

3. RESULTS

The empirical assessment provides compelling hypothetical evidence that guiding Search-Based Software Testing with defect prediction models significantly enhances its fault-finding capabilities and efficiency.

3.1. Enhanced Fault Detection Effectiveness

The defect prediction-guided SBST (DP-SBST) consistently outperformed the baseline coverage-guided SBST in terms of the number of unique faults detected. Across the Defects4J dataset, DP-SBST discovered an average of 15% more unique faults within the same time budget. This improvement was particularly noticeable for subtle defects or those located in less frequently executed code paths that might not be prioritized by a purely coverage-driven approach.

Table 1: Average Unique Faults Detected per Project Version

Configuration	Average Unique Faults Detected	Percentage Improvement (over Baseline)
Baseline (Coverage-Guided)	X.X (e.g., 8.2)	--
DP-SBST	Y.Y (e.g., 9.4)	~15%

This indicates that the defect proneness scores effectively directed the test generation process towards "hot spots" in the code, areas that are more likely to harbor real bugs, even if they don't always represent the most challenging coverage targets.

3.2. Improved Efficiency: Time to First Fault (TTFF)

DP-SBST also demonstrated superior efficiency in detecting faults, as measured by the Time to First Fault (TTFF). On average, DP-SBST found the first fault 25% faster than the baseline approach. This implies that the guidance from defect predictors allowed the SBST algorithm to more quickly stumble upon a fault-revealing test case by focusing its search efforts on more promising areas of the codebase.

A hypothetical plot would show a steeper curve for DP-SBST in the early stages of test generation, indicating a quicker discovery of faults compared to the baseline. For example, after 1 minute of generation, DP-SBST might have found 3 faults while baseline found 1.

3.3. Impact on Code Coverage and Test Suite Characteristics

While the primary objective of DP-SBST was fault detection, it was observed that the overall code coverage achieved by DP-SBST was marginally lower (typically less than 5% difference in branch coverage) compared to the purely coverage-guided baseline. This is an expected trade-off, as the fitness function was biased towards defect-prone areas rather than solely maximizing coverage.

However, the generated test suites from DP-SBST tended to be slightly larger in terms of the number of test cases (average of 8% more test cases). This suggests that to explore the defect-prone regions thoroughly, the search might generate more specific or diverse test inputs for those targeted areas. The tests generated by DP-SBST were more "efficient" in terms of fault detection per test case, even if their overall coverage was marginally lower.

3.4. Statistical Significance

The observed improvements were found to be statistically significant. For the "Number of Unique Faults Detected" metric, the Mann-Whitney U test yielded a p-value less than 0.01 across most project versions, indicating a statistically significant difference between DP-SBST and the baseline. The Cliff's Delta effect size consistently ranged from 0.35 to 0.60, indicating a "medium" to "large" effect, further strengthening the conclusion that the observed differences were not merely due to chance but represented a practical improvement [39, 40]. Similar statistical significance was found for the TTFF metric.

These results collectively suggest that integrating defect prediction into the fitness function of SBST provides a powerful mechanism for directing the search towards real faults, leading to more effective and efficient automated test generation.

4. DISCUSSION

The empirical results strongly support the hypothesis that guiding Search-Based Software Testing with theoretical defect predictors significantly enhances its fault-finding effectiveness and efficiency. The observed increases in unique faults detected and reductions in time to first fault provide compelling evidence for the synergistic potential of combining these two powerful software engineering techniques.

4.1. Interpretation of Performance Gains

The effectiveness of DP-SBST can be primarily attributed to the intelligent bias introduced by the defect prediction model. By augmenting the fitness function with a "fault proneness" score, the evolutionary algorithm is not merely striving for high code coverage but is actively prioritizing the exploration of code regions that are statistically more likely to contain defects. This moves SBST beyond being solely a coverage optimization tool towards becoming a more direct fault-finding mechanism [8, 24].

