International Journal of Advanced Smart Convergence Vol.13 No.4 145-154 (2024)
http://dx.doi.org/10.7236/IJASC.2024.13.4.145

IJASC 24-4-14

Utilizing Generative AI for Test Case Generation: Comparative Analysis and Guidelines

Woochang Shin

Professor, Dept. of Computer Science, Seokyeong University, Korea
wcshin@skuniv.ac.kr

Abstract

The advancement of generative AI technologies has significantly impacted various domains in software engineering, particularly in automating test case generation. As software systems become increasingly complex, manual test case creation faces limitations in terms of efficiency and coverage. This study analyzes the capabilities and limitations of major generative AI models—ChatGPT, Copilot, and Gemini—in generating software test cases. We focus on evaluating their performance in boundary value analysis, exception handling, and property-based testing. Using the ArrayUtils.indexOf() function from the Apache Commons Lang library as the test subject, we conducted experiments to compare the quality and effectiveness of the test cases generated by each model. Our findings indicate that while generative AI can efficiently produce a substantial number of high-quality test cases, there are instances of incorrect test cases and test codes. To address these issues, we propose guidelines for developers to enhance the reliability and consistency of test case generation using generative AI. Future research will explore the application of these models to more complex software systems and further methods to improve their test generation capabilities.

Keywords: *Generative AI, Software Testing, Test Case Generation, Property Based Testing, Boundary Value Analysis*

1. Introduction

In the field of software engineering, the necessity for applying generative AI has become increasingly prominent with the advancement of artificial intelligence technologies. Generative AI leverages natural language processing and deep learning techniques to enhance developer productivity across various domains such as code auto-generation, bug detection, and documentation [1]. These technologies contribute to increasing the efficiency of the software development process and providing innovative solutions.

The application of generative AI is also essential in software testing. As the complexity of software systems increases, manually writing test cases has reached its limitations [2]. By utilizing generative AI, it is possible to automatically generate test cases for various scenarios, thereby strengthening quality assurance. Existing automation tools, based on limited patterns or rules, fail to encompass all complex situations [3]. In contrast,

Manuscript Received: October. 11, 2024 / Revised: October. 18, 2024 / Accepted: October. 22, 2024
Corresponding Author: wcshin@skuniv.ac.kr
Tel: +82-2-940-7755
Professor, Dept. of Computer Science, Seokyeong University, Korea

generative AI can generate new test cases based on learned data, improving test coverage and depth. However, several issues exist in automatically generating test cases using generative AI, such as reliability, consistency, and difficulties in testing complex functionalities [4].

In this paper, we compare and analyze the test case generation capabilities of major generative AI models. We delve into boundary value analysis techniques, exception handling, and property-based testing. Through this, we evaluate how generative AI meets various testing requirements and identify its limitations. Additionally, we propose guidelines for generating consistent and effective test cases. This will provide practical assistance to developers in developing reliable software using generative AI.

The main contributions of this paper are as follows:

First, we systematically compare and analyze the test case generation capabilities of major generative AI models such as ChatGPT, Copilot, and Gemini. By doing this, we clearly identify the characteristics, strengths, and limitations of each model, evaluating the applicability of generative AI in the field of software testing.

Second, we experimentally verify the effectiveness of applying generative AI to various testing techniques, including boundary value analysis, exception handling, and property-based testing. By assessing the quality of the generated test cases through code coverage and mutation testing, we provide evidence on whether generative AI can be effectively utilized in actual testing scenarios.

Third, we present practical guidelines to address potential issues that may arise when generating test cases using generative AI. This contribution is expected to help developers effectively leverage generative AI to enhance the reliability and quality of software.

The rest of the paper is organized as follows. Section 2 reviews related works. Section 3 describes the generative AI's test case generation capabilities, including experimental scenarios and analyses. Section 4 presents a comparative analysis of the results and proposes guidelines. Finally, Section 5 summarizes and concludes the paper.

