Grammar-Based Patches Generation for Automated Program Repair

Yu Tang[1]*, **Long Zhou**[2], **Ambrosio Blanco**[2], **Shujie Liu**[2],
Furu Wei[2], **Ming Zhou**[2], **Muyun Yang**[1]
[1]Harbin Institute of Technology
19s103120@stu.hit.edu.cn yangmuyun@hit.edu.cn
[2]Microsoft Research Asia
{lozhou,ambrosio.blanco,shujliu,fuwei,mingzhou}@microsoft.com

Abstract

Automated program repair (APR) aims to find an automatic solution to program language bugs without human intervention, and it can potentially reduce debugging costs and improve software quality. Conventional approaches adopt learning-based methods such as sequence-to-sequence models for the patches generation. However, they tend to ignore the code structure information and suffer from grammar and syntax errors. To consider the grammar and syntax information, in this paper, we propose a grammar-based rule-to-rule model, which regards the repair process as the transformation of grammar rules, and leverages two encoders modeling both the original token sequence and the grammar rules, enhanced with a new tree-based self-attention. Besides, to guarantee grammar correctness, we employ a grammatically restricted inference method to generate each grammar rule in a legally constrained sub-search-space considering the generated previous rules. Experimental evaluations on a Java dataset demonstrate that the proposed approach significantly outperforms the state-of-the-art baselines in terms of generated code accuracy.

1 Introduction

Advances in machine learning and the availability of large corpora of source code have led to growing developments of software engineers. Researchers have exploited machine learning to automate several development and maintenance tasks, such as code completion (Svyatkovskiy et al., 2020), comment generation (Hu et al., 2018), code search (Gu et al., 2018), bug localization (Zheng et al., 2016) and fixing (Tufano et al., 2018). It's worth noting that localizing and fixing bugs is known to be an effort-prone and time-consuming task for software developers (Weiss et al., 2007). Hence, several

*Contribution during internship at Microsoft Research

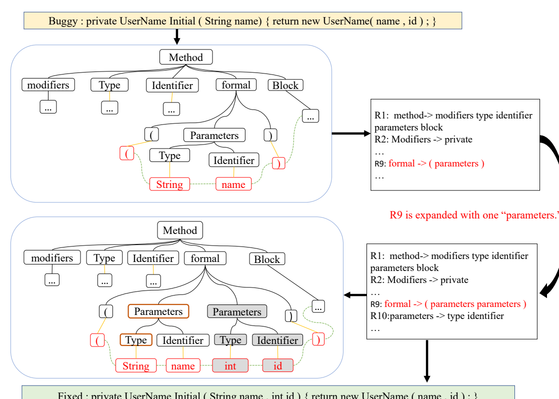


Figure 1: Schematic diagram of our model. Source code is parsed into AST (Abstract Syntax Tree) and then into a sequence of rules. Each rule consists of one head token (parent node) and several tail tokens (children nodes).

works recently focused on automatically repairing programs without human intervention, which can improve programmer productivity and software quality (Tufano et al., 2018; Chen et al., 2019; Vasic et al., 2019; Yasunaga and Liang, 2020).

Automated program repair (APR) research is very active and dominated by techniques based on static analysis (Mechtaev et al., 2016) and dynamic analysis (Wen et al., 2018). Meanwhile, APR is also challenging because fixing bugs is a difficult task. Previous approaches mainly are based on a relatively limited and manually-crafted set of fixing patterns, which need substantial effort and expertise (Saha et al., 2017; Jin et al., 2011; Nguyen et al., 2013). Moreover, these techniques can only fix bugs in a given language or a specific application domain and lack scalability and maintainability.

Very recently, deep learning based approaches, such as sequence-to-sequence learning (Sutskever et al., 2014), are proposed to automatically repair program by learning from massive open-source projects with numerous bug fixes (Tufano et al.,

2018; Chen et al., 2019). However, these sequence-to-sequence methods ignore codes' structure information because they are designed for natural language which is significantly different from program language with strict syntactic and grammatical requirements. Hence, the generated patches of these methods suffer from grammar and syntax errors.

To address the problem, in this paper, we propose a novel grammar-based approach to automatically generate fixed patches for automated program repair. More specifically, instead of using a sequence-to-sequence model with code sequence, we first introduce a grammar-based rule-to-rule model, which regards the repair process as the transformation of code grammar rules, as shown in Figure 1. Second, to guarantee the grammatical and syntactic correctness, we not only introduce a rule encoder (together with a token encoder) to directly extract grammatical features but also employ a grammatically restricted inference method to generate the fixed code. Experimental results conducted on BFPs dataset (Tufano et al., 2018) of CodeXGLUE (Lu et al., 2021) demonstrate that the proposed grammar-based approach significantly outperforms the state-of-the-art baselines.

