Grammar- and Coverage-based Augmentation of Programs for Training LLMs

Shin SAITO
IBM Research - Tokyo

Takaaki TATEISHI
IBM Research - Tokyo

Yasuharu KATSUNO
IBM Research - Tokyo

***Abstract*—Training large language models (LLMs) for programming tasks, particularly code translation, requires diverse and syntactically valid code dataset. While data augmentation can enhance generalization, uncontrolled augmentation leads to overfitting or invalid examples. In this paper, we introduce a grammar- and coverage-based augmentation method that systematically generates syntactically valid code taking the coverage of grammar rules into account. This approach ensures both syntactic correctness and diversity in the code dataset, while suppressing excessive data augmentation. Our preliminary experiment demonstrates that our method produces well-distributed training data, contributing to improved representation of the underlying grammar.**

I. Introduction

A. Background

When training large language models (LLMs), improving accuracy often requires augmenting the training data. LLMs rely heavily on the diversity and richness of their input data to generalize well across a wide range of tasks and domains. Providing diverse and sometimes complex input examples can enhance model performance by covering a broader range of possible inputs, enabling the model to learn nuanced patterns and relationships. This is particularly important in domains such as programming, where syntactic and semantic variations can be vast and subtle.

However, data augmentation is not without its challenges. Excessively complex or overly specific training data can lead to overfitting, where the model becomes too tailored to the training examples and fails to generalize to unseen inputs. This issue is commonly referred to as a regularization problem. Moreover, in the context of program-like data—where inputs are often governed by strict grammatical rules—the balance between diversity and validity becomes even more critical. Augmenting such data without violating syntactic constraints requires careful design, and the complexity of the augmented examples must be controlled to avoid introducing noise or bias into the training process.

B. Approach

Our method leverages a predefined formal grammar and enhances the coverage of grammar rules that are underrepresented in the training data. This is achieved by computing the coverage of the existing dataset over the grammar structure. The grammar-aware approach ensures that the augmented training data remain syntactically valid while introducing code patterns that are sparsely represented in the original dataset.

We also introduce an unfolding parameter, which allows us to control the structural complexity of the generated data. This helps avoid the limitations of random augmentation and prevents the introduction of overly complex or unrealistic examples.

C. Contribution of This Paper

This paper presents two primary contributions.

Grammar-based and Coverage-directed Augmentation: In this study, we propose a method for controlling the complexity of data augmentation in cases where the training data can be defined by a grammar, similar to a program. Our approach leverages the structural properties of grammars to generate syntactically valid and semantically diverse examples, while maintaining control over their complexity. By treating programs as structured data governed by formal rules, we can systematically explore the space of possible inputs and outputs, enabling augmentation that is both meaningful and scalable.

Empirical Evaluation: Through preliminary experiments, we demonstrate that our grammar-based augmentation method enables well-distributed training datasets that better represent the target domain. This results in improved generalization of the training datasets without introducing invalid or overly complex examples.

II. Augmentation Method

The proposed augmentation method consists of the following four steps: (1) Visualizing the given grammar for training data, (2) Unfolding the repetition rules in the grammar based on a specified level, (3) Calculating the coverage of existing training data over the expanded grammar, and (4) Generating new training data using an LLM, guided by the coverage analysis.

A. Preliminaries

Our approach assumes that valid training data is defined by a grammar. For simplicity, we consider a grammar G expressed as a regular language, i.e., represented by regular expressions.

The grammar of regular expressions is defined as follows, where ε denotes the empty string and a represents an arbitrary grammatical symbol:

$$G ::= \varepsilon \mid a \mid G_1 G_2 \mid (G_1 \mid G_2) \mid G+ \tag{1}$$

2151-0849/25/$31.00 ©2025 IEEE
DOI 10.1109/ASEW67777.2025.00010

Here, G^+ denotes one or more repetitions of G. We also introduce the notations $G*$ and $G?$ to represent $\varepsilon \mid G+$ and $\varepsilon \mid G$, respectively.

Note that the expression `a|bc` is interpreted as `a|(bc)`, rather than `(a|b)c`.

Although grammars are often specified using context-free languages such as Backus–Naur Form (BNF) [2], it is frequently possible to approximate them using regular expressions [5], or to extract regular components to which our method can be effectively applied.

B. Grammar Visualization

Regular grammars can be represented as finite automata, which are often visualized using *railroad diagrams*. We refer to such visualizations as *grammar graphs*.