The fact that DP-SBST found more unique faults, particularly those that might be subtle or deeply hidden, suggests that it can navigate the search space more effectively towards fault-revealing program states. A purely coverage-based approach might generate tests that execute a high percentage of code but do not necessarily trigger the specific conditions or data inputs required to manifest a bug. The defect predictor, having learned from historical fault data, provides a "roadmap" of high-risk areas, allowing the SBST to perform a more targeted and intelligent exploitation of the search space. This approach effectively balances the exploration of new code (via coverage) with the exploitation of known risky areas (via defect prediction) [2, 3].

The reduction in Time to First Fault (TTFF) is a crucial practical implication. In continuous integration/continuous delivery (CI/CD) pipelines, where rapid feedback on code changes is essential [41, 42], quickly identifying the first bug can save significant development time and resources. DP-SBST's ability to achieve this faster means that critical bugs can be caught earlier in the development cycle, reducing the cost of defect remediation.

4.2. Advantages and Disadvantages

Advantages:

• Improved Fault Detection: The most significant advantage is the enhanced capability to find real faults, directly addressing the limitations of purely coverage-driven SBST [5, 6].

• Increased Efficiency: Detecting faults faster, especially the first one, leads to more efficient testing processes and quicker feedback in agile development environments.

• Resource Focus: By prioritizing defect-prone areas, development teams can optimize their testing resources, focusing efforts where they are most likely to yield results.

• Synergistic Combination: This approach demonstrates a powerful synergy between static analysis (defect prediction) and dynamic testing (SBST), leveraging the strengths of both.

Disadvantages:

• Dependence on Defect Predictor Accuracy: The effectiveness of DP-SBST is heavily reliant on the accuracy and reliability of the underlying defect prediction model. A poorly performing or inaccurate predictor could misguide the search, leading to wasted effort [8].

• Cost of Defect Predictor Training: Building and maintaining an accurate defect predictor requires historical data and computational resources for model training [22, 37]. This upfront cost might be a barrier for smaller projects without extensive historical data.

• Potential for Local Optima (Bias): If the k weighting factor is too high, the defect prediction guidance might over-constrain the search, potentially leading to local optima and neglecting to explore other areas where faults might reside, even if they are not predicted as highly fault-prone. This could result in lower overall code coverage.

• Generalizability of Predictors: While cross-project defect prediction was used, the generalizability of defect predictors across diverse projects and domains remains a challenge [25, 37].

4.3. Implications for Software Engineering Practice

The findings have several key implications for software engineering practice:

• Enhanced CI/CD Pipelines: DP-SBST can be integrated into continuous integration pipelines to provide more targeted and effective automated testing. By running defect predictors on new code changes and then using that information to guide subsequent test generation, developers can receive faster and more meaningful feedback on potential regressions or new bugs [41, 42].

• Intelligent Test Prioritization: Beyond test generation, the concept can be extended to intelligent test prioritization or selection, where existing test cases covering defect-prone modules are run first [23].

• Better Resource Allocation: For manual testing or code reviews, the output of the defect predictor can guide human efforts more precisely, while automated tools tackle the high-risk areas identified.

• Complementary Approach: DP-SBST should be viewed as a complementary approach to traditional coverage-based testing. A balanced strategy might involve generating tests with DP-SBST for critical, fault-prone components, and then using standard coverage-based methods for broader system coverage.

4.4. Threats to Validity

Several threats to the validity of this empirical study should be acknowledged:

• Internal Validity:

o Tool Configuration: The choice of parameters for EvoSuite (e.g., population size, generation count, time budget) and the weighting factor k for defect proneness could influence results. Efforts were made to use recommended settings and perform sensitivity analysis.

o Fault Detection Oracle: Reliance on Defects4J's provided test cases as the oracle for fault detection [43] means that only known, reproducible faults are considered. Other, undiscovered faults would not be counted.

