Grammarinator: A Grammar-Based Open Source Fuzzer

Renáta Hodován
University of Szeged
Department of Software Engineering
Szeged, Hungary
hodovan@inf.u-szeged.hu

Ákos Kiss
University of Szeged
Department of Software Engineering
Szeged, Hungary
akiss@inf.u-szeged.hu

Tibor Gyimóthy
University of Szeged
MTA-SZTE Research Group on Artificial Intelligence
Szeged, Hungary
gyimothy@inf.u-szeged.hu

ABSTRACT

Fuzzing, or random testing, is an increasingly popular testing technique. The power of the approach lies in its ability to generate a large number of useful test cases without consuming expensive manpower. Furthermore, because of the randomness, it can often produce unusual cases that would be beyond the awareness of a human tester. In this paper, we present Grammarinator, a general purpose test generator tool that is able to utilize existing parser grammars as models. Since the model can act both as a parser and as a generator, the tool can provide the capabilities of both generation and mutation-based fuzzers. The presented tool is actively used to test various JavaScript engines and has found more than 100 unique issues.

CCS CONCEPTS

• Software and its engineering → Software testing and debugging

KEYWORDS

fuzzing, random testing, grammars, security

ACM Reference Format:


1 INTRODUCTION

Fuzzing [6], or random testing, is a popular technique as it promises the creation of a large number of test cases with limited effort. Moreover, as a result of randomness, it is often capable of generating extreme test cases that are easily overlooked by a human test engineer. Of course, there is a limit to the application area of such ‘blindly’ generated test cases as it is difficult to decide in the general case whether the program that consumed the random input executed in a semantically correct way. However, some important aspects of the execution can still be monitored, such as whether the program accessed resources (e.g., code, stack, heap memory) according to its permissions or adhered to its design contracts (e.g., assertions). This makes the technique highly useful for security testing [5].

The first fuzzers were truly random, simply feeding random byte sequences to their system-under-test (SUT). Although such simple fuzzers can reveal some surprising faults, they most often cannot scratch anything but the surface of SUTs that have a complex input structure. The test cases generated by such format-unaware fuzzers are usually caught and discarded by the input parser, so deeper code paths do not get exercised. Model-based fuzzers aim at reaching these deeper parts of the SUT by generating test cases from some model of the input format, trying to ensure that the high-level structures that get checked by the SUT’s parser are valid while the ‘real’ content gets randomized. Model-based fuzzers are usually sorted into two main categories based on how they build their model and generate test cases from it: purely generation-based fuzzers expect an explicit representation of the input format, usually written by hand, while purely mutation-based approaches need a suite of existing test cases, which act as an implicit representation of the input format and can be mutated or recombined to yield new inputs. Both approaches have their advantages and disadvantages.

The building of a model of a complex input format for a purely generation-based fuzzer may be a time-consuming, tedious task: e.g., generating random programs as test inputs for compilers or interpreters needs the modelling of syntax, symbol tables, type information, etc. On the other hand, mutation-based fuzzers require a vast number of existing test cases to work with (and they may still need additional information on where and how to mutate or recombine them).

In this paper, we present a tool named Grammarinator that aims at providing the capabilities of both generation and mutation-based fuzzers with the help of grammars. Grammarinator can turn existing parser grammars into generation models while using the same grammars to analyse and evolve existing test sets. The tool is being actively used for the testing of JavaScript engines and has found more than 100 unique defects.

The rest of the paper is organised as follows: Section 2 describes the challenges of grammar-based random test generation and how Grammarinator solves them. Section 3 reports on the results of Grammarinator used in practice. Section 4 highlights some related work from the academia and from the industry. And finally, Section 5 summarises the paper.
2 GRAMMARINATOR

The primary goal of Grammarinator\(^1\) is to be a generic grammar-based random test generator that is useful out-of-the-box for simple input formats while remaining customizable to support formats with complex requirements. Both simplicity and extensibility are achieved by adapting parsing concepts to generation purposes.

Generation from grammars. Traditional parser generator systems take a grammar and create two software components from it: a lexer to tokenize an input character stream and a parser to build an abstract syntax tree (AST) from the tokens according to the grammar rules. Then, these two components can work together to process any (correct) input. Grammarinator is designed to mirror these concepts: given a grammar, it creates two software components that are capable of generating input. To highlight the mirroring of the traditional concepts, the token generator part is called unlexer while the syntactic structures are generated by an unparser. Their combined result is an AST, which can be serialized into an actual test case.

