Library Progress International *Print version ISSN 0970 1052*
Vol.44, No.5, Jul-Dec 2024: P.307-316 *Online version ISSN 2320 317X*

Original Article *Available online at www.bpasjournals.com*

Enhancing Web Application Reliability Through Integrated Sbst And Functional Testing

[1]D. Mythily,[2]Dr. N. Kamaraj

[1]Research Scholar, Department of Computer Science, Sri Ramakrishna Mission Vidyalaya College of Arts and Science, Coimbatore, Tamilnadu, India
[2]Head and Assistant Professor, Department of Information Technology, Sri Ramakrishna Mission Vidyalaya College of Arts and Science, Coimbatore, Tamilnadu, India

How to cite this article: D. Mythily,Dr. N. Kamaraj (2024). Enhancing Web Application Reliability Through Integrated Sbst And Functional Testing. *Library Progress International,* 44(5), 307-316

Abstract- This paper proposes an advanced methodology for software defect prediction and functional testing of websites, integrating traditional techniques with Search-Based Software Testing (SBST). The approach combines thorough manual testing, automated test execution, and machine learning-based defect prediction to ensure comprehensive coverage and accurate defect detection. Initial steps include detailed requirement analysis, test planning, and test case development, followed by manual and automated functional testing. The SBST process iteratively refines test cases through initialization, fitness evaluation, selection, crossover, mutation, and survivor selection, optimizing their effectiveness in identifying defects. Integration with CI/CD pipelines ensures continuous validation, while continuous monitoring and feedback facilitate ongoing improvement. Detailed reporting and analysis guide proactive measures, enhancing the overall quality and reliability of web applications.
Keywords: Software defect prediction, functional testing, Search-Based Software Testing (SBST), web application reliability, CI/CD integration;

1. Introduction

Functional testing is a critical aspect of software quality assurance, aimed at ensuring that a software application operates in conformance with its requirements. Specifically, when it comes to software defect prediction for websites, functional testing becomes an invaluable tool. The rise of complex web applications, driven by the increasing reliance on internet-based services, has significantly elevated the need for robust testing methodologies. Websites today are intricate systems with numerous interdependent components, ranging from the front-end user interface to back-end databases and APIs. Ensuring the smooth and error-free operation of these components is crucial for maintaining user satisfaction and business credibility. Software defect prediction is a proactive approach that leverages historical defect data, machine learning algorithms, and statistical techniques to identify potential areas in the software that are likely to contain defects. This predictive capability allows developers and testers to prioritize their testing efforts, focusing on the most vulnerable parts of the application. Functional testing, in this context, involves verifying that each function of the website behaves as expected under various conditions, thus playing a pivotal role in the defect prediction process.

The primary objective of functional testing in the realm of software defect prediction is to validate the functionality of the website against its specified requirements and to detect any deviations that may indicate underlying defects. This involves a systematic examination of the website's features, including user interactions, data processing, and interface elements. By simulating real-world usage scenarios, functional testing helps identify defects that may not be apparent during development but could significantly impact the user experience. One of the key challenges in functional testing for web applications is the dynamic and heterogeneous nature of web environments. Websites must operate flawlessly across different browsers, devices, and operating systems, each with its own quirks and capabilities. This variability necessitates comprehensive testing strategies that cover a wide range of configurations and user behaviors. Automated functional testing tools, such as Selenium, play a crucial role in this process by enabling repetitive and exhaustive testing across multiple environments.

Moreover, the integration of functional testing with software defect prediction models enhances the overall efficiency and effectiveness of the testing process. Predictive models can identify high-risk areas that are more likely to contain defects based on patterns in historical data. Testers can then focus their functional testing efforts on these highpriority areas, ensuring a more targeted and efficient testing process. This not only helps in early detection of defects but also reduces the overall cost and time associated with the testing phase. Another important aspect of functional testing in this context is the validation of non-functional requirements such as performance, security, and usability. These aspects are crucial for the overall quality of the website and can significantly influence the user experience. Functional tests can include scenarios that validate the website's performance under different loads, its resilience against security threats, and the intuitiveness of its user interface.

2. Literature Survey

2.1 Gated Hierarchical LSTMs (GH-LSTMs)

H. Wang et.al proposed Software Defect Prediction Based on Gated Hierarchical LSTMs. Software defect prediction, aimed at assisting practitioners in efficiently allocating test resources, focuses on identifying potentially defective modules within software products. Traditional software features have shown limitations in capturing semantic information, prompting researchers to explore semantic features for building more effective defect prediction models. However, traditional features such as lines of code (LOC) still hold significant importance. Most existing research concentrates on using either semantic or traditional features exclusively. This article introduces a defect prediction method based on gated hierarchical long short-term memory networks (GH-LSTMs). The method employs hierarchical LSTM networks to extract semantic features from word embeddings of abstract syntax trees (ASTs) and traditional features from the PROMISE repository. A gated fusion strategy is used to combine the outputs of these hierarchical networks effectively. Experimental results demonstrate that GH-LSTMs outperform existing methods in both no effort-aware and effort-aware scenarios, highlighting the method's robustness in integrating diverse feature types for enhanced defect prediction.