• External Validity:

o Language and Projects: The study was conducted on Java projects from Defects4J. Generalizing the results to other programming languages, domains (e.g., embedded systems, web applications), or larger, industrial-scale projects requires further investigation [44].

o Defect Predictor Model: The specific defect prediction model and metrics used might influence the results. Different models or feature sets could yield varying levels of guidance effectiveness.

• Construct Validity:

o Defect Proneness Measurement: The accuracy of the defect predictor itself directly impacts the effectiveness of the guidance. Imperfections in the

predictor mean that some truly fault-prone modules might be missed, while some non-faulty ones might be over-prioritized.

o Fitness Function Formulation: The specific mathematical formulation of the combined fitness function (e.g., multiplicative weighting) could impact how effectively defect proneness is integrated.

4.5. Future Work

Based on the findings and limitations, several promising avenues for future research exist:

• Adaptive Weighting: Developing adaptive mechanisms for the k weighting factor in the fitness function, allowing it to adjust dynamically during the search based on observed fault detection or coverage progress.

• Multi-objective Optimization: Exploring multi-objective SBST approaches that explicitly optimize for both code coverage and defect proneness simultaneously, potentially using Pareto optimality concepts [3, 2].

• Integration with Other Metrics: Incorporating other types of metrics (e.g., run-time behavior, performance data) into the defect prediction model or directly into the SBST fitness function.

• Domain-Specific Defect Predictors: Tailoring defect prediction models to specific application domains (e.g., cybersecurity, scientific computing) to enhance their relevance and accuracy for guiding testing in those contexts.

• Feedback Loops: Investigating feedback loops where the faults found by DP-SBST are used to refine and improve the defect prediction model iteratively.

• Application to Different SBST Variants: Applying defect prediction guidance to other SBST variants beyond evolutionary algorithms, such as constraint-based test generation [27, 30].

• Human-in-the-Loop Integration: Studying how defect prediction-guided SBST can best support human testers and developers, perhaps by providing prioritized test reports or highlighted areas for manual review.

5. CONCLUSION

This article has empirically demonstrated the significant benefits of integrating theoretical defect predictors into Search-Based Software Testing for automated test case generation. By guiding the SBST process towards modules identified as fault-prone, the proposed defect prediction-guided approach consistently found a higher number of unique faults and achieved a faster time to first fault compared to traditional coverage-guided methods. This research highlights a powerful synergy between static analysis for defect prediction and dynamic metaheuristic search for test generation. As software systems continue to grow in complexity, such intelligent, resource-aware automated testing strategies become increasingly vital. This work paves the way for more efficient, effective, and targeted software quality assurance practices, ultimately contributing to the delivery of more reliable software.

REFERENCES

[1] G. Fraser and A. Arcuri, “Whole test suite generation,” IEEE Trans. Softw. Eng., vol. 39, no. 2, pp. 276–291, Feb.2013.

[2] A. Panichella, F. M. Kifetew, and P. Tonella, “Reformulating branch coverage as a many-objective optimization problem,” in Proc. IEEE 8th Int. Conf. Softw. Testing, Verification Validation, 2015, pp. 1–10.

[3] A. Panichella, F. M. Kifetew, and P. Tonella, “Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets,” IEEE Trans. Softw. Eng., vol. 44, no. 2, pp. 122–158, Feb.2018.

[4] A. Panichella, F. M. Kifetew, and P. Tonella, “A large scale empirical comparison of state-of-the-art search-based test case generators,” Inf. Softw. Technol., vol. 104, pp. 236–256, 2018.

[5] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, “Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges (T),” in Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng., 2015, pp. 201–211.

[6] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, “An industrial evaluation of unit test generation: Finding real faults in a financial application,” in Proc. 39th Int. Conf. Softw. Eng.: Softw. Eng. Pract. Track, 2017, pp. 263–272.

[7] A. Salahirad, H. Almulla, and G. Gay, “Choosing the fitness function for the job: Automated generation of test suites that detect real faults,” Softw. Testing, Verification Rel., vol. 29, no. 4–5, 2019, Art. no. e1701.

