Exploring the Capability of ChatGPT in Test Generation

Gaolei Yi[1], Zizhao Chen[2], Zhenyu Chen[1,*], W. Eric Wong[2], and Nicholas Chau[2]
[1]State Key Laboratory for Novel Software Technology Nanjing University, Nanjing, China
[2]University of Texas at Dallas, Richardson, Texas, USA
yigaolei@smail.nju.edu.cn, zxc190007@utdallas.edu,
zychen@nju.edu.cn, ewong@utdallas.edu, NCC200000@utdallas.edu
*corresponding author

Abstract—The design of test is a crucial step in the field of software testing. The quality of test significantly impacts the effectiveness of software testing, with well-designed test cases improving the efficiency of bug detection. However, manual test case design and writing are often considered time-consuming and labor-intensive.

With the emergence of large language models (LLMs), especially ChatGPT, the potential of LLMs in the field of test generation has become evident. Pre-trained LLMs can learn and understand code in various programming languages and design test cases using multiple testing frameworks.

In this paper, we used ChatGPT to generate tests for some tested projects. Through experiments, we found that ChatGPT has some gaps compared to traditional test generation tools, but its performance is closer to manual testing. However, the tests generated by ChatGPT exhibit higher readability. We believe that ChatGPT is better suited to serve as a manual testing assistant, helping understand the tested code and providing testing ideas.

Keywords–Test Generation; LLM; ChatGPT

1. Introduction

Unit testing holds an extremely crucial position in software testing. During the software development process, as soon as the code is designed and written, unit testing can be used to test each simple or complex method and observe its detailed input and output data. Therefore, unit testing is a fundamental testing method that can be performed in the software development stage, and it is a method that helps verify the correctness of software unit functionality [1].

In unit testing, there are certain metrics used to assess the adequacy of unit tests, such as code coverage [2]. However, for manual testing, meeting these testing metrics can be a very time-consuming and labor-intensive task. This is a reason why testing professionals or developers may not enjoy unit testing work. Therefore, tools for automating the generation of unit tests are an important area of research in the field of unit testing.

To improve the efficiency of unit testing, several automated testing frameworks have been developed. For example, in the Java domain, JUnit [3] is widely used to help testing professionals easily write test cases. Testers utilize the APIs provided by JUnit to quickly create test cases, and JUnit can automatically assist in checking the correctness of the test cases. On the other hand, automated testing tools can generate unit tests using unit testing frameworks, such as Evosuite [4] and Sushi [5]. This eliminates the need for testers to manually write test code, as these tools can achieve most of the unit testing criteria automatically.

The emergence of large language models, such as ChatGPT, has shown their potential in various domains [6], [7]. ChatGPT is known for its strong language understanding and text-processing capabilities. It can automatically generate code that meets specific functional requirements based on human input [8]. This has piqued the interest of researchers in the field of software engineering [9]. The question is whether ChatGPT can automatically generate test cases when provided with code to be tested, meeting the specified requirements.

In this paper, we explore the ability of ChatGPT to automatically generate tests in the field of unit testing. After generating tests using ChatGPT, we compare them with those generated by the commonly used testing tool, Evosuite. Additionally, as ChatGPT is believed to be able to replace human effort in various aspects, we are curious to see how ChatGPT compares to manual testing. Therefore, we compare the tests generated by ChatGPT with data from manual testing.

2. Related Work and Background

2.1 Traditional Test Generation Methods

2.1.1 Test Generation Methods Based on Genetic Algorithms and Evolutionary Algorithms

In the context of software testing, the utilization of genetic algorithms (GAs) and evolutionary algorithms (EAs) for test case generation has garnered significant attention [10]–[12]. Genetic algorithms, inspired by the principles of natural selection and genetics, are employed to evolve a population of potential test cases over multiple generations. Variants of GAs, such as multi-objective genetic algorithms and co-evolutionary algorithms, have been adapted to address specific challenges in test case generation [11]. Evolutionary algorithms, a broader class that includes genetic algorithms, encompass techniques like genetic programming and differential evolution. These algorithms have been applied to test case generation in scenarios where the search space is large or nonlinear, as is often the case in software systems [13]. However, it's important to note that while these algorithms can efficiently explore the solution space, they may also suffer from convergence issues and difficulties in balancing exploration and exploitation.

2693-9371/23/$31.00 ©2023 IEEE
DOI 10.1109/QRS-C60940.2023.00013

2.1.2 Coverage-Based Test Generation Methods

Coverage-based test generation methods focus on achieving specific coverage criteria during the testing process. These criteria include statement coverage, branch coverage, path coverage, and more, which guide the creation of test cases that exercise various parts of the codebase [14]. Such methods aim to ensure that all possible execution paths are traversed and critical scenarios are adequately tested.

Consider the scenario of testing a compiler for a new programming language. To achieve branch coverage, where every possible decision point in the code is evaluated, a coverage-based test generation method could systematically create test cases that explore different paths through the compiler's code. This might involve generating inputs that trigger different branches, loops, and conditional statements, ensuring that the compiler's behavior is thoroughly assessed.

