2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)

Improving Mutation-Based Fault Localization with Plausible-code Generating Mutation Operators

Juyoung Jeon[§], Shin Hong[*]
[*]*School of CSEE, Handong Global University*
Pohang, South Korea
juyoungjeon@handong.edu, hongshin@handong.edu

***Abstract*—This paper proposes a new mutation operator using neural network to generate plausible code elements to improve performance of mutation-based fault localization on omission faults. Unlike the existing mutation operators, the proposed mutation operator synthesizes new code elements at a given mutation site with a neural language model. We extended MUSE to use the proposed mutation operator, and conducted a case study with 3 omission faults found in JFreeChart of Defects4J. As a result, the accuracy of MUSE with the new mutation operator increased significantly in all three faults.**

I. Introduction

Mutation-based fault localization (MBFL) is a promising approach to locate suspicious code lines through analyzing how the test execution results change as each code line of a target program is mutated [1]–[3]. Although MBFL provide highly accurate fault localization results for many real-world bug cases, their performance is poor at locating *omission faults* [4]. Since the most existing mutation operators modify or remove the existing code elements, the mutants that add the missing behaviors at the missing line of omission faults are hardly generated by the mutation operators. Inserting new plausible code at a certain location would provide information for locating omission faults, but there may be an infinite number of code inserting mutants if such mutation operators are designed as syntactic code transformation rules as the existing mutation operators are.

To resolve the limitation of MBFL on omission faults, we propose to extend MBFL to additionally use new mutation operator that generates plausible code elements using neural network. The new mutation operator inserts a conditional statement whose condition expression is generated by the neural network model considering the context of a mutation point. We conjecture that, among many possible expression cases, the neural network model will generate plausible code that effectively introduce likely alternative behaviors to the target program, thus help MBFL analyze how code changes impact the testing results for fault localization.

To demonstrate the effectiveness of the proposed mutators, we conducted case studies with 3 omission faults of JFreeChart for which a MBFL technique MUSE shows poor results. After extending MUSE to leverage the proposed mutation operators, MUSE showed significant improvement as the ranks of the buggy lines is reduced by 80.7% on average.

[§]This work was done when Juyoung Jeon was in Handong Global University.

II. Generative Mutation Operators

A. Neural model for generating Java expressions

We designed a neural network model that receives the context of a mutation point as input, and then predicts expressions that will likely be in that mutation point. The model used for expression generation is based on the a sequence-to-sequence model [5] that generates a sequence of tokens by decoder through the representations of input token sequence fed into the encoder. We used LSTM with the attention mechansim as the language model. For designing the decoder, we extended the pointer mixture network proposed by Li et al. [6] which predicts a token either from predefined dictionary or from input sequence. Unlike Li et al. [6] uses a single cell for predicting only a single token, our model uses multi-cell RNN to generate a sequence of tokens for composing an expression. In addition, our model receives as input a combination of three different code contexts (i.e., prefix, postfix, class member) to provide more useful information to the model.

Figure 2 illustrates the overall architecture of our model. In the encoder part, the input sequence of prefix, posfix, and class member tokens is fed to an embedding layer. Each of the embedding vector is encoded using three different LSTM layers. Our model uses total 300 cells to receive 100 tokens for each of the prefix, the postfix and the class member input (total 300 cells). The hidden state of each encoder cell is passed to the attention layer. The decoder part has total 16 cells each of which predicts the likelihood of each dictionry word on the corresponding location based on the three encoder and the attention output. At the same time, the output of the attention layer is fed to the pointer-mixture layer. For each label offset, the pointer-mixture layer has a pointer model which consists of fully connected layers that makes a prediction on the likelihood that each input offset and the label have the same word. By indicating the location of input where the specific word for a label offset can be found, our model alleviates the out-of-vocabulary problem of the encoder with the predefined dictionary. The pointer-mixture layer also has a selector network for each label offset, which determines whether the output of the corresponding decoder (i.e., a word in the predifined dictionary) or the output of the pointer model (i.e., an input location) is chosen as the final prediction result.

B. Statement Inserting Mutation Operators

The proposed mutation operators insert a new statement of a predefined template to a given mutation location, and then

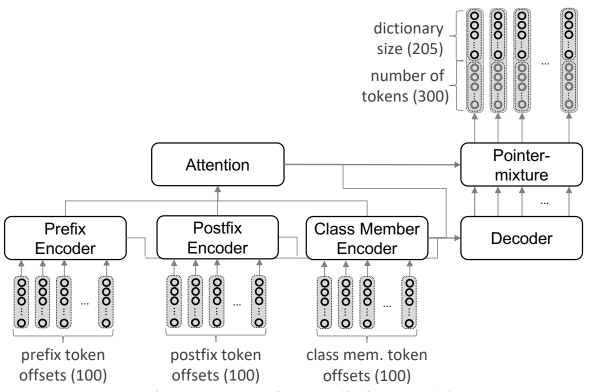

Fig. 1. Expression Prediction Model

uses the generative nerual network to fill in expression parts. The templates used for mutants are *if-break* statements and *if-return* statements with no return type and null as return type. Among the outputs of the model extracted through the beam search, the top two compilable expressions were selected based on its likelihood. Since there can be considerable number of mutation points in the source code, the mutation site in the new mutation operator has been set to three sections of code. The three sections of code targeted for mutation site are before the first statement of blocks, before the first statement after blocks, and after assignment statements with method call expression.