2 Related Work

Automatic program repair, consisting of automatically finding a solution to software bugs without human intervention, has recently received significant attention (Tufano et al., 2018; Chen et al., 2019; Yasunaga and Liang, 2020). Traditional approaches generate patch candidates by first applying a predefined set of mutation operators on the fault space. They then deploy some heuristics (Qi et al., 2014) to search among these candidates for a correct patch that passes all given test cases (Weimer et al., 2009; Qi et al., 2014). Although these methods have shown to be able to fix a wide range of bugs, they can only fix bugs in a given language or a specific application domain (Saha et al., 2017; Jin et al., 2011; Nguyen et al., 2013).

Inspired by the development of deep learning in a variety of problems, researchers attempt to employ deep learning based approaches to automatically repair code by learning from massive buggy-fixes pairs (Tufano et al., 2018; Chen et al., 2019; Vasic et al., 2019; Guo et al., 2020). Tufano et al. (2018) first presented an end-to-end approach to fix program language based on sequence-to-sequence learning. They released datasets of APR and evaluated the performance of neural machine translation. SequenceR (Chen et al., 2019) employed copy mechanism based on line level. DeepFix (Gupta et al., 2017) and SynFix (Bhatia et al., 2018) repair syntax errors in programs using neural program representations. Despite their effectiveness, the generated patches of these methods suffer from grammar and syntax errors.

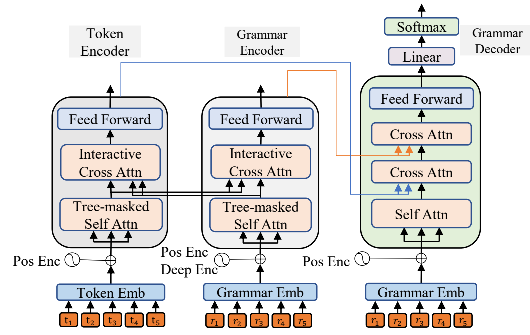


Figure 2: The framework of our proposed model.

Compared to previous works, our proposed approach has three advantages: (1) we employ the state-of-the-art Transformer model as the skeleton of code repair model; (2) we incorporate the grammar information of fixing ingredients into our model by using token and grammar encoders; (3) we propose a grammar-guided inference method to guarantee the grammar correctness.

3 Our Approach

In this section, we will first introduce a grammar-guided rule-to-rule model (Section 3.1), and then present a grammar-constrained inference method (Section 3.2).

3.1 Grammar-Driven Model

Our grammar-guided model, which is based on the state-of-the-art Transformer model (Vaswani et al., 2017), has a token encoder, a grammar encoder, and a grammar decoder, as shown in Figure 2.

3.1.1 Token and Grammar Encoder

To model token representations and grammar structures, we employ token encoder and grammar encoder to model code unit and code grammar, respectively. The two encoders have similar model architecture and different inputs, which are token sequence $\{t_1, t_2, ..., t_m\}$ and rule sequence $\{r_1, r_2, ..., r_n\}$. Considering the difference of sequence and tree structure, besides conventional sinusoidal positional embedding (Vaswani et al., 2017), we also introduce a **depth embedding** to

enhance of capacity of modeling tree structure for the grammar encoder. As for the model structure, the first sub-layer of encoders is **tree-masked self-attention**, the second sub-layer is **interactive cross-attention**, and the last sub-layer is a feed-forward layer.

Depth Embedding We extract the depth information of each rule in the corresponding abstract syntax tree (AST). For the example in Figure 1, the depth of rule *modifiers* $\rightarrow$ *private* is depending on the depth of its head token *modifiers*, which counts 1 in this example. The final position embedding is the sum of the sinusoidal position encoding and the depth embedding.

