Learning to Repair Codes via Code Coverage Testing

Hyeongjun Jeon°, Gyeongju Lee, Aditi and Sang-Ki Ko
Department of Artificial Intelligence, University of Seoul
{113bommy, lkj011218, aditimzu16, sangkiko}@uos.ac.kr

Abstract

One of the rapidly evolving topics in software engineering is the automation of program error correction. Programs can have various types of errors, including syntactic, compilation, runtime, and logical errors. The most challenging problem is solving the logical errors as the compiler could not detect them; hence, the use of the test suits plays a dominant role in identifying the logical errors. In this line of research, we employ a pre-trained language model trained on codes (CodeT5) to automatically generate patches for logical errors. In particular, we provide the result of code coverage analysis to the model to understand the runtime behavior of the codes. Our model is then tested on the publicly accessible dataset, the DeepMind's CodeContests dataset, consisting of logically correct and incorrect programs for programming contest problems along with the test cases. Experimental results demonstrate that error location is crucial for correcting logical errors.

1. Introduction

Numerous automated program repair (APR) strategies have been implemented to identify and fix program errors automatically. By saving programmers time and effort, APR significantly increases programmers' productivity. The APR techniques provide patches for erroneous programs, automatically converting incorrect codes to error-free ones. The program's errors can be broadly divided into syntactic and logical errors. Much research has been conducted to repair syntactic errors automatically. Still, less focus was drawn toward fixing logical errors, as finding the root cause of logical errors is very tough. Logical errors tend to fail the test cases, which gives a clue as to how to proceed to solve the logical errors. To address this problem, we employ the idea of code coverage, which is also called test coverage. Code coverage detects which part of the source code of a program is executed when a specific test suit is run against the source code. We use the CodeT5 [4] model, which is an encoder-decoder model; CodeT5 demonstrates impressive results on downstream tasks such as code generation , summarization, etc., and its performance is remarkable as it consists of fewer parameters making it easy to fine-tune the model. We use the DeepMind's CodeContests [2] dataset, which is a publicly available dataset. The dataset consists of correct and incorrect codes with test cases for programming problems. We show that the proposed idea performs better in repairing logical errors by successfully exploiting the information obtained from the code coverage analysis.

```
// Original Data
    4:    3:int func(int a, int b, int c) {
    -:    4:int diff;
    4:    5:a = a % 2;
    4:    6:b = b % 2;
    4:    7:c = c % 2;
    -:    8:if (a == 0 && b == 0 && c == 0) diff = 0;
    -:    9:if (a == 0 && b == 0 && c == 1) diff = 0;
    4:   10:if (a == 0 && b == 1 && c == 0) diff = 1;
    4:   11:if (a == 0 && b == 1 && c == 1) diff = 1;
...

// Clang-format Data
    4:    3:int func(int a, int b, int c) {
    -:    4:int diff;
    4:    5:a = a % 2;
    4:    6:b = b % 2;
    4:    7:c = c % 2;
    -:    8:if (a == 0 && b == 0 && c == 0)
    -:    9:diff = 0;
    -:   10:if (a == 0 && b == 0 && c == 1)
    -:   11:diff = 0;
    4:   12:if (a == 0 && b == 1 && c == 0)
    2:   13:diff = 1;
    4:   14:if (a == 0 && b == 1 && c == 1)
#####:   15:diff = 1;
...
```

Figure 1: Code coverage results after running a C++ code on a specific test case. The number on the left of each code line represents the number of times that the line was executed during runtime. The transformation of C++ code into Clang-format not only enhances code clarity but also renders the coverage results highly practical in pin-pointing erroneous line locations.

2. Related Work

Deep Learning-based APR According to a review of the published works, many studies have been conducted in APR. A deep learning-based solution to APR often uses a neural network to produce a proper patch from an incorrect program as input.

These earlier efforts, nevertheless, were primarily concerned with producing remedies for syntactic mistakes. For instance, a multi-layered seq2seq neural network end-to-end solution

†) This work was supported by the Ministry of Education of the Republic of Korea and the National Research Foundation of Korea (NRF-RS-2023-00208094).

