UNIVERSITÄT
DES
SAARLANDES

Grammar-Based Fuzzing Using Input Features

A dissertation submitted towards the degree
Doctor of Engineering (Dr.-Ing.)
of the Faculty of Mathematics and Computer Science
of Saarland University

by
Nikolas Havrikov

Saarbrücken, 2021

Day of Colloquium	March 4, 2022
Dean of the Faculty	Univ.-Prof. Dr. Thomas Schuster
Chair of the Committee	Prof. Dr. Jan Reineke
Reporters	
First Reviewer	Prof. Dr. Andreas Zeller
Second Reviewer	Prof. Dr. Lars Grunske
Third Reviewer	Prof. Dr. Thorsten Holz
Academic Assistant	Dr. Dominic Steinhöfel

Dedication

To *you*, dear reader!
Please consider doing some push-ups if you can.

Acknowledgements

I want to take this opportunity to thank the people who have helped me along the way to get to where I am today.

First, I would like to thank my advisor Andreas Zeller, who has proven to be a benevolent force always guiding me in the right direction and encouraging me with much gravitas throughout my struggles.

I would also like to thank the members of the committee Lars Grunske and Thorsten Holz who graciously agreed to undertake the task of review.

I am immensely grateful to Konstantin Kuznetsov, who was always ready to support and assist with advice and action. And although we do not seem to share much work on the record, trust me when I say that without his collaboration, I would not have been able to accomplish what I did.

A big shout out goes to Alexander Kampmann, as by working together on numerous projects we both have achieved considerable growth as engineers by bouncing ideas and designs off of each other.

I would like to thank my colleagues and postdocs at the group for their company during these trying times: Matthias, Florian, Eva, Clemens, Kevin, Andrey, Vitalii, Marcel, JP, Alessandra, Sudipta, Alessio, Maria, Konrad, Andreas, Jenny, Sascha, Nataniel, Ezekiel, Michaël, Max, Björn, Johannes, Marius, Leon, Rafael, Rahul, and Dominic.

I would also like to acknowledge my students, whose theses I have supervised, for in teaching them, I myself have also learned a lot about motivation, discipline, and perhaps more importantly, self-discipline.

Big thanks to Gordon Bolduan, a great storyteller, whose wordsmithing skills I hope to have absorbed at least a little bit. I especially appreciate him continuously indulging my sarcastic side, and yet somehow us still remaining friends.

Curd Becker deserves special appreciation as the best and friendliest administrator of computer systems under the Sun.

I want to convey my deepest thanks to my longtime friends Thomas Konietzke and Dennis Kamkar, who have helped me relieve stress throughout our countless gaming sessions while celebrating wins and commiserating losses. It is those two who had to sit there and listen to me rant and vent the most.

I would like to express gratitude to all my friends at our theater club *Thunis*, thanks to whom I became a better public speaker and obtained much needed self-confidence. Our rehearsals, plays, and projects shaped me to become a more sociable and outgoing person.

I once met Julia Rubin at the ASE conference in 2019, at a point when I was hesitating about continuing the PhD, and she randomly made me promise not to abandon it, so... uhm... here I am.

I owe a big thanks to my aunt Olga who effectively encouraged me to begin the PhD program in the first place by saying something along the lines of "you can just start and need not necessarily finish". This thought of non-obligation has turned out to be very calming at times, and has generally helped me ease into the process.

And finally, my eternal thanks goes out to my dear parents have no clue whatsoever as to what I am doing, but who have been nevertheless most supportive throughout, and thus they get to be rightfully disproportionately proud!

Abstract

In grammar-based fuzz testing, a formal grammar is used to produce test inputs that are syntactically valid in order to reach the business logic of a program under test. In this setting, it is advantageous to ensure a high diversity of inputs to test more of the program's behavior.

How can we characterize features that make inputs diverse and associate them with the execution of particular parts of the program? Previous work does not answer this question to satisfaction, with most attempts mainly considering superficial features defined by the structure of the grammar such as the presence of production rules or terminal symbols, regardless of their context.

We present a measure of input coverage called *k*-path coverage, which takes into account combinations of grammar entities up to a given context depth *k*, and makes it possible to efficiently express, assess, and achieve input diversity.

In a series of experiments, we demonstrate and evaluate how to systematically attain *k*-path coverage, how it correlates with code coverage and can thus be used as its predictor. By automatically inferring explicit associations between *k*-path features and the coverage of individual methods we further show how to generate inputs that specifically target the execution of given code locations.

We expect the presented instrument of *k*-paths to prove useful in numerous additional applications such as assessing the quality of grammars, serving as an adequacy criterion for input test suites, enabling test case prioritization, facilitating program comprehension, and perhaps beyond.

Zusammenfassung

Im Bereich des grammatik-basierten Fuzz-Testens benutzt man eine formale Grammatik, um Testeingaben zu produzieren, welche syntaktisch korrekt sind, mit dem Ziel die Geschäftslogik eines zu testenden Programms zu erreichen. Dafür ist es vorteilhaft eine hohe Diversität der Eingaben zu sichern, um mehr vom Verhalten des Programms testen zu können.

Wie kann man Merkmale charakterisieren, die Eingaben vielfältig machen und diese mit der Ausführung bestimmter Programmteile in Verbindung bringen? Bisherige Ansätze liefern darauf keine ausreichende Antwort, denn meistens betrachten sie oberflächliche, durch die Grammatikstruktur definierte Merkmale, wie das Vorhandensein von Produktionsregeln oder Terminalen, unabhängig von ihrem Verwendungskontext.

Wir präsentieren ein Maß für Eingabeabdeckung, genannt *k*-path Abdeckung, welche Kombinationen von Grammatikelementen bis zu einer vorgegebenen Kontexttiefe *k* berücksichtigt und es ermöglicht, die Diversität von Eingaben effizient auszudrücken, zu bewerten und zu erzielen.

Mit Experimenten zeigen und evaluieren wir, wie man gezielt *k*-path Abdeckung erreicht und wie sie mit der Codeabdeckung zusammenhängt und diese somit vorhersagen kann. Ferner zeigen wir wie automatisches Erlernen expliziter Assoziationen zwischen Merkmalen und der Abdeckung einzelner Methoden die Erzeugung von Eingaben ermöglicht, welche auf die Ausführung bestimmter Codestellen abzielen.

Wir rechnen damit, dass sich *k*-paths als ein vielseitiges Instrument beweisen, dessen Anwendung über solche Gebiete, wie z.B. Messung der Qualität von Grammatiken und Eingabe-Testsuiten, Testfallpriorisierung, oder Erleichterung von Programmverständnis, hinausgeht.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the domain of software testing, one of the main approaches to ensuring decent quality of a program is so called *fuzz testing* or *fuzzing* for short. Since its invention by the father of fuzzers Barton Miller [87] in the early nineties, this very simple technique of testing programs with randomly generated inputs has been widely adopted by both academia and industry and has seen a lot of improvements over the years with no sign of stopping any time soon. Its areas of application now span multiple domains ranging from command-line utilities to monolithic backend services to swarms of microservices to smart devices to generic user-interfaces.

In principle, fuzzing is a very cost-effective means to test programs for robustness: if a program has not been subjected to random inputs before, chances are high that some input will cause it to misbehave.

However, modern programs tend to have much more robust handling of their inputs, and the simple trick of feeding them random garbage will no longer work. Nowadays, it is easy to use readily-available, established validation techniques to ensure that any input which does not adhere to the expected form is simply discarded before its malformed contents can reach any of the actual business logic and cause harm.

On the other hand, fuzzing itself has neither become obsolete, nor did it stagnate. Just as software evolved to better deal with structural errors in its inputs by means of input validation, fuzzers have, in turn, taken on the burden of reaching and thereby also hardening as much of the business logic as possible. In order to be able to reach this logic though, a fuzzer must be capable of producing inputs that pass the validation criteria of the program under test.

One such common, very basic validation criterion is the syntactic validity, stating that the input must adhere to a specific *formal language*. To produce inputs that meet this requirement, both established and novel fuzzing approaches make use of *formal grammars* which specify the language of program inputs. Given a grammar, these *grammar-based* fuzzers are able to confidently produce syntactically valid inputs that pass the structural validation of the program's input parser and thus reach deeper within the business logic.

```
Expr := AddExpr;
AddExpr := MultExpr
        | AddExpr ("+" | "-") MultExpr;
MultExpr := UnaryExpr
        | MultExpr ("*" | "/" | "%") UnaryExpr;
UnaryExpr := Identifier
        | "+" UnaryExpr
        | "-" UnaryExpr
        | "++" UnaryExpr
        | "--" UnaryExpr
        | "(" AddExpr ")"
        | DecDigits;
DecDigits := DecDigit+;
DecDigit := "0" | "1" | "2" | "3" | "4"
        | "5" | "6" | "7" | "8" | "9";
Identifier := "x" | "y" | "z";
```

Figure 1.1 / Grammar for a subset of arithmetic expressions in the JavaScript programming language (excerpt). Please consider navigating to Appendix B, which contains a copy of this figure, and printing it out on an extra sheet to have it at hand for easy reference.

In a few words, a grammar-based fuzzer works by expanding a *start symbol* into further symbols, which it expands repeatedly, following the productions in the grammar and selecting from alternatives, until only terminal symbols are left. These terminals build up the resulting input string produced by the fuzzer.

For an introductory example consider the grammar given in Figure 1.1, which describes a small subset of arithmetic expressions in the language JavaScript. It allows expressions over integers and three variables. A fuzzer could, for instance, expand the start symbol `Expr` into an `AddExpr` and then a `MultExpr`, which, in turn, would expand into a `UnaryExpr`, which would eventually become a string of digits such as `"42"`.

1.1 Research Problem

While the concept of producing inputs from grammars is simple, any practical application faces several problems. One such issue is ensuring that an input does not grow beyond bounds. In Figure 1.1, if the fuzzer always selects the last expansion alternative, the result will be an infinitely long arithmetic expression because there is no way to terminate the recursion in `AddExpr` or `MultExpr`.

However, infinity is not the only bound to avoid: When the fuzzer generates inputs that are very large, there is still no guarantee of their usefulness. More likely than not, the law of diminishing returns applies to the behavior that can be reached in the program under test. Even worse if the runtime scales at least linearly with the size of the input, which is not too unrealistic of an expectation.

```
DecDigit := S0;
S0 := "0" | S1;
S1 := "1" | S2;
S2 := "2" | S3;
      ...
S8 := "8" | S9;
S9 := "9";
```

Figure 1.2 / Deeply Nested Grammar. A naïve random generator will get "stuck" early. The chances of it reaching the terminal `"9"` are very low in a uniform selection setting.

A fuzzer thus needs some means to determine which expansions to choose in order to avoid such growth.

Another issue of generating from grammars is ensuring input coverage. Intuitively, a high variation in the inputs (say, using different operators in our JavaScript example) induces a high variation in program behavior. Conversely, if some specific element is not present in the input (say, the `"+"` operator), the part of the code that is responsible for processing it will not be executed. It is therefore desirable to cover as many different input elements and productions as possible. Applied to our expression grammar, this would require us to cover all operators, identifiers, and digits.

How does one efficiently achieve high input coverage? One way, suggested by Purdom [105] is to ensure that during generation, uncovered production alternatives would be preferred over covered production alternatives. In Figure 1.1, we would first expand `AddExpr` into the first alternative (`MultExpr`), and the next time into the second alternative. Likewise, once we have covered the `"+"` alternative, we would go for the `"-"` alternative the next time. Over time, Purdom's approach would cover all alternatives.

Unfortunately, there exist grammars for which neither random generation nor Purdom's approach succeed in achieving coverage. This becomes apparent when we reformulate the `DecDigit` rule as shown in Figure 1.2. Now, the digits are no longer uniformly chosen and produced.

When choosing a `DecDigit` expansion using a naïve random approach, the probability of generating a `"0"` is 0.5, the probability of generating a `"1"` is $(1 - 0.5) \times 0.5 = 0.25$, then it is $(1 - (1 - 0.5) \times 0.5) \times 0.5 = 0.125$ for a `"2"`, and so forth, up to 0.000 976 562 5 or 1/1024 for a `"9"`.

Purdom's approach helps a bit, but is far from perfect; in the first expansion of `S0`, it would mark `"0"` as covered, then expand `S1` the next time. However, after having both covered `"0"` and `S1`, it would no longer prefer one over the other, still yielding `"0"` in 50 % of expansions.

If Figure 1.2 feels too artificially constructed, consider Figure 1.3, listing possible rules for identifiers in JavaScript. Not only does this form result in 50 % of identifiers consisting of one character only, but both `Identifier` alternatives would be quickly marked as covered by Purdom's approach, giving the fuzzer no incentive to systematically cover identifier characters or their categories.

```
Identifier := Character | Character Identifier;
Character := ASCIICharacter | UnicodeCharacter;
ASCIICharacter := ASCIIUpper | ASCIILower | "_";
```

Figure 1.3 / An `Identifier` rule that is hard to cover fully (excerpt from a real JavaScript grammar).

The deeper the grammar, the greater the extent of this coverage problem.[1] Therefore, improvements are highly desirable over the current state of the art in grammar-based input generation, in order to better cope with challenges such as deeply nested grammars, and to ensure overall effective and adequate grammar coverage of the generated test suites.

Yet another problem of grammar-based fuzzing has to do with performance: Modern fuzzers can generate countless inputs very efficiently, creating valid inputs by mutating given samples and/or leveraging language descriptions, or using guidance from program execution to cover as much code as possible. In practice, however, the effectiveness of a fuzzer heavily depends on the execution time of the program under test. If processing an input takes a millisecond, then running a million fuzzing inputs is no big problem. If it takes seconds though, then fuzzing with many inputs can quickly become infeasible.

As an example, consider a hypothetical compiler that takes JavaScript code as its input and produces an optimized version of it. Let us further assume that this compiler has an optimization pass which has a method `distributive_rule()` that splits an expression of the form $a \times (b + c)$ into $a \times b + a \times c$ such that the subterms can be optimized individually. If our goal is to focus the fuzzing efforts on this method while still executing it as part of the overall compilation, we must guide the generation of new inputs with the desired expression in a precise manner or to enable accurate selection from available inputs.

To accomplish these goals, it is desirable to leverage a readily available grammar to learn associations between features of inputs and locations in the code these inputs are reaching. Once learned, such associations can improve the performance of a fuzzer by serving as efficient oracles, helping to portion its efforts on the relevant parts of the program under test.

This dissertation approaches the above issues based on a concept of *input features*, and makes the following contributions to grammar-based fuzzing:

Input features. First and foremost, this dissertation introduces a family of novel input features called k-paths, which allow expressing the degree of diversity that a given input attains by accounting for the number of combinations of grammar elements an input contains. The k-paths further provide the basis for the definition of a grammar coverage measure.

Grammar coverage. This definition of grammar coverage gives rise to an algorithm which ensures quick coverage of all input features. Specifically, it

[1] Technically, Purdom's algorithm cannot handle grammar productions containing alternatives as it expects them to be separate productions with the same left-hand side. While it is possible to rewrite productions accordingly, in practice, this results in large and not well readable grammars.

chooses expansion alternatives that lead to input elements not yet covered. Applied on Figure 1.2, its first ten productions consist of the elements `"0"` to `"9"`, and on Figure 1.3, it produces long identifiers that cover all valid `Character` elements.

Combination coverage. The algorithm also ensures coverage of combinations of grammar elements, whose context size is controlled by a single parameter k. When applied on Figure 1.1, for instance, it produces additions within multiplications, multiplications within additions, unary minuses within parentheses, and all sorts of combinations one would want to test, e.g., in an optimizing compiler. The presented algorithm also effectively prevents inputs from growing beyond bounds.

Coverage associations. Using a grammar as a parser allows determining the features of a given input and associate them with coverage of individual methods in the program under test, resulting in a set of classifiers that model the relationship between input elements and resulting coverage. These associations are explicit, user-facing, and reusable. For example, for `distributive_rule()`, an associated feature could be the presence of an addition as part of a multiplication.

Targeted fuzzing. The obtained classifiers can be used to construct inputs that satisfy their conditions and thus are set to cover the method in question. A fuzzer can thus target individual methods without requiring guidance from the program, which is especially useful if executing the program is expensive.

Coverage prediction. The learned classifiers can take an arbitrary input (given or generated), parse it into its elements, and predict the code coverage from the input features only. This yields a very cost-effective method for selecting inputs that cover a particular method.

1.2 Thesis Structure

The remainder of this dissertation is structured as follows:

Chapter 2 begins by giving a minimum necessary background on context-free grammars and then proceeds to introduce a practical grammar notation, a corresponding grammar graph representation, and finally a notion of grammar coverage called k-path coverage.

Chapter 3 then builds on the concept of k-paths and presents an algorithm that allows systematically and efficiently producing a set of inputs that achieve full k-path coverage while at the same time effectively avoiding boundless growth to combat the problem of bloat.

Chapter 4 explores a variant of the generation algorithm from the previous chapter allowing us to produce multiple input sets over an entire range of k-path coverage levels. This rich data set enables us to explore the nature of the relationship between k-path coverage and code coverage, which, in turn, motivates additional use cases for k-paths.

Chapter 5 demonstrates how to leverage k-paths to learn associations between input features and code locations. These associations are explicitly materialized in the form of machine learning classifiers, which are consequently used for predicting the code coverage of given inputs and for generating

inputs that are aimed at the reaching code locations of interest without involving the execution of the program under test.

Chapter 6 gives an overview of relevant fuzzing approaches related to grammar-based and machine-learning-based techniques and provides commentary comparing them with the approaches presented in this dissertation.

Chapter 7 concludes and gives an outlook on future research directions that are enabled by the approaches presented in this dissertation.

1.3 List of Publications

This section gives a brief overview over the publications that I (co-)authored and outlines my contributions to them. While all these publications influenced my work to one degree or another, not all of them find a direct representation in this thesis. Publications marked with ⋆ are the ones that build up the backbone of this dissertation.

- **Nikolas Havrikov**, Alessio Gambi, Andreas Zeller, Andrea Arcuri, and Juan Pablo Galeotti. "Generating Unit Tests with Structured System Interactions." In: *12th IEEE/ACM International Workshop on Automation of Software Testing, AST@ICSE 2017, Buenos Aires, Argentina, May 20-21, 2017*. IEEE Computer Society, 2017, pp. 30–33. DOI: 10.1109/AST.2017.2.

  This workshop paper explores the integration of my specification-based input generator [49] in the context of search-based unit testing as implemented by EvoSuite [33]. It demonstrates the benefits of a grammar-like generation approach (in this case based on an XML Schema) as opposed to traditional stubbing or mocking [34] for overcoming environmental requirements of a program under test.

- ⋆ **Nikolas Havrikov**. "Efficient Fuzz Testing Leveraging Input, Code, and Execution." In: *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*. IEEE Computer Society, 2017, pp. 417–420. DOI: 10.1109/ICSE-C.2017.26.

  In this doctoral symposium submission, I raise the initial questions about the feasibility and efficiency of a grammar-based approach to fuzzing, which have eventually matured over time and guided this dissertation.

- ⋆ **Nikolas Havrikov** and Andreas Zeller. "Systematically Covering Input Structure." In: *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 2019, pp. 189–199. DOI: 10.1109/ASE.2019.00027.

  In this work, I introduce the concept of k-paths and the k-path-algorithm for efficiently generating inputs from a context-free grammar. At this time I have also begun gathering an extensive set of real-world test subjects for performing practical empirical evaluations, which addresses one of the weak points of fuzzer evaluation I point out in my previous publication.

- Ezekiel Soremekun, Esteban Pavese, **Nikolas Havrikov**, Lars Grunske, and Andreas Zeller. "Inputs from Hell: Learning Input Distributions

for Grammar-Based Test Generation." In: *IEEE Transactions on Software Engineering* (2020). DOI: 10.1109/TSE.2020.3013716.

In this publication, which evaluates data-driven, probabilistic grammar-based generation, I have contributed to the discussion about the design of the probability acquisition and inversion mechanisms, as well as the implementation and test subjects. This study has greatly benefitted from the strong technical foundation and subject collection efforts introduced in my previous publication.

- Alexander Kampmann, **Nikolas Havrikov**, Ezekiel O. Soremekun, and Andreas Zeller. "When Does my Program do This? Learning Circumstances of Software Behavior." In: *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. ACM, 2020, pp. 1228–1239. DOI: 10.1145/3368089.3409687.

  This paper introduces an approach to automated reasoning about program behavior, based on features of its inputs. By leveraging a grammar, an input can be both deconstructed into its parts and (re-)generated according to desired constraints. This enables the use of decision trees to automatically learn and refine theories explaining a given behavior, provided an oracle, of course. My contributions comprise the technical implementation of a grammar representation, software design considerations, code reviews, and the addition of a complex, real-world test subject. This publication represents my first foray into applying machine-learning techniques in the context of test input generation, which later led me to elaborate on the mapping of input features to locations in code.

- Rahul Gopinath, Alexander Kampmann, **Nikolas Havrikov**, Ezekiel O. Soremekun, and Andreas Zeller. "Abstracting Failure-Inducing Inputs." In: *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*. ACM, 2020, pp. 237–248. DOI: 10.1145/3395363.3397349.

  This publication introduces the DDSET algorithm, which is able to deduce an abstract failure-inducing input pattern, given a grammar and a failing input to start from. I am very proud to have contributed to the idea of this best-paper award winning publication. In fact, the pattern aspect of this approach has inspired me to investigate the possibility of manifesting input features as human-readable entities.

- ⋆ **Nikolas Havrikov**, Alexander Kampmann, and Andreas Zeller. "From Input Coverage to Code Coverage: Systematically Covering Input Structure with k-Paths." In: *ACM transactions on software engineering and methodology* (2021). TOSEM-2021-0220. ISSN: 1049-331X. submitted.

  In extension of previous work, *k*-paths and decision tree learning are combined to leverage grammar-based features to produce human-readable conditions under which methods under test can be reached. This allows generating relevant tests with pinpoint accuracy even on the system level. A fortunate side effect is the prediction capability granted by the learned classifiers, allowing to easily select from a given set of test inputs the

ones that are of interest. My contributions include the conception of the approach, the extension of the technical base to support *k*-paths, the implementation of a parallel evaluation pipeline, the collection of test subjects and grammars.

Chapter 2

Structural Input Features

This chapter introduces the concept of k-paths, which lay the foundation for addressing the problems outlined in the introduction. The following chapters will build on this concept and demonstrate its application to fuzzing, as well as achieving high input and code coverage.

2.1 Context-Free Grammars

A concept that is perhaps most central to this dissertation is the concept of formal grammars, or more specifically, context-free grammars. A context-free grammar is a formalism to define a context-free language as introduced by Chomsky [23]. It is often formally described as a tuple (N, T, P, s), where N is the set of non-terminal symbols, T is the set of terminal symbols, which are distinct from the non-terminal symbols. P denotes the set of productions of the form `A := X`, where $A \in N$ and $X \in (N \times T)^*$. Finally, $s \in N$ is the so-called start symbol. All words belonging to the language described by the grammar can be obtained from the start symbol by repeatedly rewriting non-terminals according to matching productions in a process called deriving or expanding.

For example, the following is a grammar describing the language whose words consist of n contiguous "a" characters, followed by a single "b", followed by n contiguous "c" characters, for $n \geq 0$.

```
A := "b"
A := "a" A "c"
```

In this example, the four components of the tuple that define the grammar have the following values:

- $N = \{A\}$
- $T = \{a, b, c\}$
- $s = A$
- $P = \{A := b,\ A := a \times A \times c\}$

Oftentimes, grammars are written in the Backus-Naur form (BNF), which allows combining multiple productions that share the same left-hand side non-terminal into so-called alternations by means of the "|" character. BNF allows us to rewrite the above example more concisely as follows:

```
A := "b" | "a" A "c"
```

For practical reasons, it often makes sense to assume that the start symbol does not occur anywhere on the right-hand side of any production. Any grammar is easily re-written such that this holds by introducing an additional production $s' := s$. In our example this transformation looks as follows:

```
Start := A
A := "b" | "a" A "c"
```

This formal tool of expressing languages through grammars gives us a solid base from which to start our consideration of grammar-based input generation for our use case of fuzz testing.

2.2 Practical Grammar Notation

The notation for context-free grammars as presented in Section 2.1 is feature-complete, but not very comfortable to work with in practice. Therefore, in the interest of readability and engineering practicality, this section introduces additional grammatical constructs, inspired, among others, by the domain-specific language of the popular parser generator ANTLR [100]. To demonstrate this new notation to its full extent, let us revisit some aspects of the grammars we have seen before.

2.2.1 Inline Alternatives

Consider the following fragment from the JavaScript grammar excerpt from Figure 1.1. Note that apart from the top-level alternations representing the possible derivations of the non-terminals, there are also alternations inside the parenthesized expressions that are part of the concatenations.

```
AddExpr := MultExpr
         | AddExpr ("+" | "-") MultExpr;
MultExpr := UnaryExpr
         | MultExpr ("*" | "/" | "%") UnaryExpr;
```

This parenthesized inline notation lends itself well to a human reader because it does not interrupt the flow of reading – in this case, by introducing additional non-terminals to capture the different operators, like this:

```
AddExpr := MultExpr
         | AddExpr AddOp MultExpr;
AddOp := "+" | "-";
MultExpr := UnaryExpr
         | MultExpr MultOp UnaryExpr;
MultOp := "*" | "/" | "%";
```

However, by failing to explicitly designate the concept of operators, this notation does not make them entities in their own right when it comes to grammar coverage. This excludes them from participating in coverage metrics that are based on named entities. This trade-off between convenience and explicitness should be considered by engineers when creating grammars for testing purposes.

2.2.2 Quantifications

Going back to another example from the JavaScript grammar, we have the following rule:

```
DecDigits := DecDigit+;
```

This derivation rule leverages the *Kleene plus* notation to conveniently indicate that a repetition of the `DecDigit` must occur at least once. A new operator rarely comes alone, so we can extend our notation to include the *Kleene star* (`*`) and the *optionality indicator* (`?`) as well. Going even further, we may want to express bounded repetitions, which explicitly set a minimum and maximum number of occurrences of particular symbols. As an example, consider the following rules:

```
UnaryExpr := "-"{0,2} Identifier | ... ;
DecNumber := "0" | "-"? NonZeroDecDigit DecDigit*;
```

Here, the `"-"`$^{\{0,2\}}$ prescribes that `"-"` must occur anywhere between zero and two times, both extremes included. Meanwhile, `"-"`$^{?}$ is a shorthand which indicates that `"-"` may occur exactly once or not at all, while `DecDigit*` allows `DecDigit` to occur arbitrarily many times, zero included.

Without these operators, the above rules would have to be written in a manner that is much more cumbersome and requires to introduce an additional rule:

```
UnaryExpr := Identifier
           | "-" Identifier
           | "-" "-" Identifier
           | ... ;
DecNumber := "0"
           | NonZeroDecDigit DigitKleeneStar;
           | "-" NonZeroDecDigit DigitKleeneStar;
DigitKleeneStar := "" | DecDigit DigitKleeneStar;
```

Together, these operators allow us to quantify the occurrences of grammar symbols, so we shall refer to them as *quantifications*.

2.2.3 Regular Expressions

Why stop there, though? Since we can fully include any regular language inside a context-free grammar as shown by Chomsky [23], we might as well allow full embeddings of regular expressions as part of our notation. For example, we could simplify the way we define `DecNumber` by writing it as a single regular expression:

```
DecNumber := /0|(-?[1-9][0-9]*)/;
```

This notation for regular expressions is actually much more powerful than the quantifications that we have just defined. Since regular languages are closed under complement and intersection, they allow us to very tersely express negative and intersecting expressions:

```
NonDigit := /[^0-9]/;
NonKeyword := /~(for|if|break|continue)&([a-z]+)/;
```

The `NonDigit` rule allows instantiating any character as long as it is not a decimal digit, while `NonKeyword` derives into any non-empty string of lowercase Latin characters as long as it is not equal to any of `for`, `if`, `break`, or `continue`. Inside regular expressions, `~` signifies the *complement* of the following expression, while `&` is the *intersection* of two regular languages.[1] If we had to resort to standard context-free grammar notation, we would have had a very hard time expressing these same derivations, as doing so would require us to enumerate *all possible* alternatives.

2.2.4 Grammar for Grammars

When applied together, the above extensions of the original context-free grammar notation give us one that is both convenient to use in practice and provides a way to control which parts of the language are more *interesting* to explore than others.

With the set of additional grammar constructs complete, we are now finally ready to define the full language of our extended context-free grammars as a context-free grammar itself. Figure 2.1 formalizes the most important structures of this meta-grammar.

A grammar consists of *productions*, each of which expands a *non-terminal* into an *alternation*, delimited by the `":="` sign. A production is always terminated by a semicolon; this is useful when implementing a whitespace-insensitive parser for files containing grammars.

An alternation is a non-empty sequence of *alternatives*, which are *concatenations* delimited with the `"|"` character. A concatenation consists of *atoms*, which can be parenthesized alternations, *literals*, *regular expressions*, or *references*. While references are essentially non-terminals on the right side of a production, it

[1] For implementation reasons, the exact notation and the extent of supported features for regular expressions is borrowed from Møller [91].

```
Grammar := Production+;
Production := NonTerminal ":=" Alternation ";";
Alternation := Concatenation ("|" Concatenation)*;
Concatenation := Atom+;
Atom := ("(" Alternation ")"
        |Regex
        |Literal
        |Reference) Quantifier?;
Quantifier := "?"|"+"|"*"
        |"{," Number "}"
        |"{" Number ",}"
        |"{" Number "," Number "}";
Regex := "/" regexp "/";
Number := /[0-9]+/;
Reference := NonTerminal;
NonTerminal := /[A-Za-z_'][A-Za-z0-9_$]*/;
Literal := """ (/[^\"\\]/|"\" /[nrt\"\\]/)* """;
```

Figure 2.1 / A meta-grammar for extended context-free grammars. The definition of `regexp` is omitted here, but it holds no secrets as it corresponds to classical regular expression notation, which, in this case, is taken directly from [91]. Also, the construct `"""` denotes a string consisting of one quote character.

is important to make the distinction because it is the references, and not the non-terminals, that will play an important role in the upcoming definition of grammar coverage.

An optional *quantifier* allows expressing how often an atom can be repeated in accordance with the definition from Section 2.2.2. As an additional detail of convenience, beside the notation $\texttt{foo}^{\{m,n\}}$, which indicates at least m repetitions and at most n repetitions of `foo`, we also allow partial specifications such as $\texttt{foo}^{\{x,\}}$ and $\texttt{foo}^{\{,y\}}$, which simply specify at least x, and at most y repetitions of `foo`, respectively.

Literals are what comes closest to the *terminals* from Section 2.1, except that they allow encoding arbitrary strings as specified by the Scala language [96].

Regular expressions are delimited with `"/"` and admit any pattern that is legal in the specification by Møller [91] with the exception that forward slashes must be escaped. As per Chomsky [23], these regular languages can be fully embedded into the context-free grammar; and while my implementation does this to some extent by synthesizing appropriate derivation rules, in some cases it is more convenient to simply replace the regular expressions by corresponding automata in the in-memory model of the grammar, instead.

2.3 Grammar Graph Representation

Now that we have established the set of concepts that make up our grammars, we can move on to build a representation of the underlying model that is suitable for us to work with. This step can be considered to serve the same function as an intermediate representation does in a compiler.

A representation that lends itself particularly well for our purposes, comes in the form of a directed graph, to which we shall refer as the *grammar graph*. Its nodes can be seen as manifestations of some of the grammar parts described in Figure 2.1, while its edges represent derivations. As an example, consider Figure 2.2, which shows an excerpt from the grammar graph, obtained from the grammar in Figure 1.1.

A grammar graph is constructed from the productions of a grammar by applying the following rules to their right-hand sides:

1. For each alternation, a synthetic alternation node (|) is created, whose children are the graph representations of its corresponding alternatives. If there is only one alternative, no node is created for the alternation, and its only alternative takes its place instead.
2. Analogously, concatenations become synthetic (~) nodes with their atoms as children, while handling single atoms as in the case of alternations.
3. Each quantifier becomes a dedicated node (e.g., (+)), whose child is the graph representation of the atom the quantifier is attached to.
4. Literals become nodes with no children.
5. Regular expressions also become childless nodes. In the first instance, it is sufficient to leave such regex nodes as is, but for practical purposes, they are translated into their equivalent context-free form, and replaced with the graph representation of this form. In cases where this translation causes a blow-up in the number of nodes that exceeds a given threshold, the regex node is left as is, but it is also enriched with a state machine.
6. References to non-terminals become nodes that have as their child the graph representation of the production they refer to.
7. All nodes are assigned a unique numeric identifier. This is necessary to distinguish between equal nodes occurring in different contexts.

Not unlike a tree, the resulting graph has a single "root" node corresponding to the right-hand side of the production of the grammar's start symbol, that has no incoming edges, which lends the graph a familiar, tree-like appearance.

Definition 1: Root Node

Let G be a grammar graph.

Then $root(G)$ shall be the root node of G.

