Proceedings of the 7th International Conference on Computing and Data Science
DOI: 10.54254/2755-2721/2025.26220

A Survey and Prospects on Unit Test Case Generation Techniques Based on Large Language Models

Zhuoying Yang[14*], Lei Wang[1234*]

[1]*College of Mathematics and Computer Science, Yan'an University, Yan'an, China*
[2]*School of Computer Science & Technology, Beijing Institute of Technology, Beijing, China*
[3]*Tangshan Research Institute of Beijing Institute of Technology, Hebei Key Laboratory of Big Data Science and Intelligent Technology, Tangshan, China*
[4]*Key Laboratory of Industrial Internet and Big Data, China National Light Industry, Beijing, China*
**Corresponding Author. Email: wanglei0823@foxmail.com*

Abstract. Automated unit test case generation is a critical research topic in the field of software engineering. Numerous scholars at home and abroad have devoted considerable efforts to addressing this problem and have achieved fruitful results. In recent years, the emergence of large language models (LLMs), such as the GPT series, LLaMA series, DeepSeek series, T5, and their variants tailored for source code, has introduced new technical pathways for unit test case generation, thereby reigniting widespread scholarly interest in this area. To further promote the theoretical and practical development of LLM-based unit test case generation, this study presents a comprehensive survey and an outlook on future research. It reviews and analyzes the evolution of unit test case generation techniques, from early exploratory efforts to the current stage driven by large models. Starting from two main categories—techniques that rely solely on LLMs and those that integrate traditional static analysis—the current state of research and recent advancements in LLM-based test generation are discussed. On this basis, the study summarizes the major issues and challenges in the field and envisions possible future research directions. This work systematically presents the developmental trajectory, latest progress, and future prospects of this domain, offering valuable insights and guidance for subsequent research and innovation.

Keywords: Software Engineering, Unit Test Case Generation, Literature Review, Large Language Models, Prompt Learning

1. Introduction

With the increasing complexity of software systems, unit testing has become a critical means to ensure software quality and system maintainability [1]. The design of test cases constitutes the most important activity in unit testing. Initially, the generation of unit test cases relied on developers to manually write test code, which was inefficient and exhibited limited coverage. To alleviate the burden of manual effort, researchers proposed automated unit test generation methods based on static analysis [2-7] and dynamic analysis [8-11], as well as approaches leveraging neural networks

DOI: 10.54254/2755-2721/2025.26220

and other machine learning algorithms [12-16]. However, these methods still face challenges in handling complex semantic understanding and assertion construction. In recent years, with the rapid advancement of large language models (LLMs) in code comprehension and generation tasks, LLM-based test case generation methods [17-20] have gradually emerged as a cutting-edge research direction. By employing techniques such as prompt learning, these methods enable the automatic modeling of test inputs and assertions, demonstrating superior coverage and generalizability [21]. Representative frameworks such as TestPilot [18] and ChatUniTest [19] have continuously emerged, propelling this field into a new stage of development. At present, unit test case generation based on LLMs has become a prominent research focus in the domain of software engineering.

Numerous researchers have conducted comprehensive surveys on automated unit test case generation techniques [22-27]. However, most of these surveys have not analyzed or discussed the development, evolutionary trajectories, and trend changes of unit test case generation techniques from a chronological perspective.

To this end, this paper adopts a chronological perspective to systematically examine and organize the development history, recent advances, and the key problems and challenges of automated unit test case generation techniques, with a particular focus on those based on LLMs. It further outlines the future trends of research in this domain, aiming to provide a structured reference for subsequent studies and to promote the continuous advancement of test case generation techniques toward higher quality, greater explainability, and enhanced intelligence. Specifically, the paper first reviews the transformative evolution of unit test case generation techniques alongside the emergence and progression of technologies such as machine learning, deep learning, and LLMs. It then categorizes the current methods into two main types—those that rely solely on LLMs and those that integrate traditional static analysis—and systematically explores the application of cutting-edge LLMs in unit test case generation for software systems. Finally, it analyzes the major problems and challenges facing this field, identifies promising research directions, and proposes potential solutions. This study aspires to offer comprehensive and systematic guidance for researchers in the field, thereby fostering ongoing innovation and broader application of unit test case generation technologies.