For instance, the grammar $G_0 =$ `a{b*}c?d+` is visualized in Figure 1. The subsequent explanation of our algorithm will refer to this graph.

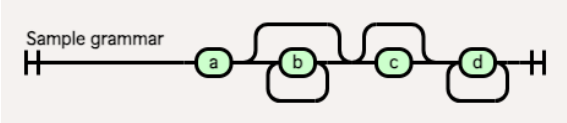

Fig. 1. Grammar graph for the example grammar

Grammars often include repetition rules, which manifest as loops within the graph. This is a natural feature, as programs typically consist of multiple statements and expressions, leading to recurring grammatical patterns.

C. Graph Unfolding

Given a positive integer parameter N, we unfold the repetition rules in the grammar up to N times. For example, the rule `a+` is unfolded to `a|aa+` when $N = 1$, and to `a|aa|aaa+` when $N = 2$.

Formally, the *unfold* operation is defined as follows:

$$\mathit{unfold}(N, \varepsilon) = \varepsilon \tag{2}$$

$$\mathit{unfold}(N, a) = a \tag{3}$$

$$\mathit{unfold}(N, G_1 G_2) = \mathit{unfold}(N, G_1)\ \mathit{unfold}(N, G_2) \tag{4}$$

$$\mathit{unfold}(N, G_1 \mid G_2) = \mathit{unfold}(N, G_1) \mid \mathit{unfold}(N, G_2) \tag{5}$$

$$\mathit{unfold}(N, G+) = \bigcup_{i=1}^{N} G^i \;\middle|\; G^N G+ \tag{6}$$

In Equation 6, G^1 is defined as G, and G^{m+1} is defined recursively as GG^m.

Applying this unfolding operation to the earlier grammar G_0 with $N = 2$ yields the following expression after simplification:

$$\texttt{a(}\varepsilon\texttt{|b|bb|bbb+)c?(d|dd|ddd+)} \tag{7}$$

This unfolded grammar is visualized in Figure 2, where empty strings are explicitly represented by nodes labeled `(ε)`.

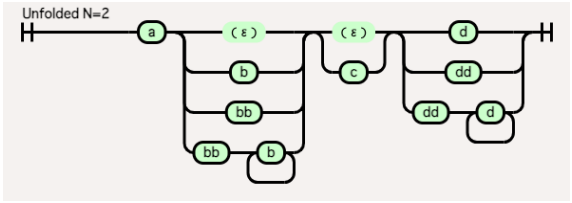

Fig. 2. Unfolded example grammar with $N = 2$

D. Construction of Coverage Graph

Next, we compute the coverage of the original training dataset over the grammar graph by counting how frequently each node is traversed.

For the example grammar G_0, the coverage resulting from a single training instance `abdd` is illustrated in Figure 3. Each node is labeled in the format ⟨node label⟩ : ⟨number of visits⟩, and nodes that were visited are highlighted in red.

By aggregating these counts across the entire dataset, we obtain a *coverage graph*, a grammar graph annotated with coverage information. An example of such a graph is shown in Figure 4.

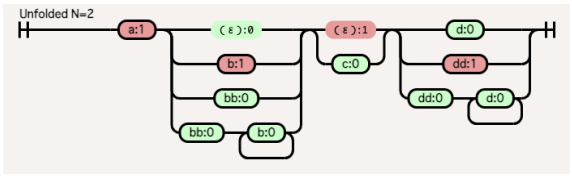

Fig. 3. Coverage graph for a single training instance.

E. Coverage-based New Data Generation

We identify underrepresented paths in the coverage graph, specifically, paths that pass through sparsely covered nodes, which are highlighted in green. These paths are treated as candidates for new training data. They are computed using the n-th shortest path algorithm for positively weighted directed graphs.

For example, the path `abbcd` may be identified as a candidate from the coverage graph shown in Figure 4.

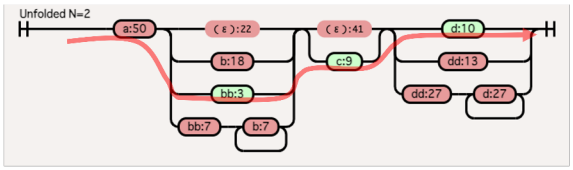

Fig. 4. An example of an identified path in the coverage graph.