The existence of grammars is key to the usefulness of this approach. To facilitate putting in practice, Grammarinator uses the widespread ANTLR v4 grammar syntax to model the input format. The central grammar repository of the ANTLR v4 tool\(^2\) and the more than 150 open-source grammar definitions therein save a lot of modelling effort and open up the possibility of fuzz testing of a large number of SUTs.

Figure 1 gives a high-level overview of the workflow of Grammarinator: the top-left part depicts the (one-off) creation of the generator components and the right-hand side of the diagram visualizes the generation of ASTs and their serialization to test cases (some yet-undiscussed parts of the chart will be detailed later).

Listing 1 shows an example grammar of simple additive expressions and variable assignments (which can also be seen as an extremely restricted subset of JavaScript), while Listings 3 and 4 give excerpts from the Grammarinator-generated Python 3 sources of the unparser and unlexer. The code snippets show how the parser and lexer rules of the grammar are mapped to methods (e.g., defined at Lst. 3 line 17 and Lst. 4 line 36, used at Lst. 3 lines 26 and 77), how

\(^1\)https://github.com/renatahodovan/grammarinator or pip install grammarinator

\(^2\)https://github.com/antlr/grammars-v4

the AST is built (e.g., child node created and added to its parent at Lst. 3 line 25), how rule alternatives are handled (e.g., selected at Lst. 4 lines 38, 40, and 42), and how quantifiers are dealt with (e.g., the Kleene plus at Lst. 3 line 22). Thus, to give more control over the generated content, Grammarinator also borrows the concepts of rule element labels, actions, and predicates from the ANTLR v4 syntax. Labels can capture generated subtrees for later reuse, actions allow the injection of arbitrary code but it will be probably discarded early on by a hypothetical SUT). To highlight the mirroring of the forbidden use of undefined identifiers, it is clear that the generated test case is not perfect (i.e., its structure is seemingly correct but it will be probably discarded early on by a hypothetical SUT). Thus, to give more control over the generated content, Grammarinator also borrows the concepts of rule element labels, actions, and predicates from the ANTLR v4 syntax. Labels can capture generated subtrees for later reuse, actions allow the injection of arbitrary code into the generators, while predicates can guard alternatives during generation (also with the help of injected expressions).

The grey additions to the grammar in Listing 1 and their counterparts in the unparser and unlexer in Listings 3 and 4 exemplify how these constructs can be used to avoid undefined identifier references in generated test cases. I.e., at the beginning of the generation of a
Listing 3: Excerpt from unparsers

```python
from grammarinator.runtime import *
from random import sample

class ExampleUnparser(Grammarinator):
    def __init__(self, unlexer):
        super(ExampleUnparser, self).__init__()
        self.unlexer = unlexer
        @depthcontrol
    def program(self):
        local_ctx = dict()
        current = self.create_node(UnlexerRule(name='program'))
        self.symtab = set()
        if self.unlexer.max_depth == 0:
            for _ in self.one_or_more():
                current += self.unlexer.ID()
                local_ctx['vardef'] = current.last_child
                current += self.create_node(UnlexerRule(src='='))
                current += self.expression()
                current += self.create_node(UnlexerRule(src=';'))
                self.symtab.add(str(local_ctx['vardef']))
        return current

    def value(self):
        current = self.create_node(UnlexerRule(name='value'))
        choice = self.choice([0, 1])
        if self.unlexer.max_depth else w + self.unlexer.weights.get((alt_25, 0, 1)) for i, w in enumerate([1, len(self.symtab) != 0, 1])]
        self.unlexer.weights((alt_25, choice)) = self.unlexer.weights get((alt_25, choice))
        alt_25, choice, 1) * self.unlexer.cooldown
        if choice == 0:
            current += self.create_node(UnlexerRule(src='='))
            current += self.expression()
            current += self.create_node(UnlexerRule(src=';'))
        elif choice == 1:
            current += self.unlexer.ID()
            current.last_child = UnlexerRule(src='=sample(self.symtab, 1)[0]')
        elif choice == 2:
            current += self.unlexer.NUM()
        return current
    value.min_depth = 1

Listing 4: Excerpt from unlexer

```
branches. The listings show an example to guide numeric literals away from zero by explicitly giving a small weight in a predicate (Lst. 1 line 9, Lst. 4 line 38).

