ChatUniTest: A Framework for LLM-Based Test Generation

Yinghao Chen
Zhejiang University
China
yh_ch@zju.edu.cn

Zehao Hu
Zhejiang University
China
Huzehao@zju.edu.cn

Chen Zhi*
Zhejiang University
China
zjuzhichen@zju.edu.cn

Junxiao Han
Hangzhou City University
China
hanjx@hzcu.edu.cn

Shuiguang Deng
Zhejiang University
China
dengsg@zju.edu.cn

Jianwei Yin
Zhejiang University
China
zjuyjw@cs.zju.edu.cn

ABSTRACT

Unit testing is an essential yet frequently arduous task. Various automated unit test generation tools have been introduced to mitigate this challenge. Notably, methods based on large language models (LLMs) have garnered considerable attention and exhibited promising results in recent years. Nevertheless, LLM-based tools encounter limitations in generating accurate unit tests. This paper presents ChatUniTest, an LLM-based automated unit test generation framework. ChatUniTest incorporates an adaptive focal context mechanism to encompass valuable context in prompts and adheres to a generation-validation-repair mechanism to rectify errors in generated unit tests. Subsequently, we have developed ChatUniTest Core, a common library that implements core workflow, complemented by the ChatUniTest Toolchain, a suite of seamlessly integrated tools enhancing the capabilities of ChatUniTest. Our effectiveness evaluation reveals that ChatUniTest outperforms TestSpark and EvoSuite in half of the evaluated projects, achieving the highest overall line coverage. Furthermore, insights from our user study affirm that ChatUniTest delivers substantial value to various stakeholders in the software testing domain. ChatUniTest is available at https://github.com/ZJU-ACES-ISE/ChatUniTest, and the demo video is available at https://www.youtube.com/watch?v=GmfxQUqm2ZQ.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Large Language Models, Automatic Unit Testing Generation

ACM Reference Format:
Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. ChatUniTest: A Framework for LLM-Based Test Generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE Companion '24), July 15–19, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3663529.3663801

*Chen Zhi is the corresponding author.

FSE Companion '24, July 15–19, 2024, Porto de Galinhas, Brazil

ACM ISBN 979-8-4007-0658-5/24/07
https://doi.org/10.1145/3663529.3663801

1 INTRODUCTION

Unit testing is a critical practice in software development, ensuring the quality of software applications. However, manually writing and maintaining high-quality unit tests can be a daunting task. Automated unit test generation tools have been introduced to alleviate the burden on developers of writing essential unit tests. Existing tools [4, 6, 8, 17, 20] primarily rely on program analysis techniques for the generation of unit tests. For instance, EvoSuite [17], a notable search-based tool, demonstrates effectiveness in achieving reasonable coverage. However, a common drawback is that tests generated by these tools often lack explainability and readability.

Recently, the exploration of deep learning techniques, particularly leveraging large language models (LLMs), for the generation of unit tests has shown significant promise. TestPilot [26], a leading LLM-based tool developed by GitHub, introduces an adaptive test generation mechanism based on Codex [16]. TestSpark [9], a JetBrains IDEA plugin, provides dual modes for test generation: one leverages OpenAI and JetBrains' AI for LLM-based generation, and the other uses EvoSuite for search-based generation.

However, we have identified two primary limitations associated with llm-based tools. Firstly, the constraint on context length imposes a limitation on the capacity of LLMs to process all relevant information and generate a comprehensive unit test. Although recent LLMs exhibit the ability to handle longer contexts as input, it remains imperative to provide concise and precise context to LLMs for reasons of economic cost and the lost-in-the-middle effect [19]. Secondly, due to insufficient validation mechanisms, LLMs often produce incorrect tests with various errors, such as *"cannot find symbol"* during compilation and *"AssertionFailedError"* during runtime, which require considerable effort for manual repair.