Additionally, because edges are directed, we can conveniently call the node at the start of an edge the *parent* node, and the one on the end the *child* node. However, very much unlike a tree, a grammar graph can have cycles because references can occur in multiple derivation rules. This is also the reason why nodes may

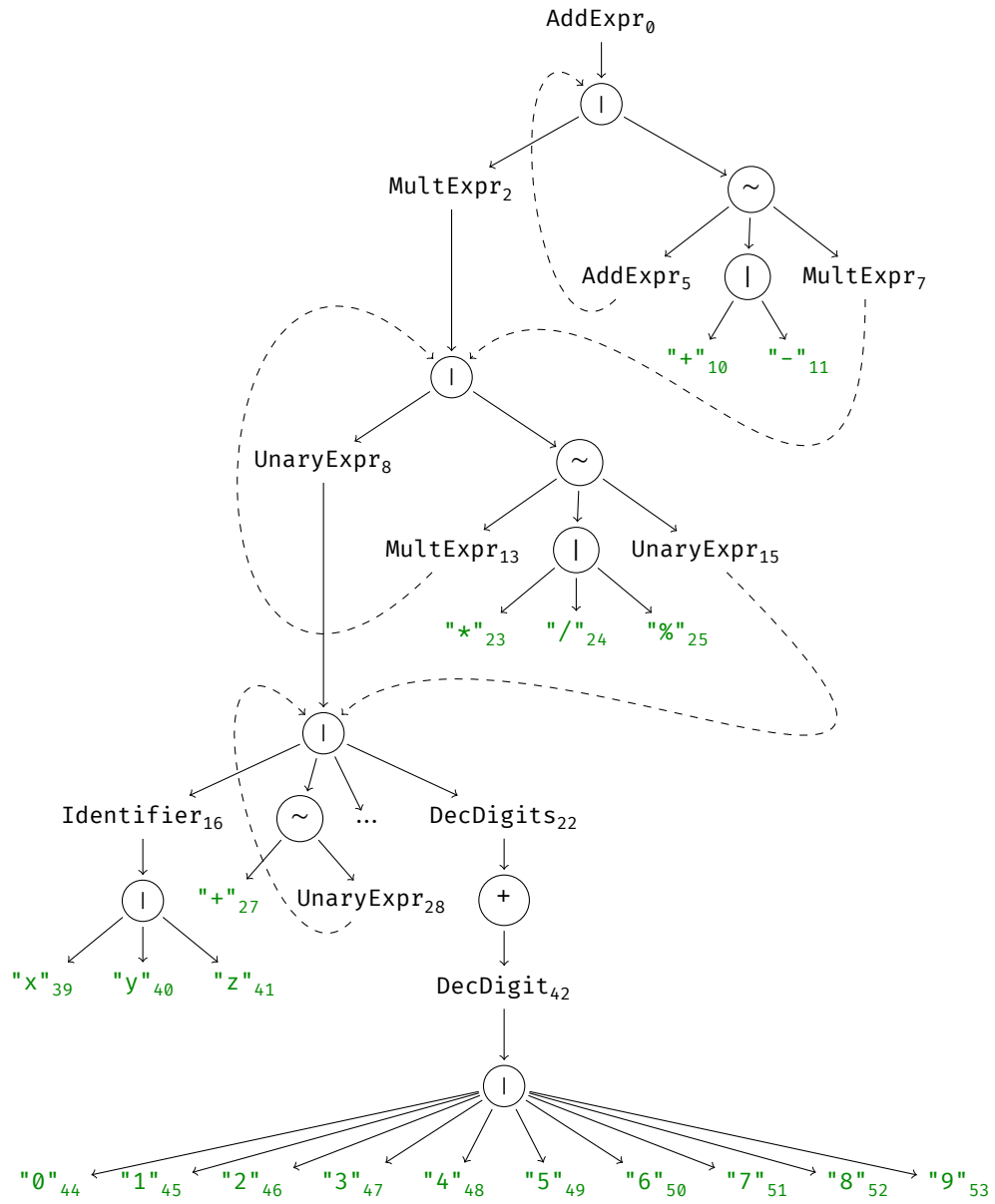


Figure 2.2 / An excerpt from the graph representation of the (partial) grammar from Figure 1.1. The root node is `AddExpr`$_0$ because it is the right-hand side of the production of the start non-terminal `Expr`. The "backward" dashed lines indicate derivations of references that prevent the graph from being a tree or even a DAG. Numeric identifiers are only shown for symbolic nodes. Please consider navigating to Appendix B, which contains a copy of this figure, and printing it out on an extra sheet to have it at hand for easy reference.

have not only multiple children, but also multiple parents. In Figure 2.2, the edges corresponding to such additional occurrences are marked as dashed lines, to leave the underlying tree-like structure clearly visible.

We refer to nodes corresponding to alternations, concatenations, and quantifiers as *synthetic* nodes, while all others are *symbolic* nodes. This serves to indicate the purpose of the nodes: While the symbolic nodes are parts of the textual representation of a grammar that were explicitly named by its creator, presumably a human, synthetic nodes only serve the purpose of orchestrating how the former relate to each other.

Note how the grammar graph sometimes breaks out of the structure given by the textual representation of the grammar. For example, in contrast to what Figure 2.1 claims, an alternation need not necessarily have concatenations as its children, in case these concatenations consist of a single element. One such case can be seen in Figure 2.2, where the atomic reference node $\texttt{MultExpr}_2$ is the direct child of an alternation node (|).

Another difference from the textual representation lies in the way quantifiers are modeled: Because they merely modify other derivation rules, we can represent them using synthetic quantification nodes, which have the subject of the modification as their child, and the nature of the modification as attributes encoded in the node. This approach allows handling of quantifiers in a manner consistent with the other ADT[2]-like grammar constructs, without the need of reformulating them in terms of recursion.

However, possibly the most significant difference between the grammar graph and the textual grammar representation is the introduction of unique numeric identifiers to all nodes of the grammar. The addition of such a seemingly small technical detail enables us to distinguish between multiple occurrences of the same (symbolic) node in different contexts. As an example, consider the terminal `"+"` as it occurs in Figure 1.1. Without identifiers, we have no way to specify that we mean the binary `"+"` operator that occurs between the `AddExpr` and `MultExpr` as opposed to the unary `"+"` which precedes a `UnaryExpr`. Using the identifiers, we can easily differentiate between these cases by writing $\texttt{"+"}_{10}$ or $\texttt{"+"}_{27}$, respectively.

2.4 Derivation Trees

We can now lean on the model provided by the grammar graph whenever we parse or generate inputs. Hereby, we will not stray far from literature [1], which defines parse trees and derivation trees for the two use cases above, respectively. Since they only differ in naming and purpose, and, in fact, effectively describe the same structure, we will refer to them as derivation trees, for simplicity.

In the classical definition, a derivation tree represents the syntactic structure of an input according to a given grammar. Specifically in our case, derivation trees shall describe the structure according to the *graph* of a grammar. Thus, the nodes and edges that make up the derivation trees conveniently come from the same set as the nodes and edges in the graph of the grammar at hand. This

[2]ADT refers to **A**lgebraic **D**ata **T**ypes.

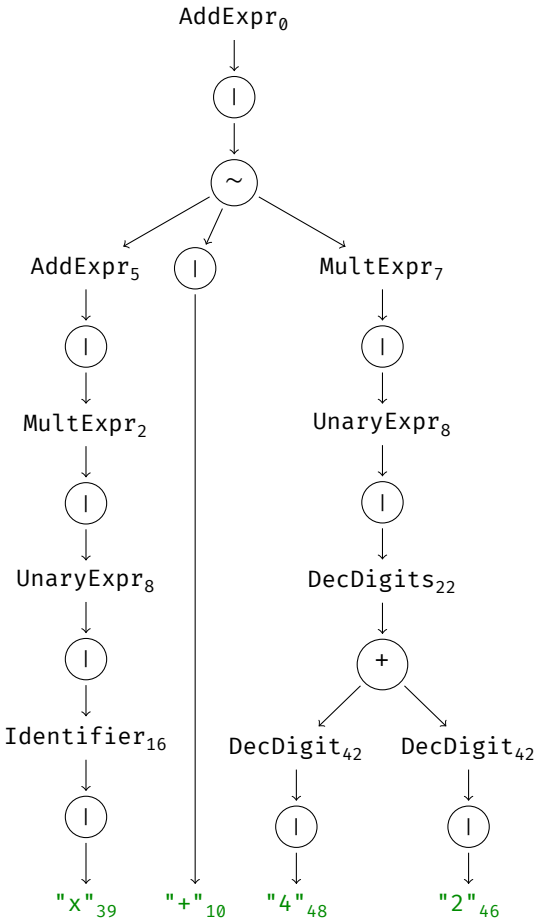


Figure 2.3 / A derivation tree representing the input string "x+42" according to the grammar graph from Figure 2.2.

definition provides us with the useful property that every path in a derivation tree is simultaneously also a path in its grammar graph.

As an example of a derivation tree, consider Figure 2.3, which shows the derivation tree for the input "x+42", parsed according to the grammar from Figure 1.1, or, more accurately, its graph representation as shown in Figure 2.2. Observe how it is perfectly valid for a derivation tree to contain multiple occurrences of the same node. This can happen if the derivation contains multiple references to the same non-terminal as is the case with the $\texttt{UnaryExpr}_8$ node, or when a quantification node has multiple occurrences of its subject, like the two $\texttt{DecDigit}_{42}$ nodes.

For the upcoming use case of generating derivation trees to obtain inputs for fuzzing, let us consider the following definition. A tree node has numbered *slots*, which are said to be filled with their children:

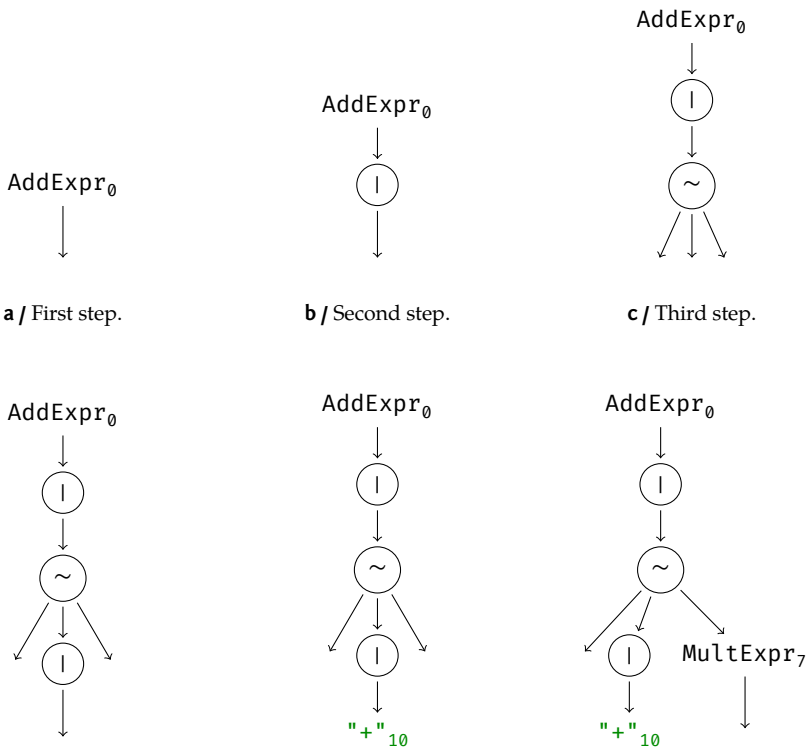


Figure 2.4 / Derivation tree generation in progress filling one slot at a time.

Definition 2: Slots

A slot is a tuple $(\text{parent}, \text{child}, \text{index}, \text{depth})$, where *parent* is the node whose child this slot represents, *child* is the node that can fill the slot, *index* signifies the position of this child node among its siblings, and *depth* corresponds to the number of derivations from the root to the child node.

Informally, it may be helpful to think of slots simply as the edges in a derivation tree. We also say that the child node of a slot is its *expansion*.

In a fully formed derivation tree such as the one given in Figure 2.3 all slots are filled. However, during the generation of a tree there may be unfilled slots where the generation algorithm has not yet added an appropriate expansion. Consider Figure 2.4, which shows consecutive steps that a derivation algorithm might take when generating a tree from the JavaScript grammar excerpt. After the first step given in Figure 2.4a, the tree under construction consists of only the root node $\texttt{AddExpr}_0$ and a single unfilled slot $(\texttt{AddExpr}_0, \emptyset, 0, 1)$. Here, we use the $\emptyset$ symbol to indicate that the child of a slot is not yet filled in. Further, this slot has index zero and depth one. According to our grammar graph, this slot can only be filled with a (|) node.

Figure 2.4b shows the state of the tree after a second derivation step. The slot from the previous step is now filled with the (|) node, which, in turn results in an unfilled slot ((|), ∅, 0, 2) at depth 2. This particular slot can now be filled by either a $\texttt{MultExpr}_2$ or a (~) node. Our generator decides to fill it with the latter in Figure 2.4c, which leaves us with three unfilled slots.

An advantage of the slot-based derivation scheme consists in the ability to fill outstanding slots in an arbitrary order. This is demonstrated as the generator proceeds to fill the ((~), ∅, 1, 3) slot with its appropriate (|) expansion in Figure 2.4d. At this point our outstanding slots are ((~), ∅, 0, 3), ((~), ∅, 2, 3), and the newly added ((|), ∅, 0, 4). The latter is filled first with the $\texttt{"+"}_{10}$ node in Figure 2.4e, which, for the first time since starting the generation does not add any more unfilled slots to the tree.

Finally, our generator chooses to expand the slot with index = 2 of the (~) node in Figure 2.4f, which results in the now filled slot ((~), $\texttt{MultExpr}_7$, 2, 3), but also brings with it the unfilled slot ($\texttt{MultExpr}_7$, ∅, 0, 4) for the generator to fill at a later point. To be consistent, as a special case, we define a *root* slot of our derivation tree as (∅, $\texttt{AddExpr}_0$, 0, 0).

2.5 Grammar-Based Input Features

With the main concepts put into place, we are now ready to define features that express properties of the syntactic makeup of inputs. Recall that we initially set out to define a notion of grammar coverage, so it makes sense to start by consulting the basic constructs of the grammar.

As we have seen in Section 2.3, the model of a grammar can be represented by its graph. The nodes in a grammar graph can be divided into synthetic nodes, which express the *structure*, and symbolic nodes, which express the *content* of the language described by the grammar. Notably, the latter are entities that are explicitly and purposefully named or stated, which makes them in some way meaningful to the designer of the grammar. In the interest of brevity, we shall refer to symbolic nodes simply as symbols going forward.

2.5.1 Symbol Coverage

Let us attempt to quantify the content diversity provided by the symbols of a grammar. The very first measure we can take is to enumerate the symbols that a grammar contains by enumerating all symbolic nodes in its grammar graph. Thinking ahead, and also for convenience, we can leverage the fact that a derivation tree consists of the same nodes as its grammar graph. Thus, we can formulate a definition encompassing both of these structures:

Definition 3: Unique Symbols

Let x be a grammar graph or a derivation tree. Then $sym(x)$ shall be the set of unique symbolic nodes in x.

Armed with this primitive analysis tool, we can express the concept of *symbol coverage* of any given input, provided that it is valid according to our grammar:

Definition 4: Symbol Coverage

Let G be a grammar graph and d a derivation tree of the given input i that was parsed according to G.

Then $symcov(i)$ shall be the symbol coverage of i computed as follows:

$$symcov(i) = \frac{|sym(d)|}{|sym(G)|}$$

In other words, the symbol coverage of an input is the fraction of unique symbolic nodes of the grammar graph that are present in its derivation tree.

As an example, consider the input "$x + 42$" and its derivation tree shown in Figure 2.3. We can now calculate its symbol coverage as per Definition 4. The derivation tree has 12 unique symbolic nodes. The grammar graph shown in Figure 2.2 contains eleven *synthetic* nodes, however, it omits several descendants of the $\texttt{UnaryExpr}_8$ node, for brevity. To compute the symbol coverage correctly, we can consult its original grammar definition from Figure 1.1, and observe that the omitted part only contains four synthetic concatenation nodes with the rest being symbolic. Leveraging the fact that the nodes are assigned contiguous, incrementing identifiers, and the largest identifier having a value of 53, we can calculate that the grammar graph contains $54 - 11 - 4 = 39$ symbolic nodes.

Returning to our goal of determining the symbol coverage, we calculate it as follows: $symcov(\text{"}x + 42\text{"}) = {}^{12}/_{39}$, or approximately 30.77 %.

In practice, we might be interested in the symbol coverage of not one, but many inputs at once. This is especially relevant to the use case of fuzzing, where it is common to reason about sets of inputs and their coverage. With this in mind, we can extend our definition of symbol coverage as follows:

Definition 5: Symbol Coverage of an Input Set

Let G be a grammar graph and D a set of derivation trees corresponding to the inputs S, parsed according to G.

Then $symcov(S)$ shall be the symbol coverage of S computed as follows:

$$symcov(S) = \frac{|\bigcup \{sym(d) \mid d \in D\}|}{|sym(G)|}$$

Intuitively, the symbol coverage of an input set is the fraction of unique symbolic nodes of the grammar graph that are present across its derivation trees.

Perhaps a noteworthy detail is that the symbol coverage is always in the $(0, 1]$ range. It cannot be zero because any derivation tree will have at least one terminal

node, which will find its correspondence in the grammar graph. The coverage also cannot exceed one because no derivation tree can contain nodes that are not part of the grammar graph. Based on this information, we can say that a set of inputs S achieves full symbol coverage if $symcov(S) = 1$.

2.5.2 k-Path Coverage

While the symbol coverage can provide us with insights into the nature of inputs, these insight are rather limited. To showcase some shortcomings, consider a fuzzing scenario in which we want to fuzz a calculator program for arithmetic expressions. Referring again to our grammar from Figure 1.1, let us assume we have the two inputs "$(x + 2) * 5$" and "$(x * 2) + 5$".

These two inputs will cause our calculator to use different code paths because it will attempt to apply the distributive rule to the first input, whereas the second input can no longer be simplified and must be processed as is.

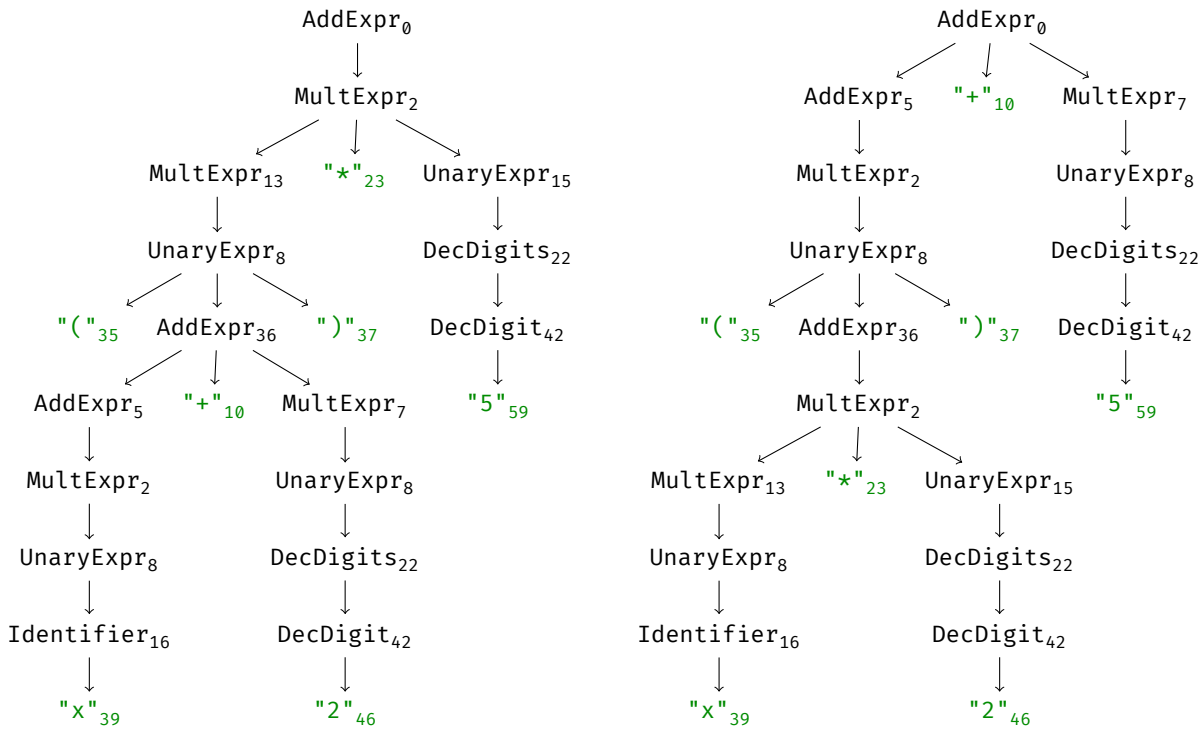
Unfortunately for us, the symbol coverage for these two inputs is identical. Even though their derivation trees have a different structure, they share the exact same set of symbolic nodes, and the way they are arranged is of no consequence to the symbol coverage measure. Figure 2.5 shows the two trees side by side.

If we want to be able to reason about the structure of inputs, then we must additionally consider the symbols' location inside their context. In this particular case, for example, we might want to express that one input has the "+" inside a parenthesized context, while the other has it on the outside. Note that in both cases, it is the same $\texttt{"+"}_{10}$ node with the same identifier, only it is located in a different context in the respective derivation trees.

One way to uniquely identify the context of a node inside a tree is by considering the path from the root to the node in question. Continuing our example, let us inspect the paths to the $\texttt{"+"}_{10}$ node in the two derivation trees. For the input "$(x + 2) * 5$", the path is $\texttt{AddExpr}_0 \rightarrow \texttt{MultExpr}_2 \rightarrow \texttt{MultExpr}_{13} \rightarrow \texttt{UnaryExpr}_8 \rightarrow \texttt{AddExpr}_{36} \rightarrow \texttt{"+"}_{10}$ if we leave out the synthetic nodes for convenience. For the second input "$(x * 2) + 5$", the path is merely $\texttt{AddExpr}_0 \rightarrow \texttt{"+"}_{10}$.

The two paths already differ in the parent of the node in question, so considering the entire path from the root seems excessive. In fact, if our goal is to simply differentiate between a "+" inside and outside parentheses for testing our calculator, we would still want to consider the $\texttt{"+"}_{10}$ nodes as equal in the two inputs "$(x + 2)$" and "$((x + 2))$". However, if the entire path counts as the context, this goal cannot be achieved.

Therefore, we want to be able to express a context that is just large enough to be relevant but at the same time small enough so as not to over-specialize. At the same time, we can clearly recognize that the size of the context is heavily dependent on the use case, which is why we choose to make the size a user-adjustable parameter in the upcoming formalizations.



a / Derivation tree for the input "$(x + 2) * 5$".

b / Derivation tree for the input "$(x * 2) + 5$".

Figure 2.5 / Derivation trees for two inputs that have different structures but share the same symbol coverage. Only symbolic nodes are shown here for reasons of simplicity and space.

Definition 6: k-Path

Let x be a grammar graph or a derivation tree, and k a positive natural number. Then a k-path shall be a path in x, containing exactly k symbolic nodes, and whose beginning and end are symbolic nodes.

We have already seen examples of k-paths above, when comparing the contexts of the `"+"`$_{10}$ node in the two inputs "$(x + 2) * 5$" and "$(x * 2) + 5$". Note that k-paths need not start at the root node, so if we were to consider only 2-paths for comparing the two inputs, we would observe that the first has the 2-path `AddExpr`$_{36} \rightarrow$ `"+"`$_{10}$, while the context of the `"+"`$_{10}$ in the second input is `AddExpr`$_{0} \rightarrow$ `"+"`$_{10}$. Clearly, setting $k = 2$ is sufficient for this use case.

Furthermore, both "$(x+2)$" and "$((x+2))$" would have the same 2-path context `AddExpr`$_{36} \rightarrow$ `"+"`$_{10}$, making them indistinguishable, exactly as desired. If we wanted to make the distinction between these two parenthesized constructs, we would have to increase the value of k to at least 5. This is because the doubly parenthesized input has an extra loop through the `MultExpr`$_{2} \rightarrow$ `UnaryExpr`$_{8} \rightarrow$ `AddExpr`$_{36}$ nodes, so we require at least a 5-path, which includes the `"+"`$_{10}$ itself as well as one node before the loop, to tell them apart.

Now that we know what it means for a symbol to have a context of a given size, we can use this notion to measure the makeup of inputs in a way that is sensitive to such contexts. Again, let us first define a way to enumerate these contexts:

Definition 7: Unique k-Paths

Let x be a grammar graph or a derivation tree, and k a positive natural number. Then $paths_k(x)$ shall be the set of unique k-paths that exist in x.

From here, we can proceed in a similar fashion to how we introduced symbol coverage to introduce the context-aware k-path coverage of an input:

Definition 8: k-Path Coverage

Let k be a positive natural number, G a grammar graph, and d a derivation tree of the given input i that was parsed according to G.

Then $pathcov_k(i)$ shall be the k-path coverage of i computed as follows:

$$pathcov_k(i) = \frac{|paths_k(d)|}{|paths_k(G)|}$$

Similarly to what we had before, the k-path coverage of an input is the fraction of unique k-paths of the grammar graph that are present in its derivation tree.

Returning to the example input "$x + 42$" and its derivation tree in Figure 2.3, we can compute its 2-path coverage. Let us start by enumerating the unique 2-paths contained in the derivation tree:

1. $\texttt{AddExpr}_0 \rightarrow \texttt{AddExpr}_5$
2. $\texttt{AddExpr}_5 \rightarrow \texttt{MultExpr}_2$
3. $\texttt{MultExpr}_2 \rightarrow \texttt{UnaryExpr}_8$
4. $\texttt{UnaryExpr}_8 \rightarrow \texttt{Identifier}_{16}$
5. $\texttt{Identifier}_{16} \rightarrow \texttt{"x"}_{39}$
6. $\texttt{AddExpr}_0 \rightarrow \texttt{"+"}_{10}$
7. $\texttt{AddExpr}_0 \rightarrow \texttt{MultExpr}_7$
8. $\texttt{MultExpr}_7 \rightarrow \texttt{UnaryExpr}_8$
9. $\texttt{UnaryExpr}_8 \rightarrow \texttt{DecDigits}_{22}$
10. $\texttt{DecDigits}_{22} \rightarrow \texttt{DecDigit}_{42}$
11. $\texttt{DecDigit}_{42} \rightarrow \texttt{"4"}_{48}$
12. $\texttt{DecDigit}_{42} \rightarrow \texttt{"2"}_{46}$

As we can see, the tree contains twelve distinct 2-paths. The grammar graph, on the other hand, contains 125 unique 2-paths.[3] This leaves us with $pathcov_2(\text{“}x + 42\text{”}) = {}^{12}/_{125}$, or a 2-path coverage of 9.6 %.

And again, as we did before, we can extend the definition of k-path coverage so that it, too, applies to a set of inputs instead of just one:

Definition 9: k-Path Coverage of an Input Set

Let k be a positive natural number, G a grammar graph, and D a set of derivation trees corresponding to the inputs S, parsed according to G.

Then $pathcov_k(S)$ shall be the k-path coverage of S computed as follows:

$$pathcov_k(S) = \frac{|\bigcup\{paths_k(d) \mid d \in D\}|}{|paths_k(G)|}$$

Defined this way, the k-path coverage will always produce values that are in the $[0, 1]$ range. This time, zero is included because a derivation tree might be too short to even contain k-paths for a given k, whereas the grammar graph can fit them just fine by going through a cycle. Analogously to symbol coverage, we can say that a set of inputs S achieves full k-path coverage if $pathcov_k(S) = 1$.

It might also be worth noting that while the number of k-paths in a grammar graph is always finite, it tends to increase exponentially with the value of k. For example, our JavaScript expression grammar graph from Figure 2.2 contains 39 1-paths, 125 2-paths, 523 3-paths, 2331 4-paths, and 10245 5-paths.

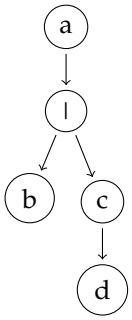
It is no coincidence that the number of 1-paths is the same as the number of symbolic nodes that we have seen earlier. In fact, recalling Definition 6 and setting $k = 1$, we achieve $paths_1(x) = sym(x)$. Therefore, we can conclude that k-path coverage is powerful enough to express symbol coverage. And that is one formalism fewer we have to lug around, which cannot fail to make us happy.

2.5.3 Coverage Subsumption

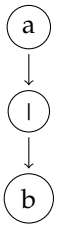
Given that in order to cover k-paths with a higher value of k, more and larger inputs must be generated, one might wonder if achieving full k-path coverage for a higher k also automatically gives us full k-path coverage for all lower k.

Consider a scenario like the one given in Figure 2.6, where Figure 2.6a shows a very simple grammar graph, and two derivation trees corresponding to this

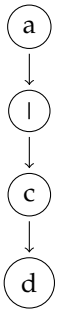
[3] At least that is what my implementation says. I choose to trust it on this one.



a / A simple grammar graph.



b / The left tree.



c / The right tree.

Figure 2.6 / Disproving the subsumption relationship between k-path coverages. The right tree achieves full 3-path coverage, but it alone does not suffice to achieve full 1-path and 2-path coverage. Therefore, k-path coverages with higher values of k do not subsume those with lower k.

graph are given in Figures 2.6b and 2.6c. As a matter of fact, these are the only two possible derivation trees that belong to this grammar graph.

If we now seek to achieve full 2-path coverage, we will inevitably also achieve full 1-path coverage because we have to generate both trees to get all three existing 2-paths (a) → (b), (a) → (c), and (c) → (d). Generating the two trees gives us all four symbolic nodes, and thus also full 1-path coverage. We can say that the 2-path coverage fully subsumes the 1-path coverage.

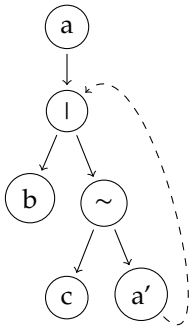
One could expect that given $i < j$, the j-path coverage will always subsume the i-path coverage. However, such a subsumption relationship does not hold in the general case.

Considering the 3-path coverage, we see that there is only one 3-path available: (a) → (c) → (d). We only need to generate the right derivation tree given in Figure 2.6c to reach this 3-path. This way we are still missing the (b) node, which is a necessary requirement for both the 1-path and 2-path coverages.

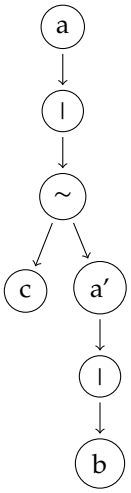
Interestingly, achieving full 1-path or 2-path coverage actually also gives us full 3-path coverage. Therefore, in this particular case, both the 1-path and the 2-path coverage are the ones fully subsuming the 3-path coverage.

In turn, this might lead us to suspect that for $i < j$, it is the i-path that subsumes the j-path. However, consider the setting pictured in Figure 2.7, where Figure 2.7a gives a small grammar graph that contains a back-reference, which results in a cycle. The derivation tree given in Figure 2.7b achieves full 1-path coverage because it contains all symbolic nodes. However, to attain full k-path coverage for any $k > 1$, we expect to see a k-path containing k contiguous occurrences of the (a′) reference. Full 2-path coverage requires (a′) → (a′), while for 3-path it is (a′) → (a′) → (a′), and so on.

From these examples, we can safely conclude that in the general case there is no subsumption relationship between the different k-path coverages.



a / Grammar graph with a cycle.



b / Tree with full 1-path coverage.

Figure 2.7 / Further disproving the subsumption relationship between k-path coverages. The derivation tree achieves full 1-path coverage, but it does not achieve full k-path coverage for $k > 1$. Therefore, k-path coverages with lower values of k do not subsume those with higher k .

Chapter 3

Systematically Covering Input Features

Up until now, we were concerned with how to quantify features of already existing inputs. However, in the context of fuzzing, we are especially interested in *creating* inputs that are rich in such features. Since k-paths signify instances of meaningful symbols located in a particular context, it is desirable for us to generate inputs exhibiting as many of them as possible, and preferably even all of them.

3.1 Generating Rich Inputs

From the way we introduced k-paths in Definition 7 we can see that the set of k-paths is readily available to be enumerated for any given grammar. Therefore, it should also be possible to construct an algorithm that systematically produces a forest of derivation trees that, together, contain all k-paths. An algorithm for doing just that is given as Algorithm 1.

In the name of terseness, and to avoid not being able to make heads or tails of k-paths, let us first introduce the following useful definition:

Definition 10: Head and Tail

Let k be a positive natural number and p a k-path, consisting, in-order, of nodes n_i for $0 \leq i < k$.

Then we set $head(p) = n_0$ and $tail(p) = p'$, where p' consists, in-order, only of nodes n_i for $1 \leq i < k$.

In short, the head of a k-path is its first node, while its tail consists of the rest.

Let us now familiarize ourselves with how the algorithm works. Given a positive natural number k and a grammar graph G, the algorithm begins by storing all k-paths available in G into a list P. It then iterates over this list, generating for

Algorithm 1 The k-Path Algorithm

```
1:  function GenerateKPaths(grammar graph G, positive natural number k)
2:      forest ← {}
3:      r ← root(G)
4:      P ← paths_k(G)                          ▷ Enumerate all k-paths as per Definition 7
5:      while P ≠ {} do
6:          t ← tree(r)                         ▷ Start new derivation tree rooted at r
7:          s ← child slot of r
8:          p ← remove next k-path from P       ▷ Select next k-path to pursue
9:          if head(p) = r then
10:             p ← tail(p)
11:         while p is not empty do
12:             n ← expansion of s
13:             Add n to t by filling slot s
14:             if n = head(p) then
15:                 p ← tail(p)
16:             if p is not empty then
17:                 s ← child slot of n with shortest derivation path to head(p)
18:         t ← CloseOff(G, t)                  ▷ Finalize t by expanding all unfilled slots
19:         forest ← forest ∪ {t}
20:         Remove from P all k-paths found in t
21:     return forest
```

each k-path p a new derivation tree t, whose root node r is the same as the root node of the grammar graph.