2. Related Works

Recent years have seen active research on the utilization of generative AI in the field of software testing. Generative AI possesses the ability to learn from large amounts of data and generate new data, and it is expected to bring revolutionary changes in the automatic generation of test cases [5].

Research on automatic generation of software test cases using generative AI can be broadly divided into black-box testing and white-box testing approaches. In black-box testing, test cases are generated based solely on inputs and outputs without knowledge of the internal structure of the system. Studies are underway to extract test cases from user requirements or specifications by leveraging the natural language processing capabilities of generative AI [6, 7]. For example, Utting et al. proposed a method to automatically generate test scenarios using generative techniques in model-based testing [8]. On the other hand, white-box testing considers the internal structure and logic of the code to generate test cases. In this area, research is being conducted to analyze source code using generative AI and explore paths where potential defects may occur [9, 10]. Specifically, Korel developed an automated test data generation technique for program path exploration [11].

Studies addressing the problems of automatic test case generation using generative AI are also being conducted. Wang et al. pointed out that due to biases in the training data, it is challenging to guarantee the quality and reliability of the test cases generated by generative AI models [4]. Zhang et al. mentioned the limitations of generative AI in sufficiently considering complex scenarios such as exception handling and

boundary conditions [12]. These issues arise because generative AI does not fully understand the deep semantics of the code or business logic.

Other related research includes hybrid approaches that combine generative AI with existing testing techniques. McMinn introduced a method to enhance the efficiency and reliability of test cases by combining search-based software testing with generative AI [13]. Just et al. developed Defects4J, targeting software containing real defects, to generate test cases and evaluate testing techniques [14]. They contributed to detecting and correcting actual defects using generative AI.

Based on these existing studies, this paper aims to analyze the current status and problems of automatic test case generation using the latest generative AI models and propose effective utilization strategies.

3. Generative AI's Test Case Generation Capabilities

This study assumes a specification-based unit test situation based on black-box testing. In this section, we compare and analyze the test case generation capabilities of representative generative AI models—ChatGPT, Copilot, and Gemini. The models of each generative AI are as follows:

- OpenAI's ChatGPT (GPT-4o with Canvas model)
- GitHub's Copilot (Based on the GPT-4o model, September 2024 version)
- Google's Gemini (Gemini 1.5 Pro-002 model)

As the test subject, we selected the *indexOf()* function of the *ArrayUtils* class included in the Apache Commons Lang library, and its specification is shown in Figure 1 [15].

public static int indexOf(boolean[] array, boolean valueToFind, int startIndex)
; Finds the index of the given value in the array starting at the given index. This method returns INDEX_NOT_FOUND (-1) for a null input array. A negative startIndex is treated as zero. A startIndex larger than the array length will return INDEX_NOT_FOUND (-1).

[Parameters]
- array - the array to search through for the object, may be null
- valueToFind - the value to find
- startIndex - the index to start searching at

[Returns]
The index of the value within the array, INDEX_NOT_FOUND (-1) if not found or null array input.

Figure 1. Specification of *ArrayUtils.indexOf()*

3.1 Test Case Generation According to Boundary Value Analysis

With the specification of the *indexOf()* function, the prompt content for test case generation provided to the generative AI is shown in Figure 2.

[Prompt-1]
- You are an experienced Java program tester. There is a class called ArrayUtils that has an indexOf() function. The requirements for the function are described below. We are trying to test this function using specification-based testing without source code. Generate code that can be tested with JUnit. When writing JUnit test cases, please use the ParameterizedTest feature.

Figure 2. Prompt-1 for Test Case Generation

When the specification of *ArrayUtils.indexOf()* and the Prompt-1 are input into the generative AI, test cases like those in Figure 3 were generated (Some code omitted).

