Token-level Input Grammar Synthesis

1st Yunlai Luo
College of Computer, *National University Of Defense Technology*, Changsha, China
luoyl@nudt.edu.cn

***Abstract*—It is challenging to synthesize the input grammars for complex parsing programs. To address this issue, this paper proposes a novel token-based synthesis method for learning input grammars. The key idea is to synthesize the input grammar at the token level, rather than the character level, which improves both the synthesis efficiency and the grammar's completeness. Specifically, we propose using token-based symbolic execution to automatically generate valid token sequences. Then, we propose a token-level grammar synthesis method that incorporates a novel generalization operation to improve the generalization of the grammar. Additionally, we utilize SMT optimization to generalize the character representation of each token to enhance the grammar's precision. The preliminary experimental results is promising.**

***Index Terms*—Grammar, Synthesis, Token, Symbolic Execution**

I. Introduction

Software inputs are typically expected to adhere to specific formats. If an input does not comply with the format requirements, the software will typically reject it and may generate input errors. For more intricate and structured input formats, there are often input grammars that can be expressed using regular expressions or context-free grammars, such as programming language grammars. Nonetheless, there may be situations where the input grammar is not readily available [1]. Input grammars are generally considered to be the domain of software developers. Even in cases where there are descriptions of the input grammar in natural language, such descriptions are often incomplete and rarely in a machine-readable format. Furthermore, manually creating an input grammar can be a time-consuming task [2]. When the input grammar of the software is complex and not available, the automatic testing of the software becomes significantly challenging.

State-of-the-art methods for grammar synthesis [3]–[5] generally presume that a set of initial valid inputs are available. These inputs are typically provided in character-level format, such as strings or byte-level files. However, the availability of valid inputs for a program can be limited, and it may be challenging to ensure that such inputs have a representative distribution that covers all production rules and terminals in the grammar. One promising approach to address this issue is to leverage symbolic execution to generate valid character-level inputs for grammar learning [1]. This approach involves using a set of sample strings to derive a series of generalizations, but the large search space associated with character-level methods may render the learning process inefficient and limit the effectiveness of the learned grammar.

DOI reference number: 10.18293/SEKE2023-230

Our key observation is that *tokens* play a critical role in grammar. The typical parsing process involves tokenizing the input into a sequence of tokens during lexical analysis, followed by verification of the token sequence based on syntax rules during syntax analysis. As token sequences are more closely aligned with the productions in the grammar, learning the grammar at the token level could potentially lead to more efficient and effective learning, resulting in a more precise and comprehensive learned grammar.

Based on this insight, we present an approach for synthesizing input grammars by leveraging token-level symbolic execution. Unlike the existing methods, our approach is a white-box learning approach that leverages grammar-agnostic symbolic execution [6] to generate token sequences for programs with complex input formats. These generated token sequences are then used to synthesize the corresponding input grammars. To enhance the generalization process, we introduce a new operation that enables the exchange of sub-expressions within regular expressions, thereby expanding the scope of generalization. Furthermore, we propose a novel method for generating token values that utilizes Satisfiability Modulo Theories (SMT) optimization to generalize the possible character-level values of a given token, thereby improving the precision of the synthesized grammar.

II. Synthesis Framework

Our synthesis framework is illustrated in Figure 1. It is composed of two distinct phases: the token learning phase and the grammar synthesis phase. The first phase is carried out by means of grammar-agnostic dynamic symbolic execution (GADSE) [6]. Specifically, GADSE analyzes a given program in two stages. In the first stage, it performs the dynamic symbolic execution of the program's tokenization code, collecting the character-level constraints for each token value. In the second stage, GADSE symbolizes the generated tokens rather than each individual character of the input, producing the corresponding token-level path constraints. After executing the program concretely, GADSE generates a new token-level path constraint that corresponds to an alternative input grammar case, thereby enabling the generation of a new token sequence.

The second phase of our synthesis framework utilizes the token sequence and token constraints generated during the first phase to synthesize the input grammar of the program. The Grammar Synthesis phase comprises two distinct steps. Firstly, a token-level context-free grammar is learned through token-based grammar synthesis. Subsequently, the learned token-level grammar is generalized into a character-level grammar by means of token generalization.

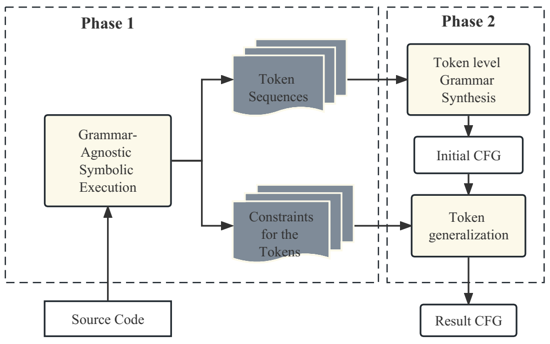
Phase 1
Phase 2
Grammar-Agnostic Symbolic Execution
Token Sequences
Constraints for the Tokens
Token level Grammar Synthesis
Initial CFG
Token generalization
Source Code
Result CFG

Fig. 1: Token-level Grammar Synthesis Framework

A. *Token-level Context-free Grammar Synthesis*

1) Regular Expression Generalization: During the first phase of synthesis, the input string is generalized into a regular expression. In contrast to the generational operations employed in GLADE, we propose a novel operation called **Exchange**, which facilitates generalization by merging the first and third parts of the regular expression if they are interchangeable. This operation enhances GLADE's generalization ability on single input. Moreover, **Exchange** ensures that the resulting generalized expression remains a valid regular expression and retains the structure of the expression prior to generalization. The generalization conforms to the following meta-grammar.

