Robustness-Enhanced Assertion Generation Method Based on Code Mutation and Attack Defense

Min Li, Shizhan Chen, Guodong Fan, Lu Zhang, Hongyue Wu(✉), Xiao Xue, and Zhiyong Feng

Tianjin University, Tianjin, China
{plainminbao,shizhan,guodongfan,zlu_4435,jzxuexiao,
hongyue.wu,zyfeng}@tju.edu.cn

Abstract. Writing high-quality unit tests plays a crucial role in discovering and diagnosing early-stage errors and preventing their further propagation throughout the development cycle. However, the low readability of existing automated test case tools hinders developers from directly using them. In addition, current approaches exhibit sensitivity to individual words in the input code, often producing completely different results for minor changes in the input code. To tackle these problems, we propose AssertGen, a powerful Java assertion generation model that maintains consistent output for minor variations in code snippets. Inspired by software mutation testing, we propose 11 heuristic strategies for code mutation, aiming to generate variant code that is human-readable but misleading to the model, by making minor changes to code text or structural information. Then, we use the variant code to attack the model to test the model's robustness. We observe that the variant based on variable names (VM), the mutation based on method names (FM), and the mutation method False_Control_Flow, which adds additional control flow, have the greatest impact on the quality of generated assertions by the model. To enhance the robustness of AssertGen, we use multiple mutations to expand the original dataset, allowing the model to learn how to counter the instability caused by mutations during the training process. Experiment results show our assertion generation model achieves a BLEU score of 60.08 and a perfect prediction rate of 47.91%, surpassing previous work significantly.

Keywords: Unit Tests · Model Robustness · Code Mutation · Attack Defense

1 Introduction

Unit testing is a testing method in software development that involves checking and validating the smallest testable software units that are isolated from other parts of the program [1–4]. While unit testing can quickly identify issues within

H. Gao et al. (Eds.): CollaborateCom 2023, LNICST 562, pp. 281–300, 2024.
https://doi.org/10.1007/978-3-031-54528-3_16

the tested modules, previous work [5] has shown that as the scale and complexity of software products increase, merging unit tests incurs significant costs in terms of traceability [6]. This reduces the feasibility of adding unit tests to software, making legacy code difficult to maintain and evolve. Adding unit tests to previously written code has become a challenge in the software development lifecycle. The software development research community has conducted extensive research [7,8] to assist developers by generating automated testing methods. However, some work points out limitations of current automated test generation tools and questions their ability to generate high-quality unit tests [9,10].

The application of pre-trained models in code-related tasks includes code autocompletion [11,12], code defect repair [13–15], code refactoring [16], code comment generation [17,18], and more. Among them, code autocompletion is one of the most common tasks. Currently, some large pre-trained models have made significant breakthroughs in code-related tasks, such as Codet5 [19], CodeBERT [21], CodeGPT [20], etc. These models have achieved good performance in code completion, code defect detection, and code search tasks. Among them, the CodeT5 model launched by the Salesforce research team has been proven to excel in various works related to code-related tasks [22,23].

While pre-trained models have greatly enhanced the software development process, according to our observations, these models are not omnipotent or perfect. Sometimes, they can produce sequences that are highly problematic or even incorrect. We do not expect the models to generate 100% accurate sentences, but such suboptimal or erroneous results are not desirable. They diminish the performance and effectiveness of code generation by the models, indicating a lack of robustness. So far, research on the issues with pre-trained models has primarily focused on their widespread use in various domains, model security, and reducing training time [24–28]. The problem of inconsistency in model-generated outputs has only been recognized in a few works [29], and these works have merely designed a pipeline for selecting the optimal output, rather than directly enhancing the consistency of model outputs. This issue has not been addressed at all in the context of automated test case generation.

In Fig. 1(left), we can see that after providing the test method code before the tested method and the assertion, the T5-based model generates 'perfect' assertion statements. However, when the variable name 'list' is changed to 'l', the model outputs different assertion statements. The assertions generated on the left are completely correct and comprehensive, while the assertions obtained from fine-tuned input have some errors because 'list.size()' should clearly be 3. This example illustrates the issue of input sensitivity in pre-trained models leading to inconsistent outputs. Figure 1(right) is an incorrect generation that cannot run correctly. In summary, the model's inconsistency in outputting results is due to its sensitivity to certain flags in the input, indicating that when the model learns on a large-scale dataset, it overly focuses on variable names, leading to a lack of robustness. Unlike traditional code generation, during the process of generating test cases, only the code's flow needs to be known, without requiring details such as variable names. The goal is to verify the correctness of program execution results, and the model only needs to generate a set of test cases that

cover all program branches. In other words, the variable name being 'list' or 'l' should not affect the generated assertion content; the same assertions should be generated. We can unify the variable names in the tested function to make the model more focused on learning the program flow.

Fig. 1. Inconsistency in the output of pre-trained models with slightly modified inputs.

Inspired by the mutation testing technique [30], we propose 11 code mutation strategies based on code text information and structural information. Then, using mutations of different tested functions to attack the model. We explore the robustness of the model under different variant influences and quantitatively evaluate the quality of the generated assertions, quantifying the model's sensitivity to the training data. Finally, the enhanced data is used to augment the original data for fine-tuning large pre-trained models, enabling the model to learn how to counter uncertainties and enhance its sensitivity resistance during the training process.

In summary, this paper offers the following contributions:

1. We propose 11 code mutation rules, where four code mutation strategies are introduced starting from the perspective of variable names in obfuscated code. Additionally, we propose 7 control flow mutations from the angle of code control flow.
2. We employ code attack models generated from the tested function under different mutation rules. Through this process, we identify three mutations that have the most significant impact on the quality of assertions generated by pre-trained models. Furthermore, we use multiple mutations to augment the training data, enabling the model to learn how to counteract unstable factors during the training process.
3. Experiment results show our assertion generation model achieves a BLEU score of 60.08 and a perfect prediction rate of 47.91%.
4. We provide an open source implementation of AssertGen at https://github.com/Bossism/test_gen.

Paper Organization. Section 1 describes the motivation of the problem. Section 2 presents the problem background and limitations of existing methods. Section 3 provides a detailed description of our method, AssertGen. Sections 4 introduce the experimental setup and results. Section 5 discusses threats to validity. Section 6 concludes the paper.