2.2Deep Learning-Based Defect Prediction Model

Qiao L et.al proposed Deep learning based software defect prediction. Software systems have grown increasingly large and complex, making the prevention of software defects a formidable challenge. Automatically predicting the number of defects in software modules is crucial for helping developers allocate limited resources efficiently. Various approaches have been developed to identify and rectify these defects at minimal cost, yet their performance still requires substantial improvement. This paper introduces a novel approach leveraging deep learning techniques to predict the number of defects in software systems. Initially, we preprocess a publicly available dataset through log transformation and data normalization. Subsequently, we model the data to prepare it as input for the deep learning model. This modeled data is then fed into a specially designed deep neural network to predict defect counts. Our approach is evaluated on two well-known datasets, demonstrating its accuracy and superiority over current state-of-the-art methods. On average, our method reduces the mean square error by over 14% and increases the squared correlation coefficient by more than 8%.

2.3Tree-Based Ensembles

Aljamaan H et.al proposed Software defect prediction using tree-based ensembles. Software defect prediction remains a vital area of research in software engineering, crucial for guiding quality assurance efforts effectively. Ensemble learning in machine learning has shown substantial promise in improving predictive performance compared to individual models. This paper empirically investigates seven Tree-based ensemble methods for defect prediction, a domain where their effectiveness has not been extensively studied. Among these ensembles, two belong to the bagging category: Random Forest and Extra Trees, while the remaining five are boosting ensembles: AdaBoost, Gradient Boosting, Hist Gradient Boosting, XGBoost, and CatBoost. The study employs 11 publicly available MDP NASA software defect datasets to evaluate these methods. Results indicate that Tree-based bagging ensembles, specifically Random Forest and Extra Trees, outperform Tree-based boosting ensembles in defect prediction tasks. However, none of the Tree-based ensembles studied showed significantly lower performance than individual decision trees. Notably, Adaboost emerged as the least effective ensemble among the Tree-based methods evaluated in this study.

2.4 WSHCKE Model

Zhu K et.al proposed Software defect prediction based on enhanced metaheuristic feature selection optimization and a hybrid deep neural network. Software defect prediction seeks to preemptively identify potential defects in new software modules using effective prediction models. However, achieving robust model performance is hindered by irrelevant and redundant features. Previous studies predominantly rely on conventional data mining or machine learning techniques, which often fall short in predictive accuracy. To address these challenges, we propose two innovations. First, inspired by search-based software engineering principles, we introduce an enhanced metaheuristic feature selection algorithm named EMWS. EMWS combines the Whale Optimization Algorithm (WOA) with Simulated Annealing (SA) to efficiently identify a concise set of highly relevant features. Second, we present WSHCKE, a unified defect prediction model that integrates feature selection results using a hybrid approach. WSHCKE leverages Convolutional Neural Networks (CNN) for abstract deep semantic feature extraction and Kernel Extreme Learning Machines (KELM) for robust classification. Extensive experiments across 20 diverse software projects validate the efficacy of EMWS and WSHCKE, demonstrating significant improvements in prediction accuracy across four evaluation metrics. These advancements highlight the potential of integrating advanced metaheuristic techniques with hybrid deep learning models for enhanced software defect prediction.

2.5 Tree boosting algorithm (XGBoost)

Esteves G et.al proposed Understanding machine learning software defect predictions. Software defects are a significant concern in software development, impacting both users and developers. Researchers have employed various techniques to mitigate these issues, with defect prediction using machine learning methods being prominent. These methods aim to preemptively identify potential defects in software modules before deployment, aiding developers in proactive defect management. However, existing literature often focuses on predicting defects using a wide array of software features, lacking clarity on the underlying reasons for software defects. Our study addresses this gap by employing the XGBoost tree boosting algorithm, which predicts module defectiveness based on easily computable module characteristics. To enhance model interpretability, we propose a model sampling approach that identifies optimal models using a minimal set of influential features. This approach not only improves predictive accuracy but also enhances model explainability, crucial for developers seeking insights into module-specific defect-prone features. Evaluation across diverse projects from Jureczko datasets demonstrates that our approach identifies project-specific influential features and achieves effective models with improved understandability, providing valuable insights for developers.

3. Proposed Methodology

This methodology aims to leverage traditional functional testing techniques, enhanced by data-driven approaches and automated tools, to predict and identify defects in websites effectively. The approach combines thorough manual testing, automated test execution, and machine learning-based defect prediction to ensure comprehensive coverage and accurate defect detection.The methodology aims to integrate traditional functional testing with advanced search-based software testing (SBST) to predict and identify defects in websites effectively. This approach leverages both manual and automated testing techniques to ensure comprehensive coverage and accurate defect detection.