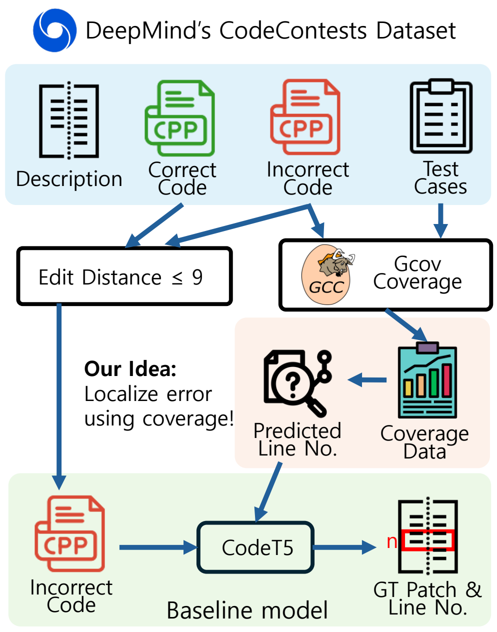

Figure 2: The overview of the proposed APR framework

called DeepFix [1] has been developed to cure various syntax faults by iteratively predicting the statement patches to repair the syntax errors. Research also focuses on improving the APR by using bug reports and tests and has suggested a significant positive impact on repair quality [3].

Spectrum-Based Fault Localization (SBFL) SBFL is one of the best and the most popular methods to locate software bugs as it utilizes the execution results of the test cases along with the information of the coverage to evaluate the likelihood of the program's specific line being erroneous [5]. In this paper, we aim to exploit the result of coverage testing as in SBFL by predicting the the suspicious lines and providing the CodeT5 model to generate more accurate patches.

3. Main Contribution

To train the CodeT5 model to predict the correction of logical errors, we use the data provided by Google DeepMind's CodeContest dataset [2]. We calculate the edit distance between logically correct code and incorrect code. Based on the edit distance, we create a pair of data in which the distance between the two data is below a fixed threshold (set to 9 here).

We first conduct experiments in the most general way, i.e., we fine-tune the CodeT5 model where the input is the incorrect code, and the output is the corrected patches for the erroneous line. However, there is room for improvement as the model needs to learn how to fix the errors from execution results on test cases. Focusing on improving the repair performance, we decided to utilize the method of providing the prediction of logical error location as additional information to the model, so we generated this information via code coverage analysis.

Extracting Coverage Data using Test Cases Code coverage testing is a well-known technique in software engineering for collecting insights on which lines of code have been executed and which lines have not. Therefore, coverage data can be a useful hint to the deep learning model to understand how the code's runtime behavior differs between successful and unsuccessful test cases. We use `gcov`, a test coverage program for C++ codes used in concert with the GCC compiler.

Dataset Construction To train the CodeT5 model to predict the correction of logical errors, we created a pair of data in which the edit distance between correct and incorrect codes is below a fixed threshold. For better usage of coverage information, we use Clang-format before the coverage analysis so that each line of code only contains an individual statement. We generated a patch using the two datasets' pair data (correct and incorrect code pair), which is the difference between the logically correct and incorrect code pair. We provide the model with a code containing the line number and train it to give a corrected line along with the number of that code. We modify the logical error C++ code using the prediction to generate new fixed C++ code.

4. Experiments

Methods We compare the following methods to see if the coverage analysis effectively helps CodeT5 better localize and fix errors:

- **Baseline**: The fine-tuned CodeT5 model.
- **w/ GT Line**: The CodeT5 model fine-tuned with ground-truth line number with the error given as a comment on the first line of the input code.
- **w/ Predicted Line**: The CodeT5 model fine-tuned with the predicted line number calculated from the coverage results. For each successful and failed test case, we count the number of executions of each line and calculate *suspicion score* of lines as the difference between the number of executions on the successful test cases and failed test cases. We select the suspicious code lines that fall within the top third in terms of the suspicion score and add a comment on the first line of the input code with the selected line numbers.

Performance Metrics The test cases provided by CodeContests have been utilized to evaluate the logical error correction ratio. We measure performance based on how many test cases are passed for original erroneous and newly generated modified codes. We check the number of the modified code