2 Background and Related Work

Traditional unit test generation techniques utilize search-based [31–33], constraint-based [34–36], or random-based strategies [37,38] to generate a set of unit tests with the main goal of maximizing the coverage of the tested software. Although these automatically generated tests achieve reasonable coverage, they often lack readability and meaningfulness compared to manually written tests, which is why developers are reluctant to adopt them directly in practice [9].

The two most widely used tools in this field are Randoop [8] and EvoSuite [7]. Previous work [39] compared the effectiveness of Randoop, EvoSuite, and Agitar, three automatic unit test generation tools, in detecting 357 real defects in the Defects4J dataset. The results showed that while the generated test suites overall detected 55.7% of the defects, only 19.9% of individual test suites detected defects. There is still significant room for improvement in automatically generated test cases for defect detection.

In recent years, there have been some approaches that leverage knowledge from the NLP field to address automated test case generation. TESTNMT [40] demonstrated the feasibility of using neural machine translation models for automated testing. It used a modified neural machine translation model to generate test cases for Java methods. However, the effect of generating test cases across projects is not satisfactory. ATLAS [5] also applies DL methods to the generation of automated test cases. Unlike TESTNMT, it focuses on generating meaningful assertion statements rather than the entire test. It takes the tested function and the test method after removing the assertion as input to the model and outputs predicted assertion statements. However, to simplify the problem, this model can only generate a single assertion statement. Michele Tufano [41] achieved good results by pre-training a BART-based model called ATHENATEST on a large-scale English and source code corpus, followed by fine-tuning on an assertion generation dataset. White R [42] proposed the ReAssert model, which is based on the Reformer model, to address the limitation of ATLAS in generating only a single assertion statement. ReAssert can generate multiple consecutive assertion statements, but the evaluation metrics such as BLEU and ROUGE indicate suboptimal performance. CONTEST [43], for the first time, leverages the structural information of the code and creatively utilizes the functions called in the test method and focal method to generate different content AST trees, which are combined as inputs to the model. However, a drawback is that the model serializes the entire combined AST tree and inputs it into a Transformer-based model. This may lead to truncation of excessively long sequences, resulting in information loss, incomplete context, and high computational complexity. TOGA [49] is a neural methodology based on a unified Transformer architecture that employs

a focus-based approach for context-based inference of anomalies and assertion test oracles. Its primary emphasis lies in identifying exceptions within methods. However, our model places a greater emphasis on generating assertions of high quality and strong consistency.

While traditional testing tools such as Evosuit have achieved commendable coverage, they suffer from limited readability and demand a comprehensive code execution environment. Furthermore, source code analysis necessitates compilation. In contrast, our methodology eliminates the requirement for executable code environments. It merely entails declaring the targeted functions and variables within the testing functions, rendering it markedly pragmatic. This approach effectively assists developers in composing test cases in real-time. In contrast to models based on RNN that are plagued by gradient vanishing, and LSTM models that suffer from extended training times and gradient explosion, AssertGen employs relative positional encoding to ensure the model's stability when processing lengthy sequences, thereby mitigating the risks of gradient explosion or vanishing. Additionally, we incorporate layer normalization during the training process to enhance the model's robustness. In comparison to ATLAS, our approach generates multiple consecutive assertion statements, as opposed to singular assertion statements, rendering it more versatile and impactful.

3 Approach

This chapter mainly describes the heuristic rules for generating different mutations and the use of these mutations to attack the model, exploring the parts of the tested code that have a significant impact on the model's robustness. The selected mutations are used to augment the training dataset, enabling the model to learn how to handle the instability caused by these mutations during training.

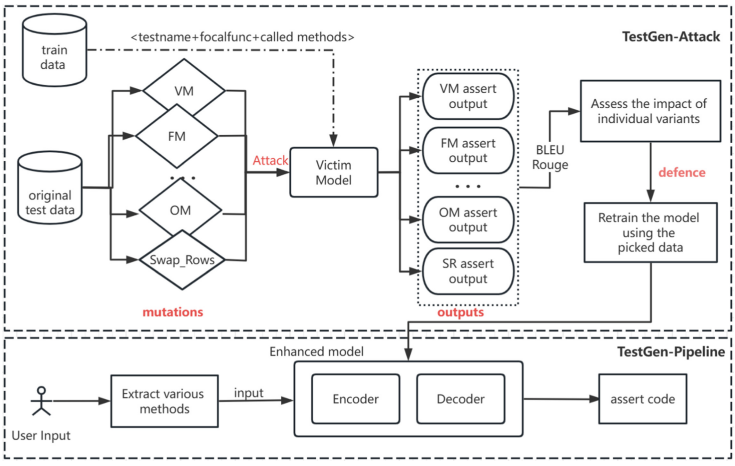

Fig. 2. Structure diagram of AssertGen.

3.1 Overview

AssertGen is an assertion generation model based on the T5 architecture. The structure diagram is shown in Fig. 2. For a given method under test, in Step 1, we apply heuristic mutation rules to generate mutations at both the code text and structural levels. We first use these mutations individually to attack the model and investigate which mutations have the greatest impact on the quality of generated assertions. These mutated mutations are then processed in Step 2 using BPE tokenization to obtain subword tokenization. Position vectors are added to the embedded input code, and the new embeddings go through N encoder blocks to obtain hidden layer outputs. The decoder maps the hidden layer outputs from the encoder to code sequences and consists of N decoder blocks. The generated code sequences are then passed through a linear layer with softmax activation to obtain a probability distribution over the vocabulary. AssertGen uses beam search on the probability distribution to generate the final candidate words as the predicted results. The following sections will provide detailed explanations of the mutation rules and the attack and adversarial steps.

3.2 Mutation Rules

The quality of model results is greatly influenced by the quality of the training data. Code blocks with the same structural and functional information may have different variable names across different projects. This "uncertainty" poses challenges for the model to learn the semantic information of code blocks with the same functionality. To address this issue, we propose heuristic mutation rules for variable names and expressions, incorporating control flow information for variable names.