Tree-Masked Self-Attention To focus on the local information from directly contacted tokens or rules, we propose tree-based attention applied to one head of multi-head self-attention. Formally, we first build a distance-based mask M_{tree}, where $M_{tree}[i, j] = 0$ if node i is the parent or one of the children of j, else $M_{tree}[i, j] = -e^9$. Then, we employ the proposed M_{tree} to one dot-product attention:

$$\mathrm{ATT}(Q, K, V) = \mathrm{softmax}(\frac{QK^T}{\sqrt{d_k}} + M_{tree})V \tag{1}$$

Interactive Cross-Attention The goal of interactive cross-attention is to make full use of token information and syntax information interactively. Specifically, given the outputs of tree-masked self-attention in token encoder and grammar encoder, e.g., H_{tok} and H_{gra}, the output of interactive cross-attention can be expressed as:

$$\begin{aligned} O_{tok} &= \mathrm{Attention}(H_{tok}, H_{gra}, H_{gra}) \\ O_{gra} &= \mathrm{Attention}(H_{gra}, H_{tok}, H_{tok}) \end{aligned} \tag{2}$$

where $\mathrm{Attention}(\cdot)$ is the same as standard self-attention in Transformer.

3.1.2 Grammar Decoder

For each layer in the grammar decoder, the lowest sub-layer is the masked multi-head self-attention network, and the top layer is a feed-forward layer, as shown in Figure 2. Moreover, we design three attention strategies to integrates source token and grammar information.

(1) The **standard strategy** is the same as the traditional cross-attention in Transformer. Specifically, the query Q comes from the output of decoder self-attention, and the key-value pair $\{K, V\}$ is transformed only from the output of the grammar encoder. (2) Figure 2 shows the **cascade strategy**, in which we first compute the cross-attention with token encoder, then use the output as the query to calculate the cross-attention with grammar encoder. (3) The **parallel strategy** attends to each encoder independently and then sums up the context vectors. We will compare the three strategies in Section 4.4.

3.2 Grammar-Constrained Inference

Considering the sensitivity and strictness of program language, we further propose a grammar-constrained inference method to guarantee the grammatical correctness of the output. More specifically, we first build an AST in the inference process according to the currently generated rule sequence and maintain an indicator to locate the AST node where the extension is happening. Then, we filter out the unsatisfactory rules whose head token is not the current extending node, by using mask operation in softmax function. Finally, our AST and indicator can be updated for the next prediction. It is worth noting that the node pointed by the indicator is the one that should be expanded as a parent (head token) in the next inference.

Take figure 1 as an example, when R9 has been predicted, the indicator is pointing the node *parameters*. We limit the search space of the next rules, which have to satisfy its head node is *parameters* (marked by a brown border). Then applicable rule with the highest probability (R10 in this case) is chosen and tail tokens *type* and *identifier* of R10 are added into AST. Finally indicator is transferred from *parameters* to *type*, denoting that *type* is the next expanded node.

4 Experiments

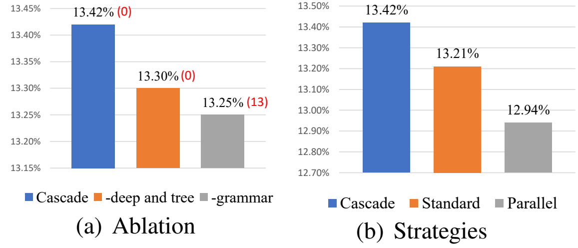
4.1 Datasets and Metrics

Datasets We evaluate our approach on BFPs dataset (Tufano et al., 2018), a collection of Java functions on Github Archive. BFPs consists of 58K bug-fixes data and is divided into training, validation and test sets by 8:1:1. We extracted 4.5K grammar constraints from the dataset in total, among which 0.5K are related to vocab and others related to grammar rules.

Metrics Following Tufano et al. (2018), we employ **XMatch**, a metric indicating the percentage of model's outputs that exactly match the reference, including Top XMatch and All XMatch. Top Match only utilizes top 1 of the beams as output, and All Match uses all beams as outputs to match the refer-

Method	Beam	XMatch (Top)	XMatch (All)	XBug
NMT (Tufano et al., 2018)	1	9.22%*	9.22%*	-
	5	-	27.33%*	-
Token-Trans	1	10.03%	10.03%	56
	5	10.32%	24.02%	44
Rule-Trans	1	10.87%	10.87%	31
	5	11.95%	24.94%	14
Our Model	1	11.47%	11.47%	0
	5	13.42%	28.03%	0

Table 1: Main results of our model on BFPs. Results marked with * are from Tufano et al. (2018).



(a) Ablation (b) Strategies

Figure 3: Comparison on different modules (a) and strategies (b). "deep and tree" means the depth embedding and tree-based attention, and "grammar" demotes the grammar-constrained inference method.

ence. The correlation coefficient analysis of (Ren et al., 2020) also shows that human evaluation is more correlated with **XMatch** than n-gram matching for code repair task. Besides, we employ **XBug** as the auxiliary metric to evaluate the grammatical correctness of output, which examines basic grammar structure of output, like symbol usage and context matching.

