Evaluating Program Coverage for Code-Model Training

Nandakishore Menon
IBM Research
Bangalore, India
nandakishore@ibm.com

Diptikalyan Saha
IBM Research
Bangalore, India
diptsaha@in.ibm.com

***Abstract*—In recent years, CodeLLMs have revolutionized the way developers interact with code. One notable application has been program translation, such as converting COBOL to Java or C to Rust. A critical challenge in this domain is ensuring that CodeLLMs are trained on programs that cover all syntactic features of the target language. This issue is especially pronounced for legacy languages like COBOL and ABAP, which are syntactically rich and have limited availability of open-source programs. In this paper, we present a tool for evaluating the syntactic coverage of COBOL programs. At the core of our approach is a representation called the Coverage Tree, which compactly and intuitively captures the syntactic constructs covered by a set of programs. Additionally, the tool can generate code statements to address uncovered syntactic gaps. Experimental results with COBOL benchmarks demonstrate the effectiveness of the tool. Screencast URL: https://youtu.be/lM0KHzcvllY.**

I. Introduction

In recent years, the use of code-specific large language models (CodeLLMs) has rapidly grown, transforming various fields in software development, programming education, and automation [1]. These models, fine-tuned on vast repositories of code, have enabled significant improvements in tasks such as code generation, bug detection, code translation, and even automated refactoring. Despite their promise, one of the critical challenges in training and evaluating CodeLLMs is ensuring that the training data provides sufficient coverage of the programming language constructs. The "coverage" problem refers to whether the training data syntactically and semantically spans the full range of a programming language's features and behaviors. This is crucial because models trained on incomplete or biased datasets may fail to generate or comprehend certain language constructs, leading to errors, inefficiencies, and unhandled edge cases during deployment.

The coverage problem is especially critical in the context of legacy languages like COBOL and ABAP, where open-domain programs are scarce, and training data availability is limited [2]. These languages, still widely used in critical industries such as finance, healthcare, and government, present unique challenges because they continuously evolve with new constructs. For example, COBOL programs often integrate with transaction management systems like CICS (Customer Information Control System) or database systems like IMS (Information Management System), adding layers of complexity that modern CodeLLMs must handle. However, due to the lack of publicly available code and the specialized nature of legacy systems, many essential features and constructs remain underrepresented in training datasets, exacerbating the coverage gap and increasing the risk of model failures in critical systems.

In this paper, we present a tool for evaluating and mitigating code-coverage problems specifically for the COBOL language. However, the principles underlying our tool are generic and can be extended to other programming languages facing similar challenges. We explore this problem in the context of developing an industry product for COBOL-to-Java transformation called watsonx Code Assistant for Z [3], where gaps in training data could lead to inaccurate code translations and hinder the migration process. Our tool aims to ensure that all COBOL constructs are adequately represented, resulting in a more robust and reliable transformation process.

In this paper, we present a tool called SYNTHGEN which addresses the CodeLLM training data coverage problem through the concept of **coverage tree**. The tool provides a graph visualization that shows all the parse trees of training data in a compact fashion merged with the grammar, allowing us to visually track what constructs are covered by the training data and which are missing in different contexts. By using this visualization, we can gain clearer insights into syntactic gaps in language coverage. In addition, we developed a statement generation technique based on the coverage tree, enabling the generation of representative code samples that can fill coverage gaps and ultimately improve CodeLLM training and performance. This work makes the following contributions:

- We developed coverage tree, a novel grammar-based compact representation of parse trees that highlights syntactic gaps in a set of programs.
- We present an algorithm to compute coverage trees, which involves grammar transformation and merging of parse trees.
- We developed a tool called SYNTHGEN that provides users with an interactive visual representation of the coverage tree. The tool enables users to easily create coverage trees for a set of COBOL programs, viewing coverage gaps, generating statements to fill coverage gaps, testing models, and updating the training dataset.
- We created several metrics related to coverage tree and evaluated the coverage of existing COBOL benchmarks.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00379

II. Approach

The COBOL-to-Java transformation model starts with a pre-trained LLM that undergoes extended pre-training (EPT) using hand-written, semantically equivalent COBOL-Java pairs. These pairs help the model learn mappings between COBOL and Java constructs. Our goal is to build a tool that identifies and analyzes coverage gaps in the EPT dataset and helps mitigate them through a compact representation of existing coverage and gaps.

