2025 3rd International Conference on Artificial Intelligence and Machine Learning Applications Theme: Healthcare and Internet of Things (AIMLA) | 979-8-3315-3896-5/25/$31.00 ©2025 IEEE | DOI: 10.1109/AIMLA63829.2025.11040354

Test Case Generation using Graph-based Deep Learning Methods

Devansh Handa, Kindi Krishna Nikhil, Kanaganandini Kanagaraj, S. Duvarakanath, Supriya M.*
Department of Computer Science and Engineering
Amrita School of Computing, Bengaluru
Amrita Vishwa Vidyapeetham, India
devanshhandaji@gmail.com, nikhilkindi1704@gmail.com,
nandinikanagaraj03@gmail.com, duvarak05@gmail.com, m_supriya@blr.amrita.edu

***Abstract*—In today's fast-paced software development environment, the need for automated and reliable test case generation is increasingly critical. Traditional methods often fail to ensure both syntactic accuracy and semantic relevance, especially in complex domains. This paper proposes a hybrid approach that combines graph-based deep learning with large language models (LLMs) to generate test cases automatically. By incorporating both semantic and syntactic analysis, the system will produce context aware, structurally sound test cases, adaptable across diverse software domains. This solution aims to improve testing efficiency and coverage, ensuring higher-quality software development and streamlined testing processes.**

***Index Terms*—Test case generation, Large Language Models, CodeLlama, Mistral, Stability AI**

I. Introduction

Testing is one of the most important phases in the software development life cycle (SDLC) that influences the quality of the developed applications. Consequently, regarding the type of software systems and the fundamental test basis, test case generation has become inevitable for the testing process [1]. However, the disadvantages of other traditional methods to derive the test cases are, they failed to guarantee syntactical whole, proper and paralleled combination of test cases in addition to semantically correct and meaningful test cases concerning the system. All these challenges become much more intense as the feature set of the system under development increases.

New trends in graph-based deep learning are offering a solution which appears more recently within AI approaches. Among the conventional graphs used to represent the relations within and between the parts in software which also allows for semantic information to be retained, graph representations that are ideal for modelling software are preferred. However, by employing these models in test case generation, several gains could be achieved one among which is enhancing the efficiency and effectiveness of the software testing; for this reason, the models suggested in this paper are important.

Our proposed approach uses graph neural networks (GNNs) to model the structural evolution of software codes enhancing the quantity and quality of the test cases developed [2]. Graph-based deep learning approaches expose the new promising field of efficient, reliable, and automated test case generation for numerous software systems. The use of class activation mapping (CAM) for test case generation is a valuable development in software testing. In general, conventional approaches fail to preserve both syntactic correctness and semantic pertinence in most cases as the software system grows large. But, the proposed approach is safer, more scalable, and more automated compared to its previous solution because it uses the structure enhancing capacities of graph neural networks and at the same time, the contextual awareness of large language models (LLMs) [3]. This makes the tool valuable for improving software quality as it increases accuracy and covers the most important aspects of the domain. Finally, this approach contributes to proper effectiveness and credibility of software development, considering the major issues of test case generation [4].

II. Literature Survey

Son et al. presents a method for automatic generation of cause-effect graph test cases through model transformation using the ATLAS Transformation Language (ATL) in Eclipse [5]. The process involves two steps: transformation of cause-effect graphs into decision tables and then into test cases. Future work is to enlarge this method to generate the test cases from Unified Modeling Language (UML) diagrams where the mapping rules need to be defined from the cause-effect graph. Article [6] formally describes an automatic test case generation approach for Java programs to identify boundary violation errors in arrays using the System Dependence Graph (SDG) and ANother Tool for Language Recognition (ANTLR). The methodology consists of using static analysis to detect arrays and their dimensions, constructing valid/invalid usage test cases, and context-free grammar (CFG) and data dependence graph (DDG) diagrams in addition to issuing an alert about empty catch statements. Future work incorporates extending the test case generation process as well as other testing techniques to increase viability and reliability, avoiding wrong software outputs and errors.

The work proposed in [7] applies modified genetic algorithm on the basis of the graph theory to clear confusion in test case generation for software testing. The proposed work establishes a directed graph of attributes of the system's

intermediate states, which forms the population of a genetic algorithm. This involves applying the operations of crossover and mutation on particular node pairs with the view of enhancing the result producing capability of test cases. In the future, the presented approach will be further investigated and developed, with attempts to optimize the process of generating test cases, and test the applicability of the proposed approach in other testing situations to increase the effectiveness of testing.