4.2 Experimental Settting

Our model is built on PyTorch and trained on 4 GPUs of TITAN XP for 7 hours. We parse source code into AST with *tree sitter*[1], and use the same Transformer setting as Vaswani et al. (2017). The hidden size is 512. The layer of both two encoders and decoder is 6. The dropout probability is 0.5. The number of trainable parameters in our best model is 49M. We test system performance with beam size = 1 and 5, and the latter is used as default.

4.3 Main Results

Table 1 shows the results of all models. The first line is the performance of the previous RNN-based NMT model (Tufano et al., 2018) on BFPs. Token-Trans is built on the Transformer model using sequences of source code as input, while Rule-Trans employs grammar sequences as input and output.

Top Match Token-Trans and Rule-Trans models get the performance of 10.32% and 11.95% with beam search respectively, which first demonstrates the model superiority by using grammar rules. Furthermore, our proposed model outperforms Token-Trans and Rule-Trans by 3.1% and 1.5% respectively, indicating the effectiveness of our proposed grammar-aware modules and inference method in the APR task.

[1]http://tree-sitter.github.io/tree-sitter/

All Match We verify All Match metric to compare with previous work (Tufano et al., 2018), which judges correct model predictions if any one of the beams can fix the buggy code. Results show that our proposed model suggests 28.03% correct patches with beam search, and our model performs better with 2.2% and 0.7% improvements than (Tufano et al., 2018) when beam is 1 and 5 respectively.

XBug We further analyze the grammatical quality of different models. Token-Trans model suffers from grammatical errors with 56 samples in the test set, while in Rule-Trans model, the number decreases to 31. Due to the ability to model grammar rules and guide rule generation of our approach, our proposed model can effectively avoid grammatical errors, guaranteeing the grammar correctness of generated patches.

4.4 Effect of Modules and Strategies

In this section, we will evaluate the contribution of different modules and compare the three combination strategies described in Section 3.1.2.

Figure 3 shows the XMatch results of different models, and we also list the XBug number in brackets. The results in Figure 3 (a) show that all of the proposed methods have positive effects. It's worth noting that the performance significantly drops in terms of XBug if we remove the grammar-constrained inference method. Compare different combination strategies in Figure 3 (b), parallel strategy performs worse than other strategies, and the underlying reason is that concatenating token and grammar sequences results in too long sentences. Besides, the cascade strategy behaves best because it can make full use of the token information and the grammar information provided by encoders.

5 Conclusion

In this paper, we propose a grammar-guided end-to-end approach for automated program repair. Particularly, we introduce three structure-aware modules and three combination strategies, and present a grammar-based inference algorithm to guarantee grammar correctness of generated patches. Experiments on BFPs dataset demonstrate that Grammar-based system performs better than Token-based system for both model learning and grammar correctness. Moreover, system that simultaneously modeling grammar and its inside token information showed great potentiality in our works. Besides, the advantage of grammar-constrained inference inspires us to explore more about the possibility of combining grammar constraints with NLP model.

References

Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. 2018. Neuro-symbolic program corrector for introductory programming assignments. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 60–70. IEEE.

Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*.

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 933–944. IEEE.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*.

Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. Deepfix: Fixing common c language errors by deep learning. In *Proceedings of the aaai conference on artificial intelligence*, volume 31.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 200–20010. IEEE.

Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. 2011. Automated atomicity-violation fixing. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 389–400.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*, pages 691–701.

Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781. IEEE.

Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 254–265.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*.

Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. 2017. Elixir: Effective object-oriented program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 648–659. IEEE.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27:3104–3112.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. Intellicode compose: Code generation using transformer. *arXiv preprint arXiv:2005.08025*.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 832–837.

Marko Vasic, Aditya Kanade, Petros Maniatis, David Bieber, and Rishabh Singh. 2019. Neural program repair by jointly learning to localize and repair. *arXiv preprint arXiv:1904.01720*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is all

you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc.

Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*, pages 364–374. IEEE.

Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. 2007. How long will it take to fix this bug? In *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*, pages 1–1. IEEE.

Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1–11. IEEE.

Michihiro Yasunaga and Percy Liang. 2020. Graph-based, self-supervised program repair from diagnostic feedback. *arXiv preprint arXiv:2005.10636*.

Wei Zheng, Desheng Hu, and Jing Wang. 2016. Fault localization analysis based on deep neural network. *Mathematical Problems in Engineering*, 2016.