One limitation of coverage-based methods lies in their inability to guarantee the detection of all defects, as they focus primarily on code coverage metrics rather than targeting specific functional behaviors. Additionally, these methods might struggle with complex control flows and nested conditions, potentially leading to incomplete coverage despite considerable testing efforts.

In summary, traditional test generation methods based on genetic algorithms and coverage criteria have offered valuable contributions to the field of software testing. However, researchers continue to explore ways to enhance their effectiveness, address their limitations, and incorporate them into comprehensive testing strategies.

2.1.3 Model-Driven Test Generation Methods

Model-driven approaches in test case generation have gained prominence due to their ability to leverage abstract representations of software systems to guide testing efforts. These methods harness the power of models to systematically generate test cases that explore various aspects of the system's behavior. They involve constructing models that capture the software's structure, behavior, and interactions, which are then transformed into executable test cases [15].

Moreover, various model-driven testing tools have been developed to streamline this process. Tools like Spec Explorer [16] and UML-based testing tools [17] provide graphical interfaces for designing models and generating corresponding test cases. These tools enable testers to focus on the abstract representation of the system and its intended behavior, while the tool takes care of converting these representations into executable test scripts.

2.1.4 Test Generation Methods Based on Symbolic and Concolic Executions

Symbolic and concolic executions have emerged as powerful techniques for test case generation. Symbolic execution involves analyzing a program's code paths symbolically, using variables and expressions instead of concrete values [18]. Concolic execution combines both concrete and symbolic execution to explore paths through the program's codebase systematically [19]. In the domain of symbolic and concolic executions, symbolic execution explores all possible paths that conditions in the code can take based on symbolic inputs. Concolic execution enhances this process by incorporating actual concrete values from the execution environment, allowing for precise analysis of complex control flows and constraints. These techniques offer advantages such as automated constraint solving and the generation of diverse test inputs. Researchers have developed various tools and frameworks, such as SAGE [20] and KLEE [21], to facilitate symbolic and concolic execution for test case generation. However, these methods also come with challenges. They can suffer from path explosion, where the number of possible execution paths becomes unmanageably large [18]. Furthermore, complex data structures, non-linear operations, and interactions with external resources can complicate symbolic and concolic execution, limiting their applicability in certain scenarios.

2.1.5 Model-Based Test Generation Methods

Model-based test generation methods rely on abstract representations of systems, such as state machines or formal models, to guide test case creation. These methods involve creating models that capture the intended behavior of the software, and then systematically deriving test cases from these models [22]. For instance, consider the validation of a communication protocol used in distributed systems. A model-based approach might involve constructing a formal model that depicts the interactions between different components of the system. Test cases can then be generated by exploring the model's transitions and interactions, ensuring that the protocol functions correctly under various scenarios [23]. Tools like Spec Explorer [16] and Alloy Analyzer [24] provide support for model-based testing by enabling the design and analysis of system models. These tools aid in automatically generating test cases that cover different scenarios and interactions, helping to uncover potential defects and vulnerabilities.

2.2 AI Methods in Test Generation

The integration of machine learning techniques into test case generation has emerged as a promising avenue of research [25]. This subsection delves into the current state of generating test cases using machine learning models. It explores the methodologies employed to create datasets for training and the techniques used for feature engineering. Additionally, the application of common machine learning algorithms in test case generation is discussed, highlighting their role in automating the test case creation process.

2.2.1 Application of Natural Language Processing (NLP) in Test Generation

The application of natural language processing (NLP) techniques to test generation introduces innovative ways to enhance the testing process. This part investigates the potential advantages of utilizing NLP in software testing. It then delves into the ways NLP can be employed in test generation or software testing, possibly exploring the incorporation of pretrained NLP models [26]. For instance, the integration of NLP techniques could aid in the automatic extraction of requirements from textual specifications and user stories, subsequently facilitating the generation of test cases that accurately cover these requirements. Additionally, pretrained NLP models like BERT [27] or GPT-3 [28] could be fine-tuned to comprehend domain-specific jargon and generate test cases from natural language descriptions of software features.

2.2.2 Application of Large Language Models (LLMs) in Test Generation

Large language models (LLMs) represent a significant advancement in AI capabilities, offering improvements over previous AI and NLP models. Large language models offer the possibility of replacing human efforts in the field of software engineering, such as in code generation and code repair [29], [30]. This subsection explores the advantages of LLMs compared to their predecessors. It then explores the potential applications of LLMs in test generation within the realm of software testing, either through existing research or future possibilities. LLMs could revolutionize test case generation by understanding complex software requirements, specifications, and functional descriptions. By interacting with LLMs, testers could describe desired test scenarios in natural language and receive automatically generated test cases as output. Furthermore, LLMs might aid in generating edge cases and uncovering corner-case defects that might be missed by traditional test generation methods.

3. Approach

As mentioned above, our goal is to assess ChatGPT's ability to produce unit test cases. In this section, an approach is designed to measure ChatGPT's ability to generate unit test cases, comparing it with both baseline models and human beings.