Shunhui et al. presents a neural network-based method for generating test cases in data-flow-oriented testing, focusing on DU-pairs (Definition-Use pairs) [8]. It employs a backpropagation (BP) neural network to simulate the fitness function and integrates a genetic algorithm for efficient test case generation. The methodology enhances efficiency by reducing program execution times, particularly for large programs. Future work will involve validating this approach on practical programs to assess its effectiveness and applicability in real-world scenarios. Article [9] describes the algorithm for creating the test cases for the Transaction-level modeling (TLM) using the graphic structure called the Control-Transaction Graph (CTG). This approach develops coverage metrics, including edge and loop coverage, to guarantee all system-on-chip designs are exhaustively verified. Both authors present test-case generation algorithms developed from the CTG, and prove the efficiency of the proposed methods using examples of an AMBA–AHB bus and a JPEG encoder model. Future works will include the modeling of timing faults and improvement on the CTG for the production of more test cases and their verification within TLM platforms.

Cummins et al. and the team introduces a new deep learning graph-based approach call PROGRAM, where the program is transformed into graphs. Integrated within MPNNs, the model incorporates both control and data flow from compiler IR levels [10]. This structured planning and the representation of the working plan, makes program analysis and optimization highly efficient compared to other techniques. The methodology entails passing information through graphs to reason about programs or about specific instructions. Future work plans to apply this model to such endeavors as automatic parallelization and IR-to-IR transpiration for optimization purposes. Allamanis et al. aims to examine the use of deep learning (DL) to source code in the framework of code completion, bug detection, and program synthesis [11]. The approach is based on the technique in which source code is analyzed based on syntactic and semiotic characteristics using neural networks. For most current tasks including the next token or function, there is relevant application of models like recurrent neural networks (RNNs) and GNNs. The authors show that their approaches outperform the previous program analysis mechanisms. Further work includes increasing the model's ability to identify unseen code and integrating DL with symbolic execution to increase precision in hard-to-model or abstract code settings.

The work proposed in [12] has the purpose of introducing GraphFormers, a model that combines Graph Neural Networks with transformer architectures to enhance the representation of code for usage in tasks like code completion and vulnerability detection. The methodology is centered on: Local structures that are captured by GNNs and the contextual information which is captured by transformers. The performance evaluation of node classification and code summarization experiments show that there are significant increases in terms of those measures. Subsequent papers will cover improving the 'readability' of the test scores as well as making the model practical for both instant and large-scale applications such as the integrate-compile-run-loops which are frequently employed in computer programming. Xiaomeng et al. and the team propose a novel CPGVA framework, which works on the foundation of deep-learning models that do the vulnerability analysis on Code Property Graphs [13]. CPGVA is the abbreviated term of control flow graph for code, data flow graph for code, and syntax tree for code all in one. This methodology involves passing the above graph structures to neural networks to enable them to predict the likelihood of the above risk such as buffer overflows and injection attacks. The results are higher that the results achieved with other more traditional vulnerability detection techniques. Future work involves improving the presented methodology to discern more new forms of vulnerability and applying the presented methodology to different forms of vulnerability and to languages other than English.

The research described in this paper is a system that suggests test cases based on the knowledge graphs so that the developers may use it as a tool to automate the selection of good tests [14][15]. The workflow can be summarized as follows: First, generate a knowledge graph that connects code changes, bug reports, and historical test cases. HSMM graphs that represent a system's IFS are then applied to recommend associated test cases according to Graph Similarity. The recommendations are feature-based and the recommendation of a specific test is given together with a rationale for it. The further work includes extending the knowledge graph to add the current data from CI/CD, and enriching the set of suggested test cases with diverse ones

III. Methodology

A. Dataset Description

The Leetcode Dataset [16] serves as the primary training dataset, chosen for its comprehensive range of coding problems, solutions, and associated test cases, structured as follows and shown in Fig 1.

