Context-Free Shuffle Languages Parsing via Boolean Satisfiability Problem Solving

Artem Gorokhov
Saint Petersburg State University
Saint Petersburg, Russia
gorohov.art@gmail.com

Semyon Grigorev
Saint Petersburg State University
Saint Petersburg, Russia
s.v.grigoriev@spbu.ru

ABSTRACT

Verification of concurrent systems is important and nontrivial problem. One of directions in this area is modeling of sequential subsystems with push-down automata (PDA) and investigating its communication. PDA is equal to context-free languages and "communication" may be expressed as shuffle of them. In this paper we consider the problem of concurrent programs' model checking from the side of context-free languages shuffle: in order to check correctness of system we should check emptiness of intersection of shuffled context-free languages (which describe behavior of the system) with regular language (which describe set of "bad" behaviors). Even in simple case, when regular language is finite, it leads to NP-complete problem and we show how it can be solved by using SAT-solvers. Our reduction is very native and use classical parsing techniques, such as Shared Packed Parse Forest and Generalized LL parsing algorithm, and some ideas from Context-Free Language reachability framework. We do not propose solution for arbitrary regular language (existence of which looks an open problem) but we show a some possible directions of research and hope that ever for restricted case proposed solution may be useful.

CCS CONCEPTS

• **Theory of computation → Grammars and context-free languages**; • **Software and its engineering → Software reliability**;

KEYWORDS

Model checking, static analysis, concurrency, shuffle, formal languages, language intersection, context-free languages

ACM Reference Format:
Artem Gorokhov and Semyon Grigorev. 2018. Context-Free Shuffle Languages Parsing via Boolean Satisfiability Problem Solving. In *Proceedings of Formal Techniques for Java-like Programs (FTfJP'18).* ACM, New York, NY, USA, Article 4, 2 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

Concurrent systems are widely spread and its verification is a nontrivial and important problem. There are a lot of papers that describe concurrent programs behavior via Push Down Systems or Context-Free languages [2–4, 9], and our interest is around a *shuffle* of Context-Free Languages (CFL) [1]. This languages describe the interleaving of CFLs (or PDA) and look perfect to describe the interleaved behavior of concurrent programs.

First of all we introduce the notion of *shuffle* operation ($\odot$), that can be defined for sequences as follows:

- $\varepsilon \odot u = u \odot \varepsilon = u$, for every sequence $u \in \Sigma^*$;
- $\alpha_1 u_1 \odot \alpha_2 u_2 = \{\alpha_1 w | w \in (u_1 \odot \alpha_2 u_2)\} \cup \{\alpha_2 w | w \in (\alpha_1 u1 \odot u_2)\}, \forall \alpha_1, \alpha_2 \in \Sigma$ and $\forall u_1, u_2 \in \Sigma^*$.

For example, $"ab" \odot "123" = \{a123b, a1b23, 12ab3, 123ab, etc.\}$.

Shuffle can be extended to languages as

$$L_1 \odot L_2 = \bigcup_{u_1 \in L_1, u_2 \in L_2} u_1 \odot u_2.$$

We can describe required aspects of behavior of functions (or methods, or subsystems) $f_1, f_2 ... f_n$ from our system $\mathcal{S}$ that run concurrently as shuffle of context-free languages $L_{f_1}, L_{f_2} ... L_{f_n}$ generated for each of them. As a result, language $\mathcal{L} = L_{f_1} \odot L_{f_2} \odot ... \odot L_{f_n}$ over alphabet Σ describes all possible executions of our system. If we want to check a correctness of $\mathcal{S}$, then we should check whether $\mathcal{L}$ contains any "bad execution". Let suppose that the set of bad executions can be described by some regular language R_1 over the same alphabet Σ. Now we should inspect an intersection $\mathcal{L} \cap R_1$ — its emptiness means that $\mathcal{S}$ can not demonstrate bad behavior.

The idea described above is used in the paper [10]. As far as shuffled context-free languages are not closed under intersection with the regular one [1] and the problem of defining either string is in the shuffle of CFL is NP-Complete, authors use a context-free approximation of shuffle of CFL and intersect it with error traces, but since the approximation was used this approach didn't found some of known bugs.

While NP-completeness may looks like death warrant, there are SAT-solvers which deal with NP problems very successfully. In this paper we show how to reduce emptiness checking of shuffled CFL and finite regular language intersection to SAT. Our reduction is very native and use some classical parsing techniques. Generalization for arbitrary regular language is a topic for future research.

2 LANGUAGES SHUFFLE TO SAT

First, we assume that R_1 is finite regular language. This is possible in assumption that the error can usually be detected in the small number of the loops iterations, so at the first step we can approximate general regular language by finite unrolling of loops. This assumption is used in bounded model checking [**?**].

We should check wheter exists $\omega \in R_1$ such that $\omega \in \mathcal{L}$. If ω exists then it should be representable as shuffle of strings $\Omega = \{\omega^i | \omega^i \in L_i, i \in 1 \dots n\}$. Our procedure trys to finde such Ω, so if our system can demonstrate bad behavior then we will not only detect this fact, but also provide an "trace" for each function which may be useful for results understanding.

The first step is creation of the regular language R_2 of all possible subsequnces of R_1: $R_2 = \{v_1 \dots v_k | \exists \{u_i | u_i \in \Sigma^*\}_{i \in 0 \dots k+1} (u_0 v_1 u_1 \dots v_k u_{k+1} \in R_1)\}$. It can be done, for example, by adding new edges in DFA M_1 which represents R_1: $E(M_2) = E(M_1) \cup \{(v_i, l_i, v_j) | (v_k, l_i, v_j) \in E(M_1)$ and v_k is reachable from v_i in $M_1\}$. In this step we should suppose that all symbols are unique. It is necessary for futher steps and may be done, for example, by extending symbol with its position. Note that $|V(M_2)| = |V(M_1)|$ and $|E(M_2)| = O(|V(M_1)|^2)$.