For each such tree, the algorithm maintains the current slot s that needs to be filled next to reach p. Initially, it is the single child slot of the root node r.

As long as the algorithm has not yet succeeded in deriving the k-path p as part of the current tree in Line 11, it proceeds to expand the slot s into the node n and if it is indeed equal to the next element of the targeted k-path (i.e., $head(p)$), and not just a synthetic node on the way to it, we remove n from the k-path p, thus shortening it so that we come closer to completing our current target.

The newly created node n is added to the tree t by filling slot s, but it comes with unfilled slots of its own, one of which now must replace s. For this, in Line 17 the algorithm chooses the one slot that brings it to the next missing node fastest.

After the loop in Lines 11 to 17 finishes, we are left with a partial tree t which contains only the derivation of our targeted k-path, and which must be completed to represent a valid input. This task is handed off to a close-off algorithm in Line 18, which gives us certain flexibility as the k-path algorithm itself is rather agnostic of how this part is handled.

Finally, the completed tree is added to our growing forest, which will be returned as the output, and the algorithm is ready to target the next not yet reached k-path.

However, if we recall our observations from Section 2.5.2, we may notice that the number of k-paths may grow exponentially with the size of the grammar graph. Since we would very much like to avoid having to generate exponentially many

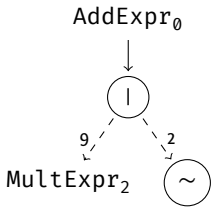


Figure 3.1 / The k-path algorithm considers the choice between the two acceptable alternatives and chooses the (~) node because its distance to the target `"+"`$_{10}$ node is shorter.

files to encompass all k-paths, in Line 20 we simply remove from P all k-paths that we happened to produce "by accident" while on our way to generating p or closing off the current tree. This simple trick dramatically reduces the number of trees generated by this algorithm.

3.1.1 Example Generation Walkthrough

Let us now walk through an example derivation to better understand what decisions the algorithm makes and how it makes them. Let us assume that we have called the algorithm with G being the JavaScript expression grammar graph from Figure 2.2 and $k = 2$, and that in Line 8 the value of p has become the 2-path `AddExpr`$_0 \rightarrow$ `"+"`$_{10}$. Then, our target path is removed from the set of not yet reached 2-paths P, and a new derivation tree is created starting with an `AddExpr`$_0$ node, whose slot for its only child node (|) becomes s in Line 7.

Here, we have triggered the edge case in Line 9, where the first node in our targeted 2-path is the root node, so we remove it from p before proceeding because our sapling already contains r by definition.

At this point, the remaining not yet reached path p consists only of the `"+"`$_{10}$ node. In Line 12 s is expanded into a (|) node n, which we then add to t in Line 13.

Since p still contains the `"+"`$_{10}$ node, we have to choose which of the child slots of n will replace s in Line 17. Looking at the graph in Figure 2.2, in this case it is the slot for the (~) node because its distance to `"+"`$_{10}$ is two, as opposed to the alternative `MultExpr`$_2$, whose distance is nine as shown in Figure 3.1.

In the next iteration, we enlarge t by the (~) node, and have to choose the next s in Line 17 from its child slots for `AddExpr`$_5$, another (|), and `MultExpr`$_7$. Here, we choose the slot for the (|) node because it leads us to `"+"`$_{10}$ fastest.

In a later iteration, we get to expand the (|) node to be the `"+"`$_{10}$ node, which lets us remove it from p in Line 15, thus rendering the currently targeted 2-path empty and exiting from the loop.

Since we have already reached our current goal, we hand off the task of completing the tree by expanding all outstanding slots to a close-off algorithm in Line 18. For instance, if we are interested in creating inputs that are terse, we might opt to use an algorithm that always produces the shortest possible derivations.

After the close-off is finished and we have obtained a complete tree in t, which in our example could correspond to an input like "x+y", we add it to our collection in forest, and very importantly, prune all 2-paths found in t from P.

In this case, t would contain such paths as $\texttt{AddExpr}_0 \rightarrow \texttt{AddExpr}_5$ and $\texttt{AddExpr}_0 \rightarrow \texttt{MultExpr}_7$, for which we no longer have to create their own trees, making the outer loop in Line 4 finish that much faster.

3.1.2 Avoiding Boundless Growth

So far, we have left the choice of the close-off procedure in Algorithm 1 entirely open. However, in practice, we are usually interested in avoiding boundless growth and useless repetitions. Algorithm 2 shows a way to generate the shortest possible inputs in terms of required expansions.

Algorithm 2 Shortest Tree Generation

```
1:  function ShortestTree(grammar graph G, derivation tree t)
2:      Q ← { unfilled slots in t }
3:      while Q ≠ {} do
4:          s ← a slot removed from Q
5:          n ← expansion of s
6:          Add n to t by filling s
7:          if n is an alternation then
8:              Q ← Q ∪ { child slot of n with least required derivation steps }
9:          else
10:             Q ← Q ∪ { all required child slots of n }   ▷ Skip optional elements
11:     return t
```

This algorithm is given a grammar graph G and an incomplete derivation tree t as inputs. It begins by identifying the set Q of all outstanding slots that still need to be filled for the tree to become complete. Its main loop in Line 3 proceeds to fill these slots until none remain. Whenever a new node n is instantiated to grow the tree t, its child slots are inspected and added to Q if appropriate.

Specifically, if n is an alternation node, in Line 8 only one of its children may exist in a valid derivation tree, and therefore its shortest alternative is added to Q. In this context, "shortest" refers not to the length of the resulting input string, but rather to the number of expansions that have to be made until the subtree is completed. In case multiple alternatives share the same shortest possible derivation, a uniform selection takes place to pick the slot to be added to Q.

Should n be a quantification node whose lower repetition limit is zero, nothing is added in Line 10 because this choice produces the shortest possible derivation from this node. In all other cases all child slots of n are added to Q, as they represent the necessary expansions that make up a valid derivation tree and are thus indispensable.

After the loop in Lines 3 to 10 finishes, we obtain a completed shortest tree in t, which is then returned.

While a way to generate minimal inputs can certainly find its use cases such as the one in [127], for the practical purposes of fuzzing, where we do not necessarily require easy human interpretability of the generated inputs, we might be interested to make another trade-off between size and feature-richness. For example, we might want to consider a global threshold depth, which our

inputs must not exceed. This way we achieve more freedom in the choice of expansions that do not lie on the critical path because we do not need to always follow the shortest possible derivation. In turn, generating non-minimal trees may lead to the inclusion of more features into the trees that we create, albeit inadvertently.

Algorithm 3 Bounded Tree Generation

```
1:  function BoundedTree(grammar graph G, derivation tree t, depth limit d)
2:      Q ← { unfilled slots in t }
3:      while Q ≠ {} do
4:          s ← a slot removed from Q
5:          n ← expansion of s
6:          Add n to t by filling s
7:          if n is an alternation or optional quantification then
8:              Q ← Q ∪ { a child slot of n fitting in d }
9:          else
10:             Q ← Q ∪ { child slots of n }
11:     return t
```

Algorithm 3 presents a way to achieve such bounded generation. Except for the additional depth limit parameter, this algorithm shares much of its structure with the shortest path algorithm.

In fact, the two algorithms differ in the way newly expanded alternations and optional quantifications are handled. The choice of the child slot for alternations in Line 8 is limited to those alternatives whose shortest required subtree will certifiably fit into the depth limit, considering the depth at which the parent node itself is located. This same condition is applied to the single child slot of a quantification whose minimal repetition count is zero.

Note however, that as opposed to always going for the minimal number of repetitions, in Algorithm 3 fitting child slots of quantifications may be added multiple times.

Also, in Line 8 a slot is chosen at random, however, it is possible to influence the behavior of the algorithm by using a heuristic instead. For example, one could imagine a selection in a least-recently-used order to enhance the variance of the generated inputs.

Using bounded generation as the close-off for Algorithm 1 requires a slight modification of the call site in that the desired depth limit must be passed along with the other parameters, however, this change is trivial to implement. In fact, when first introducing the k-path algorithm in [51] the bounded close-off variant is used inline as part of the main procedure.

Note how using either of these algorithms as the close-off call in Line 18 of Algorithm 1 avoids boundless growth by construction because the k-path algorithm always takes the shortest derivation route until the targeted path is covered, and then closes off peripheral subtrees with size-limited derivations otherwise.

While only two possible close-off algorithms are presented here, thanks to the pluggable design, it is rather straightforward to imagine others taking their

place, such as random or probabilistic generators, or even another instance of the *k*-path algorithm itself.

3.1.3 Additional Modifications

Algorithm 1 leaves out one important detail: In Line 8 the order in which the next *k*-path is picked is omitted. In practice, the order in which the algorithm aims to reach all *k*-paths influences the number of inputs that must be generated.

While in theory any heuristic would be applicable here as well, in practice, using random choice seems appropriate, especially considering a comparison against other random fuzzers as part of an empirical evaluation.

Further, there is a slight peculiarity in how the *k*-path algorithm handles optional quantifications – or rather how it does not handle them. When a derivation rule explicitly encodes the possibility of some part of the input to be absent, this absence can be reasonably treated as a feature on its own, thus making it a desirable derivation to generate. The way it is presented in Algorithm 1, the *k*-path algorithm does not explicitly offer support for this use case, however, it is possible to approach this problem from another angle. Specifically, a quantification of the form $\texttt{foo}^{\{0,n\}}$ can be rewritten as an alternation using an empty literal: $\texttt{""} \mid \texttt{foo}^{\{1,n\}}$.

Applying this modification to the input grammar allows the *k*-path algorithm to explicitly aim for generating empty expansions of optional elements without intruding too much on any intent behind the exact formulation of the textual grammar representation.

3.1.4 Requirements

The algorithms presented here owe their terseness to several requirements on the input grammars, which, albeit being perfectly reasonable, have not been explicitly stated. The following aims to address this omission.

Reachability All derivation rules in the grammar must be *reachable* from the root node, i.e., for every node *n* in the grammar graph, there must exist a sequence of slot expansions ending in *n* and starting from the root slot of the grammar.

Productivity All derivation rules are required to be *productive*, i.e., for every node *n*, there must exist a finite sequence of slot expansions, after which the subtree rooted at *n* has all its slots filled.

In the practical implementation of the *k*-path algorithm, the first phase is dedicated to checking the above properties, raising appropriate exceptions in case of violation. For a grammar engineer designing a grammar, this functionality provides a fast initial error feedback.

In addition to the above requirements, some algorithms assume the existence of auxiliary data structures that can be pre-computed statically:

- The *reachability map* contains for every node *n* a list of all nodes that are reachable from it, and in how many derivation steps.

- Every node stores *shortest derivation* information that indicates the least number of slot expansions that need to happen before the subtree rooted at this node contains no more unfilled slots.

3.1.5 Implementation Details

The *k*-path generation algorithm has been implemented as an application called TRIBBLE. It is written in the Scala programming language and its code is openly available in a repository on GitHub [47]. In order to function, the TRIBBLE tool only requires a Java 11 runtime or later.

As its main input, TRIBBLE takes a file containing a context-free grammar written in the notation outlined in Section 2.2. After parsing and turning it into a grammar graph, the tool is capable of outputting a set of inputs that achieve full *k*-path coverage using the method outlined in Algorithm 1.

The implementation is highly configurable, easily extensible, and comes with many optimizations and customizations. For example, beside the necessary reachability and productivity checks as described in Section 3.1.4, TRIBBLE can detect duplicate alternatives, or automatically merge concatenations of literals into single literals to reduce the number of nodes in the grammar graph.

Further, in addition to the heuristics and configurable modifications mentioned in Sections 3.1.2 and 3.1.3, TRIBBLE comes with implementations of multiple generation algorithms, which allows it to be used as a practical evaluation base for empirical fuzzing comparisons.

Apart from providing the implementation of the *k*-path algorithm, TRIBBLE is used as the technical base in several additional works, including the probabilistic fuzzing [114], and ALHAZEN [65] projects. This is possible thanks to TRIBBLE's well-designed grammar model and readily available documentation.

3.2 Evaluating *k*-Path Generation

Now that we have established a way of systematically covering *k*-paths, we want to study its effects on testing as compared to regular grammar-based fuzz testing. We compare our approach against the grammar-based input generator GRAMMARINATOR [54], which can be considered state of the art at the time of our implementation.

GRAMMARINATOR expects its grammars to be in the format defined by the ANTLR parser generator [100]. Therefore, for our experiments we selected the grammars of popular and well known languages, among other sources, from a popular GitHub repository hosting a variety of ANTLR grammars [38], and manually translated them into the format required by TRIBBLE as consistently as possible, i.e., only changing their notation and refraining from any refactoring or optimizing. Specifically, we chose the grammars for JSON [16], URL [124], CSV [26], and Markdown [79].

Table 3.1 provides a comparison of the grammars in terms of the number of productions as well as the average number of inputs produced by the *k*-path

Table 3.1 / Grammars and Sizes.

Grammar	Productions	Average # of inputs generated			
		$k = 1$	$k = 2$	$k = 3$	$k = 5$
JSON [16]	17	40	35	58	201
CSV [26]	12	42	38	51	221
URL [124]	27	43	45	72	552
Markdown [79]	236	653	980	1880	11409

algorithm as defined in Algorithm 1 when using bounded random generation from Algorithm 3 as the close-off procedure.

Perhaps an interesting observation about Table 3.1 is the fact that for JSON and CSV covering $k = 2$ requires generating fewer files than for $k = 1$. As we have established previously in Section 2.5, 1-path coverage corresponds to simply covering all symbols in the grammar. This means that when generating trees, the k-path algorithm is only trying to derive single symbolic nodes regardless of their context, and thus has no incentive to generate derivation trees that are deep because as soon as it attains a targeted node, the close-off is taking over. As both these grammars are rather shallow, the close-off phase terminates soon and does not cover many outstanding 1-paths "accidentally". This way, the main algorithm happens to create many trees to reach every symbol. This effect disappears for larger values of k because the k-path algorithm is now required to produce trees that are at least of depth k, which is why more k-paths are encountered along the way.

3.2.1 Test Subjects

We carry out our experiments on the open-source projects listed in the leftmost column of Table 3.2. The selection consists of some of the top search results among open-source projects consuming the previously selected formats.

For the JSON language, almost all our subjects are parsers, except for some notable exceptions: `jackson-databind`, `genson`, `gson`, `fastjson` additionally allow data-binding for automatic serialization and deserialization of JSON objects from and into data classes. The subjects `json-flattener` and `pojo` serve the purpose of flattening a JSON structure and generating Java source code, respectively.

The subjects for CSV are all parsers capable of data-binding. However, our tests only engage the part of their functionality related to parsing because it is impractical to pre-generate data classes for dynamically generated inputs. The same holds for the data-binding JSON subjects.

For URL, the projects `galimatias` and `jurl` are pure parsers, while `autolink` and `url-detector` additionally detect URLs inside arbitrary plain text before parsing them into their constituent parts.

Our subjects for the Markdown format concern themselves with rendering their inputs into HTML fragments that are suitable for display inside a web browser.

Since most of the subjects are libraries, they require a test harness to process inputs. For each of the subjects we implemented a launcher which instantiates the necessary structures, sets any available options, and feeds an input file into the main API functions covering the documented use cases. In cases where the subject is an executable, the launcher is simply a wrapper around its main method.

3.2.2 Experimental Setup

Our competitor fuzzer Grammarinator requires two parameters: d and n, the maximum depth and number of the derivation trees to be generated, respectively. For a fair comparison, we first run the k-path algorithm with a given k, and take the number of the generated inputs to be n for a corresponding run of Grammarinator. We set the depth d to 30 for both tools because we found this number in the configuration repository [35] provided by the authors of Grammarinator. Further, we set the parameter `--cooldown` to 0.9 and add a `simple_space_transformer` as described in [54]. Due to randomness, we repeat the invocation of each algorithm 50 times. We repeat the above setup for several different values of k to investigate the influence of the path length.

3.2.3 Evaluation Results

We subdivide our findings into two categories: code coverage and defect detection. The former provides an established and well-recognized metric for comparing the performance of any sort of test generator, while the latter can give valuable insights into the impact of testing in real-world scenarios.

Code Coverage

Table 3.2 shows the average branch coverage achieved by each tool over 50 runs. Since all our subjects are targeting the Java platform, we use the JaCoCo tool [55] to gather coverage data by means of offline bytecode instrumentation. The subjects are given in Table 3.2 and are grouped by the grammar describing the language of their inputs: JSON, CSV, URL, and Markdown. The columns labeled as k-path show the average branch coverage achieved with inputs generated by the k-path algorithm with the given value of k.

The columns labeled as gramm_k show the average branch coverage for runs of Grammarinator that were carried out with the n parameter mirroring the number of inputs that were produced by runs of k-path with the given k. For example, if an invocation of 2-path produced a set of 10 inputs, the corresponding gramm_2 run would also consist of 10 inputs.

Note that for each k, the values in Table 3.2 represent the average of 50 such corresponding pairs rounded to four decimals. To investigate if these average values do, in fact, indicate that one of the approaches consistently outperforms the other, we carried out a statistical significance analysis using the two-sided Mann–Whitney U test [80] as implemented in the Python SciPy library [128]. In Table 3.2, the significantly different entries (all but six) are shown in bold.

Table 3.2 / Average branch coverage achieved by TRIBBLE and GRAMMARINATOR.

	Subject	1-path	gramm_1	2-path	gramm_2	3-path	gramm_3	5-path	gramm_5
JSON	`argo` [8]	**0.4116**	0.4000	**0.4187**	0.3963	**0.4197**	0.4092	**0.4242**	0.4187
	`fastjson` [2]	0.0364	**0.0376**	**0.0404**	0.0374	**0.0413**	0.0388	**0.0431**	0.0414
	`genson` [19]	0.0842	**0.0866**	**0.0886**	0.0864	**0.0902**	0.0883	**0.0916**	0.0905
	`gson` [44]	0.2080	**0.2215**	**0.2264**	0.2213	**0.2294**	0.2266	0.2352	0.2371
	`jackson-databind` [59]	0.0886	**0.0926**	**0.0932**	0.0924	0.0938	0.0935	0.0940	**0.0952**
	`json-flattener` [61]	0.5039	**0.6246**	**0.6828**	0.6235	**0.7127**	0.6609	**0.7809**	0.7475
	`json-java` [72]	0.1093	**0.1377**	**0.1457**	0.1310	**0.1661**	0.1441	**0.1890**	0.1733
	`json-simple` [62]	0.4427	**0.4695**	**0.4870**	0.4662	**0.4931**	0.4836	**0.5093**	0.5062
	`json-cliftonlabs` [76]	0.3355	0.3325	**0.3445**	0.3301	**0.3446**	0.3410	0.3478	**0.3546**
	`minimal-json` [117]	**0.4158**	0.3970	**0.4267**	0.3936	**0.4163**	0.4054	0.4174	0.4166
	`pojo` [75]	0.1246	**0.1414**	**0.1428**	0.1423	0.1597	0.1433	**0.2112**	0.1462
CSV	`commons-csv` [4]	**0.3828**	0.3773	**0.3903**	0.3772	**0.3984**	0.3770	**0.4034**	0.3799
	`jackson-csv` [58]	**0.1665**	0.1527	**0.1666**	0.1523	**0.1700**	0.1532	**0.1777**	0.1573
	`jcsv` [60]	**0.3287**	0.3167	**0.3337**	0.3152	**0.3374**	0.3201	**0.3400**	0.3270
	`sfm-csv` [112]	0.0628	**0.0686**	0.0664	**0.0686**	0.0675	**0.0686**	0.0682	**0.0686**
	`simplecsv` [113]	**0.3472**	0.3377	**0.3482**	0.3368	**0.3481**	0.3395	**0.3489**	0.3439
	`super-csv` [119]	**0.1560**	0.1433	**0.1589**	0.1423	**0.1646**	0.1439	**0.1646**	0.1471
URL	`autolink` [118]	**0.4514**	0.2861	**0.4673**	0.2861	**0.5716**	0.2859	**0.6265**	0.2889
	`galimatias` [36]	**0.0879**	0.0343	**0.0897**	0.0341	**0.0875**	0.0346	**0.0873**	0.0356
	`jurl` [111]	0.6790	**0.6854**	0.6807	**0.6872**	0.6933	0.6904	**0.7095**	0.7012
	`url-detector` [106]	**0.4057**	0.3273	**0.4083**	0.3244	**0.4188**	0.3342	**0.4352**	0.3458
MD	`commonmark` [10]	**0.6678**	0.6253	**0.6991**	0.6322	**0.7183**	0.6419	**0.7335**	0.6634
	`markdown4j` [81]	0.6772	**0.6817**	**0.7094**	0.6851	**0.7162**	0.6931	**0.7313**	0.7129
	`txtmark` [123]	0.6017	**0.6144**	**0.6291**	0.6174	**0.6348**	0.6237	**0.6498**	0.6413

Values show the fraction of branches covered. All results are averages over 50 runs.
Bold values indicate significantly higher values according to the Mann–Whitney U test [80]. ($p < 0.005$)

Our findings in Table 3.2 show that for $k = 1$ the average coverage achieved by inputs generated by the k-path algorithm roughly equates the one produced by GRAMMARINATOR's inputs across all subjects.

When considering only subjects consuming JSON inputs, GRAMMARINATOR still outperforms TRIBBLE on all but three subjects. This is due to the 1-path algorithm not being interested in actively covering any combinations and nesting of JSON arrays and objects, which are responsible for triggering additional behavior in the subjects.

When the context depth k is set to 2, however, this disadvantage disappears as TRIBBLE now covers more code in all but two subjects. Because this time 2-path actively tries to cover pairs of elements, its code coverage is much higher than the one achieved by 1-path.

Setting $k = 3$ further strengthens the performance of k-path as it is now seeking to cover all contexts of depth 3. Once again, there is an improvement over the previous configuration.

Increasing the context depth to a value of 5 improves the achieved coverage over the previous configurations, but this time, GRAMMARINATOR is beginning to catch up again. To produce these additional combinations, however, more inputs had to be generated by the 5-path generator (see Table 3.1), and so GRAMMARINATOR also has a much higher generation budget n as well.

An additional reason for this loss of advantage of the k-path generator with growing k (or an effect of diminishing returns, if you will) is due to contexts deeper than a certain threshold no longer explicitly corresponding to variations in the executed code.

For instance, in a typical recursively descending JSON parser, there is not much difference in executed control flow between parsing nested JSON structures that are nested two, three, or more times. However, there can still be some notion of context encoded in the flow of data instead. For example, a counter could be keeping track of the current nesting depth used for matching the correct number of closing brackets. Changes in its state would not be reflected in code coverage, even though it might still make sense to strive for testing some of the values the counter can assume.

Our empirical investigation shows that the advantage of TRIBBLE over GRAMMARINATOR can be quite large. In the cases of `autolink` and `galimatias`, TRIBBLE achieves about twice the coverage of GRAMMARINATOR, even for 1-path already. There are no cases in which GRAMMARINATOR would outperform TRIBBLE by the same margin.

As an aside, when it comes to performance considerations, both tools have approximately the same wind-up time that includes parsing the grammar, building the in-memory model, and precomputing static information such as minimal required derivation depth for all nodes. And while in theory the additional computations that TRIBBLE has to perform to obtain its k-path generation agenda has worst case runtime in $O(|V|^2)$ with V being the set of all nodes in the grammar graph, we found that in practice there was barely any perceivable slowdown for our grammars.

In fact, TRIBBLE is oftentimes faster than GRAMMARINATOR because of the shortcuts in the k-path algorithm. Once a deep derivation tree has been attained, all its k-paths are removed from the generation agenda, and thus we explicitly avoid having to generate similar trees, which results in faster generation overall.

In our experiments, executing the test subjects turned out to be the real time sink, with the evaluation process spending upwards of 99 % of its time on this task. It is for this reason, and because it is simply practically more convenient, that we ensure the fairness of our comparison by equating the number of input files instead of the wall clock time.

Defect Detection

When generating test inputs, one must not forget why we test in the first place. While code coverage is a well-known proxy measure of test quality, equally as important is the tests' ability to reveal defects. A test suite may consistently achieve high coverage throughout the lifetime of a project without ever detecting a single error or bug. Such a test suite is not impactful for the project and may end up costing more in resources in maintaining it than it is worth.

During our experiments, we found a number of exceptions thrown by our test subjects. In our setting, all of these faults are triggered by system inputs, so they indicate real internal errors, that can occur in regular production use. When considering which exceptions are indeed defects and not legitimate errors in usage, we filter out those exception classes that are defined inside the subjects' own packages assuming they represent expected user-facing error behavior.

Our observations are given in Table 3.3: For each subject in which exceptions could be triggered, the exception, its origin, as well as its detection rate are given for both TRIBBLE and GRAMMARINATOR. The detection rate shows in what fraction of runs a given exception was triggered at least once at the given location.

The *location unknown* entry in the `json-flattener` subject is a result of our test harness failing to provide a stack trace for this particular failure. The `InvalidSyntaxException` thrown by `argo`, and `ParseException` thrown by `json-flattener`, which are triggered exclusively by GRAMMARINATOR might indicate a bug in GRAMMARINATOR's implementation of input generation or in its input grammar rather than in the subjects themselves. A similar effect can be observed for both exceptions thrown by `galimatias`, but for TRIBBLE instead.

If we discount the four of these likely non-issues, we see that in the configuration with $k = 1$, the k-path algorithm is able to trigger three exceptions exclusively: Two `NullPointerExceptions` and one `StringIndexOutOfBoundsException`, none of which should ever be allowed to be thrown into user code as they indicate fatal errors in the internal state.

With increasing k, the detection rate increases for both approaches, but it does so more reliably for TRIBBLE: There are only two cases of regression for TRIBBLE, both in the `txtmark` subject, while there are four for GRAMMARINATOR, distributed over three subjects expecting three different input formats.

By the 5-path configuration, out of the 23 exceptions triggered, 3 are unique to GRAMMARINATOR, 8 to TRIBBLE, and the remaining 12 were found by both.

Our evaluation shows that in the average case, thanks to the systematic approach to generating its inputs, TRIBBLE can be expected to detect issues more reliably than an approach like GRAMMARINATOR, which is inherently randomized in its generation choices.

Table 3.3 / Exception Detection Rates.

	Subject	Exception	Location	Detection Rate			
				1-path gramm$_1$	2-path gramm$_2$	3-path gramm$_3$	5-path gramm$_5$
JSON	`argo`	argo.saj.InvalidSyntax	argo…InvalidSyntaxRuntime$3:60	0% **76%**	0% **76%**	0% **84%**	0% **100%**
	`genson`	NullPointerException	com…genson.stream.JsonWriter:414	100% 100%	100% 100%	100% 100%	100% 100%
	`json-flattener`	…json.ParseException	com…wnameless…Flattener:122	0% **76%**	0% **76%**	0% **84%**	0% **100%**
		NullPointerException	com…wnameless…Unflattener:393	88% **90%**	**94%** 88%	**100%** 94%	100% 100%
			com…wnameless…Unflattener:409	**4%** 0%	6% 6%	**10%** 4%	22% **26%**
			location unknown	0% 0%	0% 0%	0% 0%	**2%** 0%
	`pojo`	StringIndexOutOfBounds	org.jsonschema…NameHelper:46	98% **100%**	100% 100%	100% 100%	100% 100%
CSV	`commons-csv`	IOException	org.apache.commons.csv.Lexer:281	100% 100%	100% 100%	100% 100%	100% 100%
			org.apache.commons.csv.Lexer:288	100% 100%	100% 100%	100% 100%	100% 100%
	`jackson-csv`	CharConversionException	com.fasterxml…CsvDecoder:429	0% 0%	0% 0%	**2%** 0%	**4%** 0%
			com.faster…ParserBootstrapper:383	0% 0%	0% 0%	**2%** 0%	**4%** 0%
	`jcsv`	IllegalStateException	com.googlecode…TokenizerImpl:73	**100%** 30%	**100%** 22%	**100%** 46%	**100%** 78%
	`sfm-csv`	IllegalStateException	org…$NoColumnCsvWriterDSL:449	100% 100%	100% 100%	100% 100%	100% 100%
	`super-csv`	NullPointerException	org…io.AbstractCsvWriter:177	0% 0%	0% 0%	0% 0%	**2%** 0%
			org.supercsv.util.Util:187	**34%** 0%	**38%** 0%	**76%** 0%	**52%** 0%
URL	`galimatias`	MalformedURLException	io.mola.galimatias.URL:527	**100%** 0%	**100%** 0%	**100%** 0%	**100%** 0%
		URISyntaxException	io.mola.galimatias.URL:509	**92%** 0%	**94%** 0%	**96%** 0%	**96%** 0%
	`jurl`	StringIndexOutOfBounds	com.anthony…PercentEncoder:176	**100%** 0%	**100%** 0%	**100%** 0%	**100%** 0%
Markdown	`markdown4j`	StringIndexOutOfBounds	org…Markdown4jProcessor:53	30% **100%**	100% 100%	100% 100%	100% 100%
	`txtmark`	StringIndexOutOfBounds	com…rjeschke.txtmark.Block:106	8% **34%**	8% **30%**	6% **62%**	6% **98%**
			com…rjeschke.txtmark.Emitter:282	4% **100%**	2% **100%**	12% **100%**	82% **100%**
			com…rjeschke.txtmark.Emitter:303	0% 0%	0% 0%	0% 0%	0% **4%**
			com…rjeschke.txtmark.Line:520	22% **76%**	**100%** 88%	**100%** 98%	100% 100%

Values show the percentage of runs in which the given exception was detected. Higher percentages are shown in **bold**.

3.2.4 Threats to Validity

The experimental evaluation is empirical in nature, especially because both GRAMMARINATOR and the *k*-path algorithm come with inherent randomness, and therefore the observations naturally face threats to validity.

When it comes to *external validity*, we have examined 24 subjects as well as four grammars, covering a variety of input and implementation features; while our observations are consistent, we cannot claim generality of the results across all programs and inputs. All our subjects are written in the Java programming language, which might come with some constraints on the represented code architecture and control structure paradigms. Additionally, we have opted to use the most easily accessible sources of grammars for our subject formats. However, variations in the formulation of derivation rules of a grammar might affect the behavior of a fuzzing algorithm. At present, there is no systematic research into the nature and extent of such effects.

In terms of *internal validity*, we have taken great care in validating our findings, notably by using well-established tools for computing code coverage, statistical significance, and validating grammar coverage during construction. We repeated the experiment with different initial seeds to offset the effects of randomness that is an essential element in both algorithms, and we used established techniques to compute statistical significance of the observations. To ensure fidelity of results we use the same initial conditions for both approaches in terms of the number of input files. In addition, the entire tool chain from experiment configuration to collected data is fully automated and tested, greatly mitigating the risk of human error; all data and tools are available for external replication and validation.

3.3 Summarizing *k*-Path Generation

Let us recap what we have observed so far. We begin by introducing a practical notation for context-free grammars and a graph representation that goes along with it. From here, we establish the concept of *k*-paths as a means to characterize variety in inputs by considering the context in which grammar symbols occur.

In the next step, we use this definition in a constructive manner for generating inputs that contain all features described by *k*-paths for a given value of *k*.

Finally, we find empirical evidence of the efficacy of the proposed generation algorithm in reaching code coverage and revealing defects. From the evaluation, we can glean a sweet spot for the size of the context *k* for practical purposes of fuzzing. It appears to lie somewhere around $k = 3$ and seems to produce relatively small sets of small files that exhibit strong structural variety. Any less, and we are not better than random grammar-based generation, any larger, and we run into diminishing returns.

Chapter 4

Correlating Input Coverage with Code Coverage

So far, we have established that sets of inputs with full k-path coverage achieve higher code coverage than random input sets of equal size. We have seen empirical evidence to support this claim in Section 3.2, but at the same time, we have also witnessed the effects of diminishing returns when choosing the context size k to be too high. Such a setting requires generating considerably more inputs to reach full k-path coverage, and while the k-path generation algorithm guarantees that all newly generated inputs contain as yet unseen k-paths, it does in no way guarantee that these inputs will contain any more than just a single new k-path each. In fact, if the close-off algorithm always chooses the shortest possible derivations, the majority of the generated inputs will differ in only few features. This effect on the number of inputs can be seen in Table 3.1.

It is important to note that the same holds to some extent irrespective of the choice of the generation algorithm. The larger the size of the requested context, the more conflicting choices must be made when deriving. In other words, if an input contains a given large k-path, it is not likely to also contain many other large k-paths because the larger the number of nodes in a path, the more conflicting alternatives are part of it. Since a derivation tree can contain only one alternative for each alternation at a time, it cannot possibly fit multiple k-paths with such conflicting alternatives. Even if the grammar graph has a loop that goes through the alternation in question, by construction it must introduce at least one additional symbolic node, which in the majority of cases will not be part of the requested k-paths. Therefore we can say that in general, the higher the value of k, the more inputs are required to cover all k-paths.

However, the results from Section 3.2 leave open the question whether there is a dependency between k-path coverage and code coverage. Therefore, in this chapter we would like to analyze the nature of the relationship between k-path coverage and code coverage.