VM(Variable Mutation). The VM technique replaces variable names with special characters. It's important to note that variable names referred to here are the ones defined by developers, such as *res* in the statement *boolean res = a >b;*. We use $\langle extra\_id\_ID \rangle$ to represent *res*. The reason for this approach is that a method may contain multiple variables, and we replace instances of the same variable with the same identifier $\langle extra\_id\_ID \rangle$. Here, *ID* represents the order of the variable's first appearance in the method.

PM(Param Mutation). The VM mutation rule is designed to handle variable names defined by developers. However, parameter names in method signatures are also commonly used custom variable names. Therefore, we have devised the PM mutation rule. The PM rule primarily focuses on parameter names within methods, replacing them with the identifier *param_id*.

FM(Function Name Mutation). Previous research [48] has found that when using natural language descriptions and method signatures as input to generate code examples using pre-trained models, the method name in the input has a significant impact on the model's output. Replacing the method name in the method signature leads to the generation of different and partially incorrect

code by the model. This finding demonstrates that even minor modifications to the method name can deceive the model and result in erroneous outputs.

In our approach, we modify the method names in the tested code. Prior to this, we trained a Word2Vec model using all the method names in the dataset. For each tested method, we split its method name into individual words using camel case notation. For each word, we select the top k similar words as candidates and introduce three additional candidates for each word: randomly swapping two characters, randomly deleting one character, and replacing a character with a similar one. This process results in combinations of candidate words for each original word. By replacing the method name in the original method with these combinations of candidates, we input the modified method into the model to obtain the output. We calculate the BLEU score between the current output and the ground truth and select the method name with the lowest BLEU score. This means that we have identified the method name that has the greatest impact on the model. By using this method name, we launch an attack on the model to maximize its performance degradation.

OM(Operator Mutation). OM is a mutation rule specifically designed for assignment expressions. We have observed that the presence of compound assignment operators in expressions can affect the model's judgment of the output. To address this, we decompose expressions containing assignment operators into simpler patterns. For example, the expression $a \mathrel{+}= b$, we represent it as $a = a + b$. By using simpler expressions, we can increase the frequency of variable occurrences and enhance the model's understanding.

In the mutation of structural information in code, the objective is to obtain mutated code that does not change the semantics and can execute correctly by introducing specific control flows or altering the code's execution flow. Different mutations of structural information are used to attack the model and observe its recognition capability regarding specific control flows.

Add_Print. We randomly insert print statements into method code, including both developer-defined variable names and parameter names from the method's parameter list. Although this type of mutation may not have a direct purpose, it simulates the coding habits of developers during the code-writing process. Developers often like to print important variables for code debugging. By simulating this process, we observe whether the randomly inserted print statements affect the model's learning of the source code.

False_Control_Flow. False_Control_Flow introduces random control flow statements such as if(false) into method code. Although these statements will never be executed, our aim is to observe the impact of randomly added control flow information on the model. By observing the model's performance in generating high-quality assertion statements under the attack of randomly added control flow, we can determine whether the model can still generate accurate assertions.

In addition to simulating the impact of programming habits in actual development, inspired by data augmentation techniques, we also made some "semantic consistency" changes to the tested code. These changes preserve syntactic

naturalness and semantic invariance from a human perspective but introduce noticeable differences for pre-trained models. By attacking the model with these semantically equivalent but structurally different mutations, we can observe their impact on the model's robustness.

While2For & For2While. For the While2For variant, we employed the SPAT [45] tool to identify the while statement blocks in the tested method and applied specific rules to transform them into syntactically equivalent for loops that are compilable. Conversely, for the For2While variant, we performed the opposite transformation by converting for loops into while loops using similar rules.

Switch2If. Similar to the previous two mutations, the Switch2If variant involves transforming switch statement blocks in the code into *if-elif-...-else* statements. This allows us to observe the impact of different code structures on the model.

Swap_If_Else. In this set of mutations, the if and else parts are swapped by exchanging the code within them while negating the if condition. This variant aims to investigate whether the model tends to excessively focus on a particular branch.

Swap_Rows. If there are no shared variables between two lines of code, the Swap_Rows variant swaps these two lines. Similar to the previous mutations, using the Swap_Rows variant to attack the model allows us to determine whether the model is sensitive to the order of certain lines of code.

3.3 Model

Our AssertGen model is based on the T5 model [19]. As shown in Fig. 3, we briefly introduce the model structure.

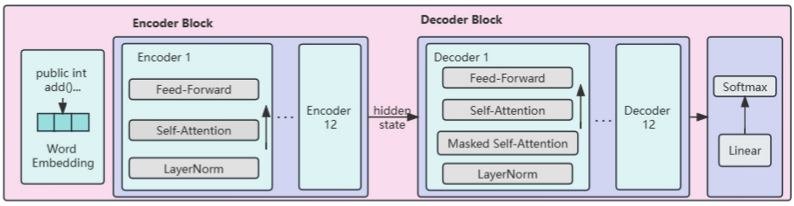

Fig. 3. Structural diagram of the AssertGen model.

Encoder Block. The encoder consists of N encoder blocks, which aim to transform the input code sequence into hidden states. Here, N is 12. We use relative positional encoding to effectively represent the relative positional information of each token in the input sequence, employing self-attention mechanisms with relative positional encoding.

We utilize a multi-head attention mechanism to force the model to jointly attend to different subspaces of code token representations from different positions in the input sequence. Specifically, we define multiple sets of queries (Q),

keys (K), and values (V) that focus on different contexts. For an input matrix X, each set of Q, K, and V produces an output matrix Z_i. We concatenate the Z_i matrices generated by multiple attention heads and feed them into a fully connected layer.

$$MultiHead(Q, K, V) = Concat(head_1, head_2, ..., head_n)W \tag{1}$$

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V) \tag{2}$$

During the calculation of attention, we perform padding and masking operations on the remaining sentences based on the maximum sentence length within each batch.