Reusing existing test cases. The advantage of grammars as fuzzers is that they can be used to parse existing – and typically both syntactically and semantically correct – test cases. This enables Grammarinator to build a pool of ASTs and use them to boost purely generation-based test case creation with evolutionary techniques. One possibility is to perform a random recombination on trees from the pool to produce new test cases. But the generation and recombination-based approaches may also be fused together by taking trees from the pool and mutating them by replacing some parts with randomly generated new subtrees. The bottom-left part of Figure 1 depicts the place of these techniques in the workflow of Grammarinator.

3 RESULTS

Grammarinator was used to generate test inputs in ECMAScript format for the JerryScript engine, using the grammar available from the ANTLR v4 repository and applying the techniques described above. Depth control was used to keep a check on the size of the generated test cases, edge weight cooling was ensuring the diversity of the inputs, predicates and actions were injected to help meet the symbolic (e.g., no reference before definition) and structural (e.g., no loop control outside loops) requirements of the language, and transformers were adding whitespaces to the output. The conformance and regression test suites of the engine were used to build a population of trees for recombination and mutation. Guided by the Fuzzinator framework—which manages the passing of test cases from the fuzzier to the engine, the detection of erroneous behaviour, the classification of issues as unique or duplicate, and the minimization of failure-inducing inputs [1]—Grammarinator has triggered 89 unique issues confirmed as valid by the JerryScript project team.

The tool was also used to fuzz the IoT.js project, a modular event-based JavaScript platform powered by the JerryScript engine. Although format (i.e., JavaScript) processing is delegated to JerryScript and IoT.js does not perform a lot of processing tasks itself, Grammarinator has revealed 15 unique issues in the platform, too.

All found issues have been reported in the issue trackers of the projects and are also listed in the wiki of Grammarinator.

4 RELATED WORK

The idea of using grammars as test generation models has several decades of history. As early as in the 70s, Purdom [7] experimented with testing parser programs with test cases generated from context-free grammars. Later in the 90s, Maurer [5] used context-free grammars to generate test cases for VLSI circuits.

One of the closest approach to our work is LangFuzz [2] by Holler et al. It also parses existing regression tests with ANTLR grammars and caches all the subtrees into a fragment pool to be used for recombination during generation. It also aims to use the grammars as a generator model, however, it supports only a subset of the ANTLR feature set. More complex grammar constructs like quantifiers or sub-alternatives are not supported and require manual grammar transformations. Moreover, the prototype tool of this research is not available as open source.

Kifetew et al. [3] use stochastic context-free grammars in BNF format as test generation model. The probabilities of alternatives are gathered automatically from existing test cases to promote valid sentence structures and to control recursion depth. Additionally, they support evolutionary operators. In a follow-up work [4], they also discuss manual annotations. Unfortunately, their prototype tool was not available for public use.

A commercially utilized model-based fuzzing approach is that of Peach. It has its own XML-based model format used only by the Peach Fuzzer. Additionally, because of its commercial focus, its open community edition has not been maintained since 2014.

5 SUMMARY

In this paper, we presented an open source tool, named Grammarinator, which is able to generate syntactically correct test cases from grammars. But, to be able to deal with not only the simplest cases, it can do more. It can limit the depth of generated structures and direct the generation toward less visited paths. By inlining code snippets into the grammar, it allows to define complex actions and decisions that a context-free grammar would not be able to describe. And with the help of tree transformers, it can manipulate test cases arbitrarily. Finally, Grammarinator can also exploit the fact that the same grammar that can generate new tests can also be used to parse existing test suites, and then create new content resulting from their recombination or mutation.

The tool has proven its usefulness in the hardening of real-life projects by revealing more than 100 valid unique issues.

ACKNOWLEDGMENTS

This research was partially supported by the Ministry of Finance of Hungary under EU-supported Hungarian national grant GINOP-2.3.2-15-2016-00037.

REFERENCES