For the GPT model, we have taken the GPT-3.5-turbo model as the object of study. During the study, the model is considered as an intelligent test generation tool that generates test cases for the project under test and then compares it with traditional testing tools and manual labor.

In this process, we first collected some Java projects of data structures or algorithms as the tested projects. Afterwards, we use the prompt to prompt the GPT model, and then input the code of the project under test to GPT to let GPT generate test cases. We will collect all test cases generated by GPT for statistical analysis. At the same time, we also use the baseline tool to generate test cases for the project under test and perform statistical analysis on the generated test cases.

3.1 Project Collection

We use some data structure and algorithm related Java projects as the projects under test. As mentioned before, in this paper, we would like to evaluate the test generation capability of ChatGPT by comparing it with human beings, so we need to have enough human samples for comparison. At the same time, we require that the items chosen to be tested should be able to guarantee quality and not be too simple.

For the above purposes, we used these items from the competition items of the National Student Software Testing Contest in China. The National Student Content of Software Testing is a software testing competition for college students organized by MoocTest. The competition is attended by students from colleges and universities across China every year and consists of several competitions, such as developer testing, web testing, and embedded software testing. The developer tests are conducted in the form of unit tests, and the purpose of this thesis is to evaluate the ability of ChatGPT to generate tests in the form of unit tests, so the test items used in the developer tests of the competition are very much in line with our requirements. The test items are all from highly rated open-source projects and are mainly concerned with data structures and algorithms. In addition, since the competition consists of three stages: selection, review, and final, the level gap of the participants in each stage is different. In order to minimize the gap in the level of manually designed test cases, we chose the test items of the final round as the items under test for the study.

For the selected test projects, we separately collect their metrics, including the number of statements (Nos), the number of branches (NoB), cyclomatic complexity (CC), and the number of mutants (NoM).

- Number of Statements (NoS) is the number of executable statements of the code, i.e., the number of executable statements of the code used in the calculation of statement coverage in the structure-based test metrics.
- Number of Branches (NoB) is the number of branches in the code under test and is an important criteria in software testing. In general, there are three structures in the code that can be considered as code branching: 1) conditional transfer of control from any node to any other node in a control flow model; 2) explicit and unconditional transfer of control from any node to any other node in a control flow model; 3) transfer of control to the entry point of a test item when the test item has multiple entry points. In most software testing processes, only the first case is considered, i.e., branches generated by if or switch statements in the code.
- Cyclomatic Complexity (CC), also known as Conditional Complexity, is a measure of code complexity and is symbolized by V(G). Cyclomatic Complexity is often used as a measure of the complexity of a module's decision structure, quantified in terms of the number of independent paths, and can also be interpreted as the minimum number of test cases used to cover all possible scenarios.
- Mutation Testing (sometimes called "mutation analysis") is a software testing methodology that improves the source code of a program in detail. These so-called mutations are based on well-defined mutation operations that simulate typical application errors that can occur during code writing, such as using the wrong operator or variable name. Code segments with mutation operations are called mutants. A mutant will be called a killed mutant if it is detected by

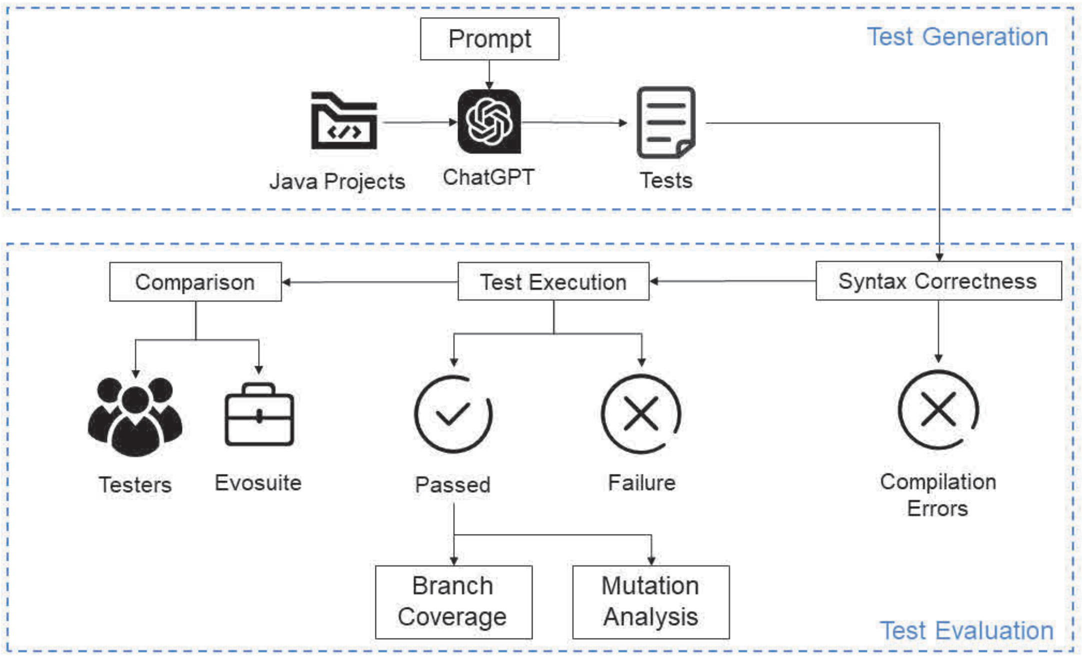


Figure 1. The workflow of generating test cases with ChatGPT and evaluating progress.

a test case, or lived if all the test cases do not detect this mutant.