In this paper, we introduce ChatUniTest, a framework to generate unit tests automatically based on LLMs. ChatUniTest employs an adaptive focal context generation mechanism to overcome the first limitation and implements a generation-validation-repair mechanism to tackle the second. These enhancements significantly improve LLM's effectiveness in generating unit tests. The main contributions of our work are as follows:

- We present ChatUniTest, a unit test generation framework based on LLM. By utilizing innovative mechanisms such as adaptive focal context and generation-validation-repair

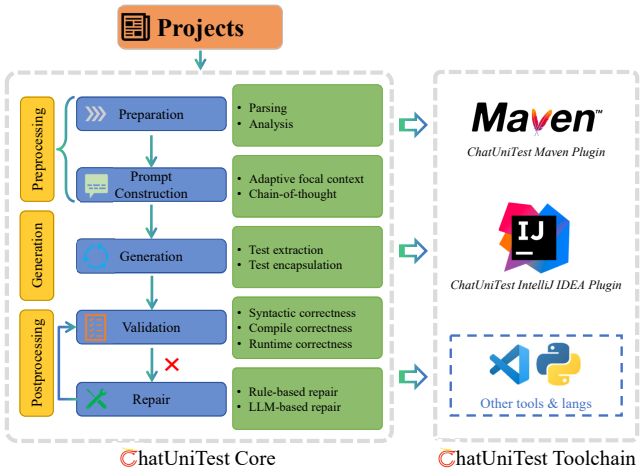

Figure 1: overview of chatunitest-core and tools derived

mechanisms, we have developed the **ChatUniTest Core** [10] to assist researchers and tool builders.

- We have implemented the **ChatUniTest Toolchain**, which includes the Maven plugin [11] and the IntelliJ IDEA plugin [12]. It offers convenience to users and caters to the requirements of various scenarios.

2 RELATED WORK

The research history of automated unit test generation spans several decades. However, traditional methods [17, 20] for generating unit tests exhibit significant deficiencies in terms of explainability and readability. Recently, LLMs have demonstrated remarkable performance across various programming tasks. In response, researchers have proposed numerous test generation approaches[9, 15, 22, 23, 25–30] based on these models, which are capable of generating well-explained and readable unit tests. Notably, TestPilot[26] and TestSpark[9] are typical test generation solutions based on LLMs, and they provide user-friendly services or tools for this purpose. TestPilot[26] automates unit test generation for JavaScript programs. It utilizes function signatures, implementation, and usage examples extracted from documentation. However, its reliance on documentation limits its applicability to undocumented or sparsely documented programs. TestSpark[9] introduces a novel approach to unit test generation within the IntelliJ IDEA. Leveraging both search-based test generation tools and large language models (LLMs), TestSpark employs a feedback cycle between the IDE and LLMs, enhancing test generation effectiveness.

3 APPROACH

As shown in Figure 1, **ChatUniTest Core** establishes the overall architecture designed for both researchers and tool builders. This architecture encompasses three phases and five steps: preprocessing (including preparation and prompt construction), generation, and postprocessing (including validation and repair). The preprocessing phase mainly focuses on how to unlock the capabilities of LLM, such as designing efficient and effective prompts. The postprocessing phase primarily focuses on how to mitigate the limitations of LLM, such as by repairing generated test cases.

3.1 Preparation

ChatUniTest initially requires preparation to optimize and accelerate generation. This includes parsing the target projects, analyzing dependencies, and obfuscating the code for privacy.

Parsing. The primary objective of the parsing step is to extract raw information from the code. In this stage, ChatUniTest scans the project folder to identify all Java files. Each Java file is then converted into an Abstract Syntax Tree (AST). As ChatUniTest explores the AST, it collects information at both the class and method levels. For classes, it notes the package, imports, extends, implements, fields, and method signatures. For methods, it records the method signature, body, field access, getter/setter invocation, dependent class names, and method invocations.