Each encoder block consists of two parts: a multi-head self-attention mechanism with relative positional encoding and a feed-forward neural network. Both parts include residual connections and layer normalization.

$$X_{attention} = LayerNorm(X + X_{attention}) \tag{3}$$

$$X_{hidden} = Linear(ReLU(Linear(X_{attention}))) \tag{4}$$

$$X_{hidden} = LayerNorm(X_{attention} + X_{hidden}) \tag{5}$$

Decoder Block. The decoder block mainly consists of three parts: Masked Multi-Head Self-Attention, Multi-Head Encoder-Decoder Attention, and a Feed-Forward Network. Similar to the encoder block, all three parts include residual connections and layer normalization. The Masked Multi-Head Self-Attention layer in the decoder is used to compute attention representations specific to the decoder itself. In this layer, each input sequence of the decoder undergoes self-attention calculations, resulting in an attention representation tailored to the current decoder input. This representation takes into account the relationships between all words in the input sequence, excluding the current token's context. It allows for better predictions in the next step. Therefore, during each time step of generating predictions, the model can only focus on the current token and the preceding context, while the sequence following the current token remains invisible to the model.

Linear & Softmax. The linear layer takes the output of the decoder as input and projects it into a vector of dimensionality equal to the size of the vocabulary. The softmax layer converts the vector values into a probability distribution that sums to 1. This produces the final output of the entire model.

3.4 Attacks and Adversarial Training

Attacks. It is well-known that deep neural networks often lack robustness. Specifically, deep neural models are easily fooled by adversarial examples, which are generated by introducing slight perturbations to the original samples. To explore the perturbations that have the greatest impact on the quality of assertion generation, we first fine-tune the T5 model on a training dataset to obtain

the base assertion generation model. Then, we generate 11 code mutations as adversarial attacks for each input code in the test set, with each variant corresponding to a generated assertion. By comparing the quality of generated assertions (mainly evaluated using BLEU and ROUGE metrics) between the model under different attack mutations and the original test set, we observe which mutations have the strongest ability to fool the model.

Adversarial Training. We select three mutations, namely VM, FM, and False_Control_Flow, which have the greatest impact on the quality of assertion generation. For each input code in the training set, we combine these three mutations (M) with the original input code (O) to form a new training set $T = M \cup O$. The assertion generation model is fine-tuned on this new dataset. During the model's learning process on the new dataset, it learns to generate correct assertion statements even when the input is a variant with slight perturbations. This enhances the model's ability to withstand minor perturbations and improves its robustness.

4 Experiments

4.1 Experimental Design

Dataset. For our research questions, we utilize the CONTEST dataset, which is an open-source dataset provided in prior work [43]. This dataset consists of test methods collected from projects on GitHub that utilize Junit as the testing framework. Focusing on the Java programming language, the dataset includes 365,450 pairs of test methods. Each test method pair consists of an input code block and an assertion code block. As shown in Fig. 4, the input code block is divided into three parts: the tested method, the test method declaration and some precode, and other methods called within the two blocks.

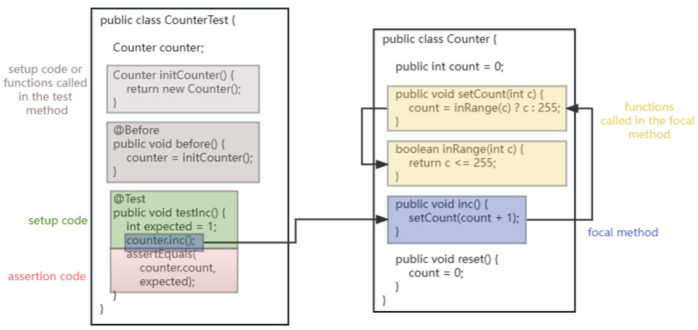

Fig. 4. Dataset composition.

In the first phase, we employ different mutations of input code to attack the model and investigate which parts of the input code have a significant impact on the model's robustness. By analyzing the effects of different mutations on the model's assertion generation quality, we identify the influential parts.

Adversarial Training and Parameter Study. In the second phase, we use the identified influential mutations to augment the training data and study the effect on the adversarial learning of the assertion generation model. We also examine different pre-trained models and parameters to assess their impact on generating assertions.

Research Questions (RQs). To evaluate the effectiveness of our approach, we address the following research questions:

- RQ1: Which parts of the tested code have a significant impact on the robustness of pre-trained models?
- RQ2: How does the assertion generation model perform after data augmentation using the influential mutations identified in RQ1? How does it compare to the baseline model?
- RQ3: How does AssertGen perform on the testing data containing a single variant?
- RQ4: What is the quality of the generated assertions?

Baselines. For our baseline models, we adopt the Contest model [43] proposed in the open-source CONTEST dataset work, as it is highly relevant to our research. Additionally, we include two other classical code pre-training models: CodeGPT [20] and CodeBERT [21]. These models share the same input format as our model, which consists of code blocks containing the tested method, with the output being the generated assertion statements.

Evaluation Metrics. BLEU [46] and ROUGE [47] are commonly used evaluation metrics in the field of machine translation. BLEU assesses translation quality based on precision, while ROUGE evaluates translation quality based on recall. Additionally, precision in assertion generation is of utmost significance, with accuracy equaling 1 only when the predicted assertion statements match the ground truth exactly; otherwise, it remains 0.

When extracting different tokens from the input code blocks, we employ the javalang[1] module for abstract syntax tree analysis. The T5 model serves as the foundational framework for our assertion generation model, implemented using Transformer[2] and PyTorch[3]. The hyperparameters of the model have been optimized based on empirical performance.

By answering these research questions, we aim to gain insights into the impact of different code mutations, data augmentation, and pre-trained models on the effectiveness of the assertion generation process.

Experimental Platform. All experiments were conducted on a Linux server equipped with an AMD EPYC 7371 CPU and an RTX A5000 GPU.

[1] https://github.com/c2nes/javalang.
[2] https://github.com/huggingface/transformers.
[3] https://pytorch.org/.

4.2 Experimental Results

RQ1: Which parts of the tested code have a significant impact on the robustness of the pre-trained model?