In this paper, we use Jacoco [31] to count the number of statements, branches, and circle complexity for each project under test. JaCoCo is a Java code coverage library that uses a series of counters to compute the code structure coverage criteria, which include statement coverage, branch coverage, and circle complexity. All of these counters are derived from information contained in Java class files, which is essentially debugging information optionally embedded in Java bytecode instructions and class files. For counting the number of mutants, a widely used Java mutation testing tool, PIT [32], is used. Pitest automates the generation of mutants for the project under test, then automatically executes all the test cases and finally calculates the percentage of killed mutants, i.e., the mutation score.

3.2 Test Generation

The purpose of test generation is to generate tests for the project under test using ChatGPT under specified conditions. Since ChatGPT is believed to have the ability to understand the code, we tried to observe the ability of ChatGPT to generate tests using a simpler prompt and a more detailed prompt under guidance. We performed tests at different levels of granularity, i.e., using ChatGPT to generate tests for the class in the project under test and for each method in the class, respectively, within a specified time frame.

The process of test generation is shown in Algorithm 1, where we first initialize the set of test cases TS for the project under test, and initialize the variables Cov and MS, which are used to record the branch coverage and mutation score of TS after each update, respectively. Initialize N, which is used to record the number of test cases in TS (Lines 1-4).

The time for the entire process of generating tests for a project using ChatGPT is limited to 90 minutes. The duration of the Developer Testing event of the National Student Contest of Software Testing is three hours and is consisted of two test projects. The two test projects are of similar difficulty, so it is assumed that the time spent by each contestant on each test project is one and a half hours, i.e., 90 minutes. In order to try to maintain fairness, the total time for generating tests for a test project with ChatGPT will be limited to 90 minutes (Line 5).

As widely acknowledged, ChatGPT heavily relies on prompts. The comprehensiveness and accuracy of a prompt in conveying task information to ChatGPT will directly impact its operational efficiency. In this context, we first provide ChatGPT with a rough prompt, instructing it to generate tests for a given class. (Include prompt design here) For a particular class, tests are generated a total of J times. Each generated test case TC_j is added to the test suite TS, and the achieved branch coverage (Cov_branch) and mutation score (MS) of the updated TS are calculated. The process of generating tests for the class is halted when it is observed that both the branch coverage (Cov_{branch}) and mutation score (MS) have ceased to change (Lines 6-10).

When the Cov_{branch} and MS of new test cases cease to change, from the perspective of code coverage, these test cases can be considered redundant. If the test generation process continues, it would lead to inefficiencies. Therefore, in such circumstances, the direct generation of test cases for this class is halted. Instead, a more detailed and specific prompt is required for ChatGPT, instructing it to generate test cases for a particular method within that class (Lines 11-14).

For a method within a class, we generate test cases K times. Each newly obtained test case TC_k is added to the test suite TS, and the updated TS's Cov_branch and MS are calculated (Lines 15-19). Similar to the previous steps, when it is observed that the Cov_branch and MS of TS have ceased to change, the generation of new tests for that method is halted. No longer generating redundant test cases is crucial to mitigate time costs. (Lines 20-22)

At the end of Algorithm 1, we will have the test suite TS generated by ChatGPT, along with its Cov_branch and MS. This will aid us in evaluating ChatGPT's test-generation capability.

Algorithm 1 Test Generation with ChatGPT

```
Input: Project under test $P$, total time $T$
Output: Test suite $TS$, branch coverage $Cov_{branch}$, mutation score $MS$
 1: initialize test suite $TS$
 2: initialize branch coverage $Cov_{branch}$
 3: initialize mutation score $MS$
 4: initialize $N$ to the number of $TS$
 5: while $T \leq 90$ minutes do
 6:     for class $C$ in $P$ do
 7:         for $j$ in $J$ do
 8:             generate tests $TC_j$ for $C$
 9:             add $TC_j$ to $TS$
10:             calculate $Cov_{branch}$ and $MS$
11:             if $Cov_{branch}$ and $MS$ not changed then
12:                 break
13:             end if
14:         end for
15:         for method $M$ in class $C$ do
16:             for $k$ in $K$ do
17:                 generate tests $TC_k$ for $M$
18:                 add $TC_k$ to test suite $TS$
19:                 calculate $Cov_{branch}$ and $MS$ for $TS$
20:                 if $Cov_{branch}$ and $MS$ not changed then
21:                     break
22:                 end if
23:             end for
24:         end for
25:     end for
26: end while
27: return $TS$, $Cov_{branch}$, $MS$
```