[ChatGPT]

```
public class ArrayUtilsTest0 {
   public static Stream<Arguments>
      provideTestArguments() {
      return Stream.of(
  Arguments.of(null,5,0,-1),
  Arguments.of(new int[]{},5,0,-1),
  Arguments.of(new int[]{1,2,3,4,5},3,0, 2),
  Arguments.of(new int[]{1,2,3,4,5},3,3,-1),
  Arguments.of(new int[]{1,2,3,4,5},6,0,-1),
  Arguments.of(new int[]{1,2,3,4,5},2,-1, 1),
  Arguments.of(new int[]{1,2,3,4,5},1,10, -1),
  Arguments.of(new int[]{1,2,3,4,5},4,2, 3) );
   }
   @ParameterizedTest
   @MethodSource("provideTestArguments")
   public void testIndexOf(int[] array, int
     valueToFind, int startIndex, int expected){
      int result = ArrayUtils.indexOf(array,
                  valueToFind, startIndex);
       assertEquals(expected, result);
   }
}
```

[Copilot]

```
public class ArrayUtilsTest0 {
   @ParameterizedTest
   @MethodSource("indexOfTestCases")
   void testIndexOf(int[] array,
                  int valueToFind, int startIndex,
                  int expected) {
      assertEquals(expected,
        ArrayUtils.indexOf(array, valueToFind,
                              startIndex));
   }
   static Stream<Arguments>
      indexOfTestCases() {
      return Stream.of(
         Arguments.of(null, 1, 0, -1),
         Arguments.of(new int[]{1,2,3},2,-1,1),
         Arguments.of(new int[]{1,2,3},2,5,-1),
         Arguments.of(new int[]{1,2,3},2,0,1),
         Arguments.of(new int[]{1,2,3},4,0,-1)
      );
   }
}
```

[Gemini]

```
public class ArrayUtilsTest0 {
   ...
   private static Stream<Arguments>
indexOfTestCases() {
      return Stream.of(
 Arguments.of(null, 5, 0, -1),
 Arguments.of(new int[]{},5,0,-1),
 Arguments.of(new int[]{1,2,3,4,5},3,0,2),
 Arguments.of(new int[]{1,2,3,4,5},1,0,0),
 Arguments.of(new int[]{1,2,3,4,5},5,0,4),
 Arguments.of(new int[]{1,2,3,3,5},3,0,2),
 Arguments.of(new int[]{1,2,3,3,5},3,3,3),
 Arguments.of(new int[]{1,2,3,4,5},6,0,-1),
 Arguments.of(new int[]{1,2,3,4,5},3,-1,2),
 Arguments.of(new int[]{1,2,3,4,5},3,-2,2),
 Arguments.of(new int[]{1,2,3,4,5},3,5,-1),
 Arguments.of(new int[]{1,2,3,4,5},3,6,-1),
 Arguments.of(new int[]{1,2,3,4,5},4,2,3),
 Arguments.of(new int[]{1,2,3,4,5},4,3,3),
 Arguments.of(new int[]{1,2,3,4,5},5,4,4) ); }
}
```

Arguments.of(new int[]{1,2,3,4,5},4,4,3),

Figure 3. Example of test code generated by generative AI for Prompt-1

Despite entering the same request, the generative AI generated different test cases and test codes each time. Occasionally, incorrectly written test cases were included among the generated ones. For example, in Figure 3, in the 15th test case generated by Gemini, "Arguments.of(new int[]{1,2,3,4,5}, 4, 4, 3)", where *valueToFind* is 4 and *startIndex* is 4, the *indexOf()* function should return -1 (*INDEX_NOT_FOUND*), but it incorrectly considered the result as 3.

To diversify the test cases further, we explicitly added the application of the boundary value analysis technique to the existing Prompt-1 to create Prompt-2 and added default boundary values for each data type to create Prompt-3. The additional sentences in Prompt-2 and Prompt-3 are shown in Figure 4.

[Prompt-2]

- Please write test cases using combinations of boundary values of all inputs according to the Boundary Value Analysis technique.