3.1 Initial Functional Testing

Requirement Analysis and Test Planning

To ensure comprehensive coverage and effective defect detection in website testing, the first step is to gather and analyze all functional requirements. This involves collaborating with stakeholders to document each feature, user interaction, and workflow the website must support. Following this, a detailed test plan is developed, outlining the scope, objectives, resources, schedule, and deliverables of the testing process. The test plan acts as a roadmap, guiding the testing team through the testing lifecycle and ensuring all aspects of the website are thoroughly examined. It defines the strategy for both manual and automated testing, ensuring alignment with project goals and timelines.

Test Case Development

Creating detailed test cases covering all website functionalities, including links, forms, navigation, and user interactions, is essential for comprehensive testing. Each test case should meticulously outline the steps to verify the correct behavior of every feature, ensuring no aspect of the website is overlooked. Additionally, it is crucial to ensure that these test cases are traceable to specific requirements. This traceability guarantees that all client and stakeholder needs are addressed, providing a clear mapping between requirements and test scenarios. This approach not only enhances the thoroughness of the testing process but also facilitates efficient tracking and management of test coverage.

Manual Functional Testing

Execute manual test cases to validate the functionality of each component comprehensively, ensuring that every feature works as intended. During this process, meticulously check all aspects of the website, including links, forms, and navigation, to confirm their performance aligns with the requirements. Record any defects or issues encountered in a detailed defect log, noting their nature, severity, and the affected components. This systematic approach ensures thorough coverage and provides valuable data for further analysis and remediation. By identifying and documenting these issues early, the process facilitates timely fixes and enhances the overall quality and reliability of the website.

Automation of Test Cases

To ensure comprehensive testing, develop automated test scripts for repetitive and regression test cases using tools like Selenium or Cypress. These scripts should be designed to cover various browsers and devices, ensuring the website's responsiveness and cross-browser compatibility. By automating these test cases, repetitive tasks are streamlined, reducing manual effort and increasing testing efficiency. The scripts should be integrated into the continuous integration/continuous deployment (CI/CD) pipeline, enabling automatic execution with every code change. This approach ensures that any issues related to browser compatibility or device responsiveness are identified and addressed promptly, leading to a more robust and user-friendly website.

3.2 Proposed Search-Based Software Testing (SBST) for Test Case Generation

Initialize Population

To effectively predict software defects in websites through functional testing, we define a population of potential test cases, each representing a possible input or sequence of actions on the website. Let $P = \{T_1, T_2, \ldots, T_n\}$ denote the population of test cases, where $T_i$an individual test case is. The initial population$P_0$is generated with a diverse set of test cases, ensuring comprehensive coverage of the website's functionalities. Diversity is achieved by sampling test cases across different user scenarios and inputs, represented as$T_i = (x_1, x_2, \ldots x_m)$, where $x_j$denotes a specific action or input. This initialization step aims to maximize the probability of uncovering defects in various components.

Fitness Evaluation

To enhance the effectiveness of test cases in identifying defects, it is essential to define fitness functions incorporating metrics such as code coverage (CC), path coverage (PC), and the ability to uncover edge cases or unusual scenarios (UE). The fitness function $F(TC)$for a test case TCcan be formulated as:

$$F(TC) = w_1 \cdot CC(TC) + w_2 \cdot PC(TC) + w_3 \cdot UE(TC)$$

Wherew_1,w_2, and $w_3$are weights assigned to each metric, reflecting their importance. This composite fitness function ensures comprehensive evaluation, guiding the selection of highly effective test cases for defect identification.

Selection

In this methodology, selection mechanisms such as tournament selection and roulette wheel selection are employed to choose parent test cases based on their fitness scores. In tournament selection, a subset of test cases is randomly chosen, and the one with the highest fitness is selected as a parent. Mathematically, if T is the tournament size, and$F(i)$ is the fitness of the$i$ -th test case, then$P_{selected} = max\{F(i_1), F(i_2), \ldots, F(i_T)\}$. In roulette wheel selection, each test case $i$is chosen with a probability $P(i) = \frac{F(i)}{\sum_j \quad F(j)}$, ensuring that higher fitness test cases have a greater chance of being selected. These mechanisms optimize the genetic algorithms ability to explore and exploit the search space efficiently, enhancing the overall effectiveness of the test case generation process.

Crossover

In implementing crossover operations for selected parent test cases, the objective is to combine their sequences to create diverse offspring while preserving logical action sequences. Let $P_1$and P_2represent two parent test cases. The crossover operation can be expressed as:

$$O_i = \propto P_1 + (1 - \propto P_2$$

Where$\propto$ is a crossover rate between 0 and 1, determining the proportion of each parent contributing to the offspringO_i. This operation ensures variations by interleaving steps from both parents, maintaining logical coherence. By introducing controlled variations, we enhance the robustness and coverage of the test suite.