Table 1. Output assertion quality under 11 variant attacks.

mutation	BLEU	BLEU-1	BLEU-2	BLEU-3	BLEU-4	ROUGE-1	ROUGE-2	ROUGE-L	Acc(%)
AssertGen	55.23	64.84	55.95	56.73	53.23	81.57	73.07	80.54	45.44
VM	**12.07**	**23.09**	**14.52**	**14.99**	**12.07**	**62.00**	**39.98**	**60.78**	**3.90**
PM	51.28	63.11	54.06	54.85	51.28	80.78	71.62	79.79	41.46
FM	**44.04**	**56.42**	**46.88**	**47.72**	**44.04**	**78.36**	**67.50**	**77.25**	**36.20**
OM	53.78	65.61	56.46	57.31	53.78	81.66	73.14	80.62	45.33
Add_Print	51.58	63.14	54.20	55.05	51.58	81.19	72.40	80.12	44.86
False_Control_Flow	**50.26**	**62.64**	**53.07**	**53.90**	**50.26**	**78.15**	**68.30**	**77.04**	**38.98**
While2For	53.25	64.86	55.97	56.74	53.25	81.57	73.07	80.54	45.41
For2While	52.80	64.54	55.54	56.33	52.80	81.43	72.86	80.40	45.35
Switch2If	53.20	64.81	55.92	56.70	53.20	81.56	73.06	80.54	45.39
Swap_If_Else	53.11	64.73	55.84	56.61	53.11	81.51	72.99	80.49	45.28
Swap_Rows	53.40	65.07	56.14	56.91	53.40	81.28	72.75	80.29	45.02

Firstly, we fine-tuned our assertion generation model using the training set. Then, we attacked the model using 11 variations of code methods, and the results of the assertion generation model on the test set under different variant attacks are presented in Table 1. Compared to the original model's performance on the test set, the model performed the worst under the VM attack, with BLEU and ROUGE-L scores dropping from 55.23 and 80.54 to 12.07 and 60.78, respectively. The perfect prediction rate also decreased from 45.44% to 3.9%. This reveals the significant impact of variable names in the input code block on the model's robustness. When the encoder learns the semantics of the source code, it focuses too much on user-defined variable names, resulting in the model providing completely different predicted assertions when faced with test code written in different developer styles. Moreover, the model's excessive attention to variable names makes its prediction quality highly dependent on the quality of the training set, thereby severely reducing the model's robustness and assertion consistency. Furthermore, the FM attack had the greatest impact on the generation performance, with the perfect prediction rate decreasing by 9.24%. The BLEU and ROUGE scores also dropped by 11.19 and 3.29 compared to the original model's performance. This validates our hypothesis that well-named tested code methods directly reveal the purpose of the methods. Learning the semantics of method names enhances the model's ability to generate accurate assertions by correctly invoking methods, thereby improving the quality of generated assertions. The FM variant constructs mutated attacks on the assertion generation model by selecting candidate method names that have the greatest impact on the model's predicted results and replacing the original method names in the code block. As a result, the model fails to extract meaningful information from method names, leading to a deterioration in the quality of generated assertions. Overall, the method names in the code block also significantly influence the generation quality of the model.

The addition of random "garbage" code statements also leads to a decrease in assertion generation quality. This is evident in the model's performance under the False_Control_Flow attack, where the perfect prediction rate is only 38.98%. The reason for this is that unnecessary control flow statements increase the difficulty for the model to comprehend the semantics of the source code, resulting in less accurate outputs. Under the Add_Print mutation attack, which adds print statements, the model's BLEU and ROUGE-L scores slightly decrease, and the perfect prediction rate also decreases by 0.58%, further supporting our observation.

Another interesting finding is that our assertion generation model remains robust in the face of certain control flow-related variations in the input code. For example, the Operator_Simple, While2For, For2While, Switch2If, Swap_If_Else, and Swap_Rows mutations. Under these attack variations, the generated assertions only experience a slight decrease in quality, with the perfect prediction rate dropping within a range of 0.6%. This indicates that different branch categories have a minimal impact on assertion generation quality, and our model has the ability to recognize different branch types and learn code structure and semantic information from them. Additionally, the model is almost unaffected when two lines of code are swapped without any shared variables. This can be attributed to the T5 model's pre-training on a large-scale code corpus, which provides strong robustness against small perturbations introduced by attackers targeting different branches.

RQ2: How does the assertion generation model perform after data augmentation using some of the variations identified in Question 1? How does it compare to the baseline model?

In RQ1, we found that the model's assertion generation quality was most affected by the variable name variation (VM), method name variation (FM), and "garbage" control flow variation (False_Control_Flow) attacks. Inspired by these findings, we incorporated these three variations into the model's training process, enabling the model to develop the ability to counteract these unstable factors and improve its robustness and assertion generation quality.

To address RQ2, we conducted experiments to evaluate the performance of the assertion generation model after data augmentation with the identified variations. We compared the augmented model's performance against the baseline model. Specifically, we examined how the model's generated assertions differ in terms of quality, robustness, and consistency.

By integrating the VM, FM, and False_Control_Flow variations into the training process, we aimed to enhance the model's ability to handle code blocks with different variable names, method names, and control flow patterns. The augmented model was expected to generate more accurate and reliable assertions, even when faced with variations and adversarial attacks.

Our experimental results demonstrate that the augmented model outperforms the baseline model in terms of assertion generation quality, robustness, and resilience to attacks. The augmentation process helps the model generalize

better and improve its ability to handle different coding styles, variable and method name variations, and control flow structures.

In Table 2, the AssertGen_O model represents the assertion generation model trained on the original training set, while the AssertGen_M model represents the model trained using the three variations and the original training set through adversarial learning. It can be observed that the model trained through adversarial learning with the variations outperforms the AssertGen_O model in terms of metrics measuring the quality of generated assertions. Specifically, the perfect prediction rate increases from 45.44% to 47.91%. This indicates that the model has learned how to generate correct assertions even in the presence of different variations in the augmented data. The model has acquired the ability to counteract minor perturbations in the input, demonstrating stronger robustness.