Analysis. The analysis step is primarily focused on leveraging static analysis techniques to extract deeper insights to promote unit test generation. For example, we intend to conduct program slicing to extract API call sample code and perform dependency analysis to build Object Construction Graphs (OCGs [18]), which can provide guidance for LLM to generate driver code.

3.2 Prompt Construction

ChatUniTest employs an adaptive focal context generation mechanism and a prompt template to construct the input prompt for LLM. The adaptive focal context generation mechanism aimed to overcome the primary limitation. This mechanism formulates a context abundant in essential details about the focal method while excluding extraneous information from the focal class. The prompt template provides an approach to designing specific prompts for various requirements, or one can simply utilize the default template.

Adaptive Focal Context Generation. The adaptive focal context generation mechanism is designed to add as much valuable information to the prompt as possible while ensuring that each addition stays within the token limit. This approach optimizes the focal method's context, facilitating the integration of the class context and the contexts of associated methods.

The mechanism dynamically generates context according to the dependency relationships of the focal methods. Initially, the context encompasses the information of the focal method and the focal class. Based on the result of dependency analysis, other relevant information (such as signatures of dependent methods and classes) might also be integrated into the context.

In the future, we plan to enhance the adaptive focal context generation mechanism. Our ultimate goal is to produce the optimal context according to the properties of each focal method, ensuring that LLMs have a clearer understanding, which in turn will elevate the quality of the generated tests.

In accordance with adaptive focal context generation, ChatUniTest initially parses the template file and substitutes the labels and placeholders with actual data. If the prompt tokens exceed the token limit, ChatUniTest will automatically eliminate the least relevant data in the user prompt, typically the last label or placeholder. By doing so, the prompt tokens are within the limit.

Prompt Template. ChatUniTest utilizes the FreeMarker Java Template Engine[2] to generate the prompt based on the template and

changing context. This approach, combined with the adaptive focal context mechanism, provides users with the flexibility to design diverse prompt templates tailored to their requirements while still reserving ample tokens for the LLM to produce helpful responses.

Chain-of-thought. Based on the dependency graph constructed during preprocessing, we are implementing to create chain-of-thought (COT) prompts. Prior research [21] has already demonstrated that the COT can effectively enhance the capability of large language models (LLMs) in addressing software engineering tasks. We believe COT could make LLMs achieve more comprehensive understanding of the code, subsequently improves the quality of the generated tests.

3.3 Generation

Utilizing the generated focal context and the user-provided prompt template (or the default template if none is provided), the generation component integrates essential information in the context into the prompt. To guarantee sufficient tokens for test case generation, a fixed threshold is set to control how many tokens the prompt can consume. The context information is then filled within this threshold. Subsequently, the LLM is invoked using the LLM API.

Upon receiving a response, ChatUniTest utilizes its CodeExtractor component to extract tests from the response. Our experiments have shown that the LLM may provide either a single test method or an entire test class. To address this, we have implemented the TestSkeleton, which can encapsulate the test method automatically to construct a complete test class. Once the test class is generated, it marks the completion of the test generation process.

Inspired by the CAT-LM [23], we also build **ChatUniTest Models**[13], which provides fine-tuned models for Java test generation tasks based on **Code Llama**[24].

3.4 Syntactic, Compile and Runtime Validation

The extracted test will be forwarded to the validation component for further verification of its correctness. The validation process consists of three sub-steps: syntactic validation, compile validation, and runtime validation. A test is considered successfully generated only if it passes all the validation steps.

Syntactic Validation. ChatUniTest first utilizes Java Parser to verify the syntax of generated tests. If a sytax error is detected at this stage, it can usually be fixed by the rule-based repair component. For example, an error *"Parse error. Found <EOF>"* in JavaParser suggests that there might be a missing *";"*, *"}"*, or another closing symbol in the code.

Compile Validation. Following the syntactic validation, ChatUniTest proceeds to compile the test. This step not only checks for syntax errors but also examines certain semantic issues in the code. If the compile fails, the tool will then move on to the repair phase. For instance, the error *"cannot find symbol"* suggests that there might be missing dependencies. It can usually be resolved with the appropriate import statement.