4.1 Empirical Evaluation

In the next experiment, we are guided by the following question: If *high* k-path coverage leads to high code coverage, does, conversely, *low* k-path coverage lead to low code coverage (i.e., is this dependency monotonic)? We now aim to find out details of any such relationship if it exists.

4.1.1 Generating Inputs

To gather any empirical evidence, we first require a sufficient number of sets of inputs that have k-path coverage ranging from almost none to full so that we can measure the code coverage they induce in subject programs. As it happens, we can adapt the original k-path algorithm to produce just such a batch of sets. Algorithm 4 shows the approach. Since a set of trees is commonly referred to as a *forest*, and the algorithm produces a set of forests, we shall refer to it as the forestation algorithm.

Algorithm 4 The Forestation Algorithm

```
1:  function Forestation(grammar graph G, positive natural number k)
2:      sets ← {}
3:      r ← root(G)
4:      for all P ⊆ paths_k(G) do          ▷ Iterate over all subsets of the k-path set
5:          forest ← {}
6:          while P ≠ {} do
7:              t ← tree(r)
8:              s ← child slot of r
9:              p ← remove next k-path from P
10:             if head(p) = r then
11:                 p ← tail(p)
12:             while p is not empty do
13:                 n ← expansion of s
14:                 Add n to t by filling slot s
15:                 if n = head(p) then
16:                     p ← tail(p)
17:                 if p is not empty then
18:                     s ← child slot of n with shortest derivation path to head(p)
19:             t ← CloseOff(G, t)
20:             forest ← forest ∪ {t}
21:             Remove from P all k-paths found in t
22:         sets ← sets ∪ {forest}
23:     return sets
```

In the k-path algorithm, as given in Algorithm 1, the main loop in Line 4 iterates over *all* k-paths contained in the given grammar graph. We can instead produce subsets of this initial set of k-paths, and then use the rest of the algorithm as is to generate inputs that reach these subsets. Of course, we cannot guarantee that the derivation trees generated in this way will not contain any additional k-paths that are not contained in the requested subsets. Nevertheless, this approach presents a practical solution for our purposes of acquiring test data.

4.1.2 Experiment Setup

To acquire a significant number of forests displaying varying k-path and code coverage, we repeat the invocation of Algorithm 4 ten times. Table 4.1 shows the number of forests that were generated by the forestation algorithm for each grammar after ten invocations. As for our test programs, we fall back on our previous subjects which we have already seen in Table 3.2.

Table 4.1 / Experiment Size.

Grammar	Forests
JSON	6421
CSV	111
URL	15 711
Markdown[1]	68 831

The experiment itself is quite straightforward: We execute every subject program with every forest generated with the appropriate grammar and measure the code coverage induced. By construction, we also know the k-path coverage of each forest we generated. For the purposes of our evaluation, we consider values of k from one to five. With this information about coverage collected for every forest, we can attempt to calculate a *correlation* and assess its strength.

As we have no reason to assume that any such correlation need necessarily be strictly linear, we must not use Pearson's correlation coefficient. We do, however, suspect that there might be a monotonic correlation instead, i.e., the higher the k-path coverage of a forest, the higher its resulting code coverage. Therefore, we compute Spearman's rank correlation coefficient [116].

Its value $\rho \in [-1, 1]$ signifies how well a monotonic function describes the relationship between two variables. When ρ is close to 1, there is a perfectly monotone *increasing* relationship, whereas a value close to -1 signifies a perfectly monotone *decreasing* relationship. Values between those extrema indicate a correlation of ever diminishing strength, with a value of zero meaning no correlation at all.

4.1.3 Interpreting Results

Table 4.2 shows the results of the experiment for our subjects. Specifically, it shows the values of ρ indicating the correlations between branch coverage and k-path coverage for values of k ranging from one to five. We can see that all reported values are positive, indicating that the two coverage measures have an *increasing* monotonic relationship, which confirms our suspicions from earlier. Further, an overwhelming majority of all observed values of ρ are much closer to 1 than to 0, which means that the correlations tend to be strong to very strong.

Notably, this correlation holds approximately equally well across all values of k and all subjects. The strength of the correlation provides evidence that the connection between grammar coverage as represented by k-path coverage and code coverage as represented by branch coverage is not merely coincidental. Further, we can say that increasing grammar coverage is highly unlikely to result in a reduction in code coverage, which makes high grammar coverage a generally desirable property.

[1] The subjects were only able to consume 68 831 out of the 113 266 inputs generated by the forestation algorithm after a week of runtime.

Table 4.2 / Spearman's rank correlation coefficient ρ between k-path coverage and branch coverage. The results are statistically significant as $p < 0.001$ holds for all entries.

	Subject	Spearman's ρ				
		1-path	2-path	3-path	4-path	5-path
JSON	argo	0.978	0.979	0.977	0.978	0.978
	fastjson	0.890	0.892	0.892	0.894	0.894
	genson	0.719	0.718	0.716	0.716	0.716
	gson	0.785	0.786	0.784	0.784	0.784
	jackson-databind	0.671	0.668	0.667	0.667	0.667
	json-flattener	0.834	0.832	0.832	0.833	0.833
	json-java	0.855	0.855	0.854	0.856	0.857
	json-simple	0.936	0.931	0.926	0.926	0.926
	json-cliftonlabs	0.923	0.918	0.913	0.914	0.914
	minimal-json	0.869	0.864	0.860	0.861	0.861
	pojo	0.921	0.918	0.915	0.916	0.917
CSV	commons-csv	0.860	0.848	0.885	0.884	0.890
	jackson-csv	0.844	0.856	0.818	0.817	0.802
	jcsv	0.873	0.822	0.858	0.858	0.880
	sfm-csv	0.516	0.481	0.462	0.462	0.464
	simplecsv	0.862	0.811	0.821	0.820	0.828
	super-csv	0.918	0.863	0.881	0.882	0.888
URL	autolink	0.692	0.791	0.796	0.796	0.797
	galimatias	0.681	0.757	0.774	0.770	0.769
	jurl	0.579	0.573	0.578	0.578	0.578
	url-detector	0.680	0.771	0.775	0.776	0.776
MD	commonmark	0.878	0.886	0.887	0.887	0.887
	markdown4j	0.875	0.885	0.886	0.886	0.886
	txtmark	0.880	0.891	0.893	0.893	0.893

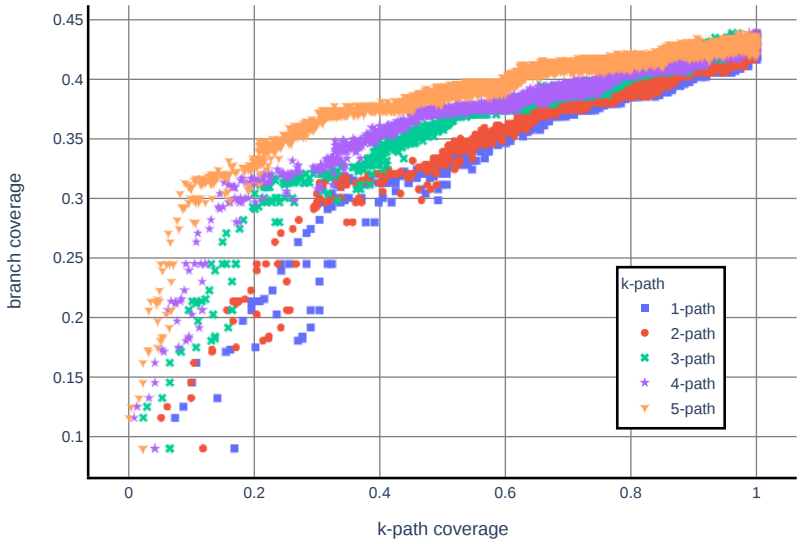


Figure 4.1 / Branch and *k*-path coverage measured for the `argo` JSON parser subject.

To inspect the nature of this connection somewhat closer, let us consider a scatter plot of the coverage values in question for the `argo` JSON parser as a representative[2] subject given in Figure 4.1. It displays characteristics that are very similar in the rest of our subjects as well.

Interestingly, we can observe that for almost any given level of branch coverage reached, the higher the value of k, the smaller is the k-path coverage that is seemingly sufficient to reach it. For example, forests that achieve a branch coverage of 0.35 also tend to have a 1-path coverage of around 0.6, a smaller 4-path coverage of around 0.4, and a much smaller 5-path coverage of only approximately 0.22.

This seems like a highly useful property in the context of test generation. For example, a test generator may consider taking advantage of this property by increasing the value of k and at the same time reducing the k-path *coverage* requirement when generating files, thus reducing the number of inputs generated but ostensibly achieving a high level code coverage nonetheless.

One explanation for this observation might lie in the not quite subsumption-like relationship that k-path coverage exhibits for different values of k. While we have seen in Section 2.5.3 that we cannot establish a strict subsumption relationship in the general case, we can still unquestionably say that longer k-paths include shorter ones.

Another explanation for this pattern of higher k-path coverage achieving certain code coverage "earlier" may have to do with specific k-paths corresponding to

[2] Figure A.1 gives the plots for all other subjects as well.

"unlocking" additional parts in the code. If some part of the code is guarded by a condition that reflects the presence of a specific structure in the input, then only those inputs will be able to execute the code, that do have this structure present. Alternatively, if this particular structure is absent, some other parts of the code may be executed instead, making this kind of inputs equally as valuable for testing.

As an example, consider a parser that turns JSON inputs into typed collection objects in memory. A JSON object with integer-valued properties would end up as a dictionary of type `Map<String, Integer>`, while a JSON array of strings would find its correspondence in a list of type `List<String>`. The allocation of these collections takes place in different parts of the code that are executed independently of each other, but dependently on the presence of objects and arrays in the JSON input. Therefore, an input combining both structures in a nested context, which can be characterized by a longer k-path, will achieve more coverage than an input that contains at most one of them, and thus also a shorter k-path. In fact, we will investigate the connection k-paths have to locations in code in more detail in the upcoming Chapter 5.

4.2 Summarizing the Correlation

In this chapter, we have seen empirical evidence for a positive correlation between input coverage, as represented by k-path coverage, and code coverage, represented by branch coverage. Such a correlation indicates that increasing the grammar coverage may also lead to an increase in code coverage in practice. Given that k-paths tend to express distinct parts of the language, the existence of such a connection to constructs in the code is not surprising.

As an unintended, yet welcome side effect, we have received an additional grammar-based generation algorithm, capable of producing sets of inputs over a wide range of k-path coverage values. Investigating its suitability as a fuzzer of its own is left for future work, however it is hard to imagine it being very effective, given its inherently large run time by construction.

Chapter 5

Associating Input Features with Code Locations

So far, in Chapter 2 we have established the construct of k-paths as an effective means for describing certain context-based features of inputs. In Chapter 3 we leveraged this notion to exhaustively cover such features by producing sets of inputs whose derivation trees contain all possible k-paths. Doing so has shown to yield adequate overall code coverage and defect detection in the tested subject programs. However, we also observed that the number of inputs required gets rather large for realistic grammars and high values of the context depth k. In Chapter 4 we have seen indications of what appears to be a monotonic relationship between k-path coverage and code coverage, and also that under some circumstances reaching a threshold in code coverage does not require *full* k-path coverage of the input set.

With this in mind, now has come the time to shift our attention to an aspect of fuzzing which, so far, we have left without consideration: performance. In realistic scenarios, it is not uncommon for the execution of the system under test (SUT) to be the main bottleneck when performing system-level (fuzz) testing. We are particularly interested in fuzzing at the system level because it completely absolves us from the costly, mostly manual, task of filtering out false positives from the findings since any and all system-level crashes are real results stemming from a user-facing interface.

In the system-level use case, one must always execute the entire program from reading in the input to finally producing a result or crashing – an act, which is likely to be expensive and time-consuming for any reasonably extensive and fuzz-worthy application. To accelerate execution, there are techniques that allow taking a snapshot of a program state and forking off copies to process different inputs [134, 63]. However, the best such approaches can do is to spare us only the initialization efforts since we still need to execute the entire business logic that we want to test. At the same time, while executing at the system level, realistically, we are mostly interested in focusing the testing efforts on some particular aspects of a subject's behavior.

```
Expr := AddExpr0;
AddExpr := MultExpr2
        | AddExpr5 ("+"10 | "-"11) MultExpr7;
MultExpr := UnaryExpr8
        | MultExpr13 ("*"23 | "/"24 | "%"25) UnaryExpr15;
UnaryExpr := Identifier16
        | "+"27 UnaryExpr28
        | "-"29 UnaryExpr30
        | "++"31 UnaryExpr32
        | "--"33 UnaryExpr34
        | "("35 AddExpr36 ")"37
        | DecDigits22;
DecDigits := DecDigit42+;
DecDigit := "0"44 | "1"45 | "2"46 | "3"47 | "4"48
        | "5"49 | "6"50 | "7"51 | "8"52 | "9"53;
Identifier := "x"39 | "y"40 | "z"41;
```

Figure 5.1 / Grammar for a subset of arithmetic expressions in the JavaScript programming language (excerpt). Duplicated from Figure 1.1 and enriched with numeric node identifiers. If not yet done, it is recommended to navigate to Appendix B, which contains a copy of this figure, and to print it out on an extra sheet to have it at hand for easy reference.

Sticking with the running example that has accompanied us this far, let us consider the following fuzzing scenario, where we try to illustrate what exactly we expect from a good fuzzer and how we can optimize its performance. First, let us revisit a setting briefly mentioned in the introduction. Our subject is going to be a compiler which takes JavaScript code as input, and produces a smaller and faster version of it as output. The optimizations implemented in its compilation passes are manifold and range from renaming identifiers to simplifying entire code regions.

Specifically, and for the sake of our example, we are interested in its capability to optimize arithmetic expressions, and, even more precisely, in its application of the distributive rule, which allows it to split expressions of the form $a \times (b + c)$ into $a \times b + a \times c$, so that the subterms can be optimized individually by further compiler passes. For example, this transformation might enable the compiler to replace expensive computations with some cheaper alternatives, such as substituting multiplication or division by powers of two with bit shift operations, or using precomputed multiplication tables for small values. For convenience, let us further assume that this functionality is implemented in a method called `distributive_rule()`.

For our example test scenario we are going to consider the realistic use case in which our target method is not considered to be sufficiently well tested. This can occur in many cases along the lifecycle of a program: from the inception of the method with no prior tests in existence, over optimizations and refactorings that may inadvertently change its functionality, to intentional changes in behavior

Table 5.1 / Sample inputs and whether they cover the `distributive_rule()` method.

Input[1]	Coverage	Input[1]	Coverage
1	☐	1 + 2	☐
(1 + 2)	☐	1 * 2	☐
1 + 2 * 3	☐	(1 + 2) * 3	☑
1 * (2 + 3)	☑	(1 + 2) * (3 + 4)	☑

such as extending or narrowing the scope of the method. In such cases, we would like for a fuzzer to be able to efficiently generate novel inputs that exercise our method of interest. In order to achieve variety among these inputs, we would like to leverage what we have learned about the make-up of inputs to establish associations between the features of inputs and the code they relate to.

Incidentally, the working principle behind this task can be seen as very similar to metamorphic testing [21]. In metamorphic testing, the goal is to leverage domain knowledge about the SUT to perform testing: If we change the inputs in a known manner, the behavior or result should change accordingly. In our use case, we wish to generate inputs that are supposed to execute the targeted method due to their make-up, and to ensure that it is indeed executed.

However, first we have to manifest these associations into entities that a fuzzer can interpret and use, and whose quality we can experimentally assess.

As an aside, throughout this chapter, we often refer to the JavaScript expression grammar first introduced in Figure 1.1. Because the numeric identifiers of its nodes are relevant, Figure 5.1 shows the grammar again, but this time with identifiers displayed for easy lookup.

5.1 Associating Coverage

Let us now begin with our first subtask, in which we would like to establish associations between features of inputs and a code location of interest such as a method. In the following, we shall refer to a method as *covered* or *reached* as soon as the system under test (SUT) enters it, i.e., we do not require its full execution. As a concrete example, consider several inputs and their coverage of our target method `distributive_rule()` as given in Table 5.1.

We as humans can, of course, easily recognize a pattern here. But how can we make a program infer such patterns automatically? Recalling from our observations in Chapter 4 that there seems to be a dependency between k-paths and code coverage, we choose to explore this connection further. Again, as we did several times previously already, we can leverage a grammar to parse the inputs and decompose them into their constituent features, which we can then use for predicting code coverage. Specifically, we can train *predictors* that enable us to make such associations.

[1]The inputs use "*" instead of "×" to indicate multiplication in accordance with the appropriate JavaScript expression grammar excerpt given in Figure 5.1.

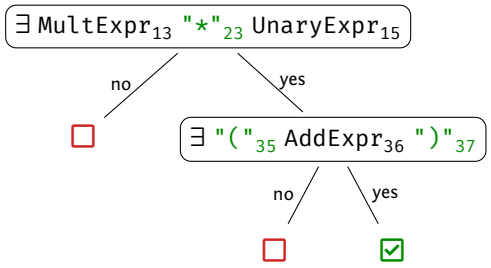


Figure 5.2 / Initial model for the coverage of `distributive_rule()`, trained on the inputs given in Table 5.1.

In terms of predictors, we need not look far, as the broad area of machine learning offers countless options, allowing to build predictive models from any data collection imaginable. Unfortunately, most of those models are uninterpretable in that any insights into their inner workings are hardly possible at all [74]. Nevertheless, we would like to profit from the approaches in this area. Since it is important for us to have our predictions be interpretable and actionable, keeping in mind our use case of generating novel inputs, we opt to fall back to some of the earliest advances in machine learning: decision trees [120].

A decision tree is a multi-level classifier, whose every node carries a constraint that splits the set of observations into those that fulfill it and those that do not. Following a path in the tree, every node adds an additional constraint, until all observations in the fulfilling set show the same behavior, or more formally, belong to the same *class*.

The properties on which the constraints of a decision tree discriminate are also called *features*. The simplest of such features can be binary in nature, e.g., representing the presence or absence of certain elements in the inputs. For our purposes, we can set the presence and absence of k-paths as the features of interest. Further, we want to classify our inputs in terms of whether or not they cover our targeted method, therefore we consider the two classes `covered` and `not-covered` also referred to as ☑ and ☐, respectively.

In order to obtain a decision tree, one requires a series of observations of the features and class of inputs, comprising so-called *training data*. The process that creates the decision tree from training data is called a decision tree learner because it learns the decisions that make up the tree from given data. An implementation of one such learner that we are going to use is freely available as part of the `scikit-learn` library [101].

So, how does a decision tree help us with our `distributive_rule()` case? Starting small, given the example inputs from Table 5.1, or more precisely, the presence or absence of productions from Figure 5.1 in each input, a decision tree learner will produce a tree like the one in Figure 5.2: If the input contains a multiplication (a production $\texttt{MultExpr}_{13}\ \texttt{"*"}_{23}\ \texttt{UnaryExpr}_{15}$) and an expression in parentheses (a production $\texttt{"("}_{35}\ \texttt{AddExpr}_{36}\ \texttt{")"}_{37}$), then `distributive_rule()` will be covered, otherwise not.

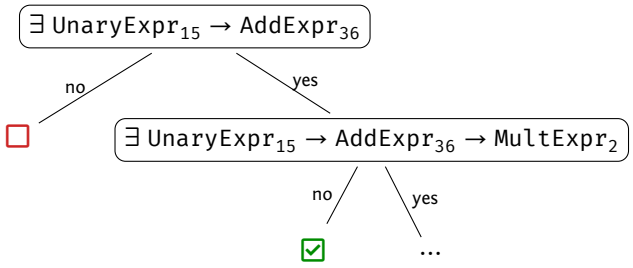


Figure 5.3 / Excerpt from an improved decision tree model for the coverage of `distributive_rule()`, obtained from 1000 generated inputs.

The decision tree in Figure 5.2 is consistent with all observations from Table 5.1. However, for the input "$(x * 2)$", it falsely predicts that `distributive_rule()` should be covered. This is because of two reasons:

a) the features in this tree do not carry any ordering information, so an input containing a $\texttt{MultExpr}_{13}$ $\texttt{"*"}_{23}$ $\texttt{UnaryExpr}_{15}$ on the inside of a $\texttt{"("}_{35}$ $\texttt{AddExpr}_{36}$ $\texttt{")"}_{37}$ is classified the same as one that has the proper expected $\texttt{MultExpr}_{13}$ $\texttt{"*"}_{23}$ $\texttt{"("}_{35}$ $\texttt{AddExpr}_{36}$ $\texttt{")"}_{37}$ expression order, and
b) the tree learner has not seen sufficient evidence to manifest the above distinction.

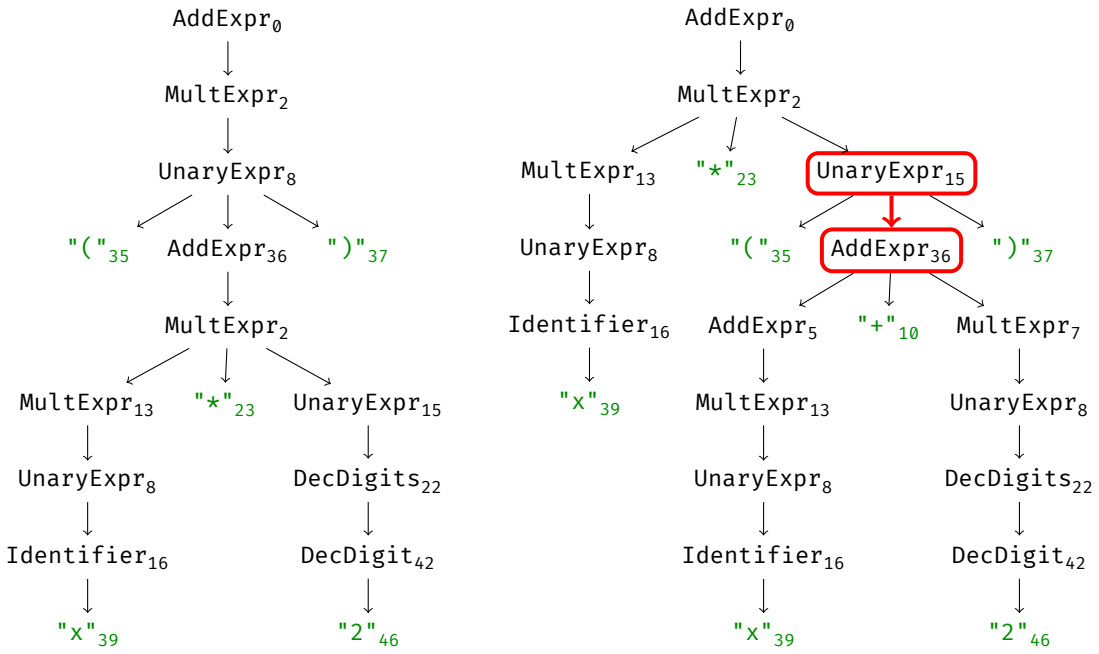
To address **a**), we opt to leverage the k-paths, which express specific derivation contexts. Chapter 4 has given us enough confidence that doing so will, in fact, enrich the trees with more precise constraints. As for **b**), the decision tree must be trained from more inputs. If we are in the fortunate situation, where we have a large body of regression test inputs available to us, we can directly profit from it to get more observations for our training data. Should this not be the case, however, remember that we still have the grammar and its proven generative capabilities at our disposal, e.g., as seen in Chapter 3. Note that while acquiring sufficiently much training data does require us to execute the SUT with many inputs, we can record the coverage of every observable method at once in a single pass.

Such an extended training results in a more refined tree, shown partially in Figure 5.3. This tree precisely expresses that a parenthesized expression (i.e., $\texttt{AddExpr}_{36}$) must occur as an expansion ($\rightarrow$) of the right-hand side of a multiplication (i.e., $\texttt{UnaryExpr}_{15}$). Further, the expression in question must not be a simple $\texttt{MultExpr}_{2}$, which guarantees by construction that it must involve an addition or a subtraction.

A decision tree such as the one in Figure 5.3 is of immediate use to us in our prediction scenario. Given any arbitrary input, we can parse it into its derivation tree and feed the k-paths found therein into the predictor trained for our method of interest, which is then able to give us a *classification* as to whether the method will be executed. The predictor does this by evaluating the criteria in its nodes and following the path until it arrives at a ☑ or a ☐ verdict. Since this does not involve executing the SUT, this process is very efficient.

As an example, consider the two inputs given in Figure 5.4. We want to use our trained decision tree to find out whether the input "$(x * 2)$" triggers the targeted method `distributive_rule()` without executing the SUT. Figure 5.4a shows its simplified derivation tree. Consulting our trained predictor from Figure 5.3, we must now answer the first question: Does the derivation tree of the input contain the 2-path $\texttt{UnaryExpr}_{15} \rightarrow \texttt{AddExpr}_{36}$? We see that it does not, and immediately conclude from the ☐ verdict that it does not cover the method.

What about "$x * (x + 2)$" given in Figure 5.4b? Here, we see that it does indeed fulfill the $\exists\ \texttt{UnaryExpr}_{15} \rightarrow \texttt{AddExpr}_{36}$ criterion. Therefore, we move further down the decision tree by following the "yes" edge. The next criterion to test is $\exists\ \texttt{UnaryExpr}_{15} \rightarrow \texttt{AddExpr}_{36} \rightarrow \texttt{MultExpr}_{2}$ and this time, it does not hold for our derivation tree. Therefore, we follow the "no" edge and end up with a ☑ verdict, from which follows that the input "$x * (x + 2)$" does cover the `distributive_rule()` method.



a / Derivation tree for input "$(x * 2)$". It is missing a relevant k-path, so its coverage class is ☐.

b / Derivation tree for input "$x * (x + 2)$". It contains a critical k-path, and is therefore labelled as ☑ by the decision tree in Figure 5.3.

Figure 5.4 / Two derivation trees demonstrating how a decision tree predicts method coverage.

Table 5.2 / Subjects and association quality of CODEINE.

	Subject	Number of Methods		Association Quality		
		Observed	Trainable	Accuracy	Precision	Recall
JSON	`argo` [8]	408	132	0.983	0.959	0.971
	`fastjson` [2]	1404	101	0.984	0.966	0.976
	`genson` [19]	986	113	0.986	0.964	0.969
	`gson` [44]	632	148	0.987	0.964	0.962
	`json-flattener` [61]	60	41	0.970	0.978	0.936
	`json-java` [72]	202	44	0.982	0.944	0.937
	`json-simple` [62]	54	17	0.997	0.988	0.994
	`json-cliftonlabs` [76]	78	17	0.987	0.989	0.988
	`minimal-json` [117]	199	100	0.974	0.946	0.928
	`pojo` [75]	451	150	0.988	0.953	0.934
URL	`autolink` [118]	43	20	0.946	0.948	0.952
	`jurl` [111]	50	38	0.941	0.956	0.977
	`url-detector` [106]	92	39	0.952	0.966	0.970
JS	`rhino` [92]	4531	599	0.954	0.846	0.825

5.1.1 Evaluating Coverage Associations

Of course, we cannot expect associations that are based on such an abstract concept as the presence and absence of k-paths to be perfect. However, we can perform an empirical evaluation to gain an estimate of their correctness in practice. In fact, it is important to ensure that we base our targeted generation efforts on an accurate representation of a method's call conditions to begin with.

The approach in question has been implemented in a tool called CODEINE, that is technically an extension of TRIBBLE as introduced previously in Section 3.1.5, which now also uses parts of the ALHAZEN tool [65] for interacting with decision trees.

And so we carry out our experiments on a set of subjects, most of which should be familiar to us already from previous chapters. In this experiment, we exclude the CSV formats due to the small size of the grammar and subject code, and Markdown due to the fact that this language has quite literally no invalid words, which makes it infinitely ambiguous. Ambiguous grammars present a big performance challenge for ALHAZEN because they admit multiple derivation trees for the same input, and so ALHAZEN tries to consider all of them at once. To make up for the loss, however, we introduce the Mozilla `rhino` subject, which is a JavaScript interpreter implemented in the Java language. Table 5.2 provides a full list of the subjects used in this experiment. The input formats are the same that we have used in Table 3.1, only the JavaScript grammar is a new addition in the heavyweight category with a total of 228 productions [32].

The experiment orchestration is implemented in Python by means of a pipeline process built on top of Spotify's Luigi framework [11]. The pipeline structure allows for an automated experiment that is easily run in parallel and is able to recover from errors.

The *Observed* column in Table 5.2 gives the number of methods CODEINE observed while executing each subject on the training input set. This number excludes constructors and static class initializers because we do not deem them to be

targets of particular interest. The *Trainable* column indicates for how many of those methods there was sufficient data to learn a decision tree (limited to 1000 at most for performance reasons). This involves filtering out methods which were either always or never executed in the training data obtained from the initial input set. As before, we use trusty JaCoCo [55] to instrument the subjects and recover coverage data.

Our experimental pipeline is set up in a way that allows us to investigate both the predictors in isolation and the targeted generation scenario in a single pass. First, the subjects are compiled and instrumented to report which methods are covered during execution and how often. Then, for each input language, we create a set of 1000 initial input files generated at random from the grammar using the Grammarinator approach [54] like we did in Section 3.2. We also adopt Grammarinator's default limit of 30 derivation steps as it seems to generate sufficiently diverse inputs across all our languages. And just to avoid any potential confusion, throughout this evaluation, the term "random" refers to *grammar-random* inputs generated this way.

Given this initial training set, we gather *features* for every input: We use the presence or absence of k-paths of up to a length of 4, which is the value used in a practical evaluation of the k-path approach by van Heerden, Raselimo, Sagonas, and Fischer [125]. Additionally, we discard features that have the same value (i.e., either present or absent) across all inputs and are thus useless to the decision tree learner.

Parallel to the feature extraction, the initial input set is fed into all subjects and method coverage data is recorded. From this data we also filter out methods that are never executed because there is no way to train a decision tree for something we have not observed. Likewise, we ignore methods that are always triggered because we do not even need a tree to explain their calling conditions. Table 5.2 gives the method numbers resulting from this filtering in its *Trainable* column.

In addition to removing methods with no variance, we also balance the dataset by sub-sampling the data, so that for every remaining method, the number of samples which reach the method is the same as the number of samples which do not reach the method, as recommended in [109]. We thus avoid the problem of learning trees that are heavily biased to either of the classes, as we want them to characterize the call conditions of a method as precisely as possible.

From here, we can train a decision tree for every trainable method using the features of the inputs. We set the maximum depth for the tree learner to be 5 as this value was successfully used in [65] to obtain good results. Also, we want to keep our trees *lean* to reduce the number of constraints Codeine will have to solve when generating inputs. Having trained the decision trees, we arrive at a central point where our "pipeline" actually branches out to address different concerns.

The first of these concerns is the evaluation of the associations between input features and code locations we have just learned in the form of our predictors. In order to evaluate these predictors, we can leverage classic evaluation methods from machine learning. Specifically, we can evaluate the precision, recall, and accuracy of the decision trees learned by Codeine.

For this, our evaluation pipeline creates another, independent set of random inputs, which we shall call our *test set*. We then run all subjects with these inputs and measure the real code coverage, establishing the so-called *ground truth*. Next, the decision trees in question are asked to classify the inputs in the test set according to their features, which gives us a *prediction* for each method and each input.

We can now compute the *accuracy* of the decision trees. We refer to our predictions as *true* or *false* depending on whether they are correct or not with respect to the ground truth, and as *positive* or *negative* depending on whether they predict that a method will be called (☑) or will not be executed (☐), respectively. Based on those definitions, we can derive the following measures:

Accuracy defined as $\frac{\text{true positives} + \text{true negatives}}{\text{number of observations}}$ is the fraction of correct predictions (`covered` or `not-covered`) over the number of samples.

Precision defined as $\frac{\text{true positives}}{\text{true positives} + \text{false positives}}$ denotes how many of the inputs reported to cover a method actually do so.

Recall defined as $\frac{\text{true positives}}{\text{true positives} + \text{false negatives}}$ denotes how many of the inputs covering a method are also classified as such.

All the above measures have values between zero and one, with one indicating the best results. Table 5.2 lists these values for all subjects in the columns *Accuracy*, *Precision*, and *Recall*, respectively.

From the results presented in Table 5.2, we can conclude that the decision trees as learned by Codeine do indeed predict the code coverage very well.

First, the predictions made by the trees are accurate at more than 94 % across all subjects. As observed earlier, the features are indeed relevant to a method being covered or not. This makes the trees valuable not only in conjunction with inputs which they can classify, but also on their own. A decision tree is very different from many other machine learning models in that its very structure gives information about the priorities with which individual features should be treated. In Section 5.1.2 we show how to translate our features into human-readable patterns, which can help in understanding the learned decision trees, making them useful even in the absence of inputs.

Second, while the accuracy is a measure of how often the tree is correct in its prediction, the precision concerns itself with how often a tree is correct in predicting the *interesting* case that a method is covered (☑). Upon inspecting our precision results, we can come to the conclusion that the trees are not only accurate because they simply predict non-coverage most of the time and just happen to be correct. Instead, the high precision indicates that the trees effectively characterize inputs that end up covering methods. The high recall further supports this interpretation: The trees identify almost all there is to identify in terms of method covering inputs.

Together, these measures indicate that the trained predictors do indeed present a good basis upon which we can attempt to implement our targeted generation use case.

5.1.2 Grammar Patterns

While having *k*-paths as features is perfectly fine for a classifier such as a decision tree, they do not give an intuitive understanding of the kind of input patterns they describe. Even in the presence of a grammar, it is difficult to mentally translate a *k*-path into something meaningful. However, the process required to do so is actually quite straightforward because it merely requires deriving a partial input by following the *k*-path.