Table 2. Comparison of the AssertGen_M model after defensive learning with the original AssertGen and other baselines.

model	BLEU	BLEU-1	BLEU-2	BLEU-3	BLEU-4	ROUGE-1	ROUGE-2	ROUGE-L	Acc(%)
Contest	38.19	56.31	42.35	32.85	27.15	68.61	70.04	55.04	-
codeBERT	30.41	43.69	33.36	34.21	30.41	73.44	60.17	72.43	27.45
codeGPT	32.62	46.60	35.76	36.61	32.62	72.84	59.72	71.82	27.42
AssertGen_O	55.23	64.84	55.95	56.73	53.23	81.57	73.07	80.54	45.44
AssertGen_M	**60.08**	**71.22**	**62.66**	**63.43**	**60.08**	**81.83**	**74.11**	**80.94**	**47.91**

Compared to the baseline model Contest, our model performs significantly better. AssertGen_O achieves a notable improvement in BLEU score and ROUGE-L score by 17.04 and 25.5, respectively, compared to Contest. In comparison to the code pre-training models CodeBERT and CodeGPT, AssertGen_O trained on the original dataset demonstrates better performance. CodeBERT and Code-GPT achieve a BLEU score of only 30.41 and 32.62, respectively, while AssertGen_O achieves a BLEU score of 55.23. Furthermore, AssertGen_M, which is trained through adversarial learning, achieves a BLEU score of 60.08. Our model significantly outperforms CodeBERT and CodeGPT in terms of generating high-quality assertions.

These findings highlight the superiority of our model in generating high-quality assertions compared to the baseline model and code pre-training models. The performance improvements can be attributed to the model's ability to effectively handle variations and adversarial attacks, as well as its robustness in the face of input perturbations. The results underscore the significance of our approach in enhancing assertion generation models for improved software development and testing practices.

RQ3: How does AssertGen perform on individual variations of the test data?

The experimental results of RQ2 demonstrated that AssertGen_M, after adversarial training against multiple attacks, learned how to generate high-quality assertions accurately. We further tested the enhanced AssertGen_M

against individual variations to assess the quality of generated assertions and determine if it exhibits stronger robustness compared to AssertGen. As shown in Table 3, AssertGen_M outperforms the original AssertGen_O in the case of VM attack. Specifically, the original AssertGen_O achieved BLEU and ROUGE-L scores of 12.07 and 60.78, respectively, while AssertGen_M achieved scores of 51.21 and 77.09, showing significant improvements. Moreover, the perfect prediction rate increased from 3.9% to 42.28%. This indicates that the model, after adversarial learning to defend against variations, has acquired the ability to identify subtle changes and generate correct assertions, thus enhancing its robustness.

Table 3. Performance of AssertGen_O and AssertGen_M under VM attack.

model	BLEU	BLEU-1	BLEU-2	BLEU-3	BLEU-4	ROUGE-1	ROUGE-2	ROUGE-L	Acc(%)
codeT5	12.07	23.09	14.52	14.99	12.07	62.00	39.98	60.78	3.90
AssertGen_O	46.18	59.51	49.00	49.92	46.18	72.53	62.37	71.49	39.70
AssertGen_M	**51.21**	**63.88**	**54.33**	**55.03**	**51.21**	**78.13**	**69.02**	**77.09**	**42.28**

Furthermore, we compared the performance of the models subjected to only VM attack and those subjected to VM, FM, and False_Control_Flow attacks on generating assertions. It can be observed that the model subjected to only VM attack shows significant improvements in BLEU and ROUGE-L compared to the original model, with the BLEU score increasing from 12.07 to 46.18 and the ROUGE-L score increasing from 60.78 to 77.09. This suggests that the model has learned how to counteract VM attacks through single-variation attacks. However, compared to the model subjected to attacks from all three variations, the performance of the model under single-variation attacks is relatively inferior. This can be attributed to the fact that multi-variation attacks and adversarial training enable the model to acquire more comprehensive knowledge of code, leading to better performance.

RQ4: How is the quality of the generated assertions?

To substantiate the quality of assertions generated by the model, we conducted a comparative analysis of the test coverage outcomes pertaining to the assertions produced by EvoSuite and AssertGen on the public methods of the NumberUtils class within the context of Lang-1-f[4]. Line coverage and branch coverage are two metrics for measuring code coverage, which describe the extent to which the source code of a program has been tested. These metrics aid in assessing the efficiency of test execution.

Table 4 presents the class-level line coverage and condition coverage for each individual public method in the class. From the results provided in Table 4, the following observations can be made: (i) Both EvoSuite and AssertGen have effectively tested all methods. (ii) ATHENATEST has generated accurate test cases for all methods, thereby achieving optimal coverage in the majority of scenarios.

[4] https://github.com/rjust/defects4j.

Table 4. Compare condition coverage and line coverage of test cases generated by EvoSuite and AssertGen in class NumberUtils for each public method.

Focal Method	EvoSuite		AssertGen	
	Lines	Conditions	Lines	Conditions
toInt(String, int)	25 (5.7%)	1(0.2%)	**26(5.9%)**	**2(0.5%)**
toLong(String, long)	25(5.7%)	1(0.2%)	**31(7.0%)**	**2(0.5%)**
toFloat(String, float)	27(6.1%)	1(0.2%)	**28(6.3%)**	1(0.2%)
toDouble(String, double)	24(5.4%)	1(0.2%)	**25(5.7%)**	1(0.2%)
toByte(String, byte)	25(5.7%)	1(0.2%)	**26(5.9%)**	**2(0.5%)**
toShort(String, short)	25(5.7%)	1(0.2%)	**26(5.9%)**	**2(0.5%)**
createFloat(String)	23(5.2%)	1(0.2%)	**24(5.4%)**	**2(0.5%)**
createDouble(String)	23(5.2%)	1(0.2%)	**24(5.4%)**	**2(0.5%)**
createInteger(String)	23(5.2%)	1(0.2%)	**24(5.4%)**	**2(0.5%)**
createLong(String)	23(5.2%)	1(0.2%)	**24(5.4%)**	**2(0.5%)**
createBigInteger(String)	33(7.5%)	6(1.4%)	**36(8.1%)**	**13(3.1%)**
createBigDecimal(String)	24(5.4%)	2(0.5%)	**25(5.7%)**	**3(0.7%)**
min(long[])	30(6.8%)	4(1.0%)	30(6.8%)	4(1.0%)
min(int, int, int)	24(5.4%)	2(0.5%)	**26(5.9%)**	**4(1.0%)**
max(long[])	24(5.4%)	1(0.2%)	**29(6.6%)**	**5(1.2%)**
max(byte, byte, byte)	25(5.7%)	2(0.5%)	**26(5.9%)**	**4(1.0%)**
isDigits(String)	22(5.0%)	1(0.2%)	22(5.0%)	1(0.2%)
isNumber(String)	37(8.4%)	10(2.4%)	**45(10.2%)**	**20(4.8%)**

Subsequently, a qualitative analysis of the quality of assertions generated by AssertGen was performed. Simulating an authentic development process, as depicted in Figure xx, the code sections comprising the method "elementMult" and the pre-test code were utilized as model inputs. The objective was to observe the assertions predicted by codeGPT, codeBERT, and AssertGen (highlighted in red in Fig. 5).