Mutation

In the context of software defect prediction for websites, applying mutation operations to offspring test cases is crucial for generating diversity and exploring new scenarios. Mutations typically involve small changes aimed at improving test case effectiveness. Examples include altering input values, rearranging action sequences, or adjusting conditional statements within test steps. These operations help in uncovering edge cases and potential defects that might not be captured by initial test scenarios. By introducing controlled variations through mutations, the methodology ensures comprehensive coverage of possible system behaviors, thereby enhancing the reliability and effectiveness of the functional testing process in identifying potential defects in web applications.

Fitness Evaluation of Offspring

In evaluating the fitness of mutated offspring test cases, it's crucial to apply consistent fitness functions used for parent test cases. This ensures that the effectiveness of each mutated offspring is assessed based on predefined criteria such as code coverage, error detection, or functional validation. By maintaining uniformity in fitness evaluation, we can objectively identify mutated test cases that demonstrate potential in uncovering defects. This process involves comparing the performance metrics of mutated offspring against those of parent test cases, aiming to select variants that not only maintain the integrity of logical sequences but also enhance the overall efficacy of the testing strategy.

Survivor Selection

In the context of evolutionary algorithms applied to software testing, the selection and replacement of test cases play a crucial role in optimizing test suite effectiveness. Following the creation of offspring through crossover and mutation operations, the next step involves selecting the fit individuals from the combined population of current test cases and offspring. This selection process is typically guided by a fitness function f, which evaluates the effectiveness of each test case based on criteria such as code coverage, fault detection capability, or execution time. Mathematically, the selection can be represented as:

$$selectedParents = Selection(population)$$

Subsequently, less effective test cases are replaced with better-performing ones, aiming to enhance the overall quality and efficiency of the test suite. This iterative process of selection and replacement contributes to the continual improvement of software testing methodologies, ensuring robust test coverage and defect detection capabilities.

Termination Criteria

Defining termination criteria is crucial in evolutionary processes like genetic algorithms used in software testing. Common criteria include reaching a maximum number of generations, achieving convergence in fitness scores, or adhering to predefined time limits. For instance, the algorithm may terminate after a fixed number of iterationsG_{max}, ensuring computational efficiency. Alternatively, convergence criteria involve halting when fitness scores stabilize over successive generations, indicating optimal test suite performance. Time limits restrict execution duration to manage computational resources effectively. These criteria collectively ensure the algorithm's efficiency and effectiveness in generating high-quality test cases while preventing unnecessary computation beyond specified constraints.

Search-Based Software Testing (SBST) focusing on test case generation

$$Function\ SearchBasedTesting():$$

Step 1: $population = InitializePopulation()$

This step generates an initial set of diverse test cases covering various website functionalities.

Step 2: $EvaluateFitness(population)$

Assess the effectiveness of each test case based on predefined criteria such as code coverage and fault detection.

Step 3: $selectedParents = Selection(population)$

Select the best-performing test cases as parents for the next generation using methods like tournament or roulette wheel selection.

Step 4: $offspring = Crossover(selectedParents)$

Combine sequences from selected parents to create new test cases, introducing variation while maintaining logical steps.

Step 5: $mutatedOffspring = Mutation(offspring)$

Apply small changes to offspring to explore new scenarios and enhance test coverage.

Step 6: $EvaluateFitness(mutatedOffspring)$

Assess the fitness of the mutated test cases to ensure they meet effectiveness criteria.

Step 7: $population = SurvivorSelection(population, mutatedOffspring)$

Combine and select the best individuals from the current population and the new offspring to form the next generation.

Step 8: Output $return\ population$

After meeting the termination criteria, return the optimized set of test cases, ready for comprehensive website validation.

This iterative process ensures continuous improvement of the test cases, enhancing their ability to predict and identify defects effectively.

Upon completing the optimization process, the methodology delivers a refined population of test cases, systematically developed through initialization, fitness evaluation, crossover, mutation, and survivor selection. These robust scenarios effectively validate website functionalities, ensuring deployment-ready comprehensive testing.

Here's an algorithm summarizing the methodology for functional testing on software defect prediction for websites:

Algorithm: Functional Testing On Software Defect Prediction

Step 1: Requirement Analysis and Test Planning

- Gather and analyze all functional requirements of the website in collaboration with stakeholders.
- Document features, user interactions, and workflows that the website must support.
- Develop a comprehensive test plan outlining scope, objectives, resources, schedule, and deliverables.

Step 2: Test Case Development

- Create detailed test cases covering all functionalities (links, forms, navigation, etc.).
- Ensure each test case is traceable to specific requirements for clear mapping and coverage.