For example, consider the *k*-path $\texttt{UnaryExpr}_{15} \rightarrow \texttt{AddExpr}_{36} \rightarrow \texttt{"-"}_{11}$. Reading from right to left and step by step, it describes a binary "-" operator in the context of an `AddExpr`. The $\texttt{"-"}_{11}$ can be obtained by choosing the second alternative of the `AddExpr` derivation rule, and the $\texttt{AddExpr}_{36}$ reference corresponds to the second to last alternative in the `UnaryExpr` rule. Working our way further up the *k*-path, we now need to locate the $\texttt{UnaryExpr}_{15}$ reference, which resides in the second alternative of the `MultExpr` rule.

With this, we can now evaluate the entire *k*-path into a single, synthesized derivation rule, which is not part of the original grammar. First, we replace the $\texttt{UnaryExpr}_{15}$ by its location, namely the second alternative of the `MultExpr` rule:

$$\texttt{MultExpr}_{13}\ (\texttt{"*"}_{23} \mid \texttt{"/"}_{24} \mid \texttt{"\%"}_{25})\ \texttt{UnaryExpr}_{15}.$$

Next, we replace herein the $\texttt{UnaryExpr}_{15}$ reference by the location of $\texttt{AddExpr}_{36}$, which is the next element in our *k*-path. After this step, we end up with a synthesized derivation rule of the form

$$\texttt{MultExpr}_{13}\ (\texttt{"*"}_{23} \mid \texttt{"/"}_{24} \mid \texttt{"\%"}_{25})\ \texttt{"("}_{35}\ \texttt{AddExpr}_{36}\ \texttt{")"}_{37}.$$

Replacing $\texttt{AddExpr}_{36}$ by its definition, we obtain the next step:

$$\texttt{MultExpr}_{13}\ (\texttt{"*"}_{23} \mid \texttt{"/"}_{24} \mid \texttt{"\%"}_{25})\ \texttt{"("}_{35}\ \texttt{AddExpr}_{5}\ (\texttt{"+"}_{10} \mid \texttt{"-"}_{11})\ \texttt{MultExpr}_{7}\ \texttt{")"}_{37}.$$

Repeating this procedure one more time for the final *k*-path element $\texttt{"-"}_{11}$ results in the following derivation rule:

$$\texttt{MultExpr}_{13}\ (\texttt{"*"}_{23} \mid \texttt{"/"}_{24} \mid \texttt{"\%"}_{25})\ \texttt{"("}_{35}\ \texttt{AddExpr}_{5}\ \texttt{"-"}_{11}\ \texttt{MultExpr}_{7}\ \texttt{")"}_{37}.$$

If we remove the identifiers and quotes, we end up with an easy to understand pattern of the form

"MultExpr (/ | * | %) (AddExpr - MultExpr)".

In fact, this functionality is implemented as part of TRIBBLE, which can turn any valid *k*-path into such a pattern, provided the original grammar. Apart from helping with interpreting the criteria from our trained decision trees, such patterns also allow conveniently checking for changes in calling conditions of methods after large refactorings. For example, if a method designed to process inputs such as "++Identifier" is now also suddenly connected to patterns like "++DecDigits" after our refactoring, we might have forgotten to include a semantic checker on the path to this method.

As an additional side effect, this transformation of *k*-paths into patterns allows applying approaches such as "evocative patterns" introduced by Gopinath, Nemati, and Zeller [42]. Since our decision trees usually give us not only a single pattern but rather a conjunction of such patterns, we can immediately

apply the *definition conjunction* operation from [42] to obtain a single pattern describing the entire conjunction.

For example, if in addition to the above 3-path $\texttt{UnaryExpr}_{15} \rightarrow \texttt{AddExpr}_{36} \rightarrow \texttt{"-"}_{11}$ we want to consider the 1-path $\texttt{"*"}_{23}$, their conjunction would result in the following pattern:

"MultExpr * (AddExpr - MultExpr)"

At this point, the astute reader might have noticed that the optimizing compiler used in our example seems to have a bug. Specifically, it would seem that it not only applies the distributive rule appropriately to multiplications of the form $a \times (b \pm c)$ but also erroneously to such patterns as $a/(b \pm c)$ and $a \,\%\, (b \pm c)$.

Note how this exemplifies the high level of impact the textual form of the grammar tends to have on grammar-based approaches. In our case, the way `MultExpr` is defined does not allow us to differentiate between multiplications and divisions by means of a single k-path. Had the same rule been formulated as follows, the single 3-path would have sufficed to produce the same pattern.

```
MultExpr := UnaryExpr8
          | MultExpr13 "*"23 UnaryExpr15
          | MultExprq "/"24 UnaryExprx
          | MultExpru "%"25 UnaryExpry;
```

Moreover, our decision tree would have been able to explicitly detect that the pattern

"MultExpr / (AddExpr - MultExpr)"

leads to the execution of `distributive_rule()`, which should have immediately raised an alarm in us as testers. As such, we are left aware of the importance of the grammar structure, and recognize that investigating its impact more closely presents an exciting future work item.

5.2 Targeted Input Generation

It is now finally time to address our actual goal of generating novel inputs that are supposed to reach our target method. Because the constrains in the decision trees that we obtained previously characterize the call conditions of a method, we would like to leverage them to generate inputs exhibiting these characteristics.

A naïve but straightforward approach would be to simply generate inputs at random and immediately feed them into the predictors to discard those that do not match our expectations of covering the target method. However, such an approach is hardly feasible in practice because the performance is necessarily going to be abysmal for any non-trivial criterion prescribed by the predictor.

Therefore, the CODEINE implementation supports what we can refer to as *feature-oriented input generation*, where it extracts constraints from a decision tree and generates samples that fulfill those constraints. Technically, this is implemented as an extension of the algorithm presented as part of the ALHAZEN tool [65].

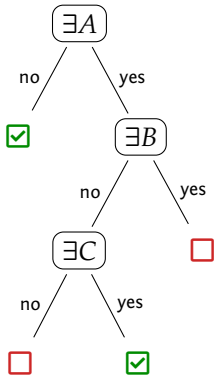


Figure 5.5 / An abstract decision tree containing three node constraints over the features A, B, and C, as well as two "covering" paths leading to a ☑ verdict. Thus, we can say that this decision tree produces two inputs.

5.2.1 Obtaining Constraints from a Decision Tree

To be precise, the constraints we seek to fulfill are conjunctions of the constraints in the nodes along a path leading to a `covered` classification in a decision tree. To collect a constraint, we follow the path from the root of the tree to a leaf that belongs to the ☑ class and create a single large constraint by constructing a conjunction. For every node we visit along the path, we add its constraint to the conjunction if the path follows the "yes" branch, otherwise we negate the constraint before adding it. In practice, most decision trees have multiple ☑ leaves, so in these cases we repeat the conjunction construction for all covering paths to obtain a set of tree constraints. This way we can say that a tree usually describes a set of inputs.

To give a visualization, consider Figure 5.5 which shows an abstract decision tree that contains three node constraints $\exists A$, $\exists B$, and $\exists C$. There are two paths from the root to a ☑ leaf, which turn into the two follwing conjunctions:

1. $\nexists A$
2. $\exists A \wedge \nexists B \wedge \exists C$

Note how we negate $\exists$ to be a $\nexists$ whenever we are following the "no" branch.

As a more concrete example, consider the tree in Figure 5.3, which proposes that the constraint

$$\exists\, \texttt{UnaryExpr}_{15} \rightarrow \texttt{AddExpr}_{36} \wedge \nexists\, \texttt{UnaryExpr}_{15} \rightarrow \texttt{AddExpr}_{36} \rightarrow \texttt{MultExpr}_{2}$$

describes an input which covers `distributive_rule()`.

In order to limit the selection of constraints to only those in which we have a high degree of confidence, we do not consider paths leading to leaves that have a Gini impurity [18] greater than 0.3. Gini impurity is a value in $[0, 1]$, and it indicates the probability to misclassify a sample if it were classified randomly according to the sample distribution of the given leaf. This measure is calculated as part of the decision tree model and is readily available for us after the decision tree learner is finished training the tree.

The next processing stage we have to perform before we can engage in the actual generation of inputs is filtering out infeasible constraints. By design, our decision tree learner optimizes to correctly classify as many *observed* inputs as possible. If

some constraint is infeasible, there will be no input satisfying it for the learner to observe. Therefore, when building the decision tree, the learner is not deterred from including such constraints because doing so does not lead to any observable misclassification. Therefore, the learner may end up placing this constraint for either class (i.e., ☑ or ☐). We remove such constraints by means of a dedicated check for incompatible k-paths. For example, a constraint of the form $\exists\ \mathtt{a} \rightarrow \mathtt{b} \rightarrow \mathtt{c} \wedge \nexists\ \mathtt{a} \rightarrow \mathtt{b}$ is infeasible because the 2-path is contained in the 3-path, and therefore the 3-path cannot be instantiated without generating the 2-path. Note that similarly to our work in [65], we do not care about the completeness of this feasibility check. If we overlook an infeasible constraint, the algorithm will stop without a result later on. A rigorous feasibility check does, however, speed up the approach significantly as the algorithm would otherwise exhaust its timeout.

5.2.2 Generating from Constraints

Having obtained a set of supposedly feasible constraints from a decision tree, we can move on to generate a derivation tree fulfilling each constraint in turn. The generation algorithm itself comprises two intertwined searches: The *outer search* is a heuristic search in the space of all possible partial derivation trees, while the *inner search* completes a partial tree and provides a heuristic value for the outer search.[2]

Because the productions of the grammar restrict the form of syntactically valid trees, most notably, the number and derivation rules of nodes and their children, a derivation tree can also be represented by its nodes listed in pre-order. The outer search maintains a list of prefixes of such pre-order node sequences, where each prefix corresponds to a partial derivation tree. In each step, the outer search chooses a prefix and expands it, thereby enlarging the corresponding partial tree, until a sequence is discovered that represents a complete derivation tree.

The inner search uses a greedy approach to complete a partial tree. This method also assigns a value which expresses how well the complete tree matches the constraints. This value is used as a heuristic within the outer search: If the given completion of a partial tree fulfils many constraints, there is a good chance that its further completion fulfills all constraints. Therefore, this partial tree is selected for expansion.

Let us now take a closer look at how this search tandem works in detail. Staying with our current example, we assume the algorithm is solving the constraint $\exists\ \mathtt{UnaryExpr}_{15} \rightarrow \mathtt{AddExpr}_{36} \wedge \nexists\ \mathtt{UnaryExpr}_{15} \rightarrow \mathtt{AddExpr}_{36} \rightarrow \mathtt{MultExpr}_{2}$ that we have obtained from the decision tree excerpt given in Figure 5.3.

The outer search begins with an empty tree. Starting with the root $\mathtt{AddExpr}_{0}$, the generator needs to make a decision which of its two alternatives to expand: $\mathtt{MultExpr}_{2}$ or $\mathtt{AddExpr}_{5}$ $(\mathtt{"+"}_{10} \mid \mathtt{"-"}_{11})$ $\mathtt{MultExpr}_{7}$. It generates partial trees for both options and calls upon the inner search to complete them. Figure 5.6 shows the state up to now in its upper part, above the dashed line.

Let us observe how the inner search completes the first partial tree, the one which expands $\mathtt{AddExpr}_{0}$ to $\mathtt{MultExpr}_{2}$. Its first decision is whether to expand $\mathtt{MultExpr}_{2}$ into a $\mathtt{UnaryExpr}_{8}$ or a $\mathtt{MultExpr}_{13}$ $(\mathtt{"*"}_{23} \mid \mathtt{"/"}_{24} \mid \mathtt{"\%"}_{25})$

[2] This scheme was designed and implemented by Alexander Kampmann as part of ALHAZEN [65].

`UnaryExpr`$_{15}$. The first clause in the constraint requires a k-path that begins with `UnaryExpr`$_{15}$. The second option generates a `UnaryExpr`$_{15}$ directly, and therefore the greedy search uses it. At this point, the greedy search deviates from the order given by the grammar[3] and immediately derives the `UnaryExpr`$_{15}$, instead of proceeding left-to-right to a `MultExpr`$_{13}$. All nodes can be easily added to the tree in the correct order regardless, thanks to the slot system as introduced in Definition 2. In turn, `UnaryExpr` expands into one of the following alternatives:

1. `Identifier`$_{16}$
2. `"+"`$_{27}$ `UnaryExpr`$_{28}$
3. `"-"`$_{29}$ `UnaryExpr`$_{30}$
4. `"++"`$_{31}$ `UnaryExpr`$_{32}$
5. `"--"`$_{33}$ `UnaryExpr`$_{34}$
6. `"("`$_{35}$ `AddExpr`$_{36}$ `")"`$_{37}$
7. `DecDigits`$_{22}$

Given that we are generating a subtree for a `UnaryExpr`, and the first clause of the constraint requires an `AddExpr`$_{36}$, the algorithm takes option 6 at this point.

Now, `AddExpr` needs to be derived, and the algorithm faces again the decision between `MultExpr`$_2$ and `AddExpr`$_5$ (`"+"`$_{10}$ | `"-"`$_{11}$) `MultExpr`$_7$. The first clause in our disjunction provides no further guidance as the requested 2-kpath is already in the tree, regardless of which decision is taken here. The second clause, however, *prohibits* a k-path. We are in a subtree below `AddExpr`$_{36}$, and there is a `UnaryExpr`$_{15}$ immediately above it, therefore the first two elements of the 3-path are present. The second clause of the constraint would be violated if we chose `MultExpr`$_2$ at this point. Therefore, the algorithm chooses the other alternative. Note how the prohibited k-path can only be considered at its last element: Simply prohibiting `MultExpr`$_2$ would also render the first clause infeasible.

The partial derivation tree below the dashed line in Figure 5.6 shows the result of the inner search so far. The tree is still not complete as there are several unfilled slots, which we have skipped along the way. However, we can simply invoke a close-off procedure as we did in Section 3.1.2 to obtain a complete tree with all elements required by our constraint.

The inner search discovered a solution for the constraint, and the outer search can terminate after just one iteration. Had this tree not turned out to be a solution, the outer search would have proceeded with the other candidate it generated earlier. In practice, we observed that k-path constraints, in contrast to some of the more involved constraints implemented in ALHAZEN, lead to termination after one round of the outer search in many cases.

With this targeted generation algorithm in place, we now have a working solution to address our use case. Given our target method, we deploy the described procedure to generate inputs that are likely to trigger its execution. What remains to be seen is how well this works in practice.

5.2.3 Evaluating Targeted Generation

In our experimental evaluation, we aim to investigate several characteristics of the presented targeted generation approach at once:

[3] If the greedy search proceeded in the order given by the grammar, cases like the one in the example would trigger an endless loop by always going for the nearest derivation of `MultExpr`.

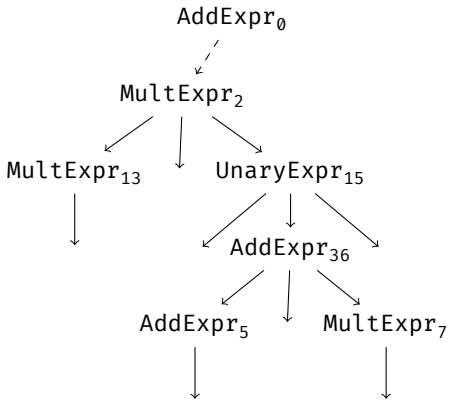


Figure 5.6 / Outer and inner search working together to generate a derivation tree that fulfills a constraint gathered from a decision tree predictor. The dotted edge signifies the hand-off from the outer search to the inner search.

Generation Success How successfully can the learned decision trees generate inputs that reach a given method?

Execution Extent Is there a difference in the number of methods reached with inputs created by the targeted generation and those created by a grammar-random approach?

Execution Frequency Are targeted methods reached *more often* with tree-guided inputs than with grammar-random inputs?

The experimental pipeline introduced in Section 5.1.1 comes in very handy in answering the above questions. Specifically, we extend it to branch out into three parts that are responsible for driving experiments to answer a question each.

Generation Success

A central question is whether we can successfully use the decision trees learned by CODEINE for generating inputs to cover a given method. To answer this question, our experimental pipeline first proceeds to invoke the generation capabilities of CODEINE for each subject. Considering that the decision trees that we have learned in Section 5.1 usually have several ☑ leaves, we obtain a set of inputs for each decision tree. From these sets, we define two coverage metrics:

Relative coverage fraction is defined as the number of inputs that do cover the targeted method divided by the number of all inputs in the set.

Absolute coverage fraction is defined as the fraction of decision trees whose input set includes at least one input that covers its intended method. This expresses how many methods could be successfully covered at all.

Some caution is advised, as there might exist the possibility that covering the methods in question is generally easy, regardless of the nature of the inputs. To investigate this, we compare the results obtained from inputs generated with

Table 5.3 / Coverage fractions achieved by CODEINE versus grammar-random generation.

	Subject	Relative		Absolute	
		Random	CODEINE	Random	CODEINE
JSON	`argo` [8]	0.111	**0.800**	0.606	**1.000**
	`fastjson` [2]	0.286	**0.857**	0.713	**0.960**
	`genson` [19]	0.300	**0.700**	0.717	**0.956**
	`gson` [44]	0.000	**1.000**	0.486	**0.966**
	`json-flattener` [61]	0.000	**1.000**	0.463	**1.000**
	`json-java` [72]	0.000	**1.000**	0.250	**1.000**
	`json-simple` [62]	0.000	**1.000**	0.471	**1.000**
	`json-cliftonlabs` [76]	0.857	**1.000**	0.941	**1.000**
	`minimal-json` [117]	0.125	**1.000**	0.560	**0.980**
	`pojo` [75]	0.000	**0.500**	0.240	**0.680**
URL	`autolink` [118]	0.500	**0.886**	1.000	1.000
	`jurl` [111]	0.948	**1.000**	1.000	1.000
	`url-detector` [106]	0.833	**0.852**	0.949	**1.000**
JS	`rhino` [92]	0.000	**0.692**	0.392	**0.928**

CODEINE to inputs generated randomly from the grammar. For a fair comparison, the pipeline generates for each decision tree as many grammar-random files using GRAMMARINATOR[4] as are generated by CODEINE. This is a scheme similar to the one we successfully employed previously in Section 3.2.2. For example, if a decision tree has four paths leading to ☑ nodes, CODEINE will end up generating four inputs, and so the pipeline will generate four random inputs to match. Table 5.3 reports the results of this comparison. The relative coverage ratio is measured individually per method, but due to space being limited even in a dissertation, its median value is reported across all methods for each subject. The absolute coverage ratio, however, is calculated per subject, and can thus be reported directly.

Looking at the relative coverage ratios, we see that given the same number of attempts, CODEINE significantly outperforms random inputs, which oftentimes fail to reach the targeted method at all. This observation is consistent across all subjects except for `jurl` and `url-detector`, where random inputs are roughly on par with CODEINE. This indicates that the trees effectively reflect the input features that are relevant to a method being covered or not.

The absolute coverage results tell us that for all but the three subjects of the URL family, inputs generated by CODEINE are indeed more likely to reach a given method than inputs generated randomly. In the aforementioned three cases, both approaches do equally well. This is due to the small size of the subjects and thus most methods being easily reachable regardless of input specifics.

Now, for the remaining subjects the difference might seem negligible, and the fact that *all* methods are indeed successfully reachable by randomly generated files might be surprising. However, this is given by the construction of our experiment: CODEINE forms decision trees from randomly generated inputs and can therefore only learn what is achievable by random inputs in the first place.

[4] Actually our own GRAMMARINATOR-like implementation, so that we can reuse the exact same grammar files.

Table 5.4 / Method coverage for CODEINE (C) and grammar-random (R).

	Subject	Covered Methods				# Calls		Cohen's d
		Median		Mean		Mean		
		R	C	R	C	R	C	
JSON	argo	40	**46**	44.2	**52.2**	4.3	8.8	0.602
	fastjson	28	**35**	29.9	**39.9**	8.8	10.5	0.101
	genson	33	**54**	35.9	**55.4**	17.4	20.1	0.066
	gson	27	**76**	35.8	**73.0**	4.0	8.8	0.366
	json-flattener	17	**27**	17.9	**25.5**	464.6	464.1	0.000
	json-java	7	**21**	9.7	**20.7**	3.1	8.0	0.458
	json-simple	7	**17**	8.7	**16.1**	5.0	6.8	0.181
	json-cliftonlabs	12	12	12.5	**13.1**	73.3	35.2	0.447
	minimal-json	24	**41**	26.8	**41.2**	2.6	7.0	**0.792**
	pojo	17	**48**	19.2	**49.6**	1.0	5.2	0.425
URL	autolink	24	**26**	24.8	**25.6**	111.1	53.6	0.487
	jurl	39	39	36.0	36.9	122.5	101.4	0.119
	url-detector	**44**	37	**41.1**	33.3	881.9	123.4	**0.796**
JS	rhino	264	**343**	270.4	**334.8**	3.8	181.0	0.082

Nonetheless, our approach does identify the conditions under which a method is reached, which is what explains why we observe a difference at all. We are convinced that given specially crafted, exotic inputs that cover hard-to-reach methods, this difference would be far more pronounced.

While it is not the main focus of the approach at hand, the inputs generated by CODEINE triggered between one and two orders of magnitude more exceptions with unique stack hashes than inputs generated by the random grammar-based approach across the seven subjects which have thrown exceptions.

Execution Extent

The next question concerns itself with whether, in general, inputs generated from decision trees cover more methods as compared to random inputs because of their supposed deeper reach into the program. To find an answer, we base this experiment on the test set data obtained from the evaluation of predictive capabilities as described in Section 5.1.1, as well as on the cumulative results from the generative evaluation from the previous section. This effectively leaves us with two large sets of inputs, where for every input we know the number of methods it reaches. One set is composed of inputs generated randomly, and the other comprises inputs generated by CODEINE.

Table 5.4 gives in the first four columns the median and mean number of unique methods covered by each file for the two input sets, denoted by R and C for grammar-random and CODEINE, respectively.

Our results show for 11 out of the 14 subjects that the inputs generated by CODEINE reach more methods per input in general. This is consistent with our hypothesis that inputs generated by CODEINE for the purpose of covering a specific method usually cover more methods that are related to it. These tend to be methods deeper on the call stack, which must be passed to reach the targeted method, and also methods that are themselves called from the now reached target.

This behavior speaks of a higher diversity of the inputs generated by Codeine, which also tend to be much smaller than randomly generated inputs. For instance, the inputs generated by Codeine have average sizes of 8.65, 12.06, and 22.09 bytes for the three grammars, while files generated by the random grammar-based approach are considerably larger averaging at 60.71, 80.75, and 60.92 bytes, respectively. In the context of testing, diverse inputs are favorable for covering more behavior in a single run of the system under test. Also, smaller inputs tend to reduce runtime and debugging effort, which is the main theme in this chapter.

Note that this does not mean that Codeine reaches *more* methods than a random generator. In our experiment, almost all methods were eventually reached with both approaches. There were no methods that could be reached by either approach *exclusively*, rather the number of methods that were reached per input was different.

For a closer look, let us first consider the JSON parser `gson` as a subject that has the most prominent difference in the number of methods covered in favor of Codeine. At its core, it has a central loop with a `switch` statement, which uses a lookahead to determine the type of the next structure and then delegates its parsing to specialized methods. For example, it has dedicated methods such as `peekNumber()` for looking up what kind of number should be read next, as well as `nextInt()` and `nextDouble()` for consuming JSON numbers as integers or double precision decimals. Due to the structure of the grammar, a random generator is very likely to generate a single, top-level value such as "true" or "null". As such inputs lack the expected lookahead characters, no specialized methods are called when parsing them. Conversely, Codeine is always guided by some non-trivial condition, leading it to generate specific, but more involved inputs that tend to cover more of those specialized methods.

On the other hand, we have `json-cliftonlabs` as a subject with no significant difference in the coverage extent. It, too, contains a single loop with a big `switch` statement as the centerpiece of its deserialization routine. For serialization, it sports an `if-else` cascade of an equally impressive size. All processing happens in these two locations, confined to two methods from which no other notable methods are called. And almost regardless of the inputs, the parser always takes the same methods to reach these central places, which explains why there cannot be a significant difference in the number of methods reached, regardless of the approach to input generation.

Finally, there is the `url-detector` subject, which is the only case where random inputs cover more methods per input file. It works similarly to `gson` by delegating a lot of its work to individual methods from a central loop. Most notably contributing to the difference in coverage, there are numerous methods that are executed only in the presence of specific characters. One such example is the `readQueryString()` method that is entered when a "?" is encountered, which is something that is only generated by Codeine when it specifically tries to reach a method that is involved in the processing of the so-called *query* part of a URL and not on other occasions. The random generator creates inputs containing a "?" much more frequently because it derives aimlessly.

Execution Frequency

One could expect that inputs generated from decision trees, which describe the precise conditions under which a method is called, would, in general, exercise those methods more often than random inputs. And so, we want to know whether this is the case. To answer this, our pipeline leverages the same inputs as in the evaluation of the generation success, which gives us for every method a set of inputs generated by CODEINE and a set of the same size of inputs generated randomly. Our experimental pipeline instruments the subjects so that not only the method coverage, but also the method execution counts are reported. We compare how often each method was executed by inputs generated specifically for it by CODEINE to how often it was executed by a set of random inputs of the same size. We use the Wilcoxon signed-rank test [132] to determine if the differences are statistically significant, and Cohen's d [25] to calculate the effect size.

While the differences are indeed statistically significant in favor of CODEINE with a p-value < 0.001 for all but the three URL subjects, they are very limited in their size. The three rightmost columns of Table 5.4 show the average number of invocations of the methods targeted by their respective decision trees for the input sets generated randomly and by CODEINE, as R and C, respectively. The rightmost column shows the value of Cohen's d calculated on the pairwise differences. Most values are below 0.5, which can be interpreted as small to medium effect sizes [108].

This observation, however, is consistent with the way CODEINE is training its decision trees: It uses the input features to learn a binary classification whether a method is covered or not without taking the number of its executions into account. Therefore, it generates samples which simply cover the targeted method, as opposed to samples which cover the targeted method as often as possible.

5.2.4 Threats to Validity

The evaluation in this chapter is, again, empirical in nature and so, it too, suffers from threats to validity. Much like our experiments from Chapter 3, the ones presented here also face several concerns.

When it comes to *external validity*, we observed a limited number of test subjects and input languages, which might be far from being generalizable to software in general. As the subjects are all *libraries*, the control flow must originate from client code, which in turn means that we had to write test drivers ourselves. We ensured that the test drivers exercised as much of the advertised functionality to the best of our ability by accessing all documented public methods, but it may be that some code was missed leading to underreporting of code coverage.

As far as *internal validity* is concerned, we do not account for the fact that some of our features may syntactically depend on other features, thus leading to feature correlation. For example, consider k-paths that are included in other k-paths by construction, in which case it would not be advantageous to include both of them in the same decision tree. Further, to avoid technical errors on our part, we use proven implementations of well-researched statistical methods throughout the experimental pipeline.

Grammar ambiguity possibly deserves its own place in this list, as it is a major source of trouble for the ALHAZEN tool when it comes to handling features of inputs. In ambiguous grammars, there exist different derivation trees for the same input, and so we must use an expensive Earley parser [30] to be able to find all such interpretations. From here, we consider that an input has a feature if it is contained in *any* of the possible derivation trees. This by itself already presents an over-approximation and thus a threat to generalizability, as different subjects may interpret the same input differently, and so the presence of a feature may or may not be factual.

5.2.5 Limitations

This approach is subject to several limitations, most of which can be addressed by further research and engineering. Let us now briefly consider some of the yet unsolved challenges.

Scaling. While the main focus is on improving the performance of the input generation, the critical step of acquiring sufficient training data for mining associations between input elements and individual input coverage is expensive. However, we assume that the purpose of individual methods rarely changes. Hence, in practice, it may suffice to extract models initially once and then re-create them only periodically.

Expressiveness. Our models express coverage conditions over basic features of input elements. As actual coverage conditions are undecidable in general, our models can only approximate coverage conditions by construction. This approximation could be improved by **1.** Expanding the feature set (e.g., make it project-specific); **2.** Including further combinations of features, such as disjunctions or comparisons; and **3.** Adding internal features relevant to program execution (e.g., features of program state).

Observability. We can only learn from what we can observe. Therefore, the extent of the methods to which our approach can be applied hinges on the amount of the training data we can obtain. In practice, we can usually expect to profit from existing regression test suites or readily available collections of test inputs found on the Internet. At the very least we can attempt to fall back onto grammar-based generation.

Undecidability. In addition to the observability problem, for some methods, it may be undecidable whether they can ever be reached and how the conditions could ever be characterized other than through the program itself. In practice, we would use our models to predict and generate inputs for those methods where the models have sufficient expressive power.

5.3 Summarizing Feature Mapping

Let us briefly recap what we have observed in this chapter. We begin by asking the question of how we can steer the generation of system-level inputs towards specific locations in the code with the goal of improving fuzzing performance by enabling inexpensive targeted generation.

We first take a detour by learning associations between features of inputs and code locations, and manifesting them as decision tree predictors. We evaluate

these predictors and deem them adequate for explaining which features relate to which methods.

We then go on to devise a guided input generation process, based on the structural properties of the decision trees, that works by means of two intertwined heuristic searches. In its evaluation we find that it copes with its task rather well by consistently reaching the targeted methods better than a random approach does.

With this in place, after having obtained decision tree predictors once, we can generate novel inputs that are aimed at specific methods in the code without the need to execute the system under test (SUT) for confirmation. This, in turn, requires far less resources that have to be committed to whole-system execution, thus freeing up more budget for improving overall testing performance in the scope of the project lifecycle.

Chapter 6

Related Work

In this chapter, we take the opportunity to explore some relevant fuzzing approaches related to grammar-based and machine-learning-based techniques. Individual sections provide a brief overview followed by some commentary comparing the approach at hand to the ones presented in this dissertation. We are mostly going to consider recent publications to keep this exploration of related work short.

6.1 Grammar-Based Fuzzing

This section gives an overview of some representative examples of grammar-based fuzzing of the recent decades. It proceeds chronologically along notable steps in the development of grammar-based techniques, while providing comparisons to the *k*-path algorithm and its implementation TRIBBLE.

6.1.1 A Sentence Generator for Testing Parsers (1972)

Purdom [105] presents an algorithm for producing short sentences while using all productions in a context-free grammar. The main motivation for this work stems from the necessity of properly testing automatically generated parsers as well as the programs that generate them. What is particularly interesting about this work is that it was published 18 years prior to what is generally regarded as the first publication on fuzzing by Barton Miller et al. [87]. Nevertheless, this work has undeniable relevance to grammar-based fuzz testing.

It must be noted that the presented algorithm expects its input grammars to be in a simplified Backus-Naur form (BNF) [12], where instead of alternations, there are top-level productions with the same non-terminal on the left-hand side, and where every production in the grammar is assigned a unique numerical identifier.

As a first step, the approach pre-computes static information about the given grammar, which comprises the shortest possible derivation production for every symbol, and a mapping indicating which alternative allows including a given symbol in the shortest derivation.

Given this information, Purdom's algorithm consecutively produces derivations of the grammar's start symbol, while choosing productions in an order that leads to expanding as yet unused productions in the least number of derivation steps, until all productions have been used. Once a production is marked as used, the algorithm derives only the shortest possible words whenever this particular production is encountered again.

The evaluation on two generated parsers [27, 28] and eight grammars demonstrates that the presented approach can be very effective in covering parser states and transitions for some grammars, while not being particularly effective on others. Further, the author reports that multiple bugs were found in his own parser generator implementation, which could not be found with other methods at that time.

In comparison to the work presented here, Purdom's algorithm actually always generates a set of inputs that achieves full 2-path coverage. To see why this is the case, we must rewrite our grammar to be in BNF, so we can apply Purdom's sentence generator in the first place. This rewriting requires us to effectively undo all transformations we introduced in Section 2.2, and for every production of the form `A := B | C` we obtain two productions `A := B` and `A := C`. Purdom's generator will aim to generate all new productions, which effectively corresponds to obtaining all paths of length two, i.e., all 2-paths in the grammar before rewriting. In comparison, the *k*-path algorithm presented here is much more generic due to the adjustable length of the production chains it can cover, and it works with a more pleasant grammar notation to boot.

6.1.2 *Geno* (2006)

Lämmel and Schulte [70] present a context-free grammar-based generation algorithm implemented as a tool called *Geno*, whose goal is to approach combinatorial coverage of a given grammar. To avoid the problem of combinatorial explosion, it features several coverage control mechanisms that allow its user to influence the derivation in terms of extent and number of explored combinations. *Geno*'s user is assumed to be a test engineer who is experienced in both the subject under test and context-free grammars. One motivation for the approach is the desire to supersede pre-existing, purely stochastic approaches in terms of control and achievable code coverage.

At the heart of the approach lies a bottom-up algorithm for generating inputs from a grammar guaranteeing full combinatorial exploration of all possible derivations. The authors concede that a bottom-up approach is less straightforward than a top-down approach, but maintain that it lends itself better to immediate reuse of produced derivations as part of the algorithm. Then they go on to introduce five control mechanisms for constraining the inputs generated by the above algorithm. Individually, they allow narrowing the scope of the targeted coverage conditions.

Depth control enables constraining the derivation depth for any non-terminal that is used as part of the final input, and not only for the entire derivation tree.
Recursion control allows limiting the number of recursive expansions of any given non-terminal reference.