The main contributions of this paper are as follows:

(1) A structured review from the methodological perspective: Based on a classification framework that considers generation mechanisms and prompting strategies, this paper proposes two major pathways for unit test case generation techniques: those that rely solely on LLMs and those that incorporate traditional static analysis. It systematically summarizes the key ideas, technical implementations, and representative achievements of each approach, offering researchers a more practical and insightful analytical dimension.

(2) Summary of key challenges and outlook for future directions: This paper identifies two major issues currently present in the field and puts forward future research suggestions along technical paths such as structural modeling, prompt automation, semantic validation, and context pruning. These insights provide a systematic reference for subsequent studies and tool development.

The overall structure of this paper is as follows: Section 2 analyzes the historical development of unit test case generation techniques; Section 3 discusses the current state of unit test case generation based on LLMs; Section 4 envisions the future directions of LLM-based unit test generation; and Section 5 concludes the paper.

2. The history of unit test case techniques

With the continuous growth in the scale and complexity of software systems [28], the importance of software testing in ensuring system quality, stability, and security has become increasingly

DOI: 10.54254/2755-2721/2025.26220

prominent [29]. Among various testing strategies, unit testing—serving as the fundamental means for verifying the correctness of individual functional modules—has emerged as a critical component in modern software development processes [30].

However, the traditional process of writing unit tests has largely relied on manual efforts by developers, resulting in significant efficiency bottlenecks and subjective inconsistencies [17]. On the one hand, manually written test cases often exhibit limited coverage and are prone to overlooking boundary conditions and exceptional paths. On the other hand, differing understandings of code logic among developers may lead to inconsistent testing standards, thereby affecting the quality and reliability of tests. This reality has driven sustained interest from researchers and practitioners in exploring automated methods for test case generation.

Early approaches to automated test case generation primarily focused on non–machine learning techniques, including static analysis [2-4], symbolic execution [5-7], path exploration [8,9], and genetic algorithms [10,11]. These methods systematically analyzed program structures and control flow graphs to generate test inputs that cover specific execution paths, supplemented by rule-based assertion design. This stage was marked by the development of toolchains such as EvoSuite [31], Randoop [32-34], and KLEE [35], which significantly improved test generation efficiency and structural coverage. However, these methods still face challenges when dealing with complex business logic, object dependencies, and large input spaces.

Since around 2010, with the gradual integration of machine learning into the field of software engineering, researchers began exploring the use of neural networks [12-14], graph neural networks [15,16], and other models to represent code features, aiming to enable more intelligent test generation processes. These approaches have achieved preliminary progress in areas such as code behavior prediction and automatic assertion construction. However, they remain constrained by issues such as data dependency and insufficient contextual understanding [35].

In recent years, with the rapid advancement of LLMs such as GPT [36], CodeBERT [37], LLaMA [38], and DeepSeek [39], these models have demonstrated strong capabilities in code understanding, generation, and reasoning by leveraging self-supervised learning on large-scale code corpora. These LLMs not only support multiple programming languages but also possess the ability to perform complex code analysis tasks through few-shot learning and contextual reasoning. As a result, they have introduced a novel paradigm for intelligent source code processing tasks, including design pattern detection [40], code generation [41], unit test case generation [42], and bug repair [43]. This progress significantly propels software engineering towards greater efficiency, automation, and intelligence. Consequently, test case generation methods based on pretrained models have emerged as a research frontier. These approaches leverage the contextual modeling capabilities and natural language generation abilities of LLMs to automatically produce structurally sound and semantically clear test cases with few-shot examples or prompt-based inputs. Representative works such as TestPilot [18] and ChatUniTest [19] have demonstrated superior coverage and adaptability compared to traditional methods across multiple open-source projects, driving test generation techniques into a new “model-driven” era.