- Problem Descriptions: Detailed problem statements are included, outlining task requirements, edge cases, and constraints.
- Solution Folder: Sample solutions are provided in code format, representing multiple coding languages and logic implementations, offering a solid foundation for understanding code structure and purpose the model will

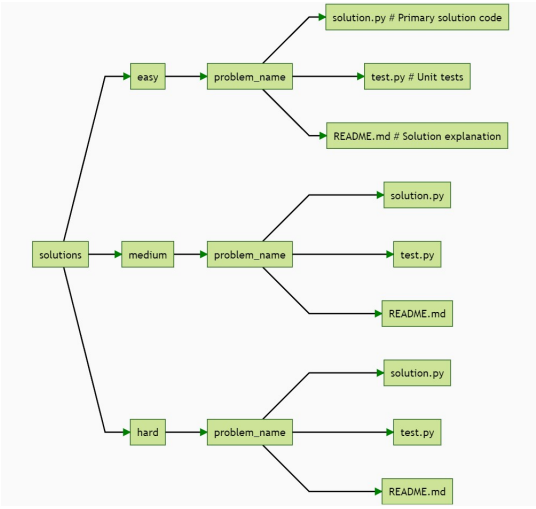

Fig. 1. Dataset Flowchart

learn to replicate and adapt for new code inputs.

- Test Cases Folder: Each problem includes pre-existing test cases aligned with the solutions, offering ready-made validation scenarios that the model will learn to replicate and adapt for new code inputs.

B. Workflow

Static and dynamic code analysis, along with test case generation, will help improve software development, minimize individual's mistakes, and enhance coding standards [17]. Current programming language models, specifically LLMs pre-trained for coding tasks, provide a fundamental architecture for constructing such systems [18]. However, there are still obstacles in inputs manipulation containing long code sequences, the question of staying consistent with context, and the problem of generating necessary test cases within given context. The proposed work addresses the problems faced in the present implementation and helps in generating the test cases using graph-based DL methods. The workflow of the proposed approach presented in Fig. 2 is explained in the following section.

C. Base Model: Code Llama 2 7B

Based on the capability of Code Llama 2 7B in code-specific language tasks, it is chosen as the foundation model. This model, a 7-billion parameter version of Meta's Llama series, employs transformer architecture that has been optimized for the understanding and generation of code. The model structure is explained here.

- *Tokenization:* Code Llama 2 7B is equipped with a precise tokenizer which segments the syntax and semantics of programming language with a very high rate of efficiency.

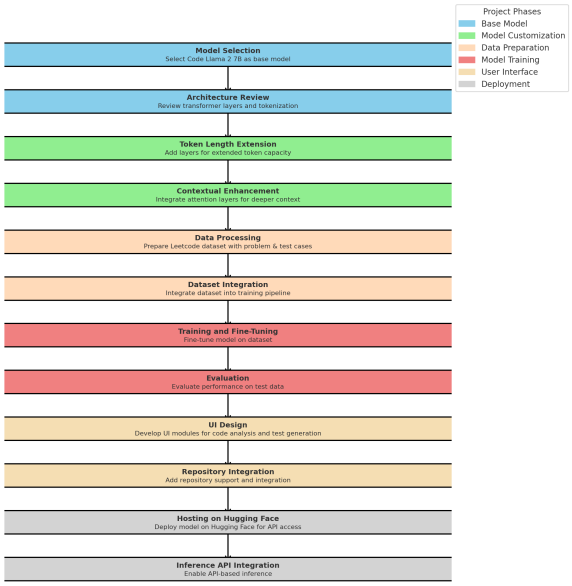

Fig. 2. Workflow of the Proposed Approach

- *Attention Mechanism:* It uses multi-head self-attention to understand the interactions within longer sequences of code, which is important for natural coding cognition.

- *Transformer Layers:* The depth of the model and the presence of multiple transformer layers enable it to handle sequential dependencies in code, particularly in programs with multiple files and dependencies [19].

- *Cross-Language Adaptability:* The model is programmed on multiple languages, and this increases the performance and ability of the model to perform in any environment.

D. Enhancing Code Llama 2 7B for Code Analysis and Test Case Generation

To extend and enhance the model for Code Llama 2 7B, more Transformer layers are added for the further token sequences allowing a more profound analysis of the contexts across the repositories. It adds conditional attention layers and adaptive token embeddings that ensure more attention is paid to significant code areas, e.g., function declarations and variable dependencies, and enhances syntax-specific token representation. Long-context embeddings are good up to a sequence of 4,096 tokens, allowing full code inclusion without being reduced.