Balance control expands on the two previous ideas by enabling to specify the range of depths of children that can be considered by the bottom-up algorithm when constructing new nodes.
Dependence control gives power over the exhaustive derivation of concatenated rules; e.g., *one-way* testing ensures productions be tested independently of each other, *two-way* testing prescribes the enumeration of all pairwise combinations, while *all-way* testing forces total combinatorial exploration.
Construction control is a very general means of control as it allows attaching conditions and computations to individual derivation rules, which makes the grammar equivalent to a grammar with *synthetic attributes* [68]. The authors show that the previous control mechanisms can all be implemented using construction control, however, they also point out that due to performance reasons, they opted not to do so.

To present the control mechanisms, the authors make use of two graph representations for context-free grammars, which find direct counterparts in our work. Their *constructor graph* corresponds to what we call the *grammar graph* in that it models all terminals, non-terminals, and their derivations. A difference between these two graphs, however, is that our grammars come with additional kinds of derivation rules, namely quantifications and regular expressions. The *sort graph*, on the other hand, corresponds to the *reachability map* from Section 3.1.4 as it serves the same function of mapping out which derivation rules are reachable from which other ones. However, their sort graph does not include terminals.

The prototypical implementation *Geno* is written in the C# language and has been evaluated, among others, on five small to medium-sized grammars. A case study was presented that centered around a de-serialization of an XML-based language, whose grammar contained 21 non-terminals and 34 alternatives. In order to focus the testing efforts on different aspects, seven grammar variants have been created, resulting in the generation of 200 000 test cases. The authors report that the test cases generated by *Geno* were able to uncover approximately 25 % of all filed bugs for the subject, but alas, it is not clear whether these bugs were novel findings filed by the authors themselves or already known previously.

The authors further point out that other tools, which were considered state-of-the-art at the time, such as Korat [15], AsmL [86], or Unit Meister [121], were not able to cope with the size of the problem.

Comparing *Geno* to our tool TRIBBLE, we can see that the latter requires much less domain knowledge from its users, and is therefore more immediately applicable to any given grammar and test subjects. To successfully deploy tribble in its capacity as a grammar-based fuzzer it is sufficient to decide on the value of the single parameter k without careful consideration of the productions of the grammar. On the other hand, *Geno* requires setting many control parameters that are specific to the productions in the grammar.

6.1.3 Nighthawk and GCOs (2007)

Beyene and Andrews [13] present an approach that applies metaheuristic optimization to grammar-based input generation. To point out a technicality, they opt to automatically turn grammars into Java classes to leverage *Nighthawk* [3], which is their already available genetic algorithm implementation specializing

in generating method call sequences as unit tests. By calling special initialization methods and providing them with appropriate parameter values, *Nighthawk* can synthesize Java objects that represent derivations of the grammar. One can obtain a string representation of these derivations by calling the `getString` method, which is available on all generated instances.

The authors then use this setup to compare several metaheuristic optimization approaches such as *hill-climbing* and *simulated annealing* to deterministic approaches like *depth-first search* and *rule coverage* as well as manually crafted conformance test suites. They compare the line coverage achieved by feeding the inputs to two small Java subjects processing HTML and XML inputs.

Perhaps XML is not the best choice of input language for this comparison, as it cannot be expressed by means of a context-free grammar, and the authors must apply postprocessing to achieve matching opening and closing XML tags in the inputs generated by the metaheuristic approaches.

As a result of their experiment, the authors come to the conclusion that both the deterministic and metaheuristic approaches are quite complementary when used for testing: while they already achieve significant coverage individually, they can achieve even more when combined.

This work is similar to our own work on integrating structured object instantiation into a genetic algorithm with the goal of improving the coverage of generated unit tests while decreasing their false positive rate [48]. However, our approach uses an XML schema instead of a generic context-free grammar and performs its structured string instantiations on the abstract syntax tree representation dynamically generated specifically for a given subject.

6.1.4 Csmith (2011)

Yang, Chen, Eide, and Regehr [133] present Csmith, which is a fuzzer created specifically with the goal of testing compilers of the C language. It generates random C programs and uses the principle of *differential testing* [83] by running multiple compilers and detecting disagreements in their behavior and results. To make this possible, Csmith generates programs that are well-formed and have a single interpretation when executed.

Specifically, Csmith generates programs that adhere to the C99 standard [57], which leaves plenty of unspecified aspects to be interpreted at the compiler's discretion. In fact, there are 191 kinds of undefined behavior and 52 kinds of unspecified behavior, all of which must not be part of the programs generated by Csmith. It avoids these behaviors by using special wrappers that, amongst other things, handle integer overflows in a well-defined manner, as well as provide safety for types, pointers, effects, arrays, and initializers.

Additionally, to have results that are comparable between programs compiled with different compilers, there is a test harness responsible for calculating a *checksum* of an execution. This checksum is based on the state of all non-pointer global variables in the program at the end of its execution. Whenever this checksum differs for the same program compiled with different compilers, there must be a bug in at least one of the compilers.

Another goal of Csmith is to generate programs that are expressive, containing as many features of the C language as possible, without falling victim to unspecified behavior. For this, Csmith uses a grammar for the well-defined subset of the C language, which is implemented as a collection of manually written C++ classes. This grammar models such language constructs as statements, expressions, functions, structs, arrays, pointers, various control flow constructs, etc.

Csmith always begins the generation of a new program with the creation of several `struct` types containing random members of boolean, integer, or previously generated struct types. It can then use these types in the generation of future components of the random program.

In the next stage, Csmith goes on to generate the rest of the program in a top-down fashion, beginning with a function that will be called by `main` at the end. While it is generating, Csmith holds a *global environment* containing top-level definitions as well as a *local environment* containing the current call stack and pointer analysis information about objects that may have been accessed in the current scope.

The generation proceeds by choosing from productions of the grammar that are admissible in the current context. This choice is governed by probability tables comprising 80 production probabilities, on which the authors claim to have spent substantial manual effort in order to make the resulting programs contain a balanced mix of the different language features.

Once a production is chosen, a special check called *filter* is performed that decides whether the selected choice is allowed in the current context, such as a `continue` statement only being legal inside a loop. Should this not be the case, Csmith simply retries with another choice. Otherwise, it proceeds with instantiating the production as prescribed by the grammar.

Whenever a target is required to complete the current production, such as e.g., a variable, the generator may choose to either select a fitting one from the environment, or to generate a new one. The generator recurses to process any outstanding non-terminals, while propagating an appropriate environment state, until the program is complete.

At the very end of the generation process, Csmith prints out the created components in the appropriate order: type declarations first, then global variables, and finally functions. Hereafter, it generates the `main` function, whose task it is to call the previously generated top-level function, to compute the checksum over all global variables, and, finally, to print out its value on the console.

It is very easy to generate a program that does not terminate and impossible to detect it. To address the issue of non-termination, the authors use a timeout of five minutes for compilations and five seconds for the execution of the compiled program in their practical evaluation.

Over a period of three years, Csmith found 325 bugs in popular C compilers including both commercial and open-source products. The majority of bugs were found in GCC and LLVM with 79 and 202 reports, respectively.

While one can most definitely classify Csmith as a grammar-based fuzzer, it does lack generality due to its singular, hand-coded, and highly specialized grammar

representation of the C99 language specification. This makes it rather difficult to compare against the approaches presented here. However, it is safe to say that Csmith bears most similarities with random grammar-based generators, but it comes extended with semantic constraints to guarantee successful compilation. As a possible future work item to consider, it might be of interest to investigate the grammar coverage achieved by inputs generated by Csmith, and possibly improve the approach by extending it with a *k*-path-aware generation heuristic.

6.1.5 LangFuzz (2012)

Holler, Herzig, and Zeller [56] present LangFuzz, which is a blackbox context-free grammar-based fuzzer. It works by mutating a given test suite of inputs by inserting input *fragments* mined previously, to create inputs that are *interesting*.

LangFuzz uses a grammar in the ANTLR format to model the **lang**uage it is supposed to **fuzz**. In its first step, which comprises the *learning phase*, LangFuzz parses given inputs into fragments which correspond to non-terminals. These fragments are added to a common fragment pool.

Then the second step begins, where LangFuzz iterates over a given test suite of inputs and mutates one input at a time. For this, the input is parsed using the original grammar into an abstract syntax tree, in which a small number (usually one to three) of non-terminal nodes are randomly selected for replacement. Each node slated for mutation is removed, and a new value for the expected non-terminal is inserted in its place either from the fragment pool or generated with a depth-limited breadth-first expansion. In the latter case, after the generation is finished, the remaining unexpanded nodes are filled with fragments or shortest possible derivations if no fitting fragments are available in the pool.

After inserting fragments into their new environment, LangFuzz performs a fix-up task, where identifiers inside the fragments are replaced by identifiers already found in the rest of the derivation tree. This is done to increase the chances of generating inputs that have the proper definition-reuse order for languages like JavaScript. Additionally, LangFuzz has the option to provide a list of globally known, or built-in identifiers, which will then not be rewritten in fragments. For example, the JavaScript global object `window`, which is available in the context of a browser, is one such built-in.

Once this second step is finished, LangFuzz has produced a new, mutated test suite, which is then fed into the system under test (SUT) to see if any crashes or hangs occur. After four months of operation, LangFuzz has detected 164 new defects in the two JavaScript engines employed in the browsers by Mozilla and Google. Additionally, the authors conducted a second, smaller experiment with the PHP interpreter and found 20 new bugs in two weeks.

LangFuzz is similar to tribble in that it, too, uses a generic context-free grammar to produce syntactically valid inputs. The biggest difference between the two approaches is that LangFuzz is capable of leveraging a body of input fragments, yet has no order to its generation, while tribble, on the other hand, produces only freshly synthesized inputs and strictly according to a *k*-path coverage plan. In principle, it should be a matter of engineering to enrich tribble so that it may

also leverage existing input fragments in its systematic generation, which might be worthwhile future work.

6.1.6 StGP (2014)

Kifetew, Tiella, and Tonella [66] combine genetic programming with grammar-based test generation. As a basis for their approach, they use so-called *stochastic grammars*, which are context-free grammars whose alternations are annotated with probabilities for choosing a derivation when generating inputs.

The authors mention two approaches for acquiring stochastic grammars from context-free grammars: On one hand, the probabilities can be assigned statically with 80 % chance allocated to non-recursive productions and 20 % to recursive ones. The authors claim that this approach has been shown to produce non-trivial derivations while not suffering from too much bloat when used for generating inputs in practice. On the other hand, one can learn a probability distribution from a given corpus of inputs using the *Inside-Outside* algorithm from [71]. This is also the approach they went with in their empirical evaluation.

The other part of the approach at hand is genetic programming [104], or more precisely, a genetic algorithm [78] because in this use-case the authors are not evolving a program but rather an input, albeit a highly structured one.

The algorithm begins by generating an initial population of derivation trees and randomly assigning them into suites at random. The initial generation is done in two flavors: purely randomly and using a stochastic variant of the same grammar trained on a given suite. The authors compare the performance of the two flavors later in their evaluation. These suites of inputs represent the individuals to be evolved by the genetic algorithm by applying genetic operators with the goal of achieving the best possible fitness in a given budget.

The set of genetic operators that can be applied to the suites comprises the *insertion* of a newly generated tree, the *deletion* of the least fit tree, as well as the *crossover* of a subset of trees among two suites. There are also two mutations that are applied to the individual trees with a probability of $\frac{1}{|\text{suite}|}$: A subtree is mutated by deleting a node selected at random and generating a fresh one in its place. A subtree crossover is performed on two derivation trees by exchanging subtrees rooted at the same non-terminal resulting in the creation of two new derivation trees.

The genetic algorithm determines the fitness of a test suite based on the sum of branch distances [85] to all branches of the test subject when executed. The algorithm aims to evolve inputs that minimize the number of missed branches.

The authors have implemented their approach in a prototype called StGP as an extension of EvoSuite [33]. They perform their empirical evaluation on three subjects taking structured inputs, ranging in their size from small (2 KLOC) to large (73 KLOC). Comparing the coverage achieved by inputs generated with comparable resource budgets, the authors found that, unsurprisingly, the genetic algorithm outperforms the random generator on the two bigger subjects. When starting with a learned stochastic distribution, even more coverage can be achieved even faster. The achieved coverage is equal across all approaches on

the small subject due to fast and easy saturation. The same results also hold for the fault exposing capability as evidenced by mutation score.

Three years later, the authors introduce type annotations that can be added manually to the productions of the grammar in [67]. They extend their experimental evaluation to compare their previous approach based on learning with the new type-annotated genetic generation approach on six subjects, but fail to find any statistically significant differences in the achieved code coverage.

While, again, not directly comparable to TRIBBLE as presented in the scope of this work, StGP shares some similarities with our work on probabilistic grammars in [114], where we use TRIBBLE to learn the common distribution of alternatives, but also go on to invert them to produce valid, but rarely seen inputs. While we do not use metaheuristic optimizations, our approach has also shown good results on its own. Still, the idea behind stochastic grammars may find a direct application in the *k*-path algorithm as one of its close-off stages in the future.

6.1.7 IFuzzer (2016)

Veggalam, Rawat, Haller, and Bos [126] present IFuzzer, which is a grammar-based fuzzer that uses genetic programming for testing language interpreters. It relies on ANTLR for generating the data model for derivation trees, which serve as individuals in its genetic algorithm.

To begin its work, IFuzzer requires a grammar and a suite of initial samples. All samples are first parsed into their constituents with a parser generated by ANTLR from the given grammar; and their fragments representing non-terminals are added to a pool, just like in the case of LANGFUZZ. Then, several full sample inputs are selected to build up the initial population for the genetic algorithm, which applies genetic operations to them in order to produce future generations and find bugs in the SUT.

IFuzzer implements mutation in two variants: A random node in the tree is either fully replaced with a fitting fragment from the pool, or a partial subtree is generated in its place by performing uniform random derivation up to a limited number of times. If any outstanding non-terminals remain, they are closed off with pool fragments.

Crossover is done in a straightforward way by discovering and swapping nodes that correspond to the same non-terminal in two trees.

The algorithm also makes use of the *elitism* mechanism [78] to preserve best performing individuals across generations. Unfortunately, the authors fail to specify how many individuals are preserved.

After performing mutation or crossover, the algorithm proceeds with a fix-up phase, where undeclared identifiers are renamed to match the ones that are already present in the host tree, just like LANGFUZZ used to do it.

Further, the authors devote quite some effort to limiting the bloat of the generated inputs. When calculating the fitness, parsimony pressure [115, 136, 102] is used to penalize inputs that are getting too large. When applying genetic operations, the size increase of the resulting inputs must not exceed a fairness factor, otherwise the result is discarded before even proceeding to the fitness

evaluation, and a new attempt is made with the same genetic operation. After sufficiently many such unsuccessful attempts, even the base input is permanently removed from further participation in the algorithm. Beside these two measures, IFuzzer applies an implementation of delta debugging [135] for JavaScript [40] to further reduce the size of its inputs. This implementation seems to be aware of the hierarchical version of delta debugging [89], however, the authors of IFuzzer choose only to mention the classical variant for some reason.

When it comes to the fitness function that is used to evaluate the inputs produced by the genetic algorithm, IFuzzer, interestingly, can still be called a blackbox approach because it does not seem to require any instrumentation of the SUT. The fitness of individual inputs considers two components: structural properties of the generated inputs themselves, namely the cyclomatic complexity [82], as well as feedback from the SUT, such as warnings, errors, crashes, or hangs. Unfortunately, the authors fail to specify how exactly these values are acquired.

All parameters for the genetic algorithm such as rates of crossover and mutation, or the number of generations to carry out were chosen by the authors empirically based on preliminary test runs. The authors evaluated IFuzzer on two versions of Mozilla's SpiderMonkey JavaScript interpreter [93].

To be able to compare their results to those of LangFuzz, the first experiment was targeted at the same version `1.8.5`. The initial input suite came from Mozilla's development test suite of 3000 inputs, which was also used in the original LangFuzz evaluation. IFuzzer found 40 bugs in one month of operation with 24 of them overlapping with those found by LangFuzz in three months.

In their second experiment, the authors targeted SpiderMonkey version `38` and found 17 bugs of which 4 were deemed exploitable by Mozilla. Unfortunately, the authors fail to mention the duration of this fuzzing campaign and merely mention that on average bugs are found after 90 to 95 generations.

Similarly to LangFuzz, IFuzzer is also closely related to tribble in its goal of generating inputs that optimize for variety. Targeting the generation of programs, IFuzzer mostly measures variety in terms of the cyclomatic complexity of its generated inputs, which is actually rather close to tribble's generation algorithm that is also concerned with covering all possible paths through the program, albeit limited to a given value of k. Inspired by IFuzzer, it might be interesting to investigate the performance of a genetic approach to achieving k-path coverage.

6.1.8 Skyfire (2017)

Wang, Chen, Wei, and Liu [130] introduce Skyfire, which they present as an approach to generate seed inputs for fuzzers. It works by learning a so-called probabilistic context-sensitive grammar (PCSG), given a context-free grammar and an initial set of inputs. It is also possible to treat Skyfire as a standalone black-box fuzzer by using the inputs it generates directly, but the authors' evaluation suggests that this usage significantly decreases the fuzzing effectiveness.

The PCSGs that are central to this approach can be characterized by the two augmentations they introduce on top of productions in context-free grammars. First, in a PCSG, every production rule is associated with a *context* that is supposed to model semantic constraints, in which the non-terminal given by the production

can be expanded. The context of a non-terminal is a tuple containing three levels of its ancestor nodes and its first sibling. While the authors do acknowledge that this information is insufficient to capture arbitrary semantic constraints that may occur in practice, they maintain that it is sufficient to capture their diversity and to keep production rule redundancy low.

Secondly, every production in a PCSG is associated with a *probability* representing how often it occurs in the set of inputs the grammar was learned from. The probabilities are computed such that their sum over all productions of the same non-terminal equals one, regardless of the application context. Note that this only effectively accounts for top-level alternatives, while, in its implementation, Skyfire uses ANTLR grammars, which allow alternatives to be nested at any level in the right-hand side of a production rule. The authors do not mention whether and how such cases are handled.

Skyfire operates in two phases: the learning phase and the generation phase. In the learning phase, a PCSG is extracted from a given set of inputs. Provided a context-free grammar in the ANTLR format, Skyfire parses the inputs into abstract syntax trees, and for every node stores its context and computes its probability by dividing the number of occurrences in its context by the number of occurrences across the entire input corpus. It then augments the production rules of the original grammar with this information and stores them in a pool to be used in the following phase.

The generation phase uses the PCSG learned in the previous phase to generate inputs to function as seeds for other, mutation-based fuzzers like AFL. Skyfire has three main goals for the inputs it generates: syntactic and semantic correctness, diversity, and uncommonness. The correctness is guaranteed by the grammar itself, while the responsibility of producing diverse and uncommon inputs falls to the generation algorithm.

The algorithm uses left-most random derivation that is bound by a timeout as a means to guarantee termination. In short, starting from the root non-terminal, the left-most non-terminal is expanded by a production chosen at random from the set of productions that are applicable in the current context. Skyfire employs four heuristics designed to help with termination and to constrain bloat:

1. Productions are partitioned into low and high probability sets by ordering them by probability and splitting at the largest gap. During generation, in 90 % of cases, productions are chosen from the low probability set to encourage uncommon structures.
2. No production may be applied more than three times.
3. Productions with fewer constituent non-terminals are preferred.
4. Productions are not expanded more than 200 times.

If an input cannot be generated in the given timeout, it is discarded, and the algorithm begins anew.

After a set of inputs is generated, Skyfire filters out those that do not increase the code coverage in the SUT by using `gcov` [37] for open-source subjects and PIN-based [77] instrumentation otherwise. Then, terminal symbols are randomly mutated according to the grammar rules, thus finalizing the generation phase.

The authors carried out an empirical evaluation on two XSLT processors, the `libxml2` [103] XML library (sharing the same grammar), and the closed-source JavaScript engine in the Internet Explorer 11 browser. The sample inputs were mined from the Internet using Heritrix [90], and the grammars to parse them come from ANTLR's community grammar repository [38]. Where possible, the experiments are relying on AddressSanitizer [110] to detect memory issues.

The authors compare the bug-finding capability as well as the achieved code coverage of three configurations: **a)** the initial inputs, **b)** AFL seeded with the initial inputs, and **c)** AFL seeded with inputs generated by Skyfire. The results show that configuration c) is the most effective by a significant margin in both coverage achieved and bugs found. With it finding 19 novel memory corruption issues across its test subjects, Skyfire has been shown as a useful seed generator for mutation-based fuzzing.

When comparing Skyfire to TRIBBLE, one can observe that both approaches share similarities. They both define a notion of context that is based on the derivation tree representation of inputs. Skyfire's choice of considering three levels of "inheritance" comes very close to our own recommended value k in our k-paths. However, the sibling node included in Skyfire's notion of context makes the task of systematically achieving full coverage largely infeasible because it results in exhaustive two-way combinatorial testing.

Both tools support probability annotations for production rules, although in the case of TRIBBLE, probabilistic generation was evaluated in its own setting in [114] rather than as part of the k-path algorithm as introduced in Chapter 3.

Both approaches employ bloat control mechanisms, although the hard-wired limits implemented in Skyfire seem less adaptable for use with arbitrary, yet untested grammars.

Finally, the inputs generated by both approaches are equally well suited for further processing by mutational fuzzers. It might be especially interesting to evaluate the performance of TRIBBLE's more lightweight approach to designating context in comparison with Skyfire as part of future work.

6.1.9 GRAMMARINATOR (2018)

Hodován, Kiss, and Gyimóthy [54] present GRAMMARINATOR, which is an open-source, grammar-based blackbox fuzzer implemented in Python, and whose inputs are grammars in the same format as used by ANTLR. Given a grammar file, GRAMMARINATOR generates an *un-lexer* and *un-parser*, which can be thought of as the counterparts to the traditional lexer and parser that ANTLR would normally generate. As such, these two components have methods to generate individual grammatical parts instead of parsing them. By calling the generation method corresponding to the root element of the grammar, an input can be generated from scratch. Similarly to how a parser would parse a token stream, the un-parser generates one.

Whenever there is an alternative derivation to take, a decision is made which token to generate. In order to avoid generating trees that are too deep, GRAMMARINATOR precomputes the minimum derivation depth that each alternative requires, and uses a user-defined *depth* parameter to restrict its choice to those

alternatives inside this limit. Further, to encourage covering all possible alternatives without doing so systematically, every time GRAMMARINATOR makes a decision, it decreases the probability to make the same choice again by a so called *cooldown* factor, which is a constant provided by the user.

In addition to generating random inputs from scratch, GRAMMARINATOR can parse given inputs into derivation trees and apply mutation and recombination to them, thus creating new inputs. To increase customization, a user also has the possibility to add one or more *transformers*, which get a derivation tree as input and may rewrite it arbitrarily. This mechanism can be used, for example, to insert whitespace into the output to make it syntactically valid in case it was not included in the original grammar, or to apply some context-sensitive fix-ups like setting specific values to parts of the input that depend on other parts.

In contrast to TRIBBLE, GRAMMARINATOR specifically eschews the *systematic* coverage of alternatives, relying instead on randomness to achieve variety in its inputs. This results in lower achieved code coverage under comparable generation conditions as we have seen in the empirical evaluation presented in Section 3.2. However, implementation-wise, there are several commonalities to the tools: TRIBBLE also precomputes static information about the grammar to enable efficient derivation as introduced in Section 3.1.4, and it also features a close-off mode delimited by a user-provided maximum depth shown in Algorithm 3, which was inspired by GRAMMARINATOR. And while TRIBBLE presents the technical basis for a cooldown-based alternative selection strategy like the one used by GRAMMARINATOR, it does not make use of it in its generation algorithm implementation. On the other hand, by default, TRIBBLE lacks both postprocessors and the functionality to parse and mutate existing inputs.

6.1.10 NAUTILUS (2019)

Aschermann, Frassetto, Holz, et al. [9] present NAUTILUS, which is a grammar-based, coverage-guided fuzzer. Being a white-box fuzzer, it requires the source code of the application under test so that it can be compiled with instrumentation to report the code coverage achieved during execution.

Starting from a grammar, NAUTILUS works by first generating an initial set of inputs. It features two generation modes: On the one hand, there is a naïve random generation mode, which uniformly picks from applicable derivation rules to create an input. To prevent the over-representation of trees that are easily derived, the resulting inputs are checked and discarded if they have been used recently already. On the other hand, there is the uniform generation mode, where NAUTILUS takes the length n of strings to produce, and follows the algorithm presented by McKenzie [84] producing inputs that are uniformly sampled from the set of all inputs of length n. Although this algorithm is not very efficient, being quadratic in time and space, it avoids any bias the grammar might impose on the reachability of certain inputs due to its syntactic structure.

Having generated an initial input set, NAUTILUS proceeds by executing the SUT and measuring the code coverage of each input. Upon finding an *interesting* input that triggers new code coverage, it is minimized. The input minimization is done in two steps, applied sequentially: First, individual subtrees are minimized by replacing non-terminals with their shortest possible derivations as long as

the input still gives the same coverage when fed into the SUT. Second, trees are minimized by recursively replacing subtrees with their descendant subtrees when they represent the same grammar element, thus shrinking the trees even further. This step is also repeated until the coverage achieved by the resulting input changes.

Finally, the minimized input undergoes several phases of mutation comprising different subsets of the following mutation operators:

Random Mutation replaces random nodes in the tree with freshly generated nodes that fit the same non-terminal.
Rules Mutation sequentially replaces each node of the input tree with a subtree corresponding to each admissible alternative derivation rule.
Random Recursive Mutation identifies recursive cycles in the tree and amplifies them 2^n times for a random value of n between 2 and 15.
Splicing Mutation selects random subtrees from other trees in the queue and swaps them with fitting nodes in the currently mutated tree.
AFL Mutations comprise a set of mutations as implemented in AFL [134] such as bit flips, arithmetic mutations, and insertion of interesting values.

The generation supports a post-processing phase for applying transformations to the mutated tree. Nautilus can afford to not impose any restrictions on their complexity because it is a purely generative approach with no need to parse existing inputs. These transformations can be used to ensure non-context-free conditions such as matching XML tags.

After any such transformations, the tree is *unparsed* into the string it represents and fed into the SUT. Nautilus uses a JSON format for its grammars, but also has a converter from ANTLR. However, to produce syntactically valid strings using ANTLR grammars, oftentimes whitespaces must be added manually.

In their evaluation, among other experiments, the authors performed a manual case study of security issues in the *mruby* interpreter [94]. After inspecting previous bugs, they came to the conclusion that a high lexical and syntactic variety is, in fact, not necessary to trigger the same classes of bugs. Therefore, they constrained their Ruby grammar to contain only a small set of integers, strings, and identifiers, as well as only one syntax for method calls. They were successful with this grammar variant in finding six CVEs.

This adaptation of the grammar is a good example of the old exploration vs. exploitation trade-off. The authors have chosen to restrict the exploration of all possible lexical and syntactic values to favor more exploitation of a well-known set of constructs to make better use of the fuzzing resource budget, i.e., time.

It is difficult to compare Nautilus to tribble because of how different the two approach coverage. For Nautilus, the only incentive comes in the form of novel code coverage, although the authors mainly report its performance in terms of the bugs found, whereas tribble is designed to guarantee coverage of the input language, or more precisely its grammar.

6.1.11 Superion (2019)

Wang, Chen, Wei, and Liu [131] present Superion, which is a grey-box mutation-based fuzzer. It is implemented as an extension of AFL, to which it contributes several grammar-aware components based on ANTLR.

One of the steps in the process of the original AFL is the *trimming* of inputs, where random chunks of inputs are systematically removed in a byte-wise fashion. In the context of fuzzing programs that process highly structured inputs, this technique simply breaks most inputs, thus rendering them unusable, because they are merely discarded by the parser of the program under test. Superion changes this strategy by parsing the input into its derivation tree according to a user-provided grammar and systematically removing subtrees as long as the code coverage induced by the input remains the same. This approach to minimization is much more effective at preserving the syntactic validity of inputs. However, as the authors themselves note, both the parsing and the execution of the subject in-the-loop result in a significant slowdown. What the authors fail to mention is whether any removed elements are considered obligatory by the grammar, thus producing invalid inputs after all.

In the step following the trimming, AFL employs *mutation* of inputs, for which it sports an impressive array of bit and byte twisters, which have the same input-breaking properties as the original input trimming above. Here, Superion adds its two mutation strategies.

On one hand, there is the *enhanced dictionary-based mutation*, which does not rely on a grammar: Based on the assumption that most tokens consist of alphanumeric characters, Superion identifies contiguous arrays of bytes corresponding to characters in the alphanumeric range. It then proceeds to either insert in-between or overwrite completely these arrays with values from a dictionary provided by the user, or mined by AFL from the program under test.

On the other hand, Superion adds the *tree-based mutation*, which fully leverages the grammar for parsing inputs. The algorithm takes two trees as arguments: a *target* tree to which mutations are to be applied, and one additional tree which serves as a source of subtrees. In order to constrain the size of the mutated trees and prevent excessive execution times, any inputs larger than 10 kB are skipped and left unprocessed. First, the mutation algorithm copies all subtrees smaller than 200 B from the two input trees into a common subtree pool. Then it applies up to 10 000 mutations by iterating the subtrees in the *target* tree and replacing them with subtrees from the pool taken at random. Unfortunately, the authors fail to specify the iteration order in which these replacements happen, which could have an effect in realistic fuzzing campaigns with limited time budgets. Also, the size limits have been determined by the authors empirically based on their experiments.

The authors performed a practical evaluation of their tool on one parser of an XML-based language (Apple Property List) and three JavaScript interpreters. They crawled thousands of input samples from the Internet, preprocessed them to remove comments, and used `afl-cmin` to minimize the number of samples that actually give new coverage, ending up with 534 and 2569 seed inputs for XML and JavaScript, respectively.

After a fuzzing campaign of three months, Superion found 31 new bugs, of which 21 were novel vulnerabilities and 16 received CVE identifiers. AFL only found one bug in one JavaScript interpreter and 5 in the Apple Property List parser. Further, Superion outperformed AFL on average by 16.7 % and 8.8 % in line and function coverage, respectively. Additionally, the authors compared Superion to `jsfunfuzz` [107] and the latter only found minor out-of-memory issues in the two JavaScript interpreters to which it could be applied.

Being a purely mutational fuzzer, Superion strongly relies on readily available valid inputs to operate on as well as on continuous feedback from the subject during the course of the fuzzing campaign. In this setting, TRIBBLE could easily serve as a generator of such an initial set of inputs, possibly leading to a similar effect as was observed in the case of Skyfire.

6.1.12 EvoGFuzz (2020)

Eberlein, Noller, Vogel, and Grunske [31] present evolutionary grammar-based fuzzing, which contributes a genetic algorithm as an extension to the probabilistic fuzzing approach based on learned distributions presented in [114]. Given a grammar and a set of input samples, the original approach learns a distribution of probabilities of the grammar's alternatives from the sample set once and is then able to produce inputs that have similar structure. The authors integrate this learning of probabilities as part of a feedback loop, implemented in a genetic algorithm, whose goal is to generate complex, failure-inducing inputs.

Specifically, the authors implement the algorithm's fitness function to maximize the ratio of derivation steps to the overall size of the inputs, thus encouraging the generation of inputs whose derivation trees have more nodes, while at the same time punishing large, but simply structured inputs. Further, if an input causes an exception to be thrown, its fitness score becomes infinite, and the input is kept until the end of the search process.

The authors employ classic mechanisms of elitism [29] and tournament selection [88] as part of their genetic algorithm. The fittest individuals from a given generation serve as the basis from which the original approach is invoked to learn a new probabilistic grammar for the next generation. The obligatory mutation step of the genetic algorithm is then applied to this newly learned grammar by randomly reassigning probabilities of alternatives to prevent genetic drift. However, the authors do not argue about the effectiveness of this preventative measure. The evolution terminates upon reaching a specified timeout.

The authors implement their approach in a tool called EvoGFuzz, which they evaluate on ten subjects across three grammars for JavaScript, JSON, and CSS3 that were also used in the evaluation of the original approach in [114]. They set the number of probability mutations to one per generation, the size of the population to 100 inputs, the overall timeout to 10 minutes, and they repeat the entire experiment 30 times.

Their experiments show that the inclusion of a search-based approach leads to significant increase in achieved line coverage and triggered exceptions in comparison with the original approach. However, the authors note that these better results come at the cost of increased run time.

A comparison of this approach with TRIBBLE is *interesting* to make because it actually uses TRIBBLE under the hood, albeit for the purposes of probabilistic generation. Nevertheless, it is conceivable to enhance the *k*-path-related functionality of TRIBBLE with a search-based approach. A genetic algorithm could be useful to achieve higher input variety across different *k*-paths, provided a fitness function that scores *k*-paths as well as a mutation function that is able to change inputs to contain random *k*-paths.

6.1.13 Bonsai Fuzzing (2021)

Vikram, Padhye, and Sen [127] present a grammar-based fuzzing approach called *bonsai fuzzing*, whose main goal is to generate terse inputs. It uses a bottom-up generation algorithm that is governed by three parameters m, n, d, which correspond to the number of identifiers, the expansion limit for Kleene-star repetitions, and the maximum recursive derivation depth for non-terminals, respectively. As such, this approach has tight requirements on the grammar in question, and its applicability on generic context-free grammars, which do not have dedicated *identifiers*, *repetitions*, or *recursion*, is questionable. Additionally, the set of identifiers has to be known statically, so that its values can be pre-instantiated before the fuzzing process begins.