As illustrated in Fig. 5, the assertions generated by AssertGen exhibit the highest quality, aligning with the ground truth. The assertions generated by codeGPT exhibit proximity to the ground truth; however, an error arises during the prediction of the computation method for "expected" within the code, where the Math.pow() function is erroneously employed. Conversely, the results generated by codeBERT are notably unsatisfactory, attributed to its inability to predict the extensive variables "i" and "j" within A.get(i, j), and its failure to generate assertions for the focal method elementMult(). In this instance, both codeBERT and codeGPT have produced assertions of subpar quality. In contrast, our model, AssertGen, emerges as the superior performer in terms of assertion quality.

5 Threats to Validity

Internal Threats. Internal threats refer to potential flaws in our use of baseline methods and implementation. To mitigate internal threats, we employed publicly available replicable packages of the baseline methods and strictly followed their original implementations. Additionally, to ensure the fairness of the results, we conducted double-checking of the code and peer reviews.

External Threats. The selection of pretrained models and datasets contributes to external threats. To address this, we manually examined the validity of the dataset and removed any data that failed compilation to ensure the authenticity and validity of the data. For pretrained models, we chose three of the most popular models currently available, namely CodeT5, CodeGPT, and CodeBERT.

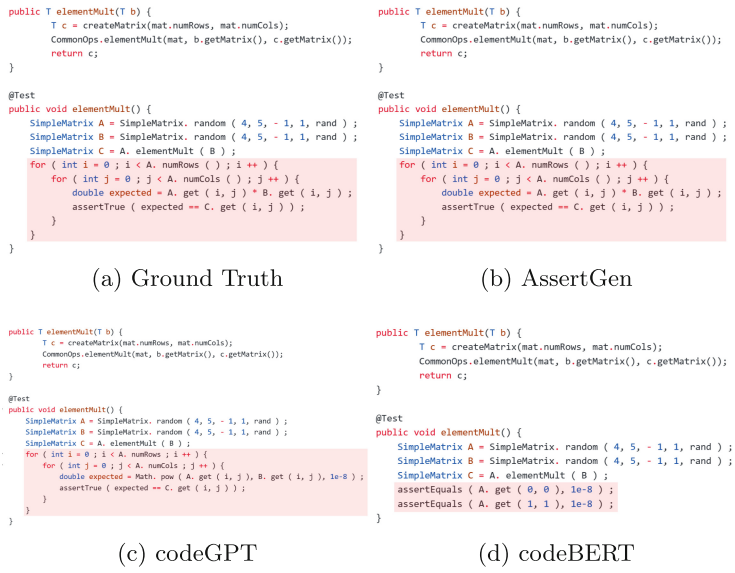

Fig. 5. Assertions generated by codeBERT, codeGPT, and AssertGen models under a specific function.

6 Conclusion

From the perspective of robustness, we employed heuristic rules to create various input mutations to attack the T5 model, aiming to explore the factors that have the greatest impact on the model's robustness. The experimental results demonstrate that variable names, method names, and randomly inserted erroneous control flow statements in the input are the factors that have the most significant influence on the model's robustness. We augmented the training data

with these three types of mutations to enable the model to learn how to counteract these unstable factors and generate high-quality assertions. In future work, we plan to investigate the performance and interpretability of pretrained models in addressing other tasks in the field of software engineering.

Acknowledgments. This project was funded by the National Natural Science Foundation of China (62032016, 61832014).

References

1. Zhu, H., Hall, P.A., May, J.H.: Software unit test coverage and adequacy. ACM Comput. Surv. (CSUR) **29**(4), 366–427 (1997)
2. Cohn, M.: Succeeding with agile: software development using Scrum. Pearson Education (2010)
3. Runeson, P.: A survey of unit testing practices. IEEE Softw. **23**(4), 22–29 (2006)
4. Olan, M.: Unit testing: test early, test often. J. Comput. Sci. Coll. **19**(2), 319–328 (2003)
5. Watson, C., Tufano, M., Moran, K., Bavota, G., Poshyvanyk, D.: On learning meaningful assert statements for unit test cases. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 1398–1409 (2020)
6. Klammer, C., Kern, A.: Writing unit tests: It's now or never! In: 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW). IEEE, pp. 1–4 (2015)
7. Fraser, G., Arcuri, A.: 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)
8. Pacheco, C., Ernst, M.D.: Randoop: feedback-directed random testing for Java. In: Companion to the 22nd ACM SIGPLAN Conference on Object- Oriented Programming Systems and Applications Companion, pp. 815–816 (2007)
9. Almasi, M.M., Hemmati, H., Fraser, G., Arcuri, A., Benefelds, J.: An industrial evaluation of unit test generation: finding real faults in a financial application. In: 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP). IEEE, pp. 263–272 (2017)
10. Shamshiri, S.: Automated unit test generation for evolving software. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pp. 1038–1041 (2015)
11. Zhang, J., Panthaplackel, S., Nie, P., Li, J.J., Gligoric, M.: Coditt5: pretraining for source code and natural language editing. In: 37th IEEE/ACM International Conference on Automated Software Engineering, pp. 1–12 (2022)
12. Fukumoto, D., Kashiwa, Y., Hirao, T., Fujiwara, K., Iida, H.: An empirical investigation on the performance of domain adaptation for t5 code completion. In: 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 693–697. IEEE (2023)
13. Xia, C.S., Wei, Y., Zhang, L.: Automated program repair in the era of large pretrained language models. In: Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery (2023)
14. Kolak, S.D., Martins, R., Le Goues, C., Hellendoorn, V.J.: Patch generation with language models: Feasibility and scaling behavior. In: Deep Learning for Code Workshop (2022)