3. Current status of unit test case generation techniques based on LLMs

Since the 1980s, automated test case generation techniques have gradually evolved, progressing from non–machine learning algorithms based on static and dynamic analysis to machine learning-based approaches, and more recently to innovative methods powered by LLMs. These advancements have led to significant achievements in the field. This paper provides a comprehensive overview of the current state of research and recent developments in two major categories of LLM-based unit

DOI: 10.54254/2755-2721/2025.26220

test case generation techniques: those that rely solely on LLMs, and those that integrate traditional static analysis.

3.1. Unit test case generation techniques relying solely on LLMs

Some researchers have entirely abandoned the static analysis typically required by traditional test case generation techniques and instead attempt to rely exclusively on the natural language understanding and source code processing capabilities inherent in LLMs for generating unit test cases.

Yuan et al. [17] designed a basic prompt that feeds the focal method along with its relevant code context into ChatGPT, enabling the direct generation of complete unit test cases containing both the test prefix and the test oracle. To further enhance the quality of generated tests, the study proposed the ChatTester framework, which adopts a workflow combining intent inference and iterative refinement: it first uses intent-oriented prompts to help ChatGPT comprehend the focal method, and then iteratively improves the test code based on compilation error messages, thereby increasing the correctness and usability of the generated test cases.

Schäfer et al. [18] developed the TestPilot tool, which constructs prompts containing function signatures, documentation comments, usage examples, and the function source code to invoke existing LLMs (such as gpt-3.5-turbo) for generating JavaScript unit test cases. The generated tests are executed automatically, and if a test fails, the system feeds the failed test case along with the corresponding error information back into the model as a new prompt, enabling the model to adaptively revise the test case. This iterative process facilitates the generation of high-quality tests.

The aforementioned techniques leverage the powerful natural language understanding and source code processing capabilities of LLMs, greatly simplifying the test case generation process and enhancing the level of automation and development efficiency. However, these approaches generally rely on the model's direct response to prompts and lack indepth modeling and utilization of structured information within the source code. As a result, the generated test cases often fall short in terms of coverage completeness, logical rigor, and adaptability to complex code scenarios. This limitation highlights the necessity of integrating traditional static analysis techniques with prompt-based learning in LLMs to further improve the quality and practical value of test case generation.

3.2. Unit test case generation techniques integrating traditional static analysis

Some researchers have attempted to incorporate traditional static analysis as an auxiliary component in the process of generating test cases based on LLMs.

Xiang et al. [19] proposed the ChatUniTest framework, which adopts an adaptive focus-context mechanism to dynamically extract key contextual information from the target code and construct high-quality prompts for input into large LLMs. These prompts guide the model to generate unit test cases with high readability and interpretability. After generation, a generate–validate–repair workflow is employed to sequentially perform syntax checking, compilation, and execution validation. Additionally, rule-based techniques and the LLM itself are used to automatically repair the test cases, thereby enhancing their correctness and practical applicability.

Zhang et al. [20] adopted a method slicing approach, in which complex methods are decomposed into multiple logical slices. Chain-of-Thought(CoT) prompts are then used to guide large LLMs to generate test cases for each slice individually, ensuring full line and branch coverage. These slice-level test cases are ultimately combined into a complete test suite. During the generation process, static analysis is introduced to extract contextual information that aids LLMs in understanding the

DOI: 10.54254/2755-2721/2025.26220

code, and a Self-Debug technique is employed to automatically repair non-executable test cases, thereby improving overall coverage and executability.

These techniques utilize static analysis to extract key contextual information, control flow, and data flow features from the source code, incorporating this information as auxiliary input into prompt design. This integration enhances LLMs' understanding of code structure and semantics, thereby improving the quality and coverage capability of the generated test cases. However, in existing methods, static analysis and prompt design are typically loosely integrated, lacking deep collaborative optimization tailored to specific testing objectives. Moreover, most prompting strategies have not systematically explored the potential of combining static analysis insights with CoT prompting. In addition, information extraction on the static analysis dimension remains incomplete and unsystematic. As a result, the rigor, coverage completeness, and self-repair capabilities of the generated test cases still require further enhancement.