The model is fine-tuned with the help of Leetcode dataset where problem descriptions are matched with test case generation by using the supervised learning so that it can develop optimal domain-related expertise. Others are a code analysis module essential for complexity and bug analysis, a test case generation for unitary test and edge test, a repository integration for multi analysis file and visualization tools. The

improved model is released into Hugging Face's Model Hub with API compatibility for seamless integration for large-scale and secure real-time code analysis and test synthesis with privacy-preserving over the data.

E. Modules Involved in Test Case Generation

1) Code Llama: Meta has many programming languages for developers through one interface called Code Llama including Python, Java, and C++ [20]. We measured its performance in code generation, completion, debugging, and explanation. Several are more focused on specific areas such as, CodeLlama-Python for Python kind of jobs and CodeLlama-Instruct for direction like talk to me. It comes in batches of 7B, 13B, and 34B for parameterization and works with most integrated development environments for automation and thus, improving developer efficiency. It is extensible and can be adapted due to its open source characteristic.

2) Stability AI's Stable-Code-3B: Stable-Code-3B, specifically tailored for programming tasks, was developed using code repositories and other documents. It helps write code, edit the code, translate between languages, and generate scripts from plain language text. Applicable to numerous domains, including web development and electronic systems, it works to improve work processes by including automation and competent debugging features. It is available for research needs and scales domain-specific fine-tuning.

3) Mistral-7B-V0.1: Mistral AI's 7B parameter model deals with natural language processing tasks including text generation, summarization, translation, etc. The lightweight architecture developed through the use of grouped-query and multi-query attention has both high efficiency and adaptability in conditions of limited resources. As a product of modular design and open access, one can fine-tune it to different fields or specialized areas, such as healthcare or finance, which always keeps the AI technologies usable and innovative.

4) Evaluation: The models are evaluated for test case generation regarding their precision, compatibility with the programming language, correctness of execution, utilization of system resources, and capability to be integrated [21]. Syntax compliance analysis of Code Llama, the domain adaptability of Stable-Code-3B, and Mistral modularity will be used to determine which model is most appropriate for implementation.

F. Code Evaluation Metrics

1) Total Test Cases: This metric represents the total number of test cases generated by the model during evaluation, serving as a baseline for performance analysis. A higher number reflects the model's scalability and its capacity to handle diverse scenarios, while consistency across inputs indicates reliability.
2) Passed Test Cases: These are test cases that meet the expected outputs without errors. This metric measures the model's accuracy in generating functional and executable code. A high number of passed test cases signifies strong performance in validation and correctness, critical for automated test generation.
3) Failed Test Cases: Test cases that do not meet expected outcomes due to errors or logical inconsistencies fall into this category. This metric highlights the model's limitations, such as misinterpreting prompts or generating invalid syntax. Analyzing failures aids in identifying areas for model improvement.
4) Accuracy: Calculated as the percentage of passed test cases out of total test cases, this metric assesses the model's reliability and precision. High accuracy reflects dependable performance in generating valid outputs, making it a key factor in model comparison.
5) Error Rate: This metric complements accuracy by quantifying the percentage of failed test cases. A low error rate signifies a robust model with minimal incorrect outputs, essential for identifying trends across varying complexities and languages.
6) Precision: Precision measures the proportion of correctly identified valid test cases, indicating the model's ability to avoid false positives. High precision ensures that the outputs are accurate and relevant, critical for generating meaningful test cases.
7) Recall: Recall evaluates the model's capability to identify all valid test cases, emphasizing thoroughness. A high recall ensures comprehensive test case generation, essential for robust software validation.
8) F1 Score: The F1 Score combines precision and recall into a balanced metric, providing a comprehensive view of model performance. A high F1 score demonstrates the model's effectiveness in generating accurate and thorough test cases.

IV. Results and Discussion

Table I lists the results obtained from the implementation of the testcase generation using the three models explained in Section III-E. The observation from the results are discussed here.

1) *Code Llama:* With a moderate accuracy of 47.06 and an error rate of 52.94, Code Llama performed better than Mistral but lagged behind Stable AI. Its precision, recall, and F1 score of 0.47 highlight some consistency but reveal limitations in reliability and complexity management.