Once the format is specified, we use generative AI to produce a code snippet that conforms to the identified structure. Details of the generation process are presented in Section III. By adding variable declarations and a procedure signature to the generated snippet, we can construct a complete compilation unit. This step can also be assisted by generative AI.

The overall augmentation process involves applying the proposed method iteratively to the initial training data. After each iteration, coverage is recalculated and further augmentation is performed. This cycle continues until the training data reaches a sufficient volume.

III. Experiment

We conduct a preliminary experiment to evaluate the effectiveness of the proposed approach.

The training dataset used in this study is referred to as *p2j-data*, which was developed within the author's organization for the purpose of converting PL/I programs to Java. This dataset consists of 96 pairs of PL/I programs and their corresponding Java translations. In this experiment, we focus exclusively on the PL/I side and describe the augmentation process. The generation of the corresponding Java programs is assumed to be handled by a separate method, which is outside the scope of this paper.

Each PL/I program in the dataset consists of a single procedure. The grammar graph for such procedures is shown in Figure 5, with certain types of statements omitted for simplicity.

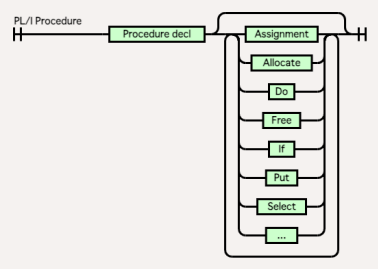

Fig. 5. Grammar graph for PL/I procedures.

As an example, we set the unfolding parameter N to 2. The resulting coverage graph, generated using the p2j-data, is presented in Figure 6. From a grammatical perspective, it is evident that the training data exhibits a bias. For example, assignment statements appear frequently. This is expected, as many programs begin with variable initialization. However, a more balanced distribution of grammatical constructs in the training data could potentially improve the overall quality of the model. Note that there are some differences between the probabilistic decision trees and the coverage graphs, one of which is a locality of statistics.

From the coverage graph, a candidate path for augmentation can be selected. For example, the sequence ⟨*Select*⟩-⟨*Allocate*⟩-⟨*Free*⟩ represents a path that passes through under-represented nodes. The prompt shown in Figure 7 is automatically generated from the path information and is used to guide generative AI in producing a new code snippet:

The resulting snippet is generated accordingly:

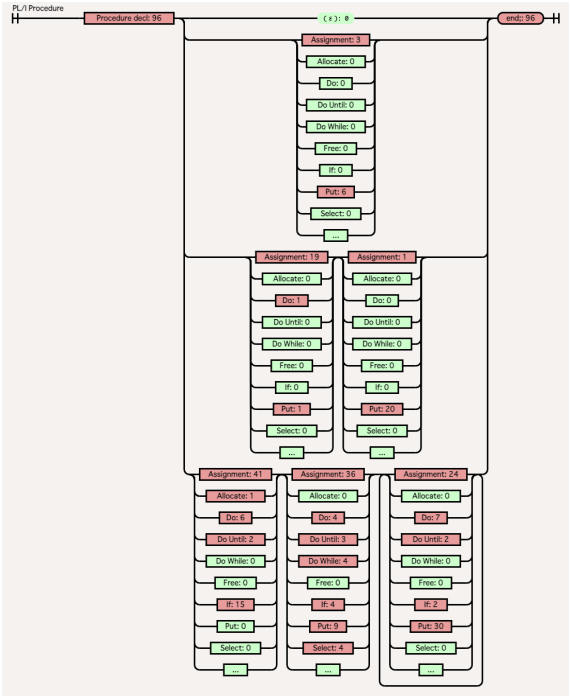

Fig. 6. Coverage graph for PL/I procedures based on the p2j-data, unfolded with $N = 2$.

```
SELECT (A > B) THEN C = A; ELSE C = B;
ALLOCATE X(100);
FREE X;
```

By adding a procedure signature and variable declarations, a complete program can be obtained. This step is also supported by generative AI:

```
SAMPLE: PROC OPTIONS(MAIN);
    DECLARE A FIXED BINARY;
    DECLARE B FIXED BINARY;
    DECLARE C FIXED BINARY;
    DECLARE X(100) FIXED BINARY;

    SELECT (A > B) THEN C = A; ELSE C = B;
    ALLOCATE X(100);
    FREE X;
END SAMPLE;
```

By iterating the described augmentation process, more diverse training data can be generated.