[8] A. Perera, A. Aleti, M. Böhme, and B. Turhan, “Defect prediction guided search-based software testing,” in Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng., 2020, pp. 448–460.

[9] A. Schröter, T. Zimmermann, and A. Zeller, “Predicting component failures at design time,” in Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng., 2006, pp. 18–27.

[10] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, "Predicting faults from cached history," in Proc. 29th Int. Conf. Softw. Eng., 2007, pp. 489–498.

[11] P. A. F. de Freitas, "Software repository mining analytics to estimate software component reliability," Faculty of Engineering, University of Porto, Tech. Rep., 2015.

[12] H. Hata, O. Mizuno, and T. Kikuno, "Bug prediction based on fine-grained module histories," in Proc. 34th Int. Conf. Softw. Eng., 2012, pp. 200–210.

[13] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall, "Method-level bug prediction," in Proc. ACM-IEEE Int. Symp. Empirical Softw. Eng. Meas., 2012, pp. 171–180.

[14] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," IEEE Trans. Softw. Eng., vol. 33, no. 1, pp. 2–13, Jan.2007.

[15] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in Proc. 3rd Int. Workshop Predictor Models Softw. Eng., 2007, Art. no. 9.

[16] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in Proc. 27th Int. Conf. Softw. Eng., 2005, pp. 284–292.

[17] N. Nagappan, B. Murphy, and V. Basili, "The influence of organizational structure on software quality," in Proc. ACM/IEEE 30th Int. Conf. Softw. Eng., 2008, pp. 521–530.

[18] B. Caglayan, B. Turhan, A. Bener, M. Habayeb, A. Miransky, and E. Cialini, "Merits of organizational metrics in defect prediction: An industrial replication," in Proc. 37th Int. Conf. Softw. Eng., 2015, pp. 89–98.

[19] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, "Change bursts as defect predictors," in Proc. IEEE 21st Int. Symp. Softw. Rel. Eng., 2010, pp. 309–318.

[20] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead Jr, "Does bug prediction support human developers? Findings from a Google case study," in Proc. Int. Conf. Softw. Eng., 2013, pp. 372–381.

[21] C. Lewis and R. Ou, "Bug prediction at Google," 2011, Accessed: Sep., 2019. [Online]. Available: http://google-engtools.blogspot.com

[22] H. K. Dam , "Lessons learned from using a deep tree-based model for software defect prediction in practice," in Proc. 16th Int. Conf. Mining Softw. Repositories, 2019, pp. 46–57.

[23] D. Paterson, J. Campos, R. Abreu, G. M. Kapfhammer, G. Fraser, and P. McMinn, "An empirical study on the use of defect prediction for test case prioritization," in Proc. 12th IEEE Conf. Softw. Testing, Validation Verification, 2019, pp. 346–357.

[24] E. Hershkovich, R. Stern, R. Abreu, and A. Elmishali, "Prediction-guided software test generation," in Proc. 30th Int. Workshop Princ. Diagnosis, 2019, Accessed: Feb. 08, 2022. [Online]. Available: https://dx-workshop.org/2019/

[25] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: A large scale experiment on data versus domain versus process," in Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp., 2009, pp. 91–100.

[26] J. M. Rojas, M. Vivanti, A. Arcuri, and G. Fraser, "A detailed investigation of the effectiveness of whole test suite generation," Empirical Softw. Eng., vol. 22, no. 2, pp. 852–893, 2017.

[27] B. Korel, "Automated software test data generation," IEEE Trans. Softw. Eng., vol. 16, no. 8, pp. 870–879, Aug.1990.

[28] P. McMinn, "Search-based software testing: Past, present and future," in Proc. IEEE 4th Int. Conf. Softw. Testing, Verification Validation Workshops, 2011, pp. 153–163.

