Anne Koziolek, Ina Schaefer, Christoph Seidl (Hrsg.): Software Engineering 2021,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2021 49

Mining Input Grammars

Rahul Gopinath[1] Björn Mathis[2] Andreas Zeller[3]

Abstract: To assess the behavior of a program, one needs to understand its *inputs*—their sources, their structure, and how they lead to individual behavior. But as syntax and semantics of inputs are almost never completely specified, humans and computers constantly have to figure out how to produce a particular behavior.

In this abstract, we show how to automatically extract accurate, well-structured *input grammars* from existing programs. Such input grammars are useful for *software testing,* as they can serve as *producers of valid, high-quality inputs for software testing* that easily pass through parsing and validation to reliably trigger the desired program behavior. Moreover, they allow testers to *control* which inputs are to be produced—in contrast to the majority of fuzzers, that operate as black boxes.

Our Mimid prototype [GMZ20] uses dynamic tainting to *extract input grammars* from given programs—grammars that are well-structured and highly readable, even for complex recursive input formats such as JavaScript or JSON. Specific *parser-directed test generators* [Ma19; MGZ20] systematically explore the input syntax, such that grammars can be mined even without any given inputs.

Keywords: grammar; grammar mining; automated testing; fuzzing; input generation

1 Introduction

Understanding the input specification is key to understanding the program behavior, but few come with an input specification. Even if an input specification is given, it may be obsolete, incomplete, or incorrect. Given that inaccuracy in the input specification is a blind spot, it is important to obtain accurate input specifications. We describe three key techniques for recovering accurate input specifications from a given program. Our techniques work on all styles of handwritten recursive descent programs. We start by generating unbiased input samples [Ma19] that explore the input space of even complex parsers with tokenisation [MGZ20]. The execution traces of these input samples are then used to decompose the given input into parse trees, which are then combined and abstracted into a general grammar [GMZ20]. The grammars obtained are well readable and accurate.

[1] CISPA Helmholtz Center for Information Security, Saarbrücken rahul.gopinath@cispa.de
[2] CISPA Helmholtz Center for Information Security, Saarbrücken bjoern.mathis@cispa.de
[3] CISPA Helmholtz Center for Information Security, Saarbrücken zeller@cispa.de

doi:10.18420/SE2021_13

2 Grammar Mining

Mimid relies on input decomposition to construct the grammar, and uses three key insights. The first insight is that the *dynamic program dependence tree* of the input processing is structurally equivalent to the parse tree. Next, Mimid limits itself to the context-free decomposition of the input. Hence, it only tracks the input decomposition in the first level parser. Finally, any input character is consumed by the node that accesses it last. Each character is attached to the node in the dynamic control dependence tree that consumed it. The parse trees resulting from the application of these techniques are then collapsed. Each node of a parse tree corresponds to a particular expansion in the corresponding grammar where the node name corresponds to the non-terminal symbol of the expansion, and the names of the child nodes correspond to the non-terminal symbols in the expansion. Any attached characters become the terminal symbols in the expansion. All such expansions are collected, and repetition of node sequences is abstracted out to produce the final grammar.

In our evaluation, the mined grammars had an accuracy of approximately 90% as producers and as recognizers.

3 Parser Directed Input Generation

For Mimid to be effective, it needs input samples that cover every input feature. One can't rely on existing input corpus as such a corpus may not exist, and even if one exists, it may not cover all features. *Parser directed test generation* provides a solution. Our technique [Ma19] injects instrumentation to differentiate between *incorrect* and *invalid* inputs. We also track the comparisons made to the last index of the input. We start with an empty input, and extend the input with a random character if the input was found to be *incomplete*. If on the other hand, the input was found to be *incorrect*, the last character added is removed, and is replaced by one of the characters it was compared against. Proceeding in this fashion, complete valid inputs are achieved. While tokenisation of inputs can reduce the effectiveness, we show [GMZ20] that tokenisation specific instrumentation can overcome this issue.

References

[GMZ20] Gopinath, R.; Mathis, B.; Zeller, A.: Mining Input Grammars from Dynamic Control Flow. In: ESEC/FSE. artifact: https://github.com/vrtrha/mimid, Nov. 2020, URL: https://publications.cispa.saarland/3101/.

[Ma19] Mathis, B.; Gopinath, R.; Mera, M.; Kampmann, A.; Höschele, M.; Zeller, A.: Parser-Directed Fuzzing. In: PLDI 2019. June 2019, URL: https://publications.cispa.saarland/2823/.

[MGZ20] Mathis, B.; Gopinath, R.; Zeller, A.: Learning Input Tokens for Effective Fuzzing. In: ISSTA 2020. Pp. 1–11, July 2020, URL: https://publications.cispa.saarland/3135/.