A. Discussion

As is widely recognized, randomly generating grammatically valid programs typically requires substantial implementation effort. However, as demonstrated in our experiments, it is possible to generate syntactically correct programs without manual implementation by leveraging LLMs for augmentation.

```
Generate a PL/I program snippet consisting of 3 statements in the following format.

# Format

<Select statement> ;
<Allocate statement> ;
<Free statement> ;

# Instructions

- Replace each placeholder (<...>) with a randomly selected, syntactically valid PL/I statement
  of the corresponding type.
- Ensure the statements are realistic and representative of typical PL/I usage.
- Use compound expressions where appropriate to reflect practical coding patterns.
```

Fig. 7. A prompt to generate a PL/1 snippet consists of three statements.

One issue that remains is the unnaturalness of some of the generated programs. Again we can lend the power of LLMs, by introducing *Evaluator-Optimizer* agents. Additionally, our current experiments focus solely on augmenting the source-side code. Extending augmentation techniques to the target-side language remains an open problem and is a critical direction for future research.

Furthermore, it is conceivable to introduce multidimensional unfolding. In this context, we consider not only horizontal expansion — such as increasing the number of statements within a program—but also vertical complexity, which involves increasing the depth of statement nesting. That could enable generation of more complex programs that contribute to the model quality.

IV. Related Work

Two recent studies [3], [6] have explored data augmentation techniques for code translation. The literature [3] proposed rule-based and retrieval-based methods to generate parallel code pairs, while the literature [6] utilized comparable corpora and multiple references to enrich training data. Notably, neither approach is based on grammar. In contrast, a grammar-based data augmentation method [4] has proposed for text classification. However, this method constructs grammar rules from text training data to perform data augmentation, whereas our method assumes a predefined grammar and focuses on increasing the coverage of rules where the training code set is sparse. This difference arises from the fact that their method is designed for natural language text rather than code.

Coverage-based techniques are widely employed in fuzzing, which involves the random generation of test data derived from original inputs. Amaya and Antonio [1] introduced a method leveraging probabilistic decision trees for fuzzing. However, their work does not detail the construction process of such trees. In contrast, our approach enables the systematic construction directly from grammar specifications. Furthermore, our coverage graph effectively captures global distribution patterns, offering a more comprehensive representation compared to the local branch probabilities used in their method —an advantage particularly relevant for large language model (LLM) generation.

V. Conclusion

We have presented a grammar-based and coverage-guided approach for augmenting program-like data to support the training of large language models. By leveraging the structural properties of formal grammars, the proposed method generates syntactically valid and semantically diverse examples while maintaining control over their structure. This enables the construction of training datasets that are both representative and generalizable, addressing key challenges in data augmentation such as overfitting and syntactic invalidity.

This study is a work in progress. As a next step, we plan to apply the algorithm in actual language model training to empirically validate its contribution to accuracy improvement. We also aim to identify the optimal unfolding parameter that balances diversity and generalization. An important future challenge is to develop augmentation techniques that produce more natural and realistic programs, thereby enhancing the relevance and utility of the generated data.

References

[1] Zamudio Amaya and José Antonio. Shaping test inputs in grammar-based fuzzing. In *Proceedings of the ISSTA 2024*, page 1901–1905, New York, NY, USA, 2024. Association for Computing Machinery.

[2] John Warner Backus. The syntax and semantics of the proposed international algebraic language of the zurich acm-gamm conference. In *IFIP Congress*, 1959.

[3] Binger Chen, Jacek Golebiowski, and Ziawasch Abedjan. Data augmentation for supervised code translation learning. In *Proceedings of the 21st IEEE/ACM International Conference on Mining Software Repositories (MSR)*, pages 444–456, Lisbon, Portugal, 2024. ACM.

[4] Joonghyuk Hahn, Hyunjoon Cheon, Elizabeth Orwig, Su-Hyeon Kim, Sang-Ki Ko, and Yo-Sub Han. Gda: Grammar-based data augmentation for text classification using slot information. In *EMNLP 2023*, pages 7291–7306, Singapore, December 2023. Association for Computational Linguistics.

[5] Mehryar Mohri and Mark-Jan Nederhof. Regular approximation of context-free grammars through transformation. In *Robustness in Language and Speech Technology*, volume 17 of *Text, Speech and Language Technology*, pages 153–163. Kluwer Academic Publishers, 2000.

[6] Yiqing Xie, Atharva Naik, Daniel Fried, and Carolyn Rose. Data augmentation for code translation with comparable corpora and multiple references. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, 2023.