Runtime Validation. After completing the compile validation, ChatUniTest begins the execution of the test. If any runtime errors are encountered, the tool will transition to the repair phase. As an example, the error *"org.mockito.exceptions.misusing"* suggests that a mockito method is used incorrectly. Such as *when()* method is used without providing a method call on a mock object as its argument.

3.5 Rule-based and LLM-based Repair

If errors occur during the validation step, ChatUniTest initiates the repair process for rectification. There are two types of repair strategies: rule-based repair and LLM-based repair. The former are used to repair specific errors, and the latter are used as a fallback measurement to repair a broader range of unexpected errors.

Rule-based Repair. ChatUniTest first employs its rule-based repair component to correct specific errors in the test. Conceptually, a repair rule consists of two parts: triggers and transformers. Triggers define when to apply the repair rule, and transformers formulate how to rewrite the code to resolve the error. Currently, ChatUniTest implements two simple repair rules.

- When the LLM's response exceeds the token limit, it can lead to truncated tests and syntactic errors. To address this, ChatUniTest first adds the necessary ending braces to validate the structure. If this fails, it searches for the *"@Test"* annotation, truncates the code preceding it, and appends closing braces. If the repair fails or no tests remain post-removal, the process ends.
- If a *"cannot find symbol"* error arises during the compile validation process, ChatUniTest tries to address it by simply copying all import statements from the focal class into the generated tests.

LLM-based Repair. If errors persist in the generated test after applying all the rule-based repairs, it should be handed over to the LLM-based repair. Within this repair phase, ChatUniTest gathers details about the error type and message, integrates this with the incorrect test's context, and generates a prompt using a predefined template. If a test remains unresolved after a specified number of rounds, the entire process is terminated.

4 ILLUSTRATIVE EXAMPLE

As shown in Figure 1, we build the **ChatUniTest Toolchain** based on **ChatUniTest Core**. The toolchain provides plugins tailored for various scenarios and offers users a more streamlined experience. To demonstrate the capabilities of ChatUniTest, we examine a specific test case generated by the IntelliJ IDEA plugin.

Consider the method *equals* within the *StringUtils* class, as depicted in Figure 2. This method aims to determine if two given *CharSequence* objects, *cs1* and *cs2*, are equal.

```
public static boolean equals(
        CharSequence cs1, CharSequence cs2) {
    if (cs1 == cs2) {
        return true;
    }
    if (cs1 == null || cs2 == null) {
        return false;
    }
    return cs1.equals(cs2);
}
```

Figure 2: Buggy method *equals* in Apache Commons.

Developers can invoke ChatUniTest to generate unit tests for the method *equals*. Figure 3 illustrates the generation results. The generated test will throw *"AssertionFailedError"* at line 15 because

StringBuilder implements *CharSequence* but does not override the *equals* method from the Object class. As a result, for *StringBuilder*, the *equals* method checks for reference equality rather than value equality. Therefore, the test detects a defect in the focal method.

```
public class StringUtilsEqualsTest {

    @Test(timeout = 8000)
    public void testEquals() {
        assertTrue(StringUtils.equals(null, null));
        assertFalse(StringUtils.equals("test", null));
        assertFalse(StringUtils.equals(null, "test"));
        assertTrue(StringUtils.equals("test", "test"));
        assertFalse(StringUtils.equals("test", "Test"));
        assertFalse(StringUtils.equals("Test", "test"));
        assertTrue(StringUtils.equals("", ""));
        assertTrue(StringUtils.equals(" ", " "));
        assertFalse(StringUtils.equals(" ", ""));
        assertFalse(StringUtils.equals("", " "));
        assertTrue(StringUtils.equals(
                new StringBuilder("test"), new StringBuilder("test")));
        assertFalse(StringUtils.equals(
                new StringBuilder("test"), new StringBuilder("Test")));
    }
}
```

Figure 3: The Test Case generated by ChatUniTest.