4. The future of unit test case generation techniques based on LLMs

4.1. Key issues and challenges

In the domain of unit test case generation based on LLMs, researchers and practitioners have conducted extensive studies and explorations, yielding a number of preliminary achievements. However, due to the inherent complexity of the unit test case generation problem and the diversity of its application scenarios, the preceding survey reveals that this research direction still faces several significant issues and challenges.

4.1.1. The challenge of limited coverage of LLMs in structurally simple projects

Although LLM-based test case generation methods demonstrate significant advantages in handling complex business logic, code with missing documentation, or strong contextual dependencies, their coverage performance may fall short when applied to traditional utility-type projects that are structurally simple and logically straightforward. For example, the Commons-CLI [44] project primarily implements commandline argument parsing and help message output. Its internal modules have low coupling and clear invocation paths, making it well-suited for structure-based testing tools such as EvoSuite [31], which achieve high statement coverage through path traversal and input enumeration. In the course of our research on the IntelliUnitGen [42] unit test generation framework, we observed that tools based on LLMs significantly underperformed compared to non-LLM tools like EvoSuite in terms of coverage on the Commons-CLI project.This phenomenon reveals that current LLM-driven methods have not yet achieved an effective balance between semantic correctness and path coverage orientation. When the testing objective is structurally driven rather than semantically driven, the model may fall into “over-reasoning,” thereby neglecting the precise control of path exploration.

This phenomenon reveals that current LLM-driven methods have not yet achieved an effective balance between semantic correctness and path coverage orientation. When the testing objective is structurally driven rather than semantically driven, the model may fall into “overreasoning,” thereby neglecting the precise control of path exploration.

DOI: 10.54254/2755-2721/2025.26220

4.1.2. Issues of generation correctness and trustworthiness caused by the “hallucination” phenomenon

Although LLMs have demonstrated outstanding generative capabilities in source code processing tasks, their inherent "hallucination" problem still severely limits their applicability in high-reliability scenarios. This type of hallucination refers to situations where the model generates code that appears semantically plausible and syntactically correct, but contains serious logical or factual errors. Such issues are widespread across tasks including code generation, API invocation, library function recommendation, comment generation, and unit test case generation.

This issue has already been observed in practical applications. For example, in the IntelliUnitGen framework [42] developed by the authors of this paper, similar phenomena were encountered during the generation of test cases for complex classes: the model would sometimes invoke non-existent methods or construct illogical input structures out of thin air, resulting in test classes that fail at the compilation stage and require multiple rounds of regeneration or even manual correction to produce valid test cases.

4.1.3. Lack of traceability and interpretability in the test case generation process

Current LLM-based test case generation methods generally adopt an end-to-end black-box paradigm in their generation logic, lacking explicit records of intermediate reasoning steps and explanatory capabilities. When the generated test cases exhibit deviations, fail to compile, or cannot be executed, users often find it difficult to accurately locate the root cause of the error. It is also challenging to determine whether the issue arises from incorrect understanding of the source code, insufficient prompt design, failure in assertion logic, or the model’s own learning bias.

In practical use, this problem becomes particularly pronounced. For instance, in extreme scenarios, test case generation frameworks such as IntelliUnitGen [42] may produce test classes that fail to compile or execute. Developers are often left to resolve such issues through repeated generation or manual intervention, without being able to thoroughly analyze the specific reasoning paths or knowledge gaps involved in the generation process of LLMs. As a result, debugging heavily relies on trial-and-error, which not only reduces debugging efficiency but also places high demands on the user’s professional expertise.