3.3 Baselines

As mentioned earlier, ChatGPT is considered as an intelligent test generation technique in this thesis, and in order to compare the test generation capabilities of ChatGPT, it is necessary to use advanced Java test generation techniques as baselines. For this purpose, we choose the traditional test case generation tools used in Java project testing, Evosuite [4], to generate test cases for the project under test. Evosuite is an automated test generation tool for Java. It generates tests as an evolutionary process using a genetic algorithm that generates a minimal set of tests for the code under test, accompanied by test assertions that react to the behavior of the corresponding class under test. In the National Student Contest of Software Testing, which provided us with artificial data samples, the scoring criteria included branch coverage and variance scores. So the contestants have to do their best to improve the branch coverage and mutation score of the tests within 3 hours during the contest. Based on this scenario, while generating tests using ChatGPT and Evosuite, the branch coverage and mutation score also need to be as high as possible. When generating tests using Evosuite, we do not limit the classes or methods for which tests can be generated, nor do we limit the number of tests that can be generated for each class or method.

4. Experiment

This section is aimed at conducting a comprehensive evaluation of the capability of GPT in test generation through the design of a series of experiments. Throughout this process, we have undertaken a sequence of targeted steps to thoroughly scrutinize the effectiveness of our approach.
To begin with, we formulate several research questions (RQs) that guide our assessment outcomes. Subsequently, we introduce the test data, which corresponds to the target projects used for test generation. Following that, we will outline the metrics employed to assess the test generation abilities, along with the details of the experimental settings. Finally, we will present an evaluation of the experimental results in relation to the various research questions.

4.1 Research Questions

We formulate the following research questions for this empirical study and attempt to address these questions through experimentation.

RQ1: Can the tests generated by GPT be directly used for test execution?

RQ2: How does the effectiveness of test generation by GPT compare to Evosuite?

RQ3: Has GPT's test generation capability surpassed that of highly skilled human testing professionals?

The purpose of **RQ1** is to analyze the accuracy of tests generated by GPT. Undoubtedly, GPT can generate a substantial number of tests within a short timeframe. However, what is the usability of these tests? How accurate are these tests? A higher accuracy of these tests implies that when we obtain them, they can be more readily utilized for testing, resulting in lower time and human resource costs for test refinement efforts. Therefore, we intend to investigate the accuracy of tests generated by GPT, assessing it from the following two perspectives:

Evosuite is widely used Java test generation tools in traditional testing techniques. In **RQ2**, we will compare ChatGPT with Evosuite to assess their respective test-generation capabilities. **RQ3** introduces the objective of comparing GPT's test generation capability with human effort. Through various experimental metrics, we will compare GPT with human testers from multiple perspectives, aiming to explore the feasibility of employing GPT for test generation in the current context.

4.2 Experiment Object

As previously mentioned in 3-A, the test items we have chosen are shown in Table I. The number of statements (Nos), number of branches (NoB), cyclomatic complexity (CC), and number of mutations (NoM) of these projects are listed in Table I.

TABLE I
PROJECTS INFORMATION

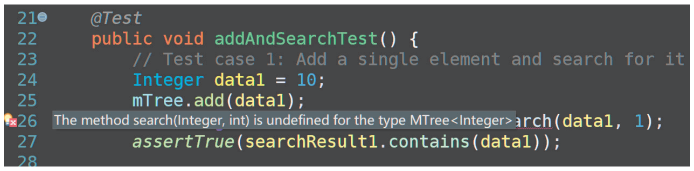
Project Name	NoS	NoB	CC	NoM
Gomoku	298	332	184	365
MTree	547	299	275	398
ITree	522	440	309	496
GSpan	507	246	165	465
CoverTree	580	268	243	421

4.3 Evaluation Metrics

For RQ1, we explore the accuracy of tests generated by GPT, assessing it from the following two perspectives:

- **Syntax Correctness.** Syntax correctness refers to whether the tests generated by GPT adhere to Java's syntax rules. For instance, when inputting the generated tests into a Java development IDE, they should not trigger any red error prompts at the very least. If the tests contain numerous syntax errors, human effort and time will be required to rectify these syntax issues.
- **JUnit Passing Rate.** We utilized JUnit, a widely-used Java unit testing framework, for our testing purposes. Similarly, when using GPT to generate tests, we require GPT to generate tests using the JUnit testing framework. Consequently, the generated tests need to adhere to the syntax requirements of the JUnit framework. Additionally, a crucial component of testing is assertions, which encompass the actual execution results of the test and the anticipated outcomes as perceived by the tester. After executing the tests using JUnit, the framework automatically provides the execution results of each test, along with the reasons for Errors or Failures.

For **RQ2**, we conduct comparisons using the following metrics:



```
    @Test
    public void addAndSearchTest() {
        // Test case 1: Add a single element and search for it
        Integer data1 = 10;
        mTree.add(data1);
 The method search(Integer, int) is undefined for the type MTree<Integer>arch(data1, 1);
        assertTrue(searchResult1.contains(data1));
```

Figure 2. An example of syntax error of the test.

- **Branch Coverage.** The coverage of branches achieved by the generated tests. We utilize Jacoco to calculate branch coverage.
- **Mutation Score.** The mutation score achieved by the generated tests. We employ Pitest to calculate the mutation score.