Step 3: Manual Functional Testing

- Execute manual test cases to validate functionality comprehensively.
- Check all aspects including links, forms, and navigation for performance alignment with requirements.

- Record defects encountered in a detailed defect log, noting severity and affected components.

Step 4: Automation of Test Cases

- Develop automated test scripts using tools like Selenium or Cypress.
- Design scripts to cover various browsers and devices for responsiveness and compatibility testing.
- Integrate scripts into CI/CD pipeline for automatic execution with each code change.

Step 5: Search-Based Software Testing (SBST) for Test Case Generation

- **Initialize Population:** Generate an initial population of diverse test cases covering various user scenarios.
- **Fitness Evaluation:** Define fitness functions incorporating metrics like code coverage, path coverage, and edge case detection.
- **Selection:** Use selection mechanisms (e.g., tournament selection, roulette wheel selection) to choose parent test cases based on fitness.
- **Crossover:** Combine selected parent test cases using crossover operations to create diverse offspring while maintaining logical sequences.
- **Mutation:** Apply mutation operations to introduce controlled variations in offspring test cases.
- **Evaluate Fitness of Offspring:** Assess the effectiveness of mutated offspring using predefined fitness functions.
- **Survivor Selection:** Replace less effective test cases in the current population with better-performing offspring based on fitness.
- **Termination Criteria:** Define criteria (e.g., maximum generations, fitness convergence, time limits) to end the evolutionary process.

Step 6: Output

- Return the final population of optimized test cases that effectively validate various functionalities of the website.
- Ensure all test cases meet termination criteria and maximize fitness through evolutionary techniques.

This algorithm outlines a systematic approach to functional testing enhanced by data-driven and automated methods, combined with SBST for effective defect prediction and identification in web applications.

3.3 Integration and Continuous Testing

Integration with CI/CD Pipelines

Integrating automated and search-based test cases into the CI/CD pipeline ensures rigorous testing throughout the software development lifecycle. Automated tests, including unit tests and integration tests, verify code functionality on every commit and merge, ensuring early detection of defects. Concurrently, search-based testing enriches the test suite by generating diverse scenarios and edge cases, enhancing test coverage. By configuring these tests to run automatically during CI/CD pipelines triggered on code commits, merges, and deployments we ensure continuous validation of code quality and functionality. This integration not only accelerates feedback cycles but also fosters reliable software releases, mitigating risks associated with bugs and ensuring robust application performance.

Continuous Monitoring and Feedback

In the context of automated testing for software systems, continuous monitoring of test results plays a crucial role in maintaining software quality. This process involves systematically collecting and analyzing data on various metrics such as test performance, defect detection rates, and test coverage. By gathering comprehensive insights from these metrics, teams can identify areas for improvement and optimize testing strategies effectively. Feedback obtained from continuous monitoring guides the refinement of test cases and testing methodologies, ensuring that subsequent test runs are more effective in identifying defects and enhancing overall software reliability before deployment. This iterative approach fosters continuous improvement in software testing practices.

Reporting and Analysis

After executing functional testing on software defect prediction for websites, detailed test reports are generated to document identified defects, their severity levels, and the functionalities they affect. These reports serve as a comprehensive record of testing outcomes, facilitating informed decision-making in bug resolution. Analyzing these results reveals patterns and identifies areas susceptible to defects, guiding proactive measures for improvement. Actionable insights derived from the analysis enable developers to prioritize and address critical issues promptly, while also implementing preventive strategies to mitigate future defects. This holistic approach ensures robust software quality assurance, enhancing overall reliability and user satisfaction.

By integrating SBST with traditional functional testing methods, this proposed methodology aims to enhance the defect prediction and detection capabilities for websites, ensuring a robust and reliable user experience.

4. Experimental Results

4.1 Coverage (%)

Indicates how much of the software is covered by the test cases. Higher percentages suggest better coverage.

$$Coverage\ (\%) = \frac{No\ of\ covered\ elements}{Total\ no\ of\ elements} * 100$$

No of Iterations	DL based Model	GH-LSTMs	Proposed SBST-FT
Iteration 1	85	80	90
Iteration 2	82	78	89
Iteration 3	88	81	92
Iteration 4	84	79	91
Iteration 5	86	77	93

Table 1.Comparison Table of Coverage

In this table 1, shows test coverage percentages for three different models across five iterations. Coverage indicates the proportion of software elements tested, with higher percentages reflecting better coverage. The DL based Model has coverage percentages ranging from 77% to 88%, the GH-LSTMs model ranges from 77% to 81%, and the Proposed SBST-FT model achieves the highest coverage, ranging from 89% to 93%. This suggests that the Proposed SBST-FT model offers superior test coverage compared to the other models.