[Prompt-3]

- Please write test cases using combinations of boundary values of all inputs according to the Boundary Value Analysis technique.
- When setting the boundary values for the function inputs in the test cases, please add the following boundary values according to the input's data type:
  - int, short, byte, long type arguments: -1, 0, 1
  - String arguments: NULL, empty string, single-character string, multi-character string
  - Array arguments: NULL, empty array, single-element array, multiple-element array

Figure 4. Additional Sentences in Prompts for Boundary Value Analysis

Each generative AI was prompted with Prompt-1, Prompt-2, and Prompt-3 to generate test cases, and each prompt was repeated three times. To minimize the influence of previous test case generation results, a new conversation window was opened each time the prompt was entered. The number of test cases generated by repeated experiments and the number of incorrectly generated test cases are shown in Table 1.

Table 1. Number of Generated Test Cases and Errors

	Prompt #	Try -1		Try -2		Try -3		Average	
		# of test-case	# of wrong test-case	# of test-case	# of wrong test-case	# of test-case	# of wrong test-case	# of test-case	# of wrong test-case
ChatGPT	Prompt-1	**8**	0	**13**	0	**8**	0	**9.67**	0.00
	Prompt-2	**12**	**1**	**13**	0	**16**	0	**13.67**	**0.33**
	Prompt-3	**19**	0	**15**	0	**20**	0	**18.00**	0.00
Copilot	Prompt-1	**7**	0	**5**	0	**7**	0	**6.33**	0.00
	Prompt-2	**13**	0	**21**	0	**13**	0	**15.67**	0.00
	Prompt-3	**20**	0	**15**	0	**11**	0	**15.33**	0.00
Gemini	Prompt-1	**16**	**1**	**12**	0	**12**	0	**13.33**	**0.33**
	Prompt-2	**21**	0	**16**	0	**12**	0	**16.33**	0.00
	Prompt-3	**36**	0	**30**	**3**	**28**	0	**31.33**	**1.00**

The results showed that the number of test cases generated generally increased as more test case-related requests were added to the prompt content input into the generative AI. Assuming that the *ArrayUtils.indexOf()* function code has no errors, most of the test cases generated by the generative AI were correct, but occasionally some incorrect test cases were generated. In the case of ChatGPT, 1 out of 39 generated test cases was incorrectly generated, and Gemini generated 4 incorrect test cases out of 73.

3.2 Test Case Generation with Exception Objects

To evaluate the ability to generate test cases for functions that throw exceptions, we modified the specification of *ArrayUtils.indexOf()* so that it throws a *RuntimeException("Invalid argument")* when the *array* argument is null or the *startIndex* argument is negative, and we created Prompt-4 accordingly. When Prompt-4 was input, the generative AI generated test cases for the modified *ArrayUtils.indexOf()* and also generated test codes handling the *Exception* object. The experimental results are shown in Table 2.

Table 2. Exception Handling Test Case Generation Results

	Prompt #	Try-1		Try-2		Try-3		Average	
		# of test-case	# of wrong test-case	# of test-case	# of wrong test-case	# of test-case	# of wrong test-case	# of test-case	# of wrong test-case
ChatGPT	Prompt-4	**8**	**2**	**13**	0	**12**	0	**11**	0.67
Copilot	Prompt-4	**9**	0	**13**	0	**15**	0	**12.33**	0
Gemini	Prompt-4	**155**	0	**96**	0	**18**	**2**	**89.67**	0.67

The generative AI wrote test cases for functions that throw exceptions well. However, some errors were found in ChatGPT and Gemini. The two errors that occurred in ChatGPT's Try-1 were not due to incorrectly generated test cases but errors that occurred in the test code executing the test cases. In Gemini's Try-3, the two errors were cases where *INDEX_NOT_FOUND* was incorrectly used as the expected result instead of the *RuntimeException* object.

Interestingly, in Gemini's Try-1 and Try-2, a significantly large number of test cases were generated. In these cases, Gemini did not simply generate an array of test cases but generated test codes that dynamically create test cases by defining arrays of boundary values for each input (*array, valueToFind, startIndex*) and combining elements of these arrays.

3.3 Property-Based Testing Technique

Property-based testing is a method that focuses on testing specific properties or characteristics of software, aiming to verify not only functional requirements but also non-functional attributes such as performance, security, and reliability. This method dynamically generates many test cases based on properties to confirm that the system behaves as expected under various inputs and conditions, effectively identifying potential defects or areas for improvement. However, writing property-based tests is significantly more challenging than example-based tests.