In **RQ3**, we compare GPT and human beings using the same evaluation metrics as in **RQ2**.

4.4 Experiment Evaluation

In this subsection, we provide the experimental results designed for each research question and offer answers to each research question.

4.4.1 **RQ1:** Can the tests generated by GPT be directly used for test execution?

TABLE II
CORRECTNESS OF THE GENERATED TESTS

Project Name	Syntax Correctness(%)	JUnit Passing Rate(%)
Gomoku	76.92	21.97
MTree	41.67	25.00
ITree	91.87	88.63
GSpan	90.15	78.57
CoverTree	31.92	26.89

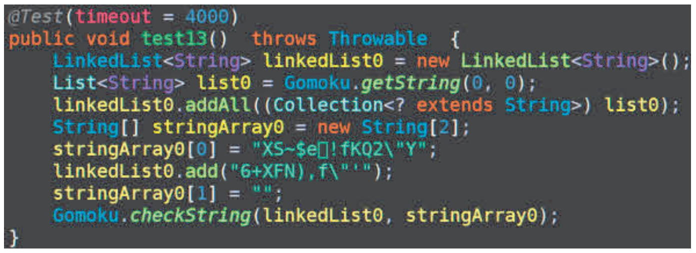
As shown in Table II, the correctness of tests generated using GPT for the five tested projects was assessed. It can be observed that the syntax correctness of the generated tests ranged from the highest rate of 90.15% to the lowest rate of 31.92%. We find that the primary reason for syntax errors was the invocation of methods that do not exist in the project under test, as shown in Figure 2. We speculate that this might be attributed to the influence of the training data on large language models during the test generation process, leading to the creation of tests resembling those from other similar projects.

In addition to syntax errors, concerning JUnit pass rates, the highest pass rate was 88.63%, while the lowest was 21.97%. The primary reason for test failures was the occurrence of assertion failures.

Answer to RQ1: The syntax correctness and JUnit pass rates of tests generated by GPT for the tested projects exhibited a wide range of variability. These rates tended to be lower when the project had complex call relationships.

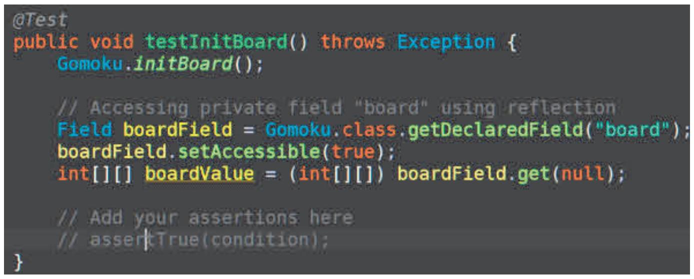
TABLE III
EFFECTIVENESS OF CHATGPT, EVOSUITE AND MANUAL TESTERS

Project Name	ChatGPT		Evosuite		Manual Testers	
	Branch Coverage(%)	Mutation Score	Branch Coverage(%)	Mutation Score	Branch Coverage(%)	Mutation Score
Gomoku	62.50	36.00	64.80	30.00	45.57	21.64
MTree	39.40	25.62	56.93	26.00	39.80	30.85
ITree	33.20	30.00	66.10	61.00	30.90	20.91
Gspan	46.80	30.00	60.95	42.00	47.20	30.40
CoverTree	53.10	19.00	67.5	52.00	65.83	27.75



```java
@Test(timeout = 4000)
public void test13()  throws Throwable  {
    LinkedList<String> linkedList0 = new LinkedList<String>();
    List<String> list0 = Gomoku.getString(0, 0);
    linkedList0.addAll((Collection<? extends String>) list0);
    String[] stringArray0 = new String[2];
    stringArray0[0] = "XS~$e[]!fKQ2\"Y";
    linkedList0.add("6+XFN),f\"'");
    stringArray0[1] = "";
    Gomoku.checkString(linkedList0, stringArray0);
}
```

Figure 3. An example of a poorly readable test generated by Evosuite



```java
@Test
public void testInitBoard() throws Exception {
    Gomoku.initBoard();

    // Accessing private field "board" using reflection
    Field boardField = Gomoku.class.getDeclaredField("board");
    boardField.setAccessible(true);
    int[][] boardValue = (int[][]) boardField.get(null);

    // Add your assertions here
    // assertTrue(condition);
}
```

Figure 4. An example of a highly readable test generated by ChatGPT

4.4.2 **RQ2:**How does the effectiveness of test generation by GPT compare to Evosuite?

In the first four columns of Table III, we recorded the test generation efficiency of ChatGPT and Evosuite and conducted a comparison. Evosuite, a relatively mature tool in the Java test generation domain, achieved code branch coverage of 64.80%, 39.80%, 66.10%, 47.20%, and 67.50% in the five tested projects and obtained mutation scores of 30.00, 26.00, 61.00, 42.00, and 52.00, respectively. According to the data in the table, we can observe that in the Gomoku, Mtree, and Gspan projects, ChatGPT's branch coverage is similar to Evosuite, while in the other two projects, ChatGPT's disadvantages are more pronounced. In the mutation analysis results, in Gomoku and MTree, ChatGPT's mutation scores closely approached or even exceeded Evosuite. However, in the other three projects, ChatGPT's mutation scores were only 30.00, 30.00, and 19.00, significantly lower than Evosuite.