Approach Summary. Essentially, our objectives are 1) to build a compact representation (called Coverage Tree) of all parse trees of COBOL programs in training data, which can show which grammar rules are used and unused at each context, e.g., which rules of the Expression non-terminal are used in the ADD statement and MULTIPLY statement separately, and 2) to generate statements from unused rules.

The architecture of SYNTHGEN is shown in Figure 1. It consists of three core stages: parsing and transformation, statement generation and program validation, and visualization. SYNTHGEN parses the source code to generate parse trees, while the input grammar (G_0) is transformed into a Kleene-style grammar (G) using * and + operators. The transformed grammar and parse trees are then processed by a Meta-Parser to construct the coverage trees. The coverage trees for multiple programs are merged to form the coverage tree for the entire dataset. Based on this structure, the Statement Generator synthesizes new statements to cover unvisited grammar rules. These statements are embedded into compilable programs using an LLM-based Program Creator. The Program Validator attempts to compile the generated program and applies corrections if necessary. The system exposes all components through an API, enabling integration with UI components such as the Coverage Tree Renderer.

Grammar Representation. A grammar G is represented as a set of *RuleSpec*s. Each *RuleSpec* contains multiple rules for a *NonTerminal*, each represented by an *Alternative*. An Alternative is a list of *Element*, representing the sequence of elements in the body of a rule. Each *Element* is either a *TermRef*, a *Block*, or a *NonTermRef* along with an occurrence symbol (?, *, +). A *Block* represents a nested set of alternatives. This grammar representation does not prohibit representing context-free grammars, but it provides occurrence symbols

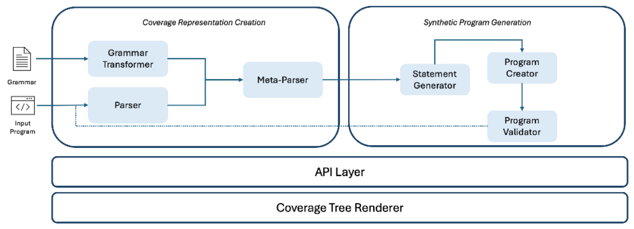

Fig. 1: SYNTHGEN Architecture

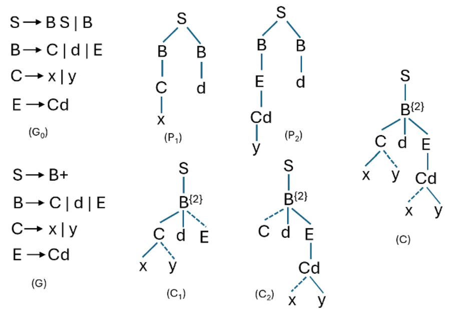

Fig. 2: Original Grammar (G_0), Transformed Grammar (G), Parse Trees (P_1 and P_2), corresponding Coverage Trees (C_1 and C_2), and merged Coverage Tree (C). $N = \{1\}$ are omitted for brevity.

to introduce a regex-style representation. Figure 2 shows an original grammar and its corresponding transformed grammar (G). Note the transformation of rules for non-terminal S.

$$\begin{aligned}
G &:= \text{Set}\langle\text{RuleSpec}\rangle \\
\text{RuleSpec} &:= \text{NonTerminal},\ \text{RuleBlock} \\
\text{RuleBlock} &:= \text{List}\langle\text{Alternative}\rangle \\
\text{Alternative} &:= \text{List}\langle\text{Element}\rangle \\
\text{Element} &:= (\text{TermRef} \mid \text{Block} \mid \text{NonTermRef})[* \mid ? \mid +] \\
\text{Block} &:= \text{List}\langle\text{Alternative}\rangle \\
\text{NonTermRef} &:= \text{NonTerminal} \\
\text{TermRef} &:= \text{Terminal}
\end{aligned}$$