In this study, we experimented to see if generative AI can effectively write property-based test codes. For this purpose, we modified Prompt-1 by replacing the requirement for *JUnit* with the property-based testing framework *jqwik* [16]. Prompt-5 for property-based test code generation is shown in Figure 5.

[Prompt-5]
- *(same as Prompt-1)*
- Please generate code that can be tested using jqwik, a property-based testing framework.

Figure 5. Prompt-5 for Property-Based Testing

The generative AI received Prompt-5 and, understanding the characteristics of the function, generated functions that dynamically test each property. The generated test functions are shown in Figure 6.

[ChatGPT]	[Copilot]	[Gemini]
• testNullArrayReturnsIndexNotFound • testNegativeStartIndexIsTreatedAsZero • testStartIndexGreaterThanArrayLength ReturnsIndexNotFound • testValueFoundInArray • testValueNotFoundInArray • **testIndexOfWithValidStartIndex**	• testNullArray • testNegativeStartIndex • testStartIndexLargerThanArrayLength • testValuePresentInArray • **testValueNotPresentInArray**	• nullArray • emptyArray • negativeStartIndex • startIndexGreaterThanLength • **valueFound** • valueNotFound • **startIndexWithinRangeValueFound**

Figure 6. Property-Based Test Functions Generated by Generative AI

Property-based test functions generate 1,000 test cases with random combinations of input values and, after filtering out those that do not meet preconditions, proceed with testing using test cases that meet the conditions. That is, not all of the 1,000 generated test cases are used for function testing.

In most cases, testing proceeded smoothly, and hundreds of test cases that met the preconditions passed for each property-based test function. However, some issues were found in the experimental results. First, in ChatGPT's *testIndexOfWithValidStartIndex()* function, when 1,000 test cases were generated randomly, more than 870 did not meet the preconditions, causing an error in *jqwik*. This can be resolved by adjusting the generation range of the parameters used in the combinations. Second, in Copilot's *testValueNotPresentInArray()* function, test cases that did not meet the preconditions among the randomly generated test cases should be filtered out, but they were incorrectly considered as test failures. This is a clear error in Copilot's generation of the property-based test function. Lastly, in Gemini's *valueFound()* and *startIndexWithinRangeValueFound()* functions, when the size of the randomly generated *array* argument was smaller than the *startIndex*, it was not properly filtered, causing an *ArrayIndexOutOfBoundsException*. This is also an error in Gemini's generation of the property-based test function.

4. Comparative Analysis

Generative AIs can generate example-based test cases and property-based test code based on function specifications for unit testing, and have shown that the generated test cases and test code mostly work correctly. However, the fact that test cases function properly does not guarantee the quality of the test cases.

In this paper, we used two methods to verify the quality of the test cases created by generative AIs. One is to measure the code coverage of the *ArrayUtils.indexOf()* function for each test suite, and the other is to perform mutation testing by deliberately inserting bugs into the *ArrayUtils.indexOf()* function. The actual code of the *ArrayUtils.indexOf()* function is shown in Figure 7.

```
01:  public class ArrayUtils {
02:      private static final int INDEX_NOT_FOUND = -1;
03:      public static int indexOf(final int[] array, final int valueToFind, int startIndex) {
04:          if (array == null)
05:              return INDEX_NOT_FOUND;
06:          if (startIndex < 0)
07:              startIndex = 0;
08:          for (int i = startIndex; i < array.length; i++ )
09:              if (valueToFind == array[i]) return i;
10:          return INDEX_NOT_FOUND;
11:      }
12:  }
```

Figure 7. Source Code of ArrayUtils.indexOf()

4.1 Code Coverage Measurement

Code coverage is a software testing metric that measures how much of the source code is executed when a specific test suite runs. It helps identify untested parts of the code, discover potential bugs, and improve software quality. To measure code coverage, we used the open-source library *JaCoCo* [17]. *JaCoCo* executes the test suite and provides a coverage report showing executed instructions, missed instructions, and missed branches.