The approach is based on a practical application of the authors' mutational fuzzer Zest [98] as part of a classic coverage-guided loop [14]. This results in mutating a pool of inputs in a time-bound loop and accumulating mutants which fulfill an *interestingness criterion*. By default, this criterion demands the inputs to produce new code coverage and to be semantically valid, i.e., not cause the SUT to return with an error.

Inspired by the iterative deepening approach to depth-first search [69], the main idea of bonsai fuzzing is to apply the above fuzzing loop inside of yet another loop, while incrementing the current values of the m, n, and d parameters one by one, thus mutating smaller, valid inputs so that they grow in size until they reach a limit given by the user-provided parameter bounds.

To additionally enable the generation of invalid inputs, the authors introduce a variant of their algorithm where they tweak the interestingness criterion to only require novel coverage and disregard the semantic validity.

The authors evaluate their technique in terms of conciseness, mutation score, and code coverage against a baseline two-stage approach consisting of first fuzzing and then reducing the size of the generated inputs. This baseline leverages state-of-the-art implementations [53, 52] of character-level and hierarchical delta debugging [135, 89].

They carry out their experimental evaluation on two subjects: ChocoPy – a compiler for a statically typed subset of the Python language for educational purposes [99] as well as Google's Closure Compiler – an optimizing JavaScript transpiler [39]. Based on the results of their evaluation, the authors conclude that their technique is effective in producing concise inputs while not significantly sacrificing the capability to detect faults and cover code compared to the more traditional baseline approach.

Compared to TRIBBLE, bonsai fuzzing seems to be applicable to a very specific setting, in which the input grammar belongs to a programming language, where the number of identifiers seems to play a particularly large role when it comes to input terseness. Since the authors did not evaluate the performance of the *k*-path algorithm in practice, this might be a great topic for future work.

6.2 Input Feature Associations

This section lists some machine-learning approaches that are related to techniques we use in Chapter 5 to establish and use associations between input features and code. Beside summaries of the individual related works, comments elaborating on the comparison with CODEINE are provided.

6.2.1 *XSS Analyzer* (2013)

Tripp, Weisman, and Guy [122] present *XSS Analyzer*, which is a commercial approach for finding XSS vulnerabilities in WEB applications. It leverages a grammar to learn features of attack inputs, which allows it to efficiently perform an attack search on its attack sample collection of half a billion inputs. They specifically highlight that storing the sample collection as a grammar results in a significant size requirement reduction as opposed to storing it as plain text because it allows denoting entire families by means of just several productions.

In the setting presented by the authors, a web application is protected by a so-called sanitizer routine, whose purpose is to filter out malicious requests by applying some attack detection criteria. The authors formalize that a primitive sanitizer transforms an input into an ostensibly harmless variant if it satisfies a predicate identifying it as malicious. Further, the authors define that generally, a sanitizer is either primitive, a composition of two sanitizers, or a nested sanitizer applied in a loop.

Based on this representation, a sanitizer can be vulnerable to an input in two ways: either is simply does not recognize a malicious construct, or it mistakenly recognizes a benign input as an attack and turns it malicious by applying its transformation. The presented approach accounts for both kinds of vulnerabilities, which are referred to as *structural* and *bypass* vulnerabilities.

The learning-based search algorithm works by systematically supplying inputs to the sanitizer under test and learning which features lead to rejection, so this knowledge can be used to avoid submitting inputs sharing these same features. This procedure is designed to save a lot of resources by avoiding an overwhelming number of sanitizer invocations that would certainly result in rejections. Since the search space, which is the sample collection, is stored as a grammar, the knowledge about rejected features can be easily manifested by removing all productions that contain these features. The features the authors chose for their learning approach comprise individual symbols of their grammar, making it corresponds to our 1-path criterion.

Bypass vulnerabilities are handled by applying a number of predefined rewrite strategies to symbols that are rejected by the sanitizer. For example, if a sanitizer transforms inputs by replacing occurrences of lowercase “`script`” with the

empty string, then the bypass rewrite strategy "`<SCRscriptIPT>`" would force the sanitizer to mistake this benign token for a malicious one and rewrite it to "`<SCRIPT>`", which is actually malicious.

XSS Analyzer is implemented in C# and uses ANTLR [100] to describe its 500 million attack input samples by means of approx. 1000 productions. It is now part of the AppScan tool by IBM.

The authors performed an empirical evaluation on 15 552 sanitizers comprising sanitizers taken from real-world applications as well as ones that were written by their in-house security experts. They evaluated the performance of *XSS Analyzer* and found that their approach of pruning the sample grammar with every newly rejected feature vastly outperforms three other contending approaches in terms of the ratio of vulnerabilities found to requests executed: **a)** the previous version of AppScan, **b)** 19 different configurations of random search applied to the sample collection, and **c)** the brute force trial of the entire sample collection.

While this approach does not have too much in common with CODEINE, it would be interesting to see if it could profit from extending its relatively simple 1-path-like features to *k*-paths, as the latter are capable of describing much more precise input constructs.

Further, the idea of pruning the grammar to obtain a specialized sub-grammar, which reflects only the relevant features, resonates really well with our use case coverage prediction use case. A sub-grammar can be parsed more efficiently, and therefore, both the feature extraction and the classification can profit from the speed up.

6.2.2 Predicting Branch Coverage (2018)

Grano, Titov, Panichella, and Gall [43] present an empirical study of three machine learning models on how well they can predict the code coverage that an automated test generation approach can achieve given a class under test. Specifically, they consider the EvoSuite [33] and Randoop [97] test generators.

The authors first seek to answer whether standard code complexity measures are suited to serve as features for the task of coverage prediction in the first place. As their candidate features they consider a selection of established code complexity measures taken from the object-oriented metrics introduced by Chidamber and Kemerer [22] such as the number of branches in a class, as well as from the tool JDepend [24], whose metrics were originally developed to assess the quality of a Java package such as the total number of abstract and concrete classes. In addition, the authors consider reserved Java keywords such as `instanceof`.

In their first experiment, they evaluated three machine learning approaches comprising Huber regression [45], support vector regression [20], and the multi-layer perceptron [95] on four different open-source subjects from different domains and ranging in size from 848 to 220 573 lines of code.

As a comparison metric they used the mean absolute error over a three-fold cross validation. Their results show that the two non-linear predictors outperform the Huber regressor, but also that is it indeed possible to predict test coverage based on classic code complexity measures applied to the class under test.

In a second, separate experiment the authors set out to find out to what extent can coverage be predicted. To this end, they apply the support vector regression model trained in the previous experiment to three additional open-source Java library subjects. On these subjects the predictor performs worse, hinting at possible overfitting to the training set. Additionally, the mean absolute error is smaller for the random Randoop generator than for the genetic EvoSuite tool.

Overall, the authors conclude that with better feature selection and a more appropriate predictor the task of predicting coverage seems achievable. With this in mind, we can see how such additional complexity-based features could be applied to Codeine as well, which could involve measuring both the complexity of the generated inputs and that of the method under test.

6.2.3 `ML-Driven` (2018)

Appelt, Nguyen, Panichella, and Briand [6] present a blackbox technique for generating SQL injection attacks for bypassing a web application firewall using machine learning and evolutionary algorithms. In their use case, a web application firewall (WAF) is deployed as a facade service in front of a SQL database, so that it can monitor incoming queries and filter out those it deems malicious to protect the database. Specifically, the authors implemented their approach titled `ML-Driven` as an extension of their previous work in [5]. To function, `ML-Driven` relies on the presence of a context-free grammar that models possible attacks in a sufficiently general manner.

First, a training set of inputs is generated using a uniform grammar-based random generation approach. This input set is used to train a machine learning classifier, whose goal is to predict whether the input will bypass the WAF or not. The features that are considered in this approach consist of all possible leaf-bearing subtrees of the derivation tree of an input. The authors evaluate the performance of using a `RandomTree` and a `RandomForest` [17] as the classifier of choice and come to the conclusion that there is no significant difference apart from the `RandomForest` being computationally more expensive.

A `RandomTree` only differs from the classic decision tree as we use it in Codeine in that it considers a random subset of features from which its decision nodes are formed. This has a clear performance advantage because not all features have to be extracted and considered, which might be a worthwhile experiment to conduct on Codeine as well. Especially so because our features, which are k-paths, sometimes correlate, e.g., when longer paths include shorter ones. This way, considering all k-paths at all times might be superfluous.

On the other hand, a `RandomForest` comprises an ensemble of `RandomTree` classifiers and applies majority voting to their individual outcomes to obtain the overall classification. While this scheme could be applied to Codeine, it is unlikely to be advantageous because in addition to worse performance, it would also be much harder to interpret.

Having obtained the initial classifier, `ML-Driven` engages a genetic algorithm, which ranks inputs by their probability of bypassing the WAF as calculated according to the classifier, and then mutates the best candidates to produce so-called offsprings. These offsprings are then evaluated against the WAF, and

observations whether they bypass it are added into the training set, which is used to re-train the classifier. This procedure is carried out in a loop until a timeout is reached.

When mutating inputs to create offsprings, `ML-Driven` replaces randomly selected subtrees with compatible subtrees that fulfill the same path constraints from the corresponding decision tree. This is similar to the heuristic search employed by Codeine, except that in Codeine this process happens only when creating fresh inputs to fulfill a given constraint.

The number of inputs mutation is applied to, and the number of offsprings created for each input are controlled by a parameter governing the trade-off between exploitation and exploration. In their previous work, the authors developed two variants of `ML-Driven`: one geared towards exploitation and one for exploration. They have found that one approach is better in the beginning of the evolutionary search, while the other performs best towards the end. Therefore, in this work, they combine the two approaches by dynamically adjusting this parameter to divide the test budget among mutating inputs proportionally to their probability of bypassing the WAF over the course of the evolution process.

The result of this process is a set of inputs that are bypassing the WAF protection and are valid according to the initial grammar. Additionally, the path constraints that can be extracted from the trained classifier cannot be directly translated into human-readable patterns because they represent opaque subtrees, whereas in Codeine grammar patterns are a direct result of using k-paths as features as we have seen in Section 5.1.2. However, the authors do present a human-in-the-loop approach for turning the path constraints into regular expressions to repair the WAF so that it correctly blocks more attacks.

Finally, the authors evaluate `ML-Driven` on two WAFs using their previously established testing tool named *Xavier* [7]. Their experiments show the improved approach, which dynamically adjusts the mutation, outperforming the other two configurations as well as two state-of-the-art tools for SQL attack generation.

It shall be noted, that the SQL attack grammar that `ML-Driven` is evaluated with lacks loops and therefore describes a regular language. Together with the fact that the WAFs evaluated use regular expressions as their means of discerning malicious queries, this raises some suspicions as to the applicability of `ML-Driven` to not only context-free grammars, but truly context-free languages.

Another concern is performance. While using a `RandomTree` classifier allows ignoring large parts of input features, the mutation procedure as described in the paper, seems to consider all possible features after all. This might lead to problems with deeply nested and cyclic grammars and inputs.

The biggest difference when comparing `ML-Driven` to Codeine, however, lies in the fact that Codeine does not employ an optimization loop, which is due to the fact that it is difficult to optimize many classifiers (i.e., one per target method) at once, but brings with it the advantage of speed, as there is no need to either re-train the classifiers or execute the subject along the way.

6.2.4 Predictive Mutation Testing (2018)

Zhang, Zhang, Harman, et al. [137] present predictive mutation testing, which is a technique for predicting whether a mutant will be killed by a given test suite without executing it. This research is motivated by the widely recognized problem that mutation testing is very computationally expensive, so much, in fact, that it hinders practical adoption beyond academia.

The presented approach works by training a machine learning model based on code features of the tests as well as the mutants themselves. The authors subscribe to the PIE theory [129, 64] stating that a mutant can be killed only in the presence of **1.** the **e**xecution of the mutated statement, **2.** immediate **i**nfection of the program state, and **3.** its **p**ropagation into the test output. Therefore, the authors manually design easily obtainable features that aim at predicting each of these conditions.

As features related to the *execution*, the authors count how often every statement of the subject is executed and by how many tests. To characterize *infection*, they keep track of the bytecode of the original statement and the type of mutation operator that is applied to it. Finally, for estimating *propagation*, the authors use features of code complexity instead, reasoning that more complex code makes an infection less likely to propagate.

When it comes to the choice of machine learning approach, the authors choose the `RandomForest` [17] predictor, which we have already seen in Section 6.2.3.

The authors evaluate their approach on nine real-world Java subjects and across multiple versions, amounting to 163 different configurations. They test their predictor in a cross-version scenario, where it would be trained on previous versions and applied on the recent one, and in a cross-project scenario, where the predictor would be trained on one set of subjects and applied on another.

The authors find that their technique is both effective and efficient, reaching speed-ups of up to two orders of magnitude compared to traditional mutation testing while providing a reasonable trade-off in prediction accuracy. Further, the authors note that features related to the *execution* criterion and those describing the code of the tests contribute more to the predictions than other features.

Comparing predictive mutation testing to Codeine, we see that both approaches have in common the aim to predict whether some part of the subject code will be *executed* by given tests or inputs, respectively. However, only two of the fifteen features the authors define relate to the tests themselves, while the majority of features pertain to properties of the program. This allocation of features is seemingly successful and thus presents an interesting opportunity to extend the set of features considered by Codeine: Perhaps considering properties of the program under test will help increase the accuracy, although it is not immediately clear how such features can be used in our targeted generation use case.

Summarizing Related Work

As we can see in this chapter, our work does not stand alone in its field, but rather it is surrounded and inspired by the work of others, and hopefully will itself encourage further exciting research.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Over the course of this dissertation, we have seen several contributions being made to the field of grammar-based fuzzing.

We have started modestly with the introduction of a very practice-oriented notation for context-free grammars. This notation allows writing elegant shorthands for repetitions of elements or embedding regular expressions, while not imposing too many strict structural requirements that might be inconvenient to uphold for a human writer. For example, the notation allows alternations to be placed anywhere inside derivation rules, instead of having to scatter the alternatives throughout multiple productions, which also makes such grammars easier to read.

We have then turned this textual representation into a graph structure to be able to better model the language it represents. This grammar graph has opened the doors for the concept of k-paths, which provides a handy way to describe certain contexts that can occur in inputs that belong to the language described by the grammar.

This concept, in turn, enabled us to define a notion of grammar coverage known as k-path coverage. Based on the grammar graph representation, this coverage metric allows us to make judgements as to how varied both individual inputs and entire input sets are.

Inspired by the promise of usefulness of the k-path coverage, we designed an algorithm to constructively attain high coverage values. We successfully tested the k-path algorithm in practice on a set of open-source subjects, where it has shown very good results in its capacity as a fuzzer by holding its ground against its closest competitor in the blackbox grammar-based fuzzing category. Beside finding new bugs in our subjects, our fuzzer implementation called TRIBBLE achieved impressive code coverage in our subjects.

Captivated by this success, we studied the effects that different levels of k-path coverage have on inputs and, more specifically, the programs they are fed into.

And so we have encountered empirical evidence of a connection between achieving *k*-path coverage and code coverage in our subjects. Specifically, this relationship appears to be a monotonic dependency, meaning that increasing the grammar coverage leads to an increase in code coverage. Given that *k*-paths tend to express distinct parts of the language, the existence of such a connection to constructs in the code is not surprising.

Evolving this observation, we have leveraged the *k*-paths to learn associations between inputs and code. Since we have originally developed the *k*-paths to represent features of inputs, we soon arrived at using *k*-paths as features in the machine learning interpretation of the word, which enabled us to use techniques from the field of machine learning to derive such associations.

Having manifested these associations as predictors comes with the immediate possibility to very efficiently predict the coverage of any given input. This is possible even without executing the program because all it takes is simply parsing the input to determine the features it carries. This is immediately applicable for the test selection use case, where we have to select from a large set of expensive tests in a limited budget setting.

The choice of these predictors to be decision trees further allows us to interpret their structure to learn exactly which features of the input are relevant to a method of interest. Reading *k*-paths as grammar patterns makes the task of understanding the purpose of a method much easier. In addition, when software evolves, we can use these features to easily spot any changes in the calling conditions of a method of interest as an early warning system possibly indicating an unintended change in behavior.

Finally, the decision trees enabled us to devise yet another algorithm for input generation, which can quickly create complete, syntactically valid system inputs that are aimed at executing a given method. This is useful in cases where test execution is expensive, and the target method is under-tested, which, let's face it, happens all the time and all over the place.

To conclude, encouraged by the promising results of our experiments, we envision the concept of *k*-paths contributing to further research which will eventually see wide adoption by practitioners.

7.2 Future Work

As stated at the very beginning of the introduction, fuzzing is not becoming obsolete, nor does it stagnate. Having contributed to this field somewhat, there are still several directions one can take going forward.

7.2.1 Addressing Limitations

The first area of improvement straightforwardly results from the fact that the approaches presented here are subject to limitations. Some of these are fundamental, e.g., when training predictors, for instance, for our approach to work well, we absolutely must rely on the training data because we simply cannot

learn what we have not seen. However, some other limitations can be addressed by further research and engineering.

When it comes to keeping the trained models up-to-date with changes in the code, we might need to (partially) re-train the predictors to reflect any differences such as when a method changes behavior. Updating our models must be approached with care to avoid performance issues because re-training from scratch is an expensive operation that does not scale well. A clever policy to determine how long to keep old data and when to deprecate it can have a significant impact on both the execution time of the subjects and the re-training of the classifiers themselves. Depending on the choice of the classifiers, an update from partial new training data might be possible instead of a full re-training from scratch.

The issue of scaling can also be approached from another side. Features that are represented by *k*-paths can be extracted from large inputs much faster by deploying specialized parsers that can only recognize the part of the grammar that is relevant for the features in question. The more *k*-paths are deemed to not be relevant features for our predictors, the more production rules can these specialized parsers ignore, and thus they gain a performance boost by having to consider fewer derivations.

As mentioned previously in Section 5.2.5, there are issues with expressiveness. Our choice of using *k*-paths as features was informed by the correlation between input and code coverage that we have observed in Chapter 4, and it seems to work well for us in practice. However, to enable the approach to learn more and more precise associations, more types of features should be considered. Such additional features can pertain to the inputs in different ways, such as expressing constraints on the length of certain sub-strings, or interpreting elements numerically and constraining their values. Such constraints have already shown some promise when implemented in the Alhazen tool with the aim to find precise conditions for program crashes, so it is not too far fetched to expect them to improve the accuracy of predictions of code locations as well.

Grammar ambiguity presents two challenges at once. Beside the performance problem of parsing all possible derivation trees for an input, it forces us to also consider all of them at the same time. However, this certainly results in an over-approximation because any realistic subject will itself only ever consider just one possible interpretation – the one determined by the implementation of its input parser. And so, if our fuzzing use case warrants the commitment of enough resources, we can consider re-using exactly the very same parser that the subject itself uses to obtain a derivation tree that is guaranteed to be correct. This should be especially feasible in cases, where the subject uses an automated parser generator such as ANTLR.

7.2.2 Further Applications

The presented approaches can also be applied in more scenarios as is, which would allow further investigation of the properties of *k*-paths.

The central concept that enables the very construction of *k*-paths is the grammar graph, which, in turn, is produced from the textual representation of the given grammar. Hence, the way the grammar is formulated has direct consequences

for the structure of the grammar graph, which also influences the *k*-paths that can be constructed. However, there are oftentimes multiple ways to express the same language construct by means of different notation. For example, in Section 2.2 we have seen different ways to express the repetition of an element: It can be defined by using a quantification, a regular expression, or an alternation together with recursion. In the latter case, there are even more options depending on which alternatives are manifested as productions and which are left inline.

All these choices impact the nodes of the grammar graph and thus also the *k*-paths that are available. Our very own *k*-path algorithm is highly and strictly dependent on which *k*-paths are there to be covered. Therefore, the differences in the *k*-paths between otherwise equivalent grammars might lead to the generation of different inputs that might have differing effects on the program under test. Regardless of the *k*-paths, similar effects might also occur for other grammar-based fuzzers, whose performance might be impacted by the grammar form. It might be worthwhile to obtain and systematize the knowledge about the influence of different grammar constructs on fuzzer performance.

Beside language-preserving grammar changes, one might consider grammar mutations that may change the input format it describes. Any effects such transformations might have on the inputs generated by the *k*-path algorithm are not yet researched.

So far, we have considered and evaluated the *k*-path algorithm as a full-fledged fuzzer on its own. However, we might want to consider it as a first stage of generating seed inputs for other fuzzers instead. One could easily imagine applying mutational and search-based techniques to the features-rich input sets that are produced by the *k*-path algorithm. A similar setup was successfully employed by Wang, Chen, Wei, and Liu [130] for their Skyfire generator.

We have seen in Section 5.1.2 how we can interpret *k*-paths as human-readable grammar patterns, but we did not have a chance to investigate whether our perceived usefulness matches reality. This presents an opportunity for a user study. We could find out whether real developers would appreciate this way of presenting information about relevant parts of inputs, and how it can help with debugging and testing tasks if at all.

As we have seen from our own experiments, there are situations where multiple programs share the same language of inputs. This offers opportunities to study transfer of knowledge. For example, it might be feasible to take a highly relevant pattern learned from one subject and measure whether it is as relevant to another subject, thus possibly indicating different or even erroneous handling of certain types of inputs.

Taking this one step further, this might enable clone detection on a semantic level. For example, we might be able to detect that in one subject the method `foo` depends on some feature, while in another subject it is methods `bar` and `baz` that together share this association. This might indicate that these methods are clones of each other, where the functionality of `foo` is spread across two methods in the other subject.

7.2.3 Extending Approaches

Finally, all approaches presented here can be extended, improved, or modified in many ways.

The *k*-paths, which are the main focus of this work, consider only the derivation context of grammar nodes, but not their surroundings. Pursuing the idea of designing features of inputs that are as expressive as possible, one could envision a sibling-based coverage, something like an *s*-path. When generating inputs that are supposed to achieve *s*-path coverage care must be taken to avoid the explosion problem, as this seems to get dangerously close to combinatorial exploration [73]. Building on this idea, there might be a way to elegantly combine *k*-path and *s*-path information in a generic notion of input context.

In many realistic cases, context-free grammars are insufficient to describe the full constraints of the language of expected inputs. And so, we might need to refer to more powerful formalisms such as attribute grammars that allow us to express almost limitless syntactic and semantic. However, together with their flexibility, attribute grammars bring challenges such as how to reason about attributes in terms of grammar coverage and how to achieve such coverage efficiently, without running into SAT-solving performance bottlenecks.

In our work, we have limited ourselves to using derivation trees as predictors due to their simplicity and easily accessible and interpretable structure. Yet there might be classifiers that are more efficient, less demanding of training data, or possibly better suited for selecting the most relevant features. Perhaps there are approaches that would enable automatic feature extraction based on raw derivation trees.

Other classifiers might support scalable multi-class prediction enabling us to model all code locations at once, thus helping with scaling the approach. However, in such cases it is unclear how to deal with targeted generation or updating parts of the model.

Currently, we only consider the initial training data when training predictors. While their quality is adequate for the subjects and methods we have seen in our experimental evaluation, we might want to consider including a feedback-loop, which automatically learns, tests, refines, and improves the predictors. While this design has shown promise in the ALHAZEN tool, we might run into performance problems if we apply the same setup to our scenario unchanged and try to optimize all predictors at once.

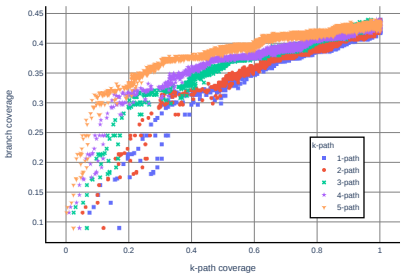
In Conclusion

Already over the course of this dissertation, *k*-paths have proven a useful instrument in several software engineering tasks, such as assessing the potential diversity of inputs enabled by a given grammar, generating diverse but small test input sets, as well as both predicting and achieving targeted code coverage. Looking into the future, only time will tell the extent to which *k*-paths will ultimately find their place in the practice of software engineering.

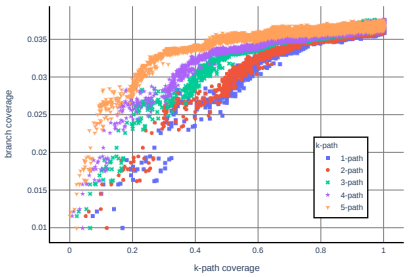
Appendix A

Supplementary Data

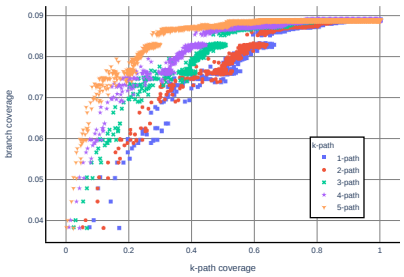
This appendix features the coverage data for all subjects used in the experimental evaluation from Chapter 4. Figure A.1 displays scatter plots of their branch and *k*-path coverage values. Please note that the y-scale is not normalized across the individual sub-figures.



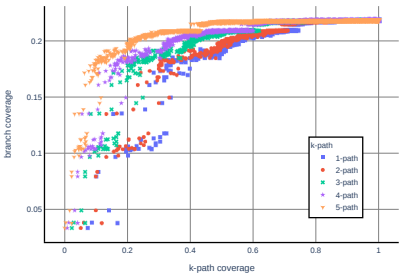
a / `argo`.



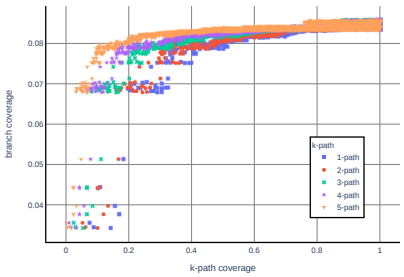
b / `fastjson`.



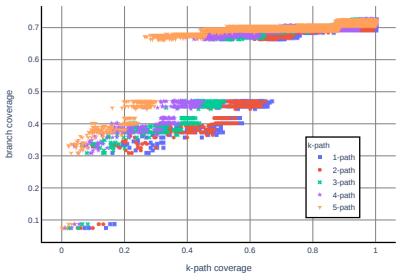
c / `genson`.



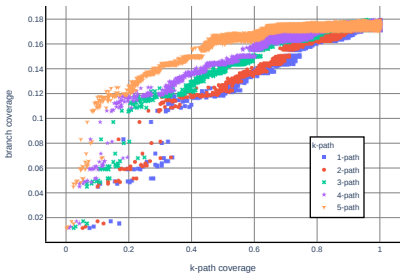
d / `gson`.



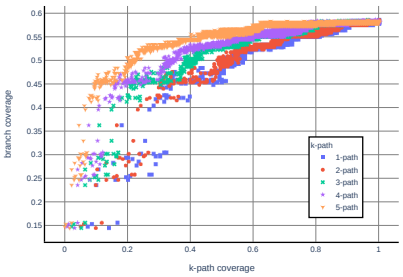
e / `jackson-databind`.



f / `json-flattener`.

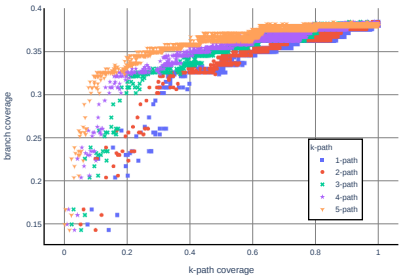


g / `json-java`.

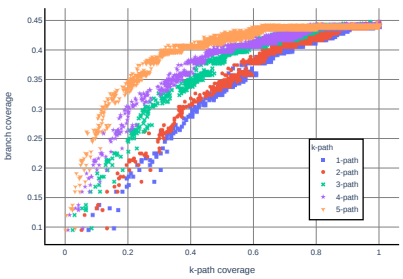


h / `json-simple`.

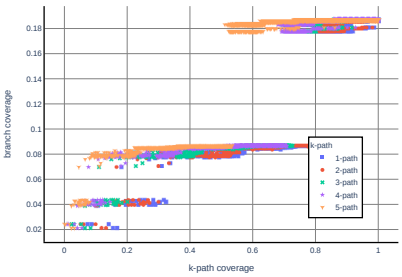
Figure A.1 / Branch and *k*-path coverage measured for subjects from Table 4.2.



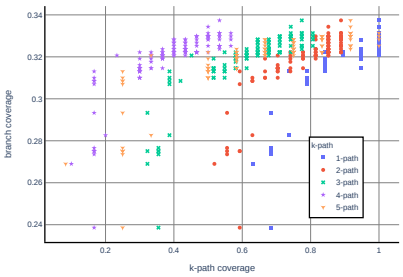
i / `json-cliftonlabs`.



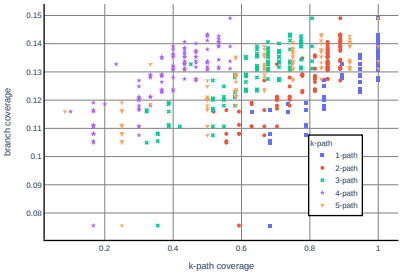
j / `minimal-json`.



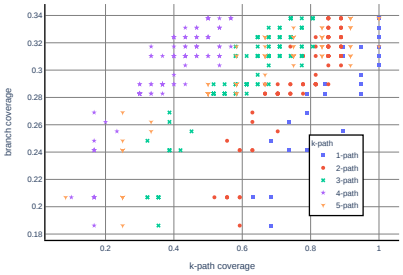
k / `pojo`.



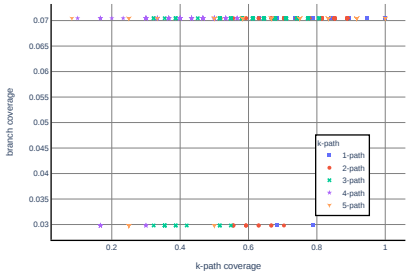
l / `commons-csv`.



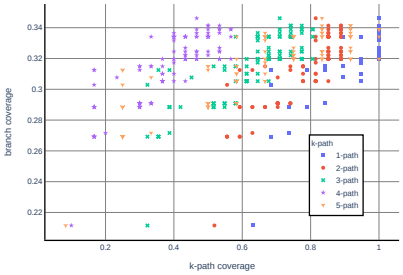
m / `jackson-csv`.



n / `jcsv`.

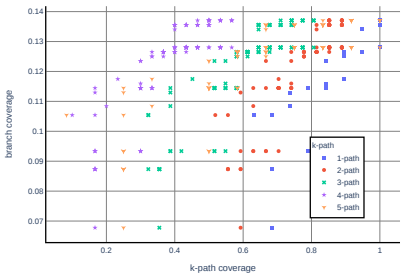


o / `sfm-csv`.

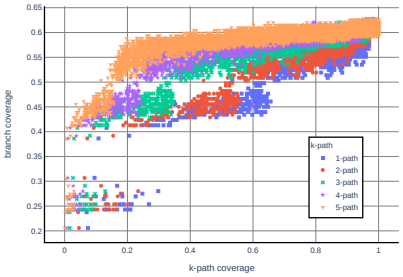


p / `simplecsv`.

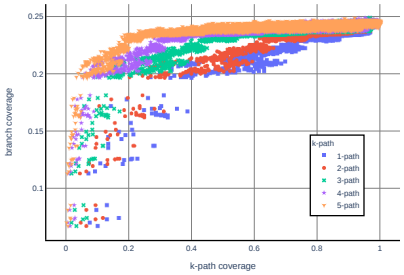
Branch and *k*-path coverage measured for subjects from Table 4.2 (continued).



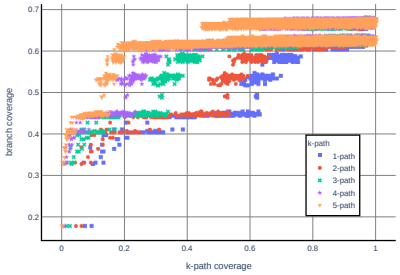
q / `super-csv`.



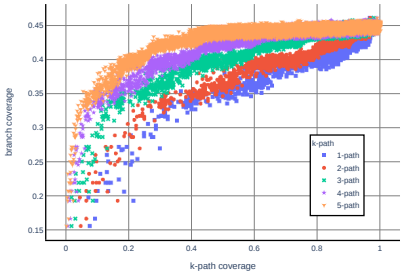
r / `autolink`.



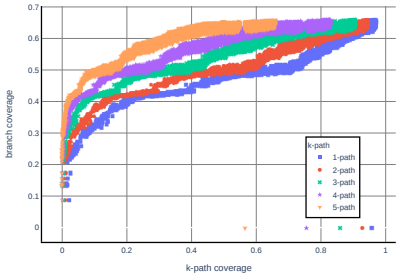
s / `galimatias`.



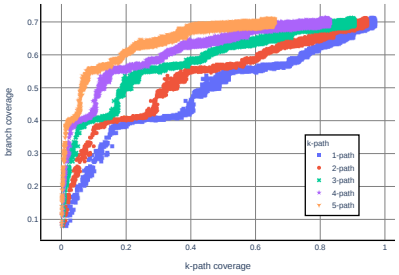
t / `jurl`.



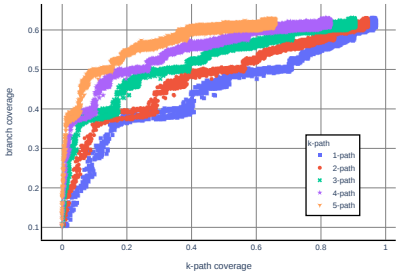
u / `url-detector`.



v / `commonmark`.



w / `markdown4j`.



x / `txtmark`.