5 EVALUATION

Effectiveness Evaluation. Our initial evaluation aims to ascertain the effectiveness of ChatUniTest by assessing the quality of the tests it generates. To establish baselines for comparison, we selected two representative tools: EvoSuite (representing program-analysis-based tools) and TestSpark (representing LLM-based tools). Notably, TestPilot was excluded from this comparison due to its lack of support for Java. In our experimental setup, we utilized the default configuration for EvoSuite and maintained consistent configurations between ChatUniTest and TestSpark. This included the use of gpt-3.5-turbo-0613 as the base model, with each round comprising one generation attempt followed by five repair attempts. This standardized configuration ensures a fair and unbiased evaluation across the selected tools.

As Table 1 shows, we selected four Java projects for our experiment: Commons-Cli[1], Commons-Csv[3], Ecommerce-microservice[5], and Binance-connector[7]. The former two projects are widely used in related work [15, 27] and likely to be included in the training data of GPT-3.5-turbo. While the latter two are unseen popular projects (more than 100 stars) that are not part of the training data (as they are created after the creation of training data). Considering the need for repeated operations of these IDEA plugins in the experiment, we employed random sampling with a 95% confidence level and a 5% margin of error. As a result, we randomly selected 264 methods from a total of 835 focal methods across these four projects. In our experimental procedure, we attempted to generate a test for each of these selected methods.

Table 1: Result of experiment

Porject Information			Line Coverage		
Name	Version	Unseen	ChatUniTest	TestSpark	EvoSuite
Cli [1]	1.5.0	no	70.9%	78.4%	**91.8%**
Csv [3]	1.10.0	no	**73.3%**	-	28.3%
Ecommerce [5]	695a6d4	yes	26.7%	**36.7%**	-
Binance [7]	2.0.0	yes	**49.2%**	29.7%	20.8%
Overall Coverage	-	-	**59.6%**	42.1%	38.2%

In Table 1, a comprehensive comparison of line coverage is provided for ChatUniTest, EvoSuite, and TestSpark across four projects. Notably, TestSpark fails to generate tests for the Csv project, attributed to the constructed prompt surpassing the token limit of the model. Additionally, in the Ecommerce project, EvoSuite fails to generate tests for focal methods, primarily due to JDK version incompatibility issues between EvoSuite and the Ecommerce project.

In summary, ChatUniTest showcased relatively reliable line coverage across diverse projects, achieving an impressive line coverage of 59.6%. This performance surpassed that of both EvoSuite and TestSpark, underscoring ChatUniTest's resilience and efficacy as a unit test generation tool.

Usefulness evaluation. We conducted a user study to investigate the usefulness of ChatUniTest. We distributed questionnaires (see [14] for the details) to 19 individuals who either starred, forked our project, raised issues on GitHub, or reached out to us via email. Out of these, we received 9 responses. The respondents comprised five students and four senior software developers. All participants had a foundational understanding of software testing; five of them are well-versed in the domain. Additionally, every participant had prior experience with large language models in programming.

The survey results indicate that 89% of the respondents use ChatUniTest to assist in writing test cases. All of them believe that using ChatUniTest is beneficial for writing test cases, with 33% of them stating that it's highly beneficial. For junior developers, especially students, ChatUniTest can effectively assist them in writing unit tests. Moreover, 33% of respondents are further developing upon ChatUniTest, such as adding support for additional languages, integrating new LLMs, and incorporating it into their development workflows. Notably, some respondents provided valuable suggestions. These include integrating performance testing and supporting more testing frameworks, especially those internal testing frameworks. They also suggested measuring and verifying the reliability of the generated test code in real production environments. These suggestions not only provide guidance for ChatUniTest but also contribute to the future development of this field. Overall, these results highlight the users' high appreciation for ChatUniTest and their expectations for its future development.