A total of 27 test suites were written, with each of the three generative AI models generating test suites three times (Try-1, Try-2, Try-3) for the three prompts (Prompt-1, Prompt-2, Prompt-3). After measuring the code coverage of the written test suites, all test suites—including Copilot/Prompt-1/Try-2, which had the fewest test cases—showed a 100% coverage rate. This is considered to be because the code of the *ArrayUtils.indexOf()* function does not have complex branching, and the loop statements are written in a simple form.

4.2 Mutation Testing

Mutation testing is a software testing technique where small changes or errors are intentionally injected into the source code to evaluate whether existing test cases can detect these changes or fail, thereby measuring the effectiveness and completeness of the tests. For testing, we wrote four mutated versions by modifying parts of the code of *ArrayUtils.indexOf()*, and the changes in each version are shown in Table 3.

Table 3. Code Modifications for Mutation Testing

Version	Line #	Original code	Modified code
(a)	4	if (array == null)	if (array != null)
(b)	6	if (startIndex < 0)	if (true || startIndex)

(c)	7	startIndex = 0	startIndex = 1
(d)	8	i++	i--

The test results for the four mutated codes showed that mutation versions (a), (b), and (d) all failed in all 27 test suites shown in Table 3. This indicates that the test suites effectively detected the intentionally inserted bugs.

In the case of mutation version (c), many test suites failed to detect the bug. Excluding the incorrectly generated test cases, after executing a total of 27 test suites, 13 test suites did not detect the bug and passed the tests. Unlike example-based testing, the test codes generated in Prompt-4, which perform property-based testing, all detected the bug in the mutated code.

Table 4. Test Suite Execution Results for Mutated Code (c)

	Prompt #	Try-1	Try-2	Try-3
ChatGPT	Prompt-1	**P**	**P**	**P**
	Prompt-2	**P**	**P**	F
	Prompt-3	**P**	**P**	**P**
Copilot	Prompt-1	**P**	**P**	**P**
	Prompt-2	F	F	F
	Prompt-3	F	F	**P**
Gemini	Prompt-1	**P**	F	F
	Prompt-2	F	F	F
	Prompt-3	F	F	F

*** P: pass, F: Fail**

4.3 Analysis Results and Guidelines

After analyzing the test case generation capabilities of major generative AIs, we found that they efficiently produce test cases at a considerable level. While there are still instances where incorrect test cases or test code are generated, the proportion is minimal (1% to 6%) and is deemed to be within an acceptable range.

The analysis results are as follows:

- The number of generated test cases varies depending on the content of the prompt input.
  ; Specifying techniques like boundary value analysis resulted in generating more test cases than general test case generation requests.
- The error rate of test cases generated by generative AI (example-based) was about 1%~6%.
  ; Since most of the errors can be easily corrected, the usefulness of the generated test cases is sufficient.
- The quality of the test cases was found to be excellent.
  ; Although the function used in the experiment is simple, all generated test suites measured a code coverage rate of 100%. In mutation testing, the detection rate was low (51.9%) in example-based testing but was 100% in property-based testing.

Based on the analysis results, we propose the following guidelines for test case generation using generative AI:

- **Guideline 1:** Explicitly include specific testing techniques in the prompt content when generating test cases.
  ; Request the generation of test cases using combinations of boundary values for each argument by specifying techniques like boundary value analysis.
- **Guideline 2:** Consider that the generated test cases and test codes may contain errors.

; If an error occurs after executing the test cases, review whether it is due to an error in the test case itself.

- **Guideline 3:** Combine example-based testing with property-based testing.

; For subtle bugs, example-based testing may not be sufficient. Specify property-based test code generation frameworks and request test code generation based on them.