4.1.4. Challenges of assertion errors and practical applicability under cross-file dependencies

Unlike source code processing tasks such as code smell detection, defect detection, vulnerability detection, and design pattern detection—which primarily focus on single-file or single-class structures—unit test case generation tasks rely more heavily on cross-class, cross-module, and even cross-project contextual semantics and invocation relationships. In practice, the method under test often depends on auxiliary methods from other classes, global variables, or specific construction logic. This makes it necessary for test case generation not only to understand the target method itself, but also to accurately model its dependency paths and execution environment.

To mitigate this challenge, some studies (such as ChatUniTest [19] and HITS [20]) have attempted to use mocking frameworks like Mockito to construct the dependencies of the object under test during testing, thereby reducing the modeling complexity caused by cross-file dependencies. While this approach improves the generatability and executability of test cases to some extent, it also introduces new semantic risks: mocked objects cannot faithfully reproduce real dependency behavior, which can easily lead to assertions that are semantically inaccurate or even

DOI: 10.54254/2755-2721/2025.26220

incorrect. As a result, the generated test cases may cover code paths but fail to cover defects and boundary conditions present in real semantic scenarios.

4.2. Future research directions

To address the multiple challenges currently faced by LLMs in the application of automated unit test case generation, future research may pursue indepth exploration and breakthroughs in the following directions:

(1) Hybrid generation mechanisms integrating structural analysis and semantic generation: To improve the performance of LLMs in structurally simple or logically straightforward projects, future studies may consider integrating traditional static analysis techniques (such as control flow graphs and data flow analysis) with the semantic reasoning capabilities of LLMs, thereby constructing a hybrid test generation framework that combines structure-driven and semantics-driven approaches. This integration can enhance adaptability and coverage across various types of projects.

(2) Reducing hallucination rates and enhancing generation trustworthiness and consistency: Future work may introduce mechanisms such as factual consistency checking, result verification modules, or search-and-execution feedback guided generation correction to mitigate the hallucination problem of LLMs. These techniques aim to improve the compilability and executability of the generated test cases.

(3) Enhancing the traceability and interpretability of the generation process: Future research may introduce visualized intermediate reasoning records—such as CoT prompting [45], generation graphs, and causal chain modeling—and leverage analysis tools with interactive debugging and visualization capabilities (e.g., CodeMind [46] and the OpenAI Code Interpreter UI, now integrated into the ChatGPT Pro environment) to construct causal paths from input to output. This would improve the debuggability of the generation process, assist users in understanding the model's generation logic and the causes of failure, and enhance the maintainability and controllability of the system.

(4) Combining real dependency modeling with contract-aware assertion generation mechanisms: To alleviate the challenges of cross-class or cross-module dependencies, future studies should explore strategies that integrate static dependency analysis (such as call graphs and global variable tracking) with dynamic behavior modeling (such as runtime dependency sampling and interface interaction logging). These methods can automatically extract the true context and invocation constraints of the method under test, and build a contract-aware test generation process. In addition, semantic guidance based on interface contracts, behavioral specifications, or method annotations can be incorporated to assist LLMs in generating logically consistent and semantically faithful assertions, thereby enhancing the semantic completeness and deployment reliability of the generated test cases.

5. Conclusion

With the continuous expansion of software system scale and the growing demand for automated testing, the automatic generation of unit test cases has become a significant research direction in the field of software engineering. This paper provides a comprehensive review of unit test case generation techniques from three perspectives: historical evolution, current state of research, and future trends. It particularly highlights the critical role of LLMs in advancing the intelligence and automation of this task, offering a systematic theoretical framework and research reference for

DOI: 10.54254/2755-2721/2025.26220

future studies, while also providing theoretical foundations and conceptual insights for the practical adoption of related technologies in real-world software engineering.

References

[1] Myers, G.J., Sandler, C., and Badgett, T. (2011) The Art of Software Testing. John Wiley & Sons, Hoboken.

[2] Bozga, M., Fernandez, J.C., and Ghirvu, L. (2000) Using static analysis to improve automatic test generation. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 235–250. Springer, Berlin, Heidelberg.