Coverage Tree Definition. A *coverage Tree* is defined as an instance of Grammar G, recursively defined as follows (with non-instance nodes are required for reference to G):

$$\begin{aligned}
\text{CovereageTree} &:= \text{RuleSpecInstance} \\
\text{RuleSpecInstance} &:= RuleSpec, Set\langle NonTerminal\rangle, \\
&\quad \text{RuleBlockInstance} \\
\text{RuleBlockInstance} &:= \text{CoveredAltInstances}, \text{UncoveredAlts} \\
\text{CoveredAltInstances} &:= \text{Set}\langle\text{AlternativeInstance}\rangle \\
\text{UncoveredAlts} &:= \text{Set}\langle\text{Alternative}\rangle \\
\text{AlternativeInstance} &:= Alternative, \text{List}\langle\text{ElementInstance}\rangle \\
\text{ElementInstance} &:= (\text{TermRef} \mid \text{BlockInstance} \\
&\quad \mid \text{RuleSpecInstance}), N \\
\text{BlockInstance} &:= \text{CoveredAltInstance}, \text{UncoveredAlts} \\
\text{N} &:= \text{Set}\langle\text{Integer}\rangle
\end{aligned}$$

A coverage tree for a program is more compact than a parse tree because it uses iteration and node merging, yet more expressive than a grammar because it preserves the hierarchical structure of the parse tree.

Each parse tree is converted to the coverage tree (e.g., P_0 to C_0), which identifies which alternatives (for RuleBlock/Block) have been used to parse (using *coveredAltInstances*) and which alternatives are unused (shown in dotted line in Fig. 2)(C_1), and also identifies the number of iterations of ?, * or + (using elements of N). Note that coverage trees still preserve the context information in the parse tree (e.g., two instances of nonterminal C are separated). However, it loses track of the correlation between the alternative and specific iterations where it was taken (like children of B).

Coverage Trees are merged pairwise to form the Coverage Tree for the dataset (e.g., C_1 and C_2 to C), thereby unioning the covered alternative at every (Rule)blockInstance level and updating N.

Coverage Tree Creation. The Coverage Tree for a COBOL program is generated by a *Meta-Parser* which parses the parse tree with G. The parser accepts the strings constituted by terminals and non-terminals in the parse tree, rather than the actual program strings.

Starting from the root, for each non-terminal of the parse tree (say N), the string constituted by its list of children (say $C = (Terminal|Nonterminal)^+$) is parsed using the *RuleSpec* corresponding to the non-terminal N. This parsing is performed by a recursive algorithm based on the dynamic-programming-based CYK algorithm. The output of the parsing is a *RuleSpecInstance* with its *TermRef*s pointing to Terminals in C and *RuleSpecInstance*s (with un-instantiated) in its leaves pointing to the Non-Terminals in C in the *RuleSpecInstance*.$Set\langle$*NonTerminal*$\rangle$. Note, one *RuleSpecInstance* can point to multiple *NonTerminal*s due to * or +. As we know, the meta-parsing is not going to fail; our algorithm can perform level-by-level parsing, parsing the children of a nonterminal, without parsing the grandchildren. In the next step, for each *RuleSpecInstance*, its corresponding non-terminals in c are parsed, and the resultant *RuleSpecInstance*s are merged based on the coverage tree merging algorithm.

Statement Generation. First, we employ traversing from the user-specified uncovered node to the root of the tree. By traversing nodes from the root to the uncovered node, we identify a sequence of rules that ultimately leads us to the input node. This path defines a string A that can be constructed by following the grammar rules specified in each node. Subsequently, we traverse forward from the uncovered alternatives to the terminal nodes in the grammar. We select at most one (could be changed through configuration) iterations for */+, select random alternatives. This yields a concatenation of tokens that form another string B. By merging strings A and B, we create a new program statement that directly addresses the uncovered grammar rule.

Statement Validation. Upon generating the proposed program statement, we create a program surrounding it using an LLM and validate its correctness by a COBOL compiler.

III. User Interface

User Flow. The user interface is organized into two primary tabs: the Editor and the Coverage View. The Editor tab provides dual panes—one for entering the source snippet and the other for its target translation. Users begin by inputting source programs and optionally invoking the latest version of the CodeLLM to generate an initial translation. If the translation is incorrect or incomplete, users can manually provide the correct version and submit the pair to the tool.

Once code snippets are ingested, users can switch to the Coverage View tab to inspect the coverage tree, which visually highlights the grammar rules exercised across all submitted snippets. Users can view the uncovered rules and select them for statement generation.