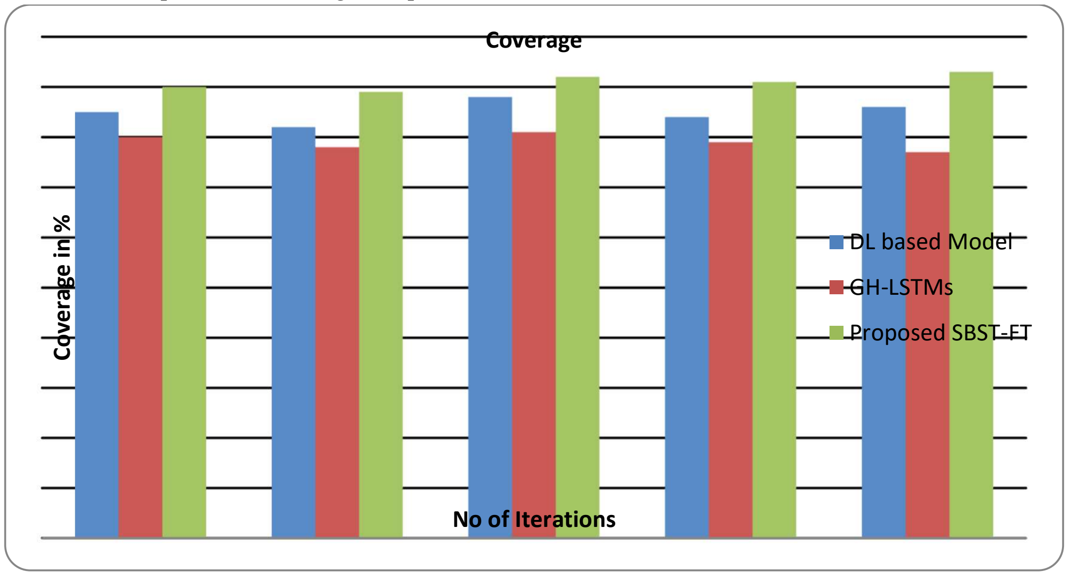

Figure 1.Comparison Chart of Coverage

The Figure 1 displays test coverage percentages for three models DL based Model, GH-LSTMs, and Proposed SBST-FT across five iterations. The Proposed SBST-FT model consistently achieves the highest coverage, ranging from 89% to 93%, indicating superior testing of software elements. In contrast, the DL based Model shows coverage between 77% and 88%, while GH-LSTMs range from 77% to 81%. These results suggest that the Proposed SBST-FT model provides the most comprehensive testing, ensuring better software quality and reliability.

4.2 Fault Detection Rate (%)

Ratio of detected faults to total faults in the software.

$$\boldsymbol{Fault\ Detection\ Rate}\ (\%) = \left(\frac{\boldsymbol{Number\ of\ detected\ faults}}{\boldsymbol{Total\ number\ of\ faults}}\right) \times \mathbf{100}$$

No of Iterations	DL based Model	GH-LSTMs	Proposed SBST-FT
Iteration 1	75	70	85
Iteration 2	78	72	88
Iteration 3	80	74	90
Iteration 4	77	71	87
Iteration 5	79	73	91

Table 2.Comparison Table of Fault Detection Rate

In this table 2, each percentage represents the ratio of detected faults to the total number of faults in the software for each iteration and model. The Proposed SBST-FT model consistently shows higher fault detection rates, indicating better performance in identifying faults compared to the DL based Model and GH-LSTMs.