6 CONCLUSION AND FUTURE WORK

We present ChatUniTest, an automated unit test generation framework that leverages the capabilities of the LLMs and provides a suite of user-friendly APIs to assist developers in their software testing tasks. In the future, we plan to improve our framework to produce the optimal context and extend our preparation and validation process to support more programming languages. Moreover, we intend to provide a broader range of benchmark implementations (such as SysPrompt [25] and ChatTester [30]), built upon the ChatUniTest.

ACKNOWLEDGMENTS

This work was supported in part by the National Natural Science Foundation of China under Grants U20A20173 and 62125206, in part by the National Key R&D Program of China (2022YFF0902702), in part by the Zhejiang Pioneer (Jianbing) Project (2023C01045), in part by the Key R&D Program of Ningbo (2023Z235), and in part by the ZJU-Hundsun Fintech Research & Development Center.

REFERENCES

[1] 2007. https://github.com/apache/commons-cli.
[2] 2007. Apache FreeMarker. https://freemarker.apache.org/.
[3] 2014. https://github.com/apache/commons-csv.
[4] 2017. https://github.com/wrdv/testme-idea.
[5] 2021. https://github.com/SelimHorri/ecommerce-microservice-backend-app.
[6] 2022. https://github.com/UnitTestBot/UTBotJava.
[7] 2022. https://github.com/binance/binance-connector-java.
[8] 2023. https://github.com/ArtemUntila/kex-intellij-plugin.
[9] 2023. https://github.com/JetBrains-Research/TestSpark.
[10] 2023. https://github.com/ZJU-ACES-ISE/chatunitest-core.
[11] 2023. https://github.com/ZJU-ACES-ISE/chatunitest-maven-plugin.
[12] 2023. https://github.com/ZJU-ACES-ISE/ChatUniTest_IDEA_Plugin.
[13] 2024. https://github.com/ZJU-ACES-ISE/chatunitest-models.
[14] 2024. https://github.com/ZJU-ACES-ISE/chatunitest-core/tree/main/docs/survey.
[15] Saranya Alagarsamy, Chakkrit Tantithamthavorn, and Aldeida Aleti. 2023. A3Test: Assertion-Augmented Automated Test Case Generation. *arXiv preprint arXiv:2302.10352* (2023).
[16] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
[17] Gordon Fraser and Andrea Arcuri. 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*. 416–419.
[18] Yun Lin, You Sheng Ong, Jun Sun, Gordon Fraser, and Jin Song Dong. 2021. Graph-based seed object synthesis for search-based unit testing. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1068–1080.
[19] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172* (2023).
[20] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816.
[21] Yun Peng, Chaozheng Wang, Wenxuan Wang, Cuiyun Gao, and Michael R Lyu. 2023. Generative Type Inference for Python. *arXiv preprint arXiv:2307.09163* (2023).
[22] Juan Altmayer Pizzorno and Emery D Berger. 2024. CoverUp: Coverage-Guided LLM-Based Test Generation. *arXiv preprint arXiv:2403.16218* (2024).
[23] Nikitha Rao, Kush Jain, Uri Alon, Claire Le Goues, and Vincent J Hellendoorn. 2023. CAT-LM training language models on aligned code and tests. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 409–420.
[24] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950* (2023).
[25] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-Aware Prompting: A study of Coverage Guided Test Generation in Regression Setting using LLM. *arXiv preprint arXiv:2402.00097* (2024).
[26] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. Adaptive test generation using a large language model. *arXiv preprint arXiv:2302.06527* (2023).
[27] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617* (2020).
[28] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* (2024).
[29] Chen Yang, Junjie Chen, Bin Lin, Jianyi Zhou, and Ziqi Wang. 2024. Enhancing LLM-based Test Generation for Hard-to-Cover Branches via Program Analysis. *arXiv preprint arXiv:2404.04966* (2024).
[30] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. arXiv:2305.04207 [cs.SE]

Received 2024-01-29; accepted 2024-04-15