Branch and *k*-path coverage measured for subjects from Table 4.2 (continued).

Appendix B

Extra Printables

This appendix contains the JavaScript grammar excerpt and its graph to be printed out separately or torn off, so that they can be placed alongside the main text for easy reference.

```
Expr := AddExpr0;
AddExpr := MultExpr2
        | AddExpr5 ("+"10 | "-"11) MultExpr7;
MultExpr := UnaryExpr8
        | MultExpr13 ("*"23 | "/"24 | "%"25) UnaryExpr15;
UnaryExpr := Identifier16
        | "+"27 UnaryExpr28
        | "-"29 UnaryExpr30
        | "++"31 UnaryExpr32
        | "--"33 UnaryExpr34
        | "("35 AddExpr36 ")"37
        | DecDigits22;
DecDigits := DecDigit42+;
DecDigit := "0"44 | "1"45 | "2"46 | "3"47 | "4"48
        | "5"49 | "6"50 | "7"51 | "8"52 | "9"53;
Identifier := "x"39 | "y"40 | "z"41;
```

Figure 1.1 / Grammar for a subset of arithmetic expressions in the JavaScript programming language (excerpt). Enriched with numeric node identifiers from the corresponding grammar graph.

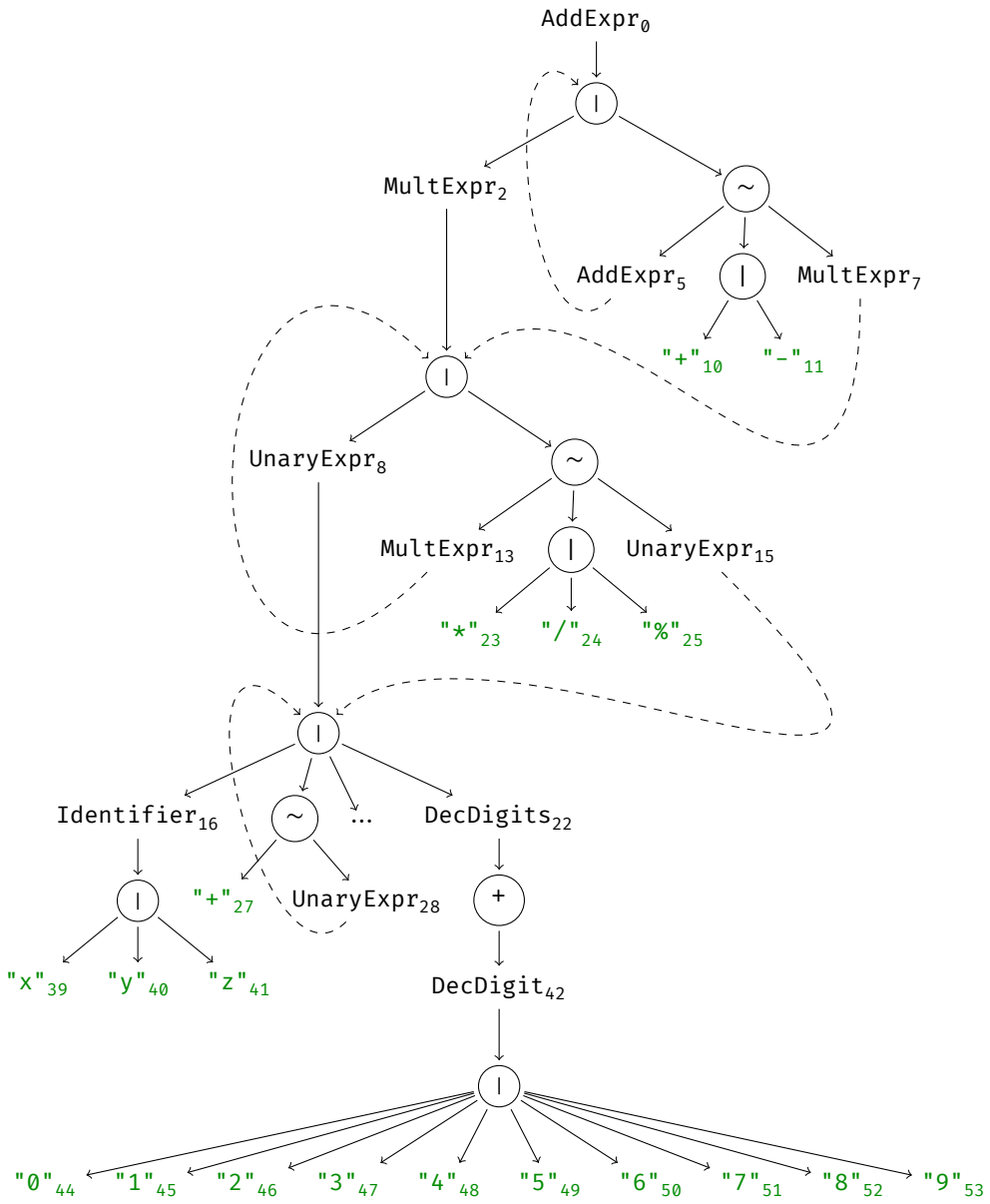


Figure 2.2 / An excerpt from the graph representation of the (partial) grammar from Figure 1.1. The root node is `AddExpr`$_0$ because it is the right-hand side of the production of the start non-terminal `Expr`. The "backward" dashed lines indicate derivations of references that prevent the graph from being a tree or even a DAG. Numeric identifiers are only shown for symbolic nodes.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986. ISBN: 0-201-10088-6. URL: https://www.worldcat.org/oclc/12285707 (cit. on p. 16).

[2] Alibaba. *fastjson*. https://github.com/alibaba/fastjson. Version 1.2.51. 2018 (cit. on pp. 36, 56, 65).

[3] James H Andrews, Felix CH Li, and Tim Menzies. "Nighthawk: A two-Level Genetic-Random Unit Test Data Generator." In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. 2007, pp. 144–153 (cit. on p. 73).

[4] *Apache Commons CSV*. https://commons.apache.org/proper/commons-csv/. Version 1.6. 2018 (cit. on p. 36).

[5] Dennis Appelt, Cu D Nguyen, and Lionel Briand. "Behind an Application Firewall, Are We Safe From SQL Injection Attacks?" In: *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*. IEEE. 2015, pp. 1–10 (cit. on p. 89).

[6] Dennis Appelt, Cu D Nguyen, Annibale Panichella, and Lionel C Briand. "A Machine-Learning-Driven Evolutionary Approach for Testing Web Application Firewalls." In: *IEEE Transactions on Reliability* 67.3 (2018), pp. 733–757 (cit. on p. 89).

[7] Dennis Appelt, Cu Duy Nguyen, Lionel C Briand, and Nadia Alshahwan. "AutOmated Testing for SQL Injection Vulnerabilities: An Input Mutation Approach." In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 2014, pp. 259–269 (cit. on p. 90).

[8] *Argo*. https://sourceforge.net/projects/argo/. Version 5.4. 2018 (cit. on pp. 36, 56, 65).

[9] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. "NAUTILUS: Fishing for Deep Bugs with Grammars." In: *NDSS*. 2019. URL: https://www.ndss-symposium.org/ndss-paper/nautilus-fishing-for-deep-bugs-with-grammars/ (cit. on p. 82).

[10] *Atlassian Commonmark*. https://github.com/atlassian/commonmark-java. Version 0.11.0. 2017 (cit. on p. 36).

[11] The Luigi Authors. *Luigi*. https://github.com/spotify/luigi. 2020 (cit. on p. 56).

[12] John W Backus. "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference." In:

Proceedings of the International Comference on Information Processing, 1959 (1959) (cit. on p. 71).

[13] Michael Beyene and James H Andrews. "Generating String Test Data for Code Coverage." In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. IEEE. 2012, pp. 270–279 (cit. on p. 73).

[14] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. "Coverage-Based Greybox Fuzzing as Markov Chain." In: *IEEE Transactions on Software Engineering* 45.5 (2017), pp. 489–506 (cit. on p. 86).

[15] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. "Korat: Automated Testing Based on Java Predicates." In: *ACM SIGSOFT Software Engineering Notes* 27.4 (2002), pp. 123–133 (cit. on p. 73).

[16] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. STD 90. RFC Editor, Dec. 2017 (cit. on pp. 33, 34).

[17] Leo Breiman. "RAndom Forests." In: *Machine learning* 45.1 (2001), pp. 5–32 (cit. on pp. 89, 91).

[18] Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984. ISBN: 0-534-98053-8 (cit. on p. 61).

[19] Eugen Cepoi. *Genson*. https://github.com/owlike/genson. Version 1.4. 2017 (cit. on pp. 36, 56, 65).

[20] Chih-Chung Chang and Chih-Jen Lin. "LIBSVM: A Library for Support Vector Machines." In: *ACM transactions on intelligent systems and technology (TIST)* 2.3 (2011), pp. 1–27. DOI: 10.1145/1961189.1961199. URL: https://doi.org/10.1145/1961189.1961199 (cit. on p. 88).

[21] Tsong Yueh Chen, S. C. Cheung, and Siu-Ming Yiu. "Metamorphic Testing: A New Approach for Generating Next Test Cases." In: *CoRR* (2020). arXiv: 2002.12543. URL: https://arxiv.org/abs/2002.12543 (cit. on p. 51).

[22] Shyam R. Chidamber and Chris F. Kemerer. "A Metrics Suite for Object Oriented Design." In: *IEEE Trans. Software Eng.* 20.6 (1994), pp. 476–493. DOI: 10.1109/32.295895. URL: https://doi.org/10.1109/32.295895 (cit. on p. 88).

[23] Noam Chomsky. "Three Models for the Description of Language." In: *IRE Trans. Inf. Theory* 2.3 (1956), pp. 113–124. DOI: 10.1109/TIT.1956.1056813. URL: https://doi.org/10.1109/TIT.1956.1056813 (cit. on pp. 9, 12, 13).

[24] Mike Clark. *JDepend*. https://github.com/clarkware/jdepend. 2015 (cit. on p. 88).

[25] Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Academic press, 2013 (cit. on p. 68).

[26] *CSV Grammar*. https://github.com/antlr/grammars-v4/tree/master/csv. 2018 (cit. on pp. 33, 34).

[27] Franklin Lewis DeRemer. "Practical Translators for LR(k) Languages." PhD thesis. Massachusetts Institute of Technology, 1969 (cit. on p. 72).

[28] Franklin Lewis DeRemer. "Simple LR(k) Grammars." In: *Communications of the ACM* 14.7 (1971), pp. 453–460 (cit. on p. 72).

[29] Haiming Du, Zaichao Wang, WEI Zhan, and Jinyi Guo. "Elitism and Distance Strategy for Selection of Evolutionary Algorithms." In: *IEEE Access* 6 (2018), pp. 44531–44541 (cit. on p. 85).

[30] Jay Earley. "An Efficient Context-Free Parsing Algorithm." In: *Communications of the ACM* 13.2 (1970), pp. 94–102 (cit. on p. 69).

[31] Martin Eberlein, Yannic Noller, Thomas Vogel, and Lars Grunske. "Evolutionary Grammar-Based Fuzzing." In: *Search-Based Software Engineering - 12th International Symposium, SSBSE 2020, Bari, Italy, October 7-8, 2020, Proceedings*. Vol. 12420. Lecture Notes in Computer Science. Springer, 2020, pp. 105–120. DOI: 10.1007/978-3-030-59762-7_8. URL: https://doi.org/10.1007/978-3-030-59762-7%5C_8 (cit. on p. 85).

[32] *ECMAScript 2015 Standard ECMA-262 6th Edition*. https://www.ecma-international.org/ecma-262/6.0/. 2015 (cit. on p. 56).

[33] Gordon Fraser and Andrea Arcuri. "EvoSuite: Automatic Test Suite Generation for Object-Oriented Software." In: *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. ACM, 2011, pp. 416–419. DOI: 10.1145/2025113.2025179. URL: https://doi.org/10.1145/2025113.2025179 (cit. on pp. 6, 77, 88).

[34] Steve Freeman, Tim Mackinnon, Nat Pryce, and Joe Walnes. "Mock Roles, not Objects." In: *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. 2004, pp. 236–246 (cit. on p. 6).

[35] *Fuzzinator Configs*. https://github.com/renatahodovan/fuzzinator-configs. 2019 (cit. on p. 35).

[36] *galimatias*. https://github.com/smola/galimatias. Version 0.2.1. 2018 (cit. on p. 36).

[37] Gcov. *Using the GNU Compiler Collection (GCC)*. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html. 2017 (cit. on p. 80).

[38] GitHub. *Community Grammars Written for ANTLR v4*. https://github.com/antlr/grammars-v4/. Accessed 2018. 2012 (cit. on pp. 33, 81).

[39] GitHub. *Google Closure Compiler*. https://github.com/google/closure-compiler. 2019 (cit. on p. 86).

[40] GitHub. *JavaScript Delta Tool*. https://github.com/wala/jsdelta. 2014 (cit. on p. 79).

[41] Rahul Gopinath, Alexander Kampmann, **Nikolas Havrikov**, Ezekiel O. Soremekun, and Andreas Zeller. "Abstracting Failure-Inducing Inputs." In: *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*. ACM, 2020, pp. 237–248. DOI: 10.1145/3395363.3397349 (cit. on p. 7).

[42] Rahul Gopinath, Hamed Nemati, and Andreas Zeller. "Input Algebras." In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE. 2021, pp. 699–710 (cit. on pp. 59, 60).

[43] Giovanni Grano, Timofey V Titov, Sebastiano Panichella, and Harald C Gall. "How High Will it be? Using Machine Learning Models to Predict Branch Coverage in Automated Testing." In: *2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*. IEEE. 2018, pp. 19–24 (cit. on p. 88).

[44] *Gson*. https://github.com/google/gson. Version 2.8.5. 2017 (cit. on pp. 36, 56, 65).

[45] Frank R Hampel, Elvezio M Ronchetti, Peter J Rousseeuw, and Werner A Stahel. *Robust Statistics: The Approach Based on Influence Functions*. Vol. 196. John Wiley & Sons, 2011 (cit. on p. 88).

[46] **Nikolas Havrikov**. "Efficient Fuzz Testing Leveraging Input, Code, and Execution." In: *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*. IEEE Computer Society, 2017, pp. 417–420. DOI: 10.1109/ICSE-C.2017.26 (cit. on p. 6).

[47] **Nikolas Havrikov**. *tribble*. https://github.com/havrikov/tribble. 2019 (cit. on p. 33).

[48] **Nikolas Havrikov**, Alessio Gambi, Andreas Zeller, Andrea Arcuri, and Juan Pablo Galeotti. "Generating Unit Tests with Structured System Interactions." In: *12th IEEE/ACM International Workshop on Automation of Software Testing, AST@ICSE 2017, Buenos Aires, Argentina, May 20-21, 2017*. IEEE Computer Society, 2017, pp. 30–33. DOI: 10.1109/AST.2017.2 (cit. on pp. 6, 74).

[49] **Nikolas Havrikov**, Matthias Höschele, Juan Pablo Galeotti, and Andreas Zeller. "XMLMate: Evolutionary XML Test Generation." In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. ACM, 2014, pp. 719–722. DOI: 10.1145/2635868.2661666. URL: https://doi.org/10.1145/2635868.2661666 (cit. on p. 6).

[50] **Nikolas Havrikov**, Alexander Kampmann, and Andreas Zeller. "From Input Coverage to Code Coverage: Systematically Covering Input Structure with k-Paths." In: *ACM transactions on software engineering and methodology* (2021). TOSEM-2021-0220. ISSN: 1049-331X. Submitted (cit. on p. 7).

[51] **Nikolas Havrikov** and Andreas Zeller. "Systematically Covering Input Structure." In: *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 2019, pp. 189–199. DOI: 10.1109/ASE.2019.00027 (cit. on pp. 6, 31).

[52] Renáta Hodován and Ákos Kiss. "Modernizing Hierarchical Delta Debugging." In: *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*. A-TEST 2016. Seattle, WA, USA: Association for Computing Machinery, 2016, pp. 31–37. ISBN: 9781450344012. DOI: 10.1145/2994291.2994296. URL: https://doi.org/10.1145/2994291.2994296 (cit. on p. 86).

[53] Renáta Hodován and Ákos Kiss. "Practical Improvements to the Minimizing Delta Debugging Algorithm." In: *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 1: ICSOFT-EA, Lisbon, Portugal, July 24 - 26, 2016*. SciTePress, 2016, pp. 241–248. DOI: 10.5220/0005988602410248. URL: https://doi.org/10.5220/0005988602410248 (cit. on p. 86).

[54] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. "Grammarinator: A Grammar-Based Open Source Fuzzer." In: *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. ACM. 2018, pp. 45–48 (cit. on pp. 33, 35, 57, 81).

[55] Marc R. Hoffmann. *JaCoCo, Version 0.8.2*. https://github.com/jacoco/jacoco. 2018 (cit. on pp. 35, 57).

[56] Christian Holler, Kim Herzig, and Andreas Zeller. "Fuzzing with Code Fragments." In: *USENIX Conference on Security Symposium*. 2012, pp. 445–458. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler (cit. on p. 76).

[57] International Organization for Standardization. *ISO/IEC 9899:TC2: Programming Languages—C*. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf. May 2005 (cit. on p. 74).

[58] *Jackson Dataformat CSV*. https://github.com/FasterXML/jackson-dataformats-text/tree/master/csv. Version 2.9.8. 2018 (cit. on p. 36).

[59] *Jackson-Databind*. https://github.com/FasterXML/jackson-databind. Version 2.9.8. 2018 (cit. on p. 36).

[60] *jcsv*. https://code.google.com/archive/p/jcsv. Version 1.4.0. 2012 (cit. on p. 36).

[61] *json-flattener*. https://github.com/wnameless/json-flattener. Version 0.6.0. 2018 (cit. on pp. 36, 56, 65).

[62] *json-simple*. https://github.com/fangyidong/json-simple. Version 1.1.1. 2014 (cit. on pp. 36, 56, 65).

[63] Jussi Judin. *Java-AFL*. https://github.com/Barro/java-afl/. 2018 (cit. on p. 49).

[64] René Just, Michael D Ernst, and Gordon Fraser. "Efficient Mutation Analysis by Propagating and Partitioning Infected Execution States." In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 2014, pp. 315–326 (cit. on p. 91).

[65] Alexander Kampmann, **Nikolas Havrikov**, Ezekiel O. Soremekun, and Andreas Zeller. "When Does my Program do This? Learning Circumstances of Software Behavior." In: *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. ACM, 2020, pp. 1228–1239. DOI: 10.1145/3368089.3409687 (cit. on pp. 7, 33, 56, 57, 60, 62).

[66] Fitsum Meshesha Kifetew, Roberto Tiella, and Paolo Tonella. "Combining Stochastic Grammars and Genetic Programming for Coverage Testing at the System Level." In: *International Symposium on Search Based Software Engineering*. Springer, 2014, pp. 138–152. URL: https://doi.org/10.1007/978-3-319-09940-8_10 (cit. on p. 77).

[67] Fitsum Meshesha Kifetew, Roberto Tiella, and Paolo Tonella. "Generating Valid Grammar-Based Test Inputs by Means of Genetic Programming and Annotated Grammars." In: *Empirical Software Engineering* 22.2 (2017), pp. 928–961. URL: https://doi.org/10.1007/s10664-015-9422-4 (cit. on p. 78).

[68] Donald E Knuth. "Semantics of Context-Free Languages." In: *Mathematical systems theory* 2.2 (1968), pp. 127–145 (cit. on p. 73).

[69] Richard E Korf. "Depth-First Iterative-Deepening: An Optimal Admissible Tree Search." In: *Artificial intelligence* 27.1 (1985), pp. 97–109 (cit. on p. 86).

[70] Ralf Lämmel and Wolfram Schulte. "Controllable Combinatorial Coverage in Grammar-Based Testing." In: *IFIP International Conference on Testing of Communicating Systems*. Springer. 2006, pp. 19–38 (cit. on p. 72).

[71] Karim Lari and Steve J Young. "The Estimation of Stochastic Context-Free Grammars Using the Inside-Outside Algorithm." In: *Computer speech & language* 4.1 (1990), pp. 35–56 (cit. on p. 77).

[72] Sean Leary. *JSON-Java*. https://github.com/stleary/JSON-java. Version 20180813. 2017 (cit. on pp. 36, 56, 65).

[73] Yu Lei and Kuo-Chung Tai. "In-Parameter-Order: A Test Generation Strategy for Pairwise Testing." In: *Proceedings Third IEEE International High-Assurance Systems Engineering Symposium (Cat. No. 98EX231)*. IEEE. 1998, pp. 254–261 (cit. on p. 97).

[74] Zachary C Lipton. "The Mythos of Model Interpretability: In Machine Learning, the Concept of Interpretability is Both Important and Slippery." In: *Queue* 16.3 (2018), pp. 31–57 (cit. on p. 52).

[75] Joe Littlejohn. *jsonschema2pojo*. https://github.com/joelittlejohn/jsonschema2pojo. Version 1.0.0. 2017 (cit. on pp. 36, 56, 65).

[76] Davin Loegering. *json-simple by Cliftonlabs*. https://github.com/cliftonlabs/json-simple. Version 3.0.2. 2018 (cit. on pp. 36, 56, 65).

[77] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. "Pin: Building Customized Program Analysis Tools With Dynamic Instrumentation." In: *Acm sigplan notices* 40.6 (2005), pp. 190–200 (cit. on p. 80).

[78] Sean Luke. *Essentials of Metaheuristics*. second. Available for free at http://cs.gmu.edu/~sean/book/metaheuristics/. Lulu, 2013 (cit. on pp. 77, 78).

[79] John MacFarlane. *Markdown PEG-Grammar*. https://github.com/jgm/peg-markdown/blob/master/markdown_parser.leg. 2013 (cit. on pp. 33, 34).

[80] H. B. Mann and D. R. Whitney. "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other." In: *The Annals of Mathematical Statistics* 18.1 (1947), pp. 50–60. ISSN: 00034851. URL: http://www.jstor.org/stable/2236101 (cit. on pp. 35, 36).

[81] *Markdown4J*. https://github.com/jdcasey/markdown4j. Version 2.2-cj-1.1. 2016 (cit. on p. 36).

[82] Thomas J McCabe. "A Complexity Measure." In: *IEEE Transactions on software Engineering* 4 (1976), pp. 308–320 (cit. on p. 79).

[83] William M McKeeman. "Differential Testing for Software." In: *Digital Technical Journal* 10.1 (1998), pp. 100–107 (cit. on p. 74).

[84] Bruce McKenzie. "Generating Strings at Random From a Context Free Grammar." 1997 (cit. on p. 82).

[85] Phil McMinn. "Search-Based Software Test Data Generation: A Survey." In: *Software testing, Verification and reliability* 14.2 (2004), pp. 105–156 (cit. on p. 77).

[86] Microsoft Research. *AsmL – Abstract State Machine Language*. http://research.microsoft.com/fse/AsmL/. 2005 (cit. on p. 73).

[87] Barton P. Miller, Louis Fredriksen, and Bryan So. "An Empirical Study of the Reliability of UNIX Utilities." In: *Commun. ACM* 33.12 (Dec. 1990), pp. 32–44. ISSN: 0001-0782. DOI: 10.1145/96267.96279. URL: https://doi.org/10.1145/96267.96279 (cit. on pp. 1, 71).

[88] Brad L Miller, David E Goldberg, et al. "Genetic Algorithms, Tournament Selection, and the Effects of Noise." In: *Complex systems* 9.3 (1995), pp. 193–212 (cit. on p. 85).

[89] Ghassan Misherghi and Zhendong Su. "HDD: Hierarchical Delta Debugging." In: *Proceedings of the 28th international conference on Software engineering*. 2006, pp. 142–151 (cit. on pp. 79, 86).

[90] Gordon Mohr, Michael Stack, Igor Rnitovic, Dan Avery, and Michele Kimpton. "Introduction to Heritrix." In: *4th International Web Archiving Workshop*. 2004, pp. 109–115 (cit. on p. 81).

[91] Anders Møller. *dk.brics.automaton – Finite-State Automata and Regular Expressions for Java*. http://www.brics.dk/automaton/. 2017 (cit. on pp. 12, 13).

[92] Mozilla. *Rhino*. https://github.com/mozilla/rhino. Version 1.7.10. 2018 (cit. on pp. 56, 65).

[93] Mozilla. *SpiderMonkey JavaScript Interpreter*. https://spidermonkey.dev. 2019 (cit. on p. 79).

[94] mruby developers. *mruby*. https://mruby.org/. 2018 (cit. on p. 83).

[95] Ali Bou Nassif, Danny Ho, and Luiz Fernando Capretz. "Towards an Early Software Estimation Using log-Linear Regression and a Multilayer Perceptron Model." In: *Journal of Systems and Software* 86.1 (2013), pp. 144–160. DOI: 10.1016/j.jss.2012.07.050. URL: https://doi.org/10.1016/j.jss.2012.07.050 (cit. on p. 88).

[96] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: Updated for Scala 2.12*. 3rd. Sunnyvale, CA, USA: Artima Incorporation, 2016. ISBN: 0981531687 (cit. on p. 13).

[97] Carlos Pacheco and Michael D. Ernst. "Randoop: Feedback-Directed Random Testing for Java." In: *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. ACM, 2007, pp. 815–816. DOI: 10.1145/1297846.1297902. URL: https://doi.org/10.1145/1297846.1297902 (cit. on p. 88).

[98] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. "Semantic Fuzzing with Zest." In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Beijing, China, 2019, pp. 329–340. ISBN: 9781450362245. DOI: 10.1145/3293882.3330576. URL: https://doi.org/10.1145/3293882.3330576 (cit. on p. 86).

[99] Rohan Padhye, Koushik Sen, and Paul N. Hilfinger. "ChocoPy: A Programming Language for Compilers Courses." In: *Proceedings of the 2019 ACM SIGPLAN Symposium on SPLASH-E*. SPLASH-E 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 41–45. ISBN: 9781450369893. DOI: 10.1145/3358711.3361627. URL: https://doi.org/10.1145/3358711.3361627 (cit. on p. 86).

[100] Terence J. Parr and Russell W. Quong. "ANTLR: A predicated-LL(k) parser generator." In: *Software: Practice and Experience* 25.7 (1995), pp. 789–810 (cit. on pp. 10, 33, 88).

[101] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. "Scikit-learn: Machine Learning in Python." In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830 (cit. on p. 52).

[102] Riccardo Poli and Nicholas F McPhee. *Covariant Parsimony Pressure in Genetic Programming*. Tech. rep. Citeseer, 2008 (cit. on p. 78).

[103] The GNOME Project. *libxml2*. http://xmlsoft.org/. 2017 (cit. on p. 81).

[104] William F. Punch. "Book Review: Genetic Programming - An Introduction: On the Automatic Evolution of Computer Programs and Its Applications." In: *Genet. Program. Evolvable Mach.* 2.2 (2001), pp. 193–195. DOI: 10.1023/A:1011508532477. URL: https://doi.org/10.1023/A:1011508532477 (cit. on p. 77).

[105] Paul Purdom. "A Sentence Generator for Testing Parsers." In: *BIT Numerical Mathematics* 12.3 (1972), pp. 366–375 (cit. on pp. 3, 71).

[106] Kanishk Rastogi. *URL Detector*. https://github.com/linkedin/URL-Detector. Version 0.1.17. 2018 (cit. on pp. 36, 56, 65).

[107] Jesse Ruderman. *Introducing jsfunfuzz*. https://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/. 2007 (cit. on p. 85).

[108] Shlomo S Sawilowsky. "New Effect Size Rules of Thumb." In: *Journal of Modern Applied Statistical Methods* 8.2 (2009), p. 26 (cit. on p. 68).

[109] *Scikit-learn Guide on Decision Trees*. https://scikit-learn.org/stable/modules/tree.html. 2020 (cit. on p. 57).

[110] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. "AddressSanitizer: A Fast Address Sanity Checker." In: *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 2012, pp. 309–318 (cit. on p. 81).

[111] Anthony N. Simon. *jurl*. https://github.com/anthonynsimon/jurl. Version v0.3.0. 2018 (cit. on pp. 36, 56, 65).

[112] *Simple Flat Mapper*. https://github.com/arnaudroger/SimpleFlatMapper. Version 6.1.1. 2018 (cit. on p. 36).

[113] *simplecsv*. https://github.com/quux00/simplecsv. Version 2.1. 2018 (cit. on p. 36).

[114] Ezekiel Soremekun, Esteban Pavese, **Nikolas Havrikov**, Lars Grunske, and Andreas Zeller. "Inputs from Hell: Learning Input Distributions for Grammar-Based Test Generation." In: *IEEE Transactions on Software Engineering* (2020). DOI: 10.1109/TSE.2020.3013716 (cit. on pp. 6, 33, 78, 81, 85).

[115] Terence Soule and James A Foster. "Effects of Code Growth and Parsimony Pressure on Populations in Genetic Programming." In: *Evolutionary computation* 6.4 (1998), pp. 293–309 (cit. on p. 78).

[116] C. Spearman. "The Proof and Measurement of Association between Two Things." In: *The American Journal of Psychology* 15.1 (1904), pp. 72–101. ISSN: 00029556. URL: http://www.jstor.org/stable/1412159 (cit. on p. 45).

[117] Ralf Sternberg. *minimal-json*. https://github.com/ralfstx/minimal-json. Version 0.9.5. 2017 (cit. on pp. 36, 56, 65).

[118] Robin Stocker. *autolink-java*. https://github.com/robinst/autolink-java. Version 0.9.0. 2018 (cit. on pp. 36, 56, 65).

[119] *super-csv*. https://github.com/super-csv/super-csv. Version 2.4.0. 2016 (cit. on p. 36).

[120] Philip H Swain and Hans Hauska. "The Decision Tree Classifier: Design and Potential." In: *IEEE Transactions on Geoscience Electronics* 15.3 (1977), pp. 142–147 (cit. on p. 52).

[121] Nikolai Tillmann and Wolfram Schulte. "Parameterized Unit Tests." In: *ACM SIGSOFT Software Engineering Notes* 30.5 (2005), pp. 253–262 (cit. on p. 73).

[122] Omer Tripp, Omri Weisman, and Lotem Guy. "Finding Your Way in the Testing Jungle: A Learning Approach to Web Security Testing." In: *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*. ACM, 2013, pp. 347–357. DOI: 10.1145/2483760.2483776. URL: https://doi.org/10.1145/2483760.2483776 (cit. on p. 87).

[123] *Txtmark*. https://github.com/rjeschke/txtmark. Version 0.13. 2017 (cit. on p. 36).

[124] *URL Grammar*. https://github.com/antlr/grammars-v4/blob/master/url/url.g4. 2018 (cit. on pp. 33, 34).

[125] Phillip van Heerden, Moeketsi Raselimo, Konstantinos Sagonas, and Bernd Fischer. "Grammar-Based Testing for Little Languages: An Experience Report with Student Compilers." In: *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2020. Virtual, USA: Association for Computing Machinery, 2020, pp. 253–269. ISBN: 9781450381765. DOI: 10.1145/3426425.3426946. URL: https://doi.org/10.1145/3426425.3426946 (cit. on p. 57).

[126] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. "Ifuzzer: An Evolutionary Interpreter Fuzzer Using Genetic Programming." In: *European Symposium on Research in Computer Security*. Springer. 2016, pp. 581–601 (cit. on p. 78).

[127] Vasudev Vikram, Rohan Padhye, and Koushik Sen. "Growing a Test Corpus with Bonsai Fuzzing." In: *arXiv preprint arXiv:2103.04388* (2021) (cit. on pp. 30, 86).

[128] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python." In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2 (cit. on p. 35).

[129] Jeffrey M. Voas. "PIE: A Dynamic Failure-Based Technique." In: *IEEE Transactions on software Engineering* 18.8 (1992), p. 717 (cit. on p. 91).

[130] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. "Skyfire: Data-Driven Seed Generation for Fuzzing." In: *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2017, pp. 579–594 (cit. on pp. 79, 96).

[131] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. "Superion: Grammar-Aware Greybox Fuzzing." In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 724–735 (cit. on p. 84).

[132] Frank Wilcoxon. "Individual Comparisons by Ranking Methods." In: *Biometrics Bulletin* 1.6 (1945), pp. 80–83. ISSN: 00994987. URL: http://www.jstor.org/stable/3001968 (cit. on p. 68).

[133] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. "Finding and Understanding Bugs in C Compilers." In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 2011, pp. 283–294 (cit. on p. 74).

[134] Michal Zalewski. *American Fuzzy Lop*. https://lcamtuf.coredump.cx/afl/. 2017 (cit. on pp. 49, 83).

[135] Andreas Zeller and Ralf Hildebrandt. "Simplifying and Isolating Failure-Inducing Input." In: *IEEE Transactions on Software Engineering* 28.2 (2002), pp. 183–200 (cit. on pp. 79, 86).

[136] Byoung-Tak Zhang and Heinz Mühlenbein. "Balancing Accuracy and Parsimony in Genetic Programming." In: *Evolutionary Computation* 3.1 (1995), pp. 17–38 (cit. on p. 78).

[137] Jie Zhang, Lingming Zhang, Mark Harman, Dan Hao, Yue Jia, and Lu Zhang. "Predictive Mutation Testing." In: *IEEE Transactions on Software Engineering* 45.9 (2018), pp. 898–918 (cit. on p. 91).