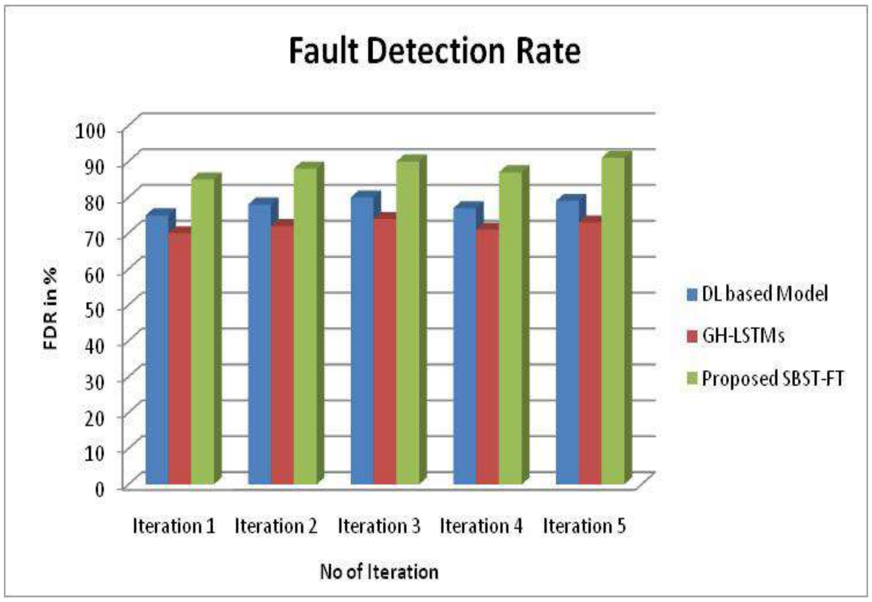

Figure 2.Comparison Chart of Fault Detection Rate

In this figure 2, the proposed SBST-FT model (green bars) consistently shows the highest Fault Detection Rate across all five iterations, indicating superior performance in detecting faults compared to the other two models. The DL based Model (blue bars) has a higher FDR than GH-LSTMs (red bars) in all iterations but is lower than the proposed SBST-FT model. The GH-LSTMs model consistently has the lowest FDR among the three models, suggesting it is the least effective at detecting faults.

4.3 Execution Time (hours)

The time taken for test case generation and evaluation. Shorter times are preferable for efficiency.

$$Exection\ Time = Time\ for\ Test\ Case\ Generation + Time\ for\ Test\ Case\ Evaluation$$

No of Iterations	DL based Model	GH-LSTMs	Proposed SBST-FT
Iteration 1	5.0	4.5	3.5
Iteration 2	5.2	4.7	3.6
Iteration 3	5.1	4.6	3.4
Iteration 4	5.3	4.8	3.7
Iteration 5	5.0	4.5	3.5

Table 3.Comparison Table of Execution Time

In this table 3, execution time for the DL based Model ranges from 5.0 to 5.3 hours across the five iterations. For the GH-LSTMs model, the execution time ranges from 4.5 to 4.8 hours across the iterations. The Proposed SBST-FT model shows an execution time ranging from 3.4 to 3.7 hours across the iterations. This model consistently demonstrates the shortest execution time, suggesting that it is the most efficient among the three models in terms of test case generation and evaluation.

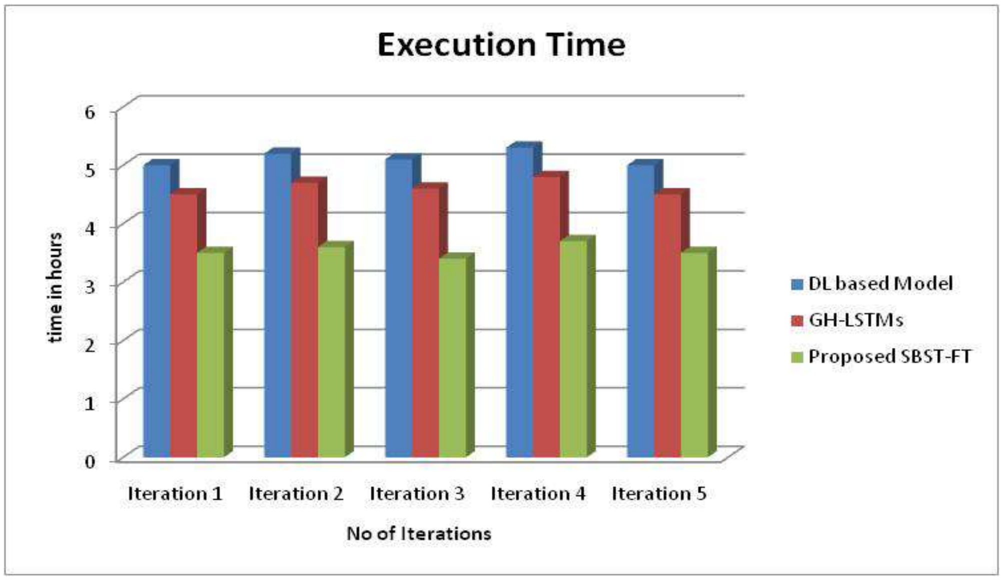

Figure 3.Comparison Chart of Execution Time

The figure 3, illustrates the Execution Time in hours across five iterations for three models: DL based Model, GH-LSTMs, and Proposed SBST-FT. The DL based Model (blue bars) consistently takes the longest time, ranging from 5.0 to 5.3 hours. The GH-LSTMs model (red bars) has a shorter execution time, between 4.5 and 4.8 hours. The Proposed SBST-FT model (green bars) consistently shows the shortest execution time, ranging from 3.4 to 3.7 hours. This indicates that the Proposed SBST-FT model is the most efficient in test case generation and evaluation. Overall, the Proposed SBST-FT outperforms the other two models in terms of execution time efficiency.

5. Conclusion

In this paper, proposed methodology effectively combines traditional functional testing with advanced Search-Based Software Testing (SBST) to enhance software defect prediction and detection for websites. By integrating manual and automated testing techniques, along with machine learning-based defect prediction, this approach ensures comprehensive coverage and accurate defect identification. Iterative refinement of test cases through SBST and seamless integration with CI/CD pipelines enable continuous validation and improvement. Detailed reporting and analysis provide actionable insights, guiding proactive measures to address defects and improve web application reliability. This comprehensive methodology ensures robust, high-quality web applications, delivering a reliable user experience.