At the same time, we observed that ChatGPT-generated tests exhibit higher readability. In the pursuit of achieving higher coverage, Evosuite-generated test data often include strings without real meaning, as illustrated in Figure 3. This can make it challenging for individuals to comprehend the test code when reading it. In contrast, ChatGPT-generated test data are typically more relevant to the tested methods, and they may include annotations, as shown in Figure 4, explaining the meaning of certain parts of the test code. These factors undoubtedly contribute to significantly improved readability.

Answer to RQ2: Compared to Evosuite, the tests generated by ChatGPT did not exhibit a significant advantage in mutation analysis. Among the test projects under test, ChatGPT was only able to closely approach or surpass Evosuite in two of the projects. In the other three projects, ChatGPT's disadvantages were more pronounced. In contrast, tests generated by ChatGPT are much more readable and highly understandable.

4.4.3 **RQ3:** Has the capability of GPT of test generation surpassed that of highly skilled human testing professionals?

Comparing the testing capabilities of ChatGPT with those of human testers is an intriguing topic. We conducted a comparison by using test cases designed by numerous real human testers and comparing them with the test cases generated by ChatGPT. The data is recorded in the fifth and sixth columns of Table III.

Based on the data in Table III, human testing is less efficient compared to Evosuite. However, when comparing human testing efficiency with ChatGPT, the difference is not particularly pronounced in Mtree, ITree, and Gspan. In Gomoku, ChatGPT has a significant advantage, while in CoverTree, human testing is more effective.

Answer to RQ3: Compared to manual testing, the difference in the effectiveness of tests generated by ChatGPT is not significantly large. We believe that to some extent, ChatGPT can replace a portion of manual testing work.

5. Threats to Validity

In the process of designing prompts, since we didn't focus extensively on the field of prompt engineering, we chose a prompt that has proven effective in practical applications. We used this prompt for generating tests in all the tested projects. We must acknowledge that if a better-designed prompt were available, the tests generated by ChatGPT might have been more effective than our current results. On the other hand, ChatGPT exhibits randomness in actual usage, meaning that the results can vary between different runs. To mitigate this randomness, we conducted 5 repetitions of experiments for each tested project to minimize the impact of variability.

Furthermore, when using Evosuite to generate tests, we employed default settings for parameters such as the number of processor cores and memory in the command. It is possible that altering these parameters and conducting multiple test generations could lead to better test outcomes with Evosuite.

In the comparison of test effectiveness, we relied on two metrics: branch coverage and mutation testing score. We did not compare the number of bugs identified by tests generated through different methods because bug identification involves a significant workload. Additionally, we believe that branch coverage and mutation testing scores, to some extent, already reflect test effectiveness. In future work, we will consider evaluating ChatGPT's ability to discover bugs.

6. Conclusion

In this paper, we conducted experiments to explore ChatGPT's capabilities in test generation. For comparison, we used the commonly used Java test generation tool, Evosuite, and manual testing. Based on the final experimental results, we found that there is still a significant gap between ChatGPT and Evosuite in terms of branch coverage and mutation test scores. However, the gap between ChatGPT and manual testing is not as significant, and in some tested projects, ChatGPT even has certain advantages. Furthermore, the tests generated by ChatGPT exhibit high readability, making them easy to understand. These tests also include annotations in the test code, facilitating the understanding of the purpose of code segments. We believe that ChatGPT is better suited to serve as a testing assistant, helping people understand the tested code and providing new testing ideas.

Acknowledgements

The authors would like to thank the anonymous reviewers for their insightful comments. This work was supported partially by the National Natural Science Foundation of China (61932012). This work was also supported by the U.S. National Science Foundation under Grant 2050869. This work was completed solely by the author at the University of Texas at Dallas Research Experience for Undergraduates (REU) under the supervision of Dr. W. Eric Wong and Mr. Zizhao Chen.

References

[1] H. Zhu, P. A. Hall, and J. H. May, "Software unit test coverage and adequacy," *Acm computing surveys (csur)*, vol. 29, no. 4, pp. 366–427, 1997.

[2] L. M. Chen, M.H. and W. Wong, "An empirical study of the correlation between code coverage and reliability estimation," in *Proceedings of the 3rd International Software Metrics Symposium*, pp. 133–141, 1996.

[3] "Junit." https://junit.org/, 2023.

[4] G. Fraser and A. Arcuri, "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, 2011.

[5] P. Braione, G. Denaro, A. Mattavelli, and M. Pezzè, "Combining symbolic execution and search-based testing for programs with complex heap inputs," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 90–101, 2017.

[6] Q. Zhang, C. Fang, T. Zhang, W. Sun, B. Yu, and Z. Chen, "Gamma: Revisiting template-based automated program repair via mask prediction," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, pp. 535–547, 2023.