[3] Babić, D., Martignoni, L., McCamant, S., et al. (2011) Statically-directed dynamic automated test generation. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis, pp. 12–22. ACM, New York, NY.

[4] Kallingal Joshy, A., Chen, X., Steenhoek, B., et al. (2021) Validating static warnings via testing code fragments. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 540–552. ACM, New York, NY.

[5] Godefroid, P. (2012) Test generation using symbolic execution. In: IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2012), pp. 24–33. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Wadern.

[6] Uehara, T. (2016) Exhaustive Test-case Generation using Symbolic Execution. FUJITSU Sci. Tech. J., 52(1), 34–40.

[7] Staats, M. and Păsăreanu, C. (2010) Parallel symbolic execution for structural test generation. In: Proceedings of the 19th International Symposium on Software Testing and Analysis, pp. 183–194. ACM, New York, NY.

[8] Ma, E., Fu, X., and Wang, X. (2022) Scalable path search for automated test case generation. Electronics, 11(5), 727.

[9] Maragathavalli, P. (2011) Search-based software test data generation using evolutionary computation. arXiv preprint arXiv: 1103.0125.

[10] Suresh, Y. and Rath, S.K. (2014) A genetic algorithm based approach for test data generation in basis path testing. arXiv preprint arXiv: 1401.5165.

[11] Smith, G.D., Steele, N.C., Albrecht, R.F., et al. (1998) Genetic algorithm based software testing. In: Artificial Neural Nets and Genetic Algorithms: Proceedings of the International Conference in Norwich, UK, pp. 325–328. Springer, Vienna.

[12] Tufano, M., Drain, D., Svyatkovskiy, A., et al. (2022) Generating accurate assert statements for unit test cases using pretrained transformers. In: Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test, pp. 54–64. ACM, New York, NY.

[13] Alagarsamy, S., Tantithamthavorn, C., and Aleti, A. (2024) A3test: Assertion-augmented automated test case generation. Information and Software Technology, 176, 107565.

[14] Dinella, E., Ryan, G., Mytkowicz, T., et al. (2022) Toga: A neural method for test oracle generation. In: Proceedings of the 44th International Conference on Software Engineering, pp. 2130–2141. ACM, New York, NY.

[15] Husakovskyi, A. (2025) Harnessing Graph Neural Networks (GNN) for automated test case prioritization: Challenges and opportunities in QA automation. The American Journal of Engineering and Technology, 7(04), 07–15.

[16] Samoaa, H.P., Longa, A., Mohamad, M., et al. (2022) Tep-GNN: Accurate execution time prediction of functional tests using graph neural networks. In: International Conference on Product-Focused Software Process Improvement, pp. 464–479. Springer, Cham.

[17] Yuan, Z., Lou, Y., Liu, M., et al. (2023) No more manual tests? Evaluating and improving ChatGPT for unit test generation. arXiv preprint arXiv: 2305.04207.

[18] Schäfer, M., Nadi, S., Eghbali, A., et al. (2023) An empirical evaluation of using large language models for automated unit test generation. IEEE Transactions on Software Engineering, 50(1), 85–105.

[19] Chen, Y., Hu, Z., Zhi, C., et al. (2024) ChatUniTest: A framework for LLM-based test generation. In: Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, pp. 572–576. ACM, New York, NY.

[20] Wang, Z., Liu, K., Li, G., et al. (2024) HITS: High-coverage LLM-based unit test generation via method slicing. In: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, pp. 1258–1268. ACM, New York, NY.