Table 1: Overall experimental results

	Data format	Training Method	Error Repair Rate Perfect	Error Repair Rate Partial	Error Repair Rate Total	Fault Localization Accuracy Perfect	Fault Localization Accuracy Partial	Fault Localization Accuracy Total
Pass@1	Original	Baseline	10.81%	**7.95%**	18.76%	17.96%	**37.19%**	55.15%
		w/ Predicted Line	**11.07%**	7.78%	**18.86%**	**19.33%**	36.39%	**55.72%**
		w/ GT Line	29.75%	10.45%	40.20%	97.40%	2.56%	99.96%
	Clang-format	Baseline	12.41%	8.20%	20.61%	18.86%	34.53%	53.39%
		w/ Predicted Line	**18.19%**	**9.88%**	**28.07%**	**36.56%**	**39.65%**	**76.21%**
		w/ GT Line	35.37%	10.05%	45.42%	99.66%	0.33%	99.99%
Pass@10	Original	Baseline	37.32%	20.89%	58.21%	**42.08%**	**18.49%**	**60.57%**
		w/ Predicted Line	**38.02%**	**21.02%**	**59.04%**	41.61%	18.36%	59.98%
		w/ GT Line	60.92%	14.13%	75.05%	99.36%	0.63%	99.99%
	Clang-format	Baseline	39.03%	21.75%	60.78%	43.40%	16.40%	59.70%
		w/ Predicted Line	**46.54%**	19.02%	**65.56%**	**59.11%**	**21.59%**	**80.70%**
		w/ GT Line	65.53%	13.48%	79.01%	99.96%	0.03%	99.99%

that passed all test cases and the number of the modified code that passed more test cases than the original erroneous error codes, but not all cases. Then, we calculated the ratio of both numbers to the total C++ code and used it as a performance metric. We name the case a *perfect repair* if the modified code passed the entire test case and the *partial repair* if the modified code improved over the existing erroneous code regarding the number of successful test cases.

Additionally, we focus on the fact that the model can not localize the error properly, so we aggregated the number of predictions whose line matches the actual logical error location. We name the case a *perfect localization* if the prediction line and actual error location match perfectly and the *partial localization* if the actual error location is included in the prediction result.

Experimental Results and Analysis Table 1 shows the overall experimental results. The baseline model, which is a fine-tuned CodeT5 model on the incorrect code and line fix pairs, can correct logical errors about 18.76% (pass@1) and 58.21% (pass@10) for original data, and 20.61% (pass@1) and 60.78% (pass@10) for revised data using Clang-format. It is readily seen that Clang-format already helps CodeT5 better localize and repair errors.

To verify the effect of including the erroneous line number in the input, we compare the baseline model and the model with the ground-truth erroneous line number (w/ GT Line). We can see that both error repair rate and fault localization accuracy skyrocket as the model clearly understands 'where to fix' from the input code.

Therefore, we include the 'predicted erroneous lines' calculated from the code coverage information in the input to CodeT5. Interestingly, our approach achieves better performance than the baseline model yet not as much as with the ground-truth line number. The surprising fact is that the predicted line numbers barely improve the performance of the baseline model without the help of Clang-format. Figure 1 explains the reason as it is difficult to localize actual erroneous lines from original data from the coverage results while it is much easier after applying Clang-format.

Fault Localization Performance In most cases, we find that the the baseline model cannot even predict the line number correctly. To verify the model's error repair capability separately from the fault localization, we perform one more experiment by providing the actual erroneous line numbers to the model to see if the correct fault localization information can improve the error repair performance. We discover that the model trained with the actual logical error location remarkably outperforms the baseline model, which shows that the correct fault localization helps the model generate the correct patches.

5. Conclusions

We have presented a novel algorithm for automated logical program repair that utilizes a pre-trained neural network model, that is, CodeT5, for fixing logical errors of C++ programs. We have efficiently utilized the test cases and the code coverage technique to fix logical errors better so that the model could understand, locate, and fix them. Evaluation results show that our model with error line prediction using coverage data successfully repairs logical errors and correctly identifies the location of logical errors compared to the baseline models.

References

[1] R. Gupta et al. "DeepFix: Fixing Common C Language Errors by Deep Learning". In: *AAAI 2017*. 2017.

[2] Y. Li et al. "Competition-Level Code Generation with AlphaCode". In: *CoRR* abs/2203.07814 (2022). arXiv: 2203.07814.

[3] M. Motwani and Y. Brun. "Better Automatic Program Repair by Using Bug Reports and Tests Together". In: *ICSE 2023*. 2023, pp. 1225–1237.

[4] Y. Wang et al. "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation". In: *EMNLP 2021*. 2021.

[5] X. Xie et al. "Spectrum-Based Fault Localization: Testing Oracles are No Longer Mandatory". In: *QSIC 2011*. 2011, pp. 1–10.