[7] Q. Zhang, C. Fang, W. Sun, Y. Liu, T. He, X. Hao, and Z. Chen, "Boosting automated patch correctness prediction via pre-trained language model," *arXiv preprint arXiv:2301.12453*, 2023.

[8] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, "A survey of learning-based automated program repair," *arXiv preprint arXiv:2301.03270*, 2023.

[9] Q. Zhang, T. Zhang, J. Zhai, C. Fang, B. Yu, W. Sun, and Z. Chen, "A critical review of large language model on software engineering: An example from chatgpt and automated program repair," *arXiv preprint arXiv:2310.08879*, 2023.

[10] X. Li, W. Wong, R. Gao, L. Hu, and S. Hosono, "Genetic algorithm-based test generation for software product line with the integration of fault localization techniques," *Empirical Software Engineering*, vol. 23, pp. 1–51, 2018.

[11] E. Rudnick, J. Patel, G. Greenstein, and T. Niermann, "A genetic algorithm framework for test generation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 9, pp. 1034–1044, 1997.

[12] T. Bartz-Beielstein, J. Branke, J. Mehnen, and O. Mersmann, "Evolutionary algorithms," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 4, no. 3, pp. 178–195, 2014.

[13] D. Mayer, B. Kinghorn, and A. Archer, "Differential evolution – an easy and efficient evolutionary algorithm for model optimization," *Agricultural Systems*, vol. 83, no. 3, pp. 315–328, 2005.

[14] Q. Yang, J. Li, and D. Weiss, "A survey of coverage-based testing tools," in *Proceedings of the 2006 Interna-*

tional Workshop on Automation of Software Test, pp. 99–103, 2006.

[15] I. Boussaïd, P. Siarry, and M. Ahmed-Nacer, "A survey on search-based model-driven engineering," *Automated Software Engineering*, vol. 24, pp. 233–294, 2017.

[16] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson, "Model-based testing of object-oriented reactive systems with spec explorer," in *Formal Methods and Testing: An Outcome of the FORTEST Network, Revised Selected Papers*, pp. 39–76, 2008.

[17] A. Da Silva, "Model-driven engineering: A survey supported by the unified conceptual model," *Computer Languages, Systems Structures*, vol. 43, pp. 139–155, 2015.

[18] R. Baldoni, E. Coppa, D. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Computing Surveys (CSUR)*, vol. 51, no. 3, pp. 1–39, 2018.

[19] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium (USENIX Security 18)*, pp. 745–761, 2018.

[20] P. Godefroid, M. Levin, and D. Molnar, "Sage: Whitebox fuzzing for security testing: Sage has had a remarkable impact at microsoft," *Queue*, vol. 10, no. 1, pp. 20–27, 2012.

[21] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, vol. 8, pp. 209–224, 2008.

[22] S. Dalal, A. Jain, N. Karunanithi, J. Leaton, C. Lott, G. Patton, and B. Horowitz, "Model-based testing in practice," in *Proceedings of the 21st International Conference on Software Engineering*, pp. 285–294, 1999.

[23] F. Stevens, T. Courtney, S. Singh, A. Agbaria, J. Meyer, W. Sanders, and P. Pal, "Model-based validation of an intrusion-tolerant information system," in *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004*, pp. 184–194, IEEE, 2004.

[24] D. Jackson, I. Schechter, and H. Shlyahter, "Alcoa: The alloy constraint analyzer," in *Proceedings of the 22nd International Conference on Software Engineering*, pp. 730–733, 2000.

[25] C. Tuncali, G. Fainekos, H. Ito, and J. Kapinski, "Simulation-based adversarial test generation for autonomous vehicles with machine learning components," in *2018 IEEE Intelligent Vehicles Symposium (IV)*, pp. 1555–1562, IEEE, 2018.

[26] L. Dong, N. Yang, W. Wang, F. Wei, X. Liu, Y. Wang, J. Gao, M. Zhou, and H. Hon, "Unified language model pre-training for natural language understanding and generation," in *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[27] A. Rogers, O. Kovaleva, and A. Rumshisky, "A primer in bertology: What we know about how bert works," *Transactions of the Association for Computational Linguistics*, vol. 8, pp. 842–866, 2021.

[28] R. Dale, "Gpt-3: What's it good for?," *Natural Language Engineering*, vol. 27, no. 1, pp. 113–118, 2021.

[29] W. Yuan, Q. Zhang, T. He, C. Fang, N. Q. V. Hung, X. Hao, and H. Yin, "Circle: Continual repair across programming languages," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 678–690, 2022.

[30] Q. Zhang, C. Fang, B. Yu, W. Sun, T. Zhang, and Z. Chen, "Pre-trained model-based automated software vulnerability repair: How far are we?," *IEEE Transactions on Dependable and Secure Computing*, 2023.

[31] "Jacoco." https://www.jacoco.org/, 2023.

[32] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: a practical mutation testing tool for java," in *Proceedings of the 25th international symposium on software testing and analysis*, pp. 449–452, 2016.