For each uncovered rule, a corresponding statement can be automatically generated and is copied to the clipboard. Users can then paste multiple such statements back into the Editor. By invoking the autocomplete functionality, the tool wraps the pasted statements into a complete, compilable program structure. This synthesized code can be translated using CodeLLM, and if the output is unsatisfactory, the user may manually provide the correct translation and resubmit the improved pair to enhance overall coverage.

Visual Representation. For Coverage Tree visualization, the SYNTHGEN uses `D3.js` to render an interactive hierarchical graph. Each instance-node has a label and color indicating coverage—green for covered and white for uncovered constructs. *RuleSpecInstance* nodes show rule names; *RuleBlockInstance* and *BlockInstance* nodes use the logical OR symbol ($\vee$) for alternatives, omitted if only one exists. *AlternativeInstance* nodes use the logical AND symbol ($\wedge$) for sequences, also omitted when singular. Terminal instances appear as labeled leaf nodes. This compact structure enables users to explore grammar coverage interactively and generate tests for uncovered rules.

IV. Experimental Evaluation

We investigate three research questions:

- *RQ1*: [**Coverage**] What is the coverage of existing datasets?
- *RQ2*: [**Compactness**] The compactness achieved by Coverage Tree compared to Parse Tree?
- *RQ3*: [**Generation**] How well can we fill up coverage gaps using statement generation?

Benchmarks. We assess the coverage of the following COBOL datasets using a grammar with 1508 nonterminals and 2900 rules (Alternatives corresponding to RuleBlock):

- **IBM C2J Dataset (Old version):** An internal dataset containing 4530 COBOL code snippets curated for COBOL-to-Java translation task. The latest version is not taken for confidentiality.
- **The Stack-V2 COBOL Subset:**[1] All COBOL programs (1033) extracted from the publicly available Stack-V2 dataset used for training StarCoder model.
- **X-COBOL:** 282 programs from the X-COBOL dataset [4].

Metrics. The coverage is measured by the following metrics:

[1]https://github.com/bigcode-project/the-stack-v2

- **Rule Coverage**: Ratio of unique grammar non-terminals ($RuleSpec$) in the coverage tree to the total number defined in the COBOL grammar.
- **Alternative Coverage**: Ratio of number of grammar rule alternatives in the coverage tree to the total number of alternatives in the grammar.
- **Contextual Alternative Coverage**: Ratio of total alternative instances in the coverage tree to the total number of uncovered alternatives and alternative instances present in the tree.

Results. RQ1. Table I reports the syntactic coverage metrics on each of the datasets. The results indicate significant coverage gaps in the dataset. Some of these coverage gaps have been mitigated in later versions of C2J dataset.

RQ2. The C2J coverage tree size is shown below: *RuleSpecInstance*: 12150, *RuleBlockInstance*: 10753, *RuleRefInstance*: 12149, *BlockInstance* 5114, *AlternativeInstance*: 16646, *TerminalDefInstance*: 1288. We performed the merging of all parse trees, which led to a merged parse tree with 171943 nodes. This shows a 13-fold size reduction compared to *RuleSpecInstance* and *TerminalDefInstance* with parse tree nodes.

RQ3. We synthetically created 7001 statement samples for the IBM C2J dataset and computed the metrics for the augmented dataset, the results for which are presented in II. Augmenting the dataset with the synthetically generated samples improves the coverage of the dataset and eliminates some of the gaps in coverage.

V. Related Work

The work most closely related to ours is AsTExplainer [5], which aggregates LLM token predictions into structures called Abstract Syntax Concepts (AsC), derived from Abstract Syntax Trees. However, their method does not address coverage gaps in the training data. Other approaches [6], [7] acknowledge the issue of language coverage and incorporate grammar information during training to alleviate such gaps. Several testing methodologies [8], [9], [10], [11], [12], [13] generate test cases based on grammar coverage for validating compilers and APIs, using metrics such as rule coverage and pattern coverage. In contrast, our approach can generate training samples for the same rule in different contexts with semantic verification using LLM. In addition, our algorithm abstracts repeating rule information to create a compact version of parse trees that is easier for human consumption, which is not addressed in other grammar fuzzing-related work.

Dataset	RuleSpec %	Alts %	Alts % (context)
IBM C2J	29.6	14.67	15.88
Stack-v2	29.73	14.8	16.28
X-COBOL	26.5	13.0	15.35

TABLE I: Baseline coverage metrics for various datasets.