2) *Stable AI:* With an accuracy of 71.43 percentage and an error rate of 28.57 percentage, Stable AI outperformed others. Its balanced precision, recall, and F1 score of

0.71 reflect consistent success in handling test cases.

3) *Mistral AI:* Achieving the lowest accuracy of 40.74 and an error rate of 59.26, Mistral struggled with test case complexity, indicating significant limitations.

TABLE I
EVALUATION MATRIX

Evaluation Matrix	Models		
	CodeLlama	***Stability AI***	***Mistral AI***
Total Test cases	51	35	27
Passed Test Cases	24	25	11
Failed Test Cases	27	10	16
Accuracy	47.06	71.43	40.74
Error Rate	52.94	28.57	59.26
Precision	0.47	0.71	0.41
Recall	0.47	0.71	0.41
F1 score	0.47	0.71	0.41

The assessment of three models–Code Llama, Stability AI, and Mistral AI shows remarkable differences in their outcomes calculated on test cases. Stability AI indicates the best reliability with 71.43 accuracy and the lowest error rate of 28.57 alongside an almost equally impressive precision, the recall, the F1 score with 0.71. This is indicated by the above metrics and they show that the method is very reliable and efficient for situations that requires accuracy and stability. Code Llama has a 47.06 of accuracy and a 52.94 of error risking, improving the ones obtained by Mistral but it is still behind Stability AI and its 61.67 of accuracy. The performance is relatively low, but it can be seen that the accuracy for handling specific test cases is around 0.47, the recall is also decent, as well as the F1 score. It is reliable for informal uses but could do with enhancements for high level applications. Mistral deliver the smallest accuracy of 40.74 and the biggest error rate 59.26 and seems to fail to respond appropriately to test cases. The model achieves a precision of 0.41, coupled by a similar recall and F1 score which shows that it is not capable of generating accurate outputs fully, in cases where thoroughness and accuracy is needed. Thus, Stability AI is considered to be the most stable model out of all three, the second one is Code Llama, and the application has a lot of issues with the Mistral model, so it needs improvements. Overall, Stable AI emerged as the most reliable model for handling test cases effectively.

V. CONCLUSION

Among the four models of AI, stable AI is the most ocularly reliable kind of deep learning applications that have achieved exemplary performance in restrictive environments. Nevertheless, Code Llama is proved to be efficient but needs further optimizing for complicated scenarios. Mistral, even if it is slower and less accurate and has a higher misidentification rate is still good for basic usage. Specific improvements to these models could include; Abstract syntax trees for better validation, fine-tuning by domain specificity, and graph neural networks (GNNs) for better context. The future work may focuses on the advanced testing methods, powerful algorithms for debugging and the methods for automatically synthesizing test cases for increasing LLM dependability.The abilities of models such as T5 or GPT-3 show that automatic test case generation holds a lot of promise. These models involved determine the context of the conversation as well as identify keywords that lead to the formulation of rather comprehensive test cases. This also saves effort, reduces overlap and likely covers scenarios fully, and cuts the need for the best designing talent. Proposed framework empowers test Automation Function through NLP by minimizing cost and efforts while providing context-aware test cases for effective Software Testing.

REFERENCES

[1] Nicha Kosindrdecha and Jirapun Daengdej, "A Test Case Generation Process and Technique," Journal of Software Engineering, 2010, 4: 265-287.

[2] Tanya Bloch, André Borrmann, Pieter Pauwels, "Graph-based learning for automated code checking – Exploring the application of graph neural networks for design review," Advanced Engineering Informatics, Volume 58, 2023, 102137, ISSN 1474-0346, https://doi.org/10.1016/j.aei.2023.102137.

[3] G. S. Murugesan and S. A. Viswanathan, "Enhancing Human-Machine Interaction: A Study on Deployment of LLM and Gen AI Hybrid Models in Responsible Humanoids for Human Assistance," 2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT), Kamand, India, 2024, pp. 1-6, doi: 10.1109/ICCCNT61001.2024.10724819.

[4] S. Sriram, C. H. Karthikeya, K. P. Kishore Kumar, N. Vijayaraj and T. Murugan, "Leveraging Local LLMs for Secure In-System Task Automation With Prompt-Based Agent Classification," in IEEE Access, vol. 12, pp. 177038-177049, 2024, doi: 10.1109/ACCESS.2024.3505298.