C. Data Preparation and Model Training

For training the model, we used 5 Java projects from the Defects4J benchmark. There was total 2594 number of java files and from each of the java file, expressions and the context data of the expressions were extracted as training data. For the context data of each expression, 100 tokens for each prefix, postfix, and class member data were tokenized. The dictionary built for the data consists of total 205 tokens. We used 102 tokens that were reserved keywords of Java language, and the 100 tokens such as identifiers, numbers, and strings in the source code that most frequently appeared in the training data. The rest of the data that were not in the predefined dictionary were labeled as `UNK`.

The maximum token length of the label expression was set to 16. There are two labels for this model, a label that maps expression tokens with the indexes in the dictionary and a label that maps expression tokens with the indexes of the same token found in the input data. For the dictionary label, the tokens that are not found in the dictionary were replaced as `UNK`. For the locational label, when multiple positions in the input data are equal to the target token, we chose the closest position to the target token. If the expression token is not found in the input sequence but only in the dictionary, the label was set to the index higher than the last index.

We trained the model described above for the input length of 100 tokens. Both encoder and decoder have 200 hidden units. In each encoder and decoder, embedding vector has 100 dimensions. The accuracy of the output is computed by the average match of tokens in the label sequence. The label sequence includes the padding values. The final model used for expression generation had training accuracy of 97.67%.

TABLE I
THE DEFECTS4J PROJECTS USED FOR TRAINING MODEL

Program	# Test Cases	# Mutants by Existing Mutators	# Mutants by New Mutators
Chart-4	1737	4930	829
Chart-5	738	110	20
Chart-25	1149	7104	1305

TABLE II
THE EXPERIMENT RESULT OF 3 OMISSION FAULTS ON MUSE

Program	W/o New Mu. Oper.	W/ New Mu. Oper.
Chart-4	2325	30
Chart-5	13	6
Chart-25	1168	126

III. Preliminary Results

To evaluate the effectiveness of the proposed mutation operators, we conducted a case study on the arbitrary selected three omission faults of JFreeChart where MBFL performs poorly. The artifiact of the selected three faulty versions were obtained from the results of an empirical study of the MBFL techniques [7] with Defects4J [8]. Table I shows the number of test cases used for localizing faults, the number of mutants generated with Major and the number of mutants generated by the statement inserting mutation operators.

To evaluate the effectiveness of the new mutation operator we compare the MUSE results on mutants generated by the existing mutation operators, and the results of mutants generated on the new mutation operator in addition to mutants generated by the existing mutation operators. An average of 718 mutants were additionally generated over the three programs. To compute the suspiciousness score of the MBFL technique, we used the MUSE formula.

Using the proposed mutation operators, MUSE can locate buggy statements more accurately than only using the existing mutation operators. Table II shows the best ranking of buggy statement of MUSE on the three omission faults. MUSE showed significant improvement when using the new mutation operator in addition to the existing mutation operator. In each of the three omission faults, the buggy line rank was reduced by 99%, 54%, and 89%.

```
1:   Number meanValue =
     dataset.getMeanValue(row, column);
/* if (meanValue == null) : PATCH */
++   if (meanValue == null) return ; //Mutant
2:   double value =
       meanValue.doubleValue();
3:   double base = 0.0;
```
Fig. 2. Example of Chart-25

For example, Figure 2 shows the mutant generated in Chart-25 between line 1 and line 2 is a bug fixing mutant and is equal to the bug patch. Both the generated mutant and the bug patch exit the method when a variable assigned in line is 1 null. The mutant is generated after the first line because the first line is an assignment statement with method call and the expression `meanValue == null` is generated by the trained code generation model.

ACKNOWLEDGMENT

This work was supported by IITP grant (No. 2021-0-01001, 60%) and NRF grants funded by the Korea government (NRF-2020R1C1C1013512 and NRF-2021R1A5A1021944).

References

[1] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *ICST*, 2014.

[2] M. Papadakis and Y. Le-Traon, "Metallaxis-FL: mutation-based fault localization," *STVR*, vol. 25, pp. 605–628, 2015.

[3] S. Hong, B. Lee, T. Kwak, Y. Jeon, B. Ko, Y. Kim, and M. Kim, "Mutation-based fault localization for real-world multilingual programs," in *ASE*, 2015.

[4] Y. Lin, J. Sun, L. Tran, G. Bai, H. Wang, and J. Dong, "Break the dead end of dynamic slicing: Localizing data and control omission bug," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 509–519. [Online]. Available: https://doi.org/10.1145/3238147.3238163

[5] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'14. Cambridge, MA, USA: MIT Press, 2014, p. 3104–3112.

[6] J. Li, Y. Wang, M. R. Lyu, and I. King, "Code completion with neural attention and pointer networks," in *IJCAI*, 2018.

[7] J. Jeon and S. Hong, "Threats to validity in experimenting mutation-based fault localization," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE-NIER '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–4. [Online]. Available: https://doi.org/10.1145/3377816.3381746

[8] R. 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*, ser. ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 437–440. [Online]. Available: https://doi.org/10.1145/2610384.2628055