Dataset	RuleSpec %	Alts %	Alts % (context)
IBM C2J	33.22	17.36	16.09
X-COBOL	29.06	14.7	15.45

TABLE II: Coverage metrics Post augementation.

VI. Conclusion

We present a tool for visualizing the syntactic coverage of COBOL programs and generating synthetic statements to address coverage gaps. Central to our approach is an intuitive representation that balances the compactness of a grammar with the structural detail of a parse tree. The tool is currently deployed in IBM's application modernization initiatives and has proven effective in identifying and mitigating syntactic coverage gaps in COBOL-to-Java translation and Generative AI-based COBOL understanding projects [14]. Since the tool is internal and proprietary, its source code cannot be publicly shared. In the future, we aim to extend this tool 1) to capture semantic coverage and integrate an LLM-as-a-Judge framework for automatically evaluating the impact of coverage gaps on model performance, and 2) to employ dynamic analysis alongside static analysis to determine areas where the model underperforms.

References

[1] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," *arXiv preprint arXiv:2406.00515*, 2024.

[2] S. Joel, J. J. Wu, and F. H. Fard, "A survey on llm-based code generation for low-resource and domain-specific programming languages," *arXiv preprint arXiv:2410.03981*, 2024.

[3] A. Kumar, D. Saha, T. Yasue, K. Ono, S. Krishnan, S. Hans, F. Satoh, G. Mitchell, and S. Kumar, "Automated validation of cobol to java transformation," in *ASE*, 2024, pp. 2415–2418.

[4] M. S. Ali, N. Manjunath, and S. Chimalakonda, "X-COBOL: A dataset of COBOL repositories," *CoRR*, vol. abs/2306.04892, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2306.04892

[5] D. N. Palacio, A. Velasco, D. Rodriguez-Cardenas, K. Moran, and D. Poshyvanyk, "Evaluating and explaining large language models for code using syntactic structures," *arXiv preprint arXiv:2308.03873*, 2023.

[6] Q. Liang, Z. Zhang, Z. Sun, Z. Lin, Q. Luo, Y. Xiao, Y. Chen, Y. Zhang, H. Zhang, L. Zhang *et al.*, "Grammar-based code representation: Is it a worthy pursuit for llms?" *arXiv preprint arXiv:2503.05507*, 2025.

[7] Q. Zhu, Q. Liang, Z. Sun, Y. Xiong, L. Zhang, and S. Cheng, "Grammart5: Grammar-integrated pretrained encoder-decoder neural model for code," in *ICSE*, 2024, pp. 1–13.

[8] X. Liu, X. Li, R. Prajapati, and D. Wu, "Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing," in *AAAI*, vol. 33, no. 01, 2019, pp. 1044–1051.

[9] F. M. Kifetew, R. Tiella, and P. Tonella, "Generating valid grammar-based test inputs by means of genetic programming and annotated grammars," *Empirical Software Engineering*, vol. 22, no. 2, pp. 928–961, 2017.

[10] A. Ibias, P. Vazquez-Gomis, and M. Benito-Parejo, "Coverage-based grammar-guided genetic programming generation of test suites," in *2021 IEEE CEC*. IEEE, 2021, pp. 2411–2418.

[11] Y. Jitsunari and Y. Arahori, "Coverage-guided learning-assisted grammar-based fuzzing," in *ICSTW*. IEEE, 2019, pp. 275–280.

[12] C. Hentz, J. J. Vinju, and A. M. Moreira, "Reducing the cost of grammar-based testing using pattern coverage," in *IFIP International Conference on Testing Software and Systems*. Springer, 2015, pp. 71–85.

[13] V. Atlidakis, R. Geambasu, P. Godefroid, M. Polishchuk, and B. Ray, "Pythia: Grammar-based fuzzing of rest apis with coverage-guided feedback and learning-based mutations," *arXiv preprint arXiv:2005.11498*, 2020.

[14] S. Shah, S. Agarwal, S. Krishnan, V. Kanvar, and S. Chimalakonda, "A-cobrex: A tool for identifying business rules in cobol programs," in *ICSE-Companion*. IEEE, 2025, pp. 5–8.