The next step is a calculation of $\Omega' = \bigcup\limits_{i \in 1 \dots n} L_{f_i} \cap R_2$. It is well-known that intersection of context-free language with regular one is a context-free language. Practical aspects of such intersection construction is actively We use an algorithm described in paper [5] because it provide useful representation of intersection result. This algorithm is based on Generalised LL (GLL) [7] and utilizes the Binarized Shared Packed Parse Forest (SPPF) [6, 8] for result representation. Binarized SPPF compresses derivation trees optimally reusing common nodes and subtrees, thus utilizing it for parsing forest representation grants worst-case cubic space complexity [7] which allows us to get compact formula for SAT-solver.

Binarized SPPF can be represented as a graph in which each node has one of four types described below. We denote the start and the end positions of substring as i and j respectively, and we call tuple (i, j) an *extension* of a node.

- **Terminal node** with label (i, T, j).
- **Nonterminal node** with label (i, N, j). This node denotes that there is at least one derivation for substring $\alpha = \omega[i..j-1]$ such that $N \Rightarrow_G^* \alpha, \alpha = \omega[i..j-1]$. All derivation trees for the given substring and nonterminal can be extracted from SPPF by left-to-right top-down graph traversal started from respective node.
- **Intermediate node**: a special kind of node used for binarization of SPPF. These nodes are labeled with (i, t, j), where t is a grammar slot.
- **Packed node** with label $(N \to \alpha, k)$. Subgraph with "root" in such node is one possible derivation from nonterminal N in case when the parent is a nonterminal node labeled with $(\Leftrightarrow (i, N, j))$.

The Ω' is closed to Ω but we should additionally guarantee, that each symbol uses only ones through all strings. It is required for shuffle and will be done at the next step.

!!!!!!. This fomula can be built via recursive traversal of SPPF. We convert the binary nodes to the conjunction of children, or in case of multiple derivations — alternation. Terminal nodes of the form (i, a_i, j) of m'th SPPF are to be transformed to bool variables $(i a_i^m j)$.

In addition to conjunction of formulas describing SPPFs, there are needed an expression to preserve the shuffle semantics: the terminals .... should be cosen exactly once, this grants the fact that the union of strings results a valid path in R_1. For the one path $abc\dots$ in R_1 and n given SPPFs the formula describing such condition is a conjunction of parts $(1a^1 2)$ XOR $(1a^2 2)$ XOR $(1a^3 2)$ $\dots$ $(1a^n 2)$ for each terminal.

3 CONCLUSION

We propose the way to reduce emptiness checking of intersection of shuffled CF languages with finite regular one to SAT. We show that result formula has a special structure (huge XOR subformula) which require to use XOR-SAT-solvers. We hope that our restriction on regular language is week enough to solve real tasks. To prove it it is necessary to evaluate our approach on real project.

Main question for future research is decidability of emptiness of shuffled CFL and regular language intersection. It is known that shuffled CFL is not closed under intersection with regular languages [1], but decidability of intersection emptiness is looks an open question. If it will be shown that it is undecidable in general case, then it is interesting to find subclasses for which this problem is decidable.

ACKNOWLEDGMENTS

The research was supported by the Russian Science Foundation grant 18-11-00100 and a grant from JetBrains Research.

REFERENCES

[1] Martin Berglund, Henrik Björklund, and Johanna Högberg. 2011. Recognizing shuffled languages. In *International Conference on Language and Automata Theory and Applications*. Springer, 142–154.
[2] Ahmed Bouajjani, Javier Esparza, and Tayssir Touili. 2003. A generic approach to the static analysis of concurrent programs with procedures. *International Journal of Foundations of Computer Science* 14, 04 (2003), 551–582.
[3] Sagar Chaki, Edmund Clarke, Nicholas Kidd, Thomas Reps, and Tayssir Touili. 2006. Verifying concurrent message-passing C programs with recursive calls. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 334–349.
[4] Graeme Gange, Jorge A Navas, Peter Schachte, Harald Søndergaard, and Peter J Stuckey. 2015. A tool for intersecting context-free grammars and its applications. In *NASA Formal Methods Symposium*. Springer, 422–428.
[5] Semyon Grigorev and Anastasiya Ragozina. 2017. Context-free Path Querying with Structural Representation of Result. In *Proceedings of the 13th Central & Eastern European Software Engineering Conference in Russia (CEE-SECR '17)*. ACM, New York, NY, USA, Article 10, 7 pages. https://doi.org/10.1145/3166094.3166104
[6] Joan Gerard Rekers. 1992. *Parser generation for interactive environments*. Ph.D. Dissertation. Universiteit van Amsterdam.
[7] Elizabeth Scott and Adrian Johnstone. 2010. GLL parsing. *Electronic Notes in Theoretical Computer Science* 253, 7 (2010), 177–189.
[8] Elizabeth Scott, Adrian Johnstone, and Rob Economopoulos. 2007. BRNGLR: a cubic Tomita-style GLR parsing algorithm. *Acta informatica* 44, 6 (2007), 427–461.
[9] Fu Song and Tayssir Touili. 2015. Model checking dynamic pushdown networks. *Formal Aspects of Computing* 27, 2 (2015), 397–421.
[10] Jari Stenman. 2011. Approximating the Shuffle of Context-free Languages to Find Bugs in Concurrent Recursive Programs.