DOI: 10.54254/2755-2721/2025.26220

[21] Alagarsamy, S., Tantithamthavorn, C., Takerngsaksiri, W., et al. (2024) Enhancing large language models for text-to-testcase generation. arXiv preprint arXiv: 2402.11910.
[22] Wang, J., Suleiman, B., and Alibasa, M.J. (2025) Automated unit test case generation: A systematic literature review. arXiv preprint arXiv: 2504.20357.
[23] Fontes, A. and Gay, G. (2023) The integration of machine learning into automated test generation: A systematic mapping study. Software Testing, Verification and Reliability, 33(4), e1845.
[24] Ali, S., Briand, L.C., Hemmati, H., et al. (2009) A systematic review of the application and empirical investigation of search-based test case generation. IEEE Transactions on Software Engineering, 36(6), 742–762.
[25] McMinn, P. (2004) Search-based software test data generation: A survey. Software Testing, Verification and Reliability, 14(2), 105–156.
[26] Pecorelli, F., Grano, G., Palomba, F., et al. (2024) Toward granular search-based automatic unit test case generation. Empirical Software Engineering, 29(4), 71.
[27] Qi, F., Hou, Y., Lin, N., et al. (2024) A survey of testing techniques based on large language models. In: Proceedings of the 2024 International Conference on Computer and Multimedia Technology, pp. 280–284. ACM, New York, NY.
[28] Alenezi, M. and Zarour, M. (2020) On the relationship between software complexity and security. arXiv preprint arXiv: 2002.07135.
[29] Ahamed, S.S. (2010) Studying the feasibility and importance of software testing: An analysis. arXiv preprint arXiv: 1001.4193.
[30] Khorikov, V. (2020) Unit Testing Principles, Practices, and Patterns. Manning Publications, New York, NY.
[31] Fraser, G. and Arcuri, A. (2011) EvoSuite: Automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, pp. 416–419. ACM, New York, NY.
[32] Pacheco, C., Lahiri, S.K., Ernst, M.D., et al. (2007) Feedback-directed random test generation. In: 29th International Conference on Software Engineering (ICSE'07), pp. 75–84. ACM, New York, NY.
[33] Pacheco, C. and Ernst, M.D. (2007) Randoop: Feedback-directed random testing for Java. In: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion, pp. 815–816. ACM, New York, NY.
[34] Randoop Project (2025) Randoop: Automatic unit test generation for Java. https: //randoop.github.io/randoop/, last accessed 2025/06/02.
[35] Cadar, C., Dunbar, D., and Engler, D.R. (2008) Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI 2008, pp. 209–224.
[36] Radford, A., Narasimhan, K., Salimans, T., et al. (2018) Improving language understanding by generative pre-training.
[37] Feng, Z., Guo, D., Tang, D., et al. (2020) CodeBERT: A pre-trained model for programming and natural languages. arXiv preprint arXiv: 2002.08155.
[38] Touvron, H., Lavril, T., Izacard, G., et al. (2023) LLaMA: Open and efficient foundation language models. arXiv preprint arXiv: 2302.13971.
[39] Bi, X., Chen, D., Chen, G., et al. (2024) DeepSeek LLM: Scaling open-source language models with longtermism. arXiv preprint arXiv: 2401.02954.
[40] Wang, L., Yuan, Y., and Wang, G.R. (2025) Design pattern detection techniques in software: Current status, challenges, and prospects. Journal of Software, 36(6), 2643–2682 (in Chinese).
[41] Jiang, J., Wang, F., Shen, J., et al. (2024) A survey on large language models for code generation. arXiv preprint arXiv: 2406.00515.
[42] Yang, Z.Y. and Wang, L. IntelliUnitGen: A unit test case generation framework based on the integration of static analysis and prompt learning. IEEE Access (Online).
[43] Zhang, Q., Fang, C., Xie, Y., et al. (2024) A systematic literature review on large language models for automated program repair. arXiv preprint arXiv: 2405.01466.
[44] Apache Software Foundation (2024) Apache Commons CLI. https: //github.com/apache/commons-cli, last accessed 2024/05/12.
[45] Wei, J., Wang, X., Schuurmans, D., et al. (2022) Chain-of-thought prompting elicits reasoning in large language models. Advances in Neural Information Processing Systems, 35, 24824–24837.
[46] Liu, C., Zhang, S.D., Ibrahimzada, A.R., et al. (2024) CodeMind: A framework to challenge large language models for code reasoning. arXiv preprint arXiv: 2402.09664.