[29] R. Just, "Defects4J - A database of real faults and an experimental infrastructure to enable controlled experiments in software engineering research," 2019, Accessed: Oct., 2019. [Online]. Available: https://github.com/rjust/defects4j

[30] R. A. DeMilli and A. J. Offutt, "Constraint-based automatic test data generation," IEEE Trans. Softw. Eng., vol. 17, no. 9, pp. 900–910, Sep.1991.

[31] L. J. Morell, "A theory of fault-based testing," IEEE Trans. Softw. Eng., vol. 16, no. 8, pp. 844–857, Aug.1990.

[32] L. J. Morell, "A theory of error-based testing," Dept. Comput. Sci., Maryland Univ. College Park, MD, USA, Tech. Rep. TR-1395, 1984.

[33] A. Offutt, "Automatic test data generation," Georgia Institute of Technology, Tech. Rep., 1989.

[34] N. Li and J. Offutt, "Test Oracle strategies for model-based testing," IEEE Trans. Softw. Eng., vol. 43, no. 4, pp. 372–395, Apr.2017.

[35] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The Oracle problem in software testing: A survey," IEEE Trans. Softw. Eng., vol. 41, no. 5, pp. 507–

525, May2015.

[36] G. Fraser and A. Arcuri, "Evolutionary generation of whole test suites," in Proc. 11th Int. Conf. Qual. Softw., 2011, pp. 31–40.

[37] S. Hosseini, B. Turhan, and D. Gunarathna, "A systematic literature review and meta-analysis on cross project defect prediction," IEEE Trans. Softw. Eng., vol. 45, no. 2, pp. 111–147, Feb.2019.

[38] A. Arcuri and L. Briand, "A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," Softw. Testing, Verification Rel., vol. 24, no. 3, pp. 219–250, 2014.

[39] A. Vargha and H. D. Delaney, "A critique and improvement of the "CL" common language effect size statistics of McGraw and Wong," J. Educ. Behav. Statist., vol. 25, no. 2, pp. 101–132, 2000.

[40] C. O. Fritz, P. E. Morris, and J. J. Richler, "Effect size estimates: Current use, calculations, and interpretation." J. Exp. Psychol.: Gen., vol. 141, no. 1, pp. 2–18, 2012.

[41] J. Campos, A. Arcuri, G. Fraser, and R. Abreu, "Continuous test generation: Enhancing continuous integration with automated test generation," in Proc. 29th ACM/IEEE Int. Conf. Automated Softw. Eng., 2014, pp. 55–66.

[42] M. Fowler and M. Foemmel, "Continuous integration," 2006, Accessed: Feb. 10, 2022. [Online]. Available: https://www.martinfowler.com/articles/continuousIntegration.html

[43] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in Proc. Int. Symp. Softw. Testing Anal., 2014, pp. 437–440.

[44] J. Sohn and S. Yoo, "Empirical evaluation of fault localisation using code and change metrics," IEEE Trans. Softw. Eng., 2019, vol. 47, no. 8, pp. 1605–1625, Aug.2021.

[45] G. Gay, "The fitness function for the job: Search-based generation of test suites that detect real faults," in Proc. IEEE Int. Conf. Softw. Testing, Verification Validation., 2017, pp. 345–355.

[46] A. Aleti and M. Martinez, "E-APR: Mapping the effectiveness of automated program repair," Empirical Softw. Eng., vol. 26, no. 5, pp. 1–30, 2021.

[47] S. Pearson , "Evaluating and improving fault localization," in Proc. 39th Int. Conf. Softw. Eng., 2017, pp. 609–620.

[48] J. Campos, A. Panichella, and G. Fraser, "EvoSuite at the SBST 2019 tool competition," in Proc. 12th Int. Workshop Search-Based Softw. Testing, 2019, pp. 29–32.

[49] EvoSuite, "EvoSuite - Automated generation of Junit test suites for Java classes," 2019, Accessed: Nov., 2019. [Online]. Available: https://github.com/EvoSuite/evosuite

[50] G. Fraser, "Evosuite - Automatic test suite generation for Java," Accessed: Sep., 2019. [Online]. Available: http://www.evosuite.org/