15. Prenner, J.A., Babii, H., Robbes, R.: Can openai's codex fix bugs? an evaluation on quixbugs. In: Proceedings of the Third International Workshop on Automated Program Repair, pp. 69–75 (2022)
16. White, J., Hays, S., Fu, Q., Spencer-Smith, J., Schmidt, D.C.: Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design, arXiv preprint arXiv:2303.07839 (2023)
17. Jiang, X., Zheng, Z., Lyu, C., Li, L., Lyu, L.: Treebert: a tree-based pre-trained model for programming language. In: Uncertainty in Artificial Intelligence. PMLR, pp. 54–63 (2021)
18. Wan, Y., Zhao, W., Zhang, H., Sui, Y., Xu, G., Jin, H.: What do they capture? a structural analysis of pre-trained language models for source code. In: Proceedings of the 44th International Conference on Software Engineering, pp. 2377–2388 (2022)
19. Wang, Y., Wang, W., Joty, S., Hoi, S.C.: Codet5: identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv preprint arXiv:2109.00859 (2021)
20. Lu, S., et al.: Codexglue: a machine learning benchmark dataset for code understanding and generation. arXiv preprint arXiv:2102.04664 (2021)
21. Feng, Z., et al.: Codebert: a pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155 (2020)
22. Fu, M., Tantithamthavorn, C., Le, T., Nguyen, V., Phung, D.: Vulrepair: a t5-based automated software vulnerability repair. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 935–947 (2022)
23. Fan, G., et al.: Dialog summarization for software collaborative platform via tuning pre-trained models. J. Syst. Softw., 111763 (2023)
24. Imai, S.: Is github copilot a substitute for human pair-programming? an empirical study. In: Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings, pp. 319–321 (2022)
25. Pearce, H., Tan, B., Ahmad, B., Karri, R., Dolan-Gavitt, B.: Can openai codex and other large language models help us fix security bugs? arXiv preprint arXiv:2112.02125 (2021)
26. Pearce, H., Tan, B., Krishnamurthy, P., Khorrami, F., Karri, R., Dolan Gavitt, B.: Pop quiz! can a large language model help with reverse engineering? arXiv preprint arXiv:2202.01142 (2022)
27. Sarsa, S., Denny, P., Hellas, A., Leinonen, J.: Automatic generation of programming exercises and code explanations using large language models. In: Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1, pp. 27–43 (2022)
28. Zhang, Z., Zhang, H., Shen, B., Gu, X.: Diet code is healthy: simplifying programs for pre-trained models of code. In: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1073–1084 (2022)
29. Li, Z., Wang, C., Liu, Z., Wang, H., Wang, S., Gao, C.: Cctest: testing and repairing code completion systems. arXiv preprint arXiv:2208.08289 (2022)
30. Ojdanic, M., Soremekun, E., Degiovanni, R., Papadakis, M., Le Traon, Y.: Mutation testing in evolving systems: studying the relevance of mutants to code evolution. ACM Trans. Softw. Eng. Methodol. **32**(1), 1–39 (2023)
31. Harman, M., McMinn, P.: A theoretical and empirical study of search-based testing: Local, global, and hybrid search. IEEE Trans. Software Eng. **36**(2), 226–247 (2009)

32. Blasi, A., Gorla, A., Ernst, M.D., Pezz'e, M.: Call me maybe: using nlp to automatically generate unit test cases respecting temporal constraints. In: 37th IEEE/ACM International Conference on Automated Software Engineering, pp. 1–11 (2022)
33. Delgado-Perez, A., Ramirez, A., Valle-Gomez, K.J., Medina-Bulo, I., Romero, J.R.: Interevo-tr: Interactive evolutionary test generation with readability assessment. IEEE Trans. Softw. Eng. (2022)
34. Ernst, M.D., et al.: The daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1–3), 35–45 (2007)
35. Csallner, C., Tillmann, N., Smaragdakis, Y.: Dysy: dynamic symbolic execution for invariant inference. In: Proceedings of the 30th International Conference on Software Engineering, pp. 281–290 (2008)
36. Xiao, X., Li, S., Xie, T., Tillmann, N.: Characteristic studies of loop problems for structural test generation via symbolic execution. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 246–256 (2013)
37. Zeller, A., Gopinath, R., B̄ohme, M., Fraser, G., Holler, C.: The fuzzing book (2019)
38. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: 29th International Conference on Software Engineering (ICSE'07), pp. 75–84. IEEE (2007)
39. Shamshiri, S., Just, R., Rojas, J.M., Fraser, G., McMinn, P., Arcuri, A.: Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 201–211 (2015)
40. White, R., Krinke, J.: Testnmt: function-to-test neural machine translation. In: Proceedings of the 4th ACM SIGSOFT International Workshop on NLP for Software Engineering, pp. 30–33 (2018)
41. Tufano, M., Drain, D., Svyatkovskiy, A., Deng, S.K., Sundaresan, N.: Unit test case generation with transformers and focal context
42. White, R., Krinke, J: Reassert: deep learning for assert generation. arXiv preprint arXiv:2011.09784 (2020)
43. Villmow, J., Depoix, J., Ulges, A.: Contest: a unit test completion benchmark featuring context. In: Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021), pp. 17–25 (2021)
44. Pascanu, R., Mikolov, T., Bengio, Y.: On the difficulty of training recurrent neural networks. In: International conference on machine learning. Pmlr, pp. 1310–1318 (2013)
45. Yu, S., Wang, T., Wang, J.: Data augmentation by program transformation. J. Syst. Softw. **190**, 111304 (2022)
46. Papineni, K., Roukos, S., Ward, T., Zhu, W.-J.: Bleu: a method for automatic evaluation of machine translation. In: Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics, pp. 311–318 (2002)
47. Lin, C.-Y.: Rouge: a package for automatic evaluation of summaries. In: Text Summarization Branches Out, pp. 74–81 (2004)
48. Yang, G., Zhou, Y., Yang, W., Yue, T., Chen, X., Chen, T.: How important are good method names in neural code generation? a model robustness perspective. arXiv preprint arXiv:2211.15844 (2022)
49. Dinella, E., Ryan, G., Mytkowicz, T., Lahiri, S.K.: Toga: a neural method for test oracle generation. In: Proceedings of the 44th International Conference on Software Engineering, pp. 2130–2141 (2022)