[5] H. S. Son, R. Y. C. Kim and Y. B. Park, "Test Case Generation from Cause-Effect Graph Based on Model Transformation," 2014 International Conference on Information Science & Applications (ICISA), Seoul, Korea (South), 2014, pp. 1-4, doi: 10.1109/ICISA.2014.6847468.

[6] A. v. karuthedath, S. Vijayan and V. K. K. S, "System Dependence Graph based test case generation for Object Oriented Programs," 2020 International Conference on Power, Instrumentation, Control and Computing (PICC), Thrissur, India, 2020, pp. 1-6, doi: 10.1109/PICC51425.2020.9362460.

[7] V. Rajappa, A. Biradar and S. Panda, "Efficient Software Test Case Generation Using Genetic Algorithm Based Graph Theory," 2008 First International Conference on Emerging Trends in Engineering and Technology, Nagpur, India, 2008, pp. 298-303, doi: 10.1109/ICETET.2008.79.

[8] S. Ji, Q. Chen and P. Zhang, "Neural Network Based Test Case Generation for Data-Flow Oriented Testing," 2019 IEEE International Conference On Artificial Intelligence Testing (AITest), Newark, CA, USA, 2019, pp. 35-36, doi: 10.1109/AITest.2019.00-11.

[9] Mohammad Reza Kakoee, M.H. Neishaburi, Siamak Mohammadi, "Graph based test case generation for TLM functional verification, Microprocessors and Microsystems," Volume 32, Issues 5–6, 2008, Pages 288-295, ISSN 0141-9331, https://doi.org/10.1016/j.micpro.2008.03.010.

[10] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, Hugh Leather, "Programl: Graph-based deep learning for program optimization and analysis.", 2020, arXiv preprintarXiv:2003.10536.

[11] Allamanis, Miltiadis, MarcBrockschmidt,and Mahmoud Khademi."Learning to represent programs with graphs", 2018, arXivpreprintarXiv:1711.00740.

[12] Junhan Yang, Zheng Liu, Shitao Xiao, Chaozhuo Li, Defu Lian, Sanjay Agrawal, Amit Singh, Guangzhong Sun, Xing Xie, "Graphformers:Gnn-nested transformers for representation learning on textual graph." Advances in Neural Information Processing Systems34, 2021, 28798-28810.

[13] W. Xiaomeng, Z. Tao, W. Runpu, X. Wei and H. Changyu, "CPGVA: Code Property Graph based Vulnerability Analysis by Deep Learning," 2018 10th International Conference on Advanced Infocomm Technology (ICAIT), Stockholm, Sweden, 2018, pp. 184-188, doi: 10.1109/ICAIT.2018.8686548.
[14] W. Ke, C. Wu, X. Fu, C. Gao and Y. Song, "Interpretable Test Case Recommendation based on Knowledge Graph," 2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS), Macau, China, 2020, pp. 489-496, doi: 10.1109/QRS51102.2020.00068.
[15] K. Vardhni, G. Devaraja, R. Dharhshita, R. K. Chowdary and A. Mahadevan, "Performance Evaluation and Comparative Ranking of LLM Variants in Entity Relationship Prediction," 2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT), Kamand, India, 2024, pp. 1-7, doi: 10.1109/ICCCNT61001.2024.10725334.
[16] https://github.com/wiseaidev/awesome-code/
[17] M. Ram, V. Mohith Krishna, M. Pranavkrishnan, P. C. Nair and D. Gupta, "Shellcode Generation: A Resource Efficient Approach using Fine-tuned LLMs," 2024 15th International Conference on Computing Communication and Networking Technologies (ICCCNT), Kamand, India, 2024, pp. 1-6, doi: 10.1109/ICCCNT61001.2024.10723327.
[18] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Neel Sundaresan, "Generating accurate assert statements for unit test cases using pretrained transformers." Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test, 2022.
[19] C. Wang, F. Pastore, A. Goknil and L. C. Briand, "Automatic Generation of Acceptance Test Cases From Use Case Specifications: An NLP-Based Approach," in IEEE Transactions on Software Engineering, vol. 48, no. 2, pp. 585-616, 1 Feb. 2022, doi: 10.1109/TSE.2020.2998503.
[20] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, Junjie Chen, "On the evaluation of large language models in unit test generation." Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering. 2024.
[21] Salman, Alzahraa. "Test case generation from specifications using natural language processing.", 2020.