5. Conclusion

In this study, we conducted an in-depth analysis of the current state and challenges of automatic software test case generation using generative AI. By comparing the test case generation capabilities of major generative AI models such as ChatGPT, Copilot, and Gemini, we confirmed that generative AI can efficiently generate a considerable number of test cases with high quality. In particular, test case generation that considers boundary value analysis techniques and exception handling showed high accuracy and comprehensiveness.

However, we found that some incorrect test cases or test codes can be generated, and that techniques such as property-based testing need to be used in parallel to fully detect subtle bugs. To address these issues, this paper presented specific guidelines to help developers effectively utilize generative AI.

These guidelines are expected to enhance the utilization of generative AI and contribute to improving the reliability and quality of software. Future research will involve an in-depth analysis of generative AI's test case generation capabilities for more complex software systems and seek methods to improve reliability and consistency.

Acknowledgement

This Research was supported by Seokyeong University in 2023.

References

[1] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A Survey of Machine Learning for Big Code and Naturalness," *ACM Computing Surveys*, vol. 51, no. 4, pp. 1–37, Jul. 2018. DOI: http://doi.org/10.1145/3212695

[2] G. Fraser and A. Arcuri, "Whole Test Suite Generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, Feb. 2013. DOI: https://doi.org/10.1109/TSE.2012.14

[3] C. C. Michael, G. McGraw, and M. A. Schatz, "Generating Software Test Data by Evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 12, pp. 1085–1110, Aug. 2002. DOI: http://doi.org/10.1109/32.988709

[4] S. Wang, T. Liu, and L. Tan, "Automatically Learning Semantic Features for Defect Prediction," *in Proceedings of the 38th International Conference on Software Engineering*, May 2016, pp. 297–308. DOI: http://doi.org/10.1145/2884781.2884804

[5] M. Islam, F. Khan, S. Alam, and M. Hasan, "Artificial Intelligence in Software Testing: A Systematic Review," *IEEE Tencon 2023*. DOI: http://doi.org/10.1109/TENCON58879.2023.10322349

[6] S. Ali, L. C. Briand, H. Hemmati, R. K. Panesar-Walawege, "A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation," *IEEE Transactions on Software Engineering*, vol. 36, no. 6, pp. 742-762, Dec. 2010, DOI: http://doi.org/10.1109/TSE.2009.52

[7] G. Fraser, A. Arcuri, "Whole Test Suite Generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276-291, 2013. DOI: http://doi.org/10.1109/TSE.2012.14

[8] M. Utting, A. Pretschner, B. Legeard, "A Taxonomy of Model-Based Testing Approaches," *Software: Testing, Verification and Reliability*, vol. 22, no. 5, pp. 297-312, 2012. DOI: http://doi.org/10.1002/stvr.456

[9] R. G. Hamlet, "Random Testing," *Encyclopedia of Software Engineering*, Wiley, 2002. DOI: https://doi.org/10.1002/0471028959.sof293

[10] P. Tonella, "Evolutionary Testing of Classes," *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 119-128, 2004. DOI: http://doi.org/10.1145/1007512.1007528
[11] B. Korel, "Automated Software Test Data Generation," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 870-879, 1990. DOI: https://doi.org/10.1109/32.57624
[12] Z. Zhang, X. Xie, H. Wang, and T. Xie, "Automated Test Input Generation for Android: Are We There Yet?" *in Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering*, 2015, pp. 429–440. DOI: https://doi.org/10.48550/arXiv.1503.07217
[13] P. McMinn, "Search-Based Software Test Data Generation: A Survey," *Software: Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004. DOI: http://doi.org/10.1002/stvr.294
[14] J. Just, D. Jalali, and M. D. Ernst, "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs," *in Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 437–440. DOI: http://doi.org/10.1145/2610384.2628055
[15] The Apache Software Foundation, Apache Common Lang 3.11 API, *https://commons.apache.org/proper/commons-lang/javadocs/api-release/index.html*
[16] Jqwik-Team, The jqwik User Guide 1.9.1, *https://jqwik.net/docs/current/user-guide.html*
[17] EclEmma Team, JaCoCo Java Code Coverage Library, *https://www.eclemma.org/jacoco/index.html*