References

1. H. Wang, W. Zhuang and X. Zhang, "Software Defect Prediction Based on Gated Hierarchical LSTMs," in IEEE Transactions on Reliability, vol. 70, no. 2, pp. 711-727, June 2021, doi: 10.1109/TR.2020.3047396.

2. Qiao L, Li X, Umer Q, Guo P. Deep learning based software defect prediction. Neurocomputing. 2020 Apr 14;385:100-10.

3. Aljamaan H, Alazba A. Software defect prediction using tree-based ensembles. InProceedings of the 16th ACM international conference on predictive models and data analytics in software engineering 2020 Nov 8 (pp. 1-10).

4. Zhu K, Ying S, Zhang N, Zhu D. Software defect prediction based on enhanced metaheuristic feature selection optimization and a hybrid deep neural network. Journal of Systems and Software. 2021 Oct 1;180:111026.

5. Esteves G, Figueiredo E, Veloso A, Viggiato M, Ziviani N. Understanding machine learning software defect predictions. Automated Software Engineering. 2020 Dec;27(3):369-92.

6. Yao J, Shepperd M. Assessing software defection prediction performance: Why using the Matthews correlation coefficient matters. InProceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering 2020 Apr 15 (pp. 120-129).

7. Aniche M, Maziero E, Durelli R, Durelli VH. The effectiveness of supervised machine learning algorithms in predicting software refactoring. IEEE Transactions on Software Engineering. 2020 Sep 4;48(4):1432-50.

8. Chen J, Xia X, Lo D, Grundy J, Luo X, Chen T. Defectchecker: Automated smart contract defect detection by analyzing evm bytecode. IEEE Transactions on Software Engineering. 2021 Jan 27;48(7):2189-207.

9. Yucalar F, Ozcift A, Borandag E, Kilinc D. Multiple-classifiers in software quality engineering: Combining predictors to improve software fault prediction ability. Engineering Science and Technology, an International Journal. 2020 Aug 1;23(4):938-50.

10. Goyal S, Bhatia PK. Comparison of machine learning techniques for software quality prediction. International Journal of Knowledge and Systems Science (IJKSS). 2020 Apr 1;11(2):20-40.
11. Tian H, Liu K, Kaboré AK, Koyuncu A, Li L, Klein J, Bissyandé TF. Evaluating representation learning of code changes for predicting patch correctness in program repair. InProceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering 2020 Dec 21 (pp. 981-992).
12. Cheng X, Wang H, Hua J, Xu G, Sui Y. Deepwukong: Statically detecting software vulnerabilities using deep graph neural network. ACM Transactions on Software Engineering and Methodology (TOSEM). 2021 Apr 23;30(3):1-33.
13. Dias Canedo E, Cordeiro Mendes B. Software requirements classification using machine learning algorithms. Entropy. 2020 Sep 21;22(9):1057.
14. Tang Y, Liu Z, Zhou Z, Luo X. Chatgpt vs sbst: A comparative assessment of unit test suite generation. IEEE Transactions on Software Engineering. 2024 Mar 29.
15. Zheng W, Xun Y, Wu X, Deng Z, Chen X, Sui Y. A comparative study of class rebalancing methods for security bug report classification. IEEE Transactions on Reliability. 2021 Oct 22;70(4):1658-70.
16. Lin G, Xiao W, Zhang J, Xiang Y. Deep learning-based vulnerable function detection: A benchmark. InInformation and Communications Security: 21st International Conference, ICICS 2019, Beijing, China, December 15–17, 2019, Revised Selected Papers 21 2020 (pp. 219-232). Springer International Publishing.
17. Chen J, Chen C, Xing Z, Xu X, Zhu L, Li G, Wang J. Unblind your apps: Predicting natural-language labels for mobile gui components by deep learning. InProceedings of the ACM/IEEE 42nd International Conference on Software Engineering 2020 Jun 27 (pp. 322-334).
18. Ghafouri SH, Hashemi SM, Hung PC. A survey on web service QoS prediction methods. IEEE Transactions on Services Computing. 2020 Mar 16;15(4):2439-54.
19. Biswas E, Karabulut ME, Pollock L, Vijay-Shanker K. Achieving reliable sentiment analysis in the software engineering domain using bert. In2020 IEEE International conference on software maintenance and evolution (ICSME) 2020 Sep 28 (pp. 162-173). IEEE.
20. Wu Y, Zou D, Dou S, Yang S, Yang W, Cheng F, Liang H, Jin H. SCDetector: Software functional clone detection based on semantic tokens analysis. InProceedings of the 35th IEEE/ACM international conference on automated software engineering 2020 Dec 21 (pp. 821-833).