$$\begin{aligned} T_{\mathsf{rep}} &::= \beta \mid T_{\mathsf{alt}}^{*} \mid \beta T_{\mathsf{alt}}^{*} \mid T_{\mathsf{alt}}^{*} T_{\mathsf{rep}} \mid && \text{(Repetition)} \\ &\quad \beta T_{\mathsf{alt}}^{*} T_{\mathsf{rep}} \mid T_{\mathsf{exh}} \mid \beta T_{\mathsf{exh}} \mid T_{\mathsf{exh}} T_{\mathsf{rep}} \mid \beta T_{\mathsf{exh}} T_{\mathsf{rep}} \\ T_{\mathsf{alt}} &::= T_{\mathsf{rep}} \mid T_{\mathsf{rep}} + T_{\mathsf{alt}} && \text{(Alternation)} \\ T_{\mathsf{exh}} &::= (T_{\mathsf{meg}} T_{\mathsf{rep}})^{*} T_{\mathsf{meg}} && \text{(Exchange)} \\ T_{\mathsf{meg}} &::= T_{\mathsf{rep}} + T_{\mathsf{rep}} \end{aligned}$$

T_{rep} represents Kleene star, T_{alt} represents alternation, T_{exh} represents exchanging sub-expressions, and ranges over all substrings of the input string. Hence, each generalization step of regular expression can be translated into one or multiple rules in the context-free grammar.

In the context of our token-based synthesis method, each generalization setup may have multiple candidates due to the existence of multiple decompositions. However, this limitation is less severe than character-level generalization, since token sequences are typically shorter than character-level inputs. As a result, we only need to consider the decompositions between tokens, which are fewer than those between characters.

2) Context-free Grammar Generalization: The second phase of grammar synthesis involves the generalization of the regular expression obtained in the first phase into a context-free grammar. This phase comprises two steps. Firstly, the regular expression is translated into an equivalent context-free grammar by leveraging the production rules of the meta-grammar corresponding to the first stage's generalization of the regular expressions. Subsequently, we generalize the context-free grammar by merging non-terminals derived from *repetition* and *exchange* operations, which may introduce recursions in the resulting grammar. *Notably, in our approach, each token is assigned a dedicated non-terminal, and the non-terminals of tokens can also be merged during this stage*, representing an advantage of synthesizing at the token-level.

B. *Token Generalization*

Since our approach synthesizes the grammar at the token-level, the resulting grammar is also a token-level context-free grammar. To fully complete the grammar, we need to generate the character-level values for each token. This is achieved by synthesizing a grammar for each token and utilizing the constraints collected by GADSE in the first stage of our synthesis method. As a token's values can often be represented by regular expressions, we employ the first stage of our synthesis method to synthesize the seed values for each token. Furthermore, to refine the search space for token generalization, we use the token's constraints to calculate the range of each character's value through SMT optimization.

III. PRELIMINARY EVALUATION AND NEXT STEP

We have implemented our approach for Java programs using GLADE [3] and GADSE [6], with the underlying Satisfiability Modulo Theories (SMT) solver of GADSE being Z3. The effectiveness of synthesized grammars is evaluated through three metrics: precision, recall, and F1 score. The precision is determined by calculating the proportion of inputs sampled from the synthesized grammar that is accepted by the oracle grammar. On the other hand, the recall is determined by calculating the proportion of inputs sampled from the oracle grammar that is accepted by the synthesized grammar. Efficiency is measured by comparing the synthesis time of different synthesis methods. Our approach is evaluated using three real-world Java parsing programs[1][2][3] with complex and diverse input grammars. We compared our approach with two existing state-of-the-art methods: GLADE and Arvada [5]. Our approach achieves 17x and 19x higher F1 scores, respectively, on average. Furthermore, our approach achieves a high efficiency compared with both GLADE and Arvada.

The subsequent steps entails three key aspects: (1) The evaluation on extensive parsing programs; (2) The application of our approach in fuzzing or symbolic execution to enhance its efficiency; (3) The exploration of a synthesis method for context-sensitive grammars.

REFERENCES

[1] Z. Wu, E. Johnson, W. Yang, O. Bastani, D. Song, J. Peng, and T. Xie, "REINAM: reinforcement learning for input-grammar inference," in *ESEC/FSE '19*. ACM, 2019, pp. 488–498.

[2] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *PLDI '08*. ACM, 2008, pp. 206–215.

[3] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing program input grammars," in *PLDI '17*. ACM, 2017, pp. 95–110.

[4] R. Gopinath, B. Mathis, and A. Zeller, "Mining input grammars from dynamic control flow," in *ESEC/FSE '20*. ACM, 2020, pp. 172–183.

[5] N. Kulkarni, C. Lemieux, and K. Sen, "Learning highly recursive input grammars," in *ASE 2021*. IEEE, 2021, pp. 456–467.

[6] W. Pan, Z. Chen, G. Zhang, Y. Luo, Y. Zhang, and J. Wang, "Grammar-agnostic symbolic execution by token symbolization," in *ISSTA '21*. ACM, 2021, pp. 374–387.

[1] https://github.com/rindPHI/FirstOrderParser

[2] https://github.com/abcdw/javacc-clojure

[3] https://github.com/mwnorman/JSONParser