Systematically Covering Input Structure

Nikolas Havrikov
CISPA Helmholtz Center for Information Security
Saarland Informatics Campus
nikolas.havrikov@cispa.saarland

Andreas Zeller
CISPA Helmholtz Center for Information Security
Saarland Informatics Campus
zeller@cispa.saarland

***Abstract*—Grammar-based testing uses a given grammar to produce syntactically valid inputs. To cover program features, it is necessary to also cover input features—say, all URL variants for a URL parser. Our k-path algorithm for grammar production systematically covers syntactic elements as well as their combinations. In our evaluation, we show that this results in a significantly higher code coverage than state of the art.**

I. Introduction

Testing programs with randomly generated inputs, or “fuzzing”, is a cost-effective means to test programs for robustness: If a program has not been subjected to random inputs before, the chances are high that some input will cause the program to fail.

To reach deeper layers of a program, though, inputs must be *syntactically valid* because invalid inputs would be rejected already during initial input processing. To this end, recent fuzzing approaches [1]–[4] make use of *grammars* to specify the language of program inputs. A grammar-based test generator uses a grammar to expand a start symbol into further symbols (often selecting from alternatives), which it would repeatedly expand until only terminal symbols are left. For the grammar shown in Figure 1, for instance, the *Expr* start symbol may expand into an *AddExpr* and then a *MultExpr*, which again may expand into a *UnaryExpr*, which would eventually become a string of digits.

While the concept of producing inputs from grammars is simple, any practical implementation has to struggle with two problems. The first issue is to ensure an input does not *grow beyond bounds*. In Figure 1, if the producer always selects the last expansion alternative, the result will be an infinitely long arithmetic expression. A producer thus needs means to determine which expansion to choose in order to avoid such growth.

The second issue of producing from grammars is to ensure *input coverage*. Intuitively, a high variation in the inputs (say, operators) induces a high variation in program behavior. Conversely, if some input element is not present in the input (say, `"+"`), the code that processes it will not be executed. It is thus desirable to *cover* as many different input elements and productions as possible. In our expression grammar, this means to cover all operators and all digits.

How does one maximize coverage? One way, suggested by Purdom [5] is to ensure that during production, uncovered production alternatives would be preferred over covered production alternatives. In Figure 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’d go for the `"-"` alternative the next time. Over time, Purdom’s approach would cover all alternatives.

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

Figure 1: Grammar for JavaScript expressions (simplified)

$DecDigit \rightarrow X_0$; $X_2 \rightarrow$ `"2"` | X_3; $X_5 \rightarrow$ `"5"` | X_6;
$X_0 \rightarrow$ `"0"` | X_1; $X_3 \rightarrow$ `"3"` | X_4; . . .
$X_1 \rightarrow$ `"1"` | X_2; $X_4 \rightarrow$ `"4"` | X_5; $X_9 \rightarrow$ `"9"`;

Figure 2: A *DecDigit* rule variant that is hard to cover

Unfortunately, there are grammars where neither the random nor Purdom’s approach help in achieving coverage. This becomes apparent when we reformulate the *DecDigit* rules as shown in Figure 2. Now, the digits are no longer chosen and produced uniformly. Choosing a *DecDigit* expansion at random yields a 50% chance of producing a `"0"`, a 25% chance of a `"1"`, and a 1/1024 = 0.0977% chance of producing a `"9"`. Clearly, we would want to cover all terminals quickly, but on average, it would take 1,024 inputs until we see `"9"` produced.

Purdom’s approach helps a bit, but is far from perfect; in the first expansion of X_0, it would mark `"0"` as covered, then expanding X_1 the second time (with a 1/512 chance of producing a `"9"`). However, after having both covered `"0"` and X_1, it would no longer prefer one over the other, still yielding 50% `"0"` expansions.

```
Identifier → Identifier | Character Identifier;
Character → ASCIICharacter | UnicodeCharacter;
ASCIICharacter → ASCIIUpper | ASCIILower | "_";
```

Figure 3: An *Identifier* rule that is hard to cover

If Figure 2 feels a bit too pathological, consider Figure 3, listing possible rules for identifiers. Not only does this formulation result in 50% identifiers consisting of one character only; also, both *Identifier* alternatives would be quickly marked as covered by Purdom's approach. In other words, our producer would have no incentive to systematically cover identifier characters or their categories. The deeper the grammar, the greater the extent of this coverage problem.

In this paper, we introduce a novel grammar production algorithm, called k-path, that addresses all these issues:

Input coverage. The k-path algorithm ensures quick coverage of all grammar features. Specifically, it chooses expansion alternatives that *lead to input elements not covered yet;* applied on Figure 2, its first ten productions consist of the elements `"0"` to `"9"`; on Figure 3, it produces long identifiers that cover all valid *Character* elements.

Combination coverage. The k-path algorithm ensures (if wanted) coverage of *combinations* of grammar features, whose context size is controlled by k. Applied on Figure 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 in a symbolic calculator, for instance.

Growth control. The k-path algorithm effectively limits growth beyond bounds. If the number or depth of elements produced exceeds a certain threshold, the k-path algorithm will always choose alternatives that eventually end in terminal symbols, thus ensuring a quick closure of production.

These features are effective. Compared against a state-of-the-art grammar producer, *Grammarinator* [2], our *tribble* prototype implementing k-path results in a higher code coverage in the same time. These benefits are not limited to text inputs alone. We show that a number of classical testing domains (configuration testing, UI testing, reactive systems) can be encoded as grammars, resulting in k-path systematically covering all features, interactions, and commands, as well as their sequences. To the best of our knowledge, k-path is the first algorithm systematically and universally striving for "deep" coverage beyond individual production rules.

The remainder of this paper is organized as follows. Section II introduces grammars, their elements, and derivations. Section III details coverage criteria for grammars, including k-path coverage. Section IV derives the k-path production algorithm, which systematically satisfies these coverage criteria. After giving implementation details (Section V), Section VI evaluates the k-path algorithm, comparing against the *Grammarinator* state-of-the-art producer. Section VII discusses further applications beyond text inputs, encoding problems such as configuration testing, UI testing, or reactive systems as grammars that can be effectively handled by the k-path algorithm. After discussing related work (Section VIII), Section IX closes with conclusion and future work.

```
Grammar → Production+;
Production → NonTerminal "→" Alternation ";";
Alternation → Concatenation ("|" Concatenation)*;
Concatenation → Atom+;
Atom → ("(" Alternation ")"
          | Literal | Reference) Quantifier?;
Quantifier → "?" | "+" | "*"
     | "{," num "}"
     | "{" num ",}"
     | "{" num "," num "}";
```

Figure 4: A grammar for context-free grammars (excerpt)

II. Grammars and Derivations

In this work, we consider test generation based on *context-free grammars.* We assume the reader is familiar with the concept of context-free grammars; for the purpose of precision, we introduce their syntax, semantics, and composition as used in this paper.

A. Composition

To name and define the individual elements of the grammars as used in this paper, we use—well—a grammar that describes their names and syntax. As detailed in Figure 4, a *grammar* consists of *productions*, each of which expands a *nonterminal* into an *alternation*. The grammar in Figure 1, for instance, has seven such productions, one for each nonterminal.

An alternation is a sequence of *alternatives* over non-empty sequences of *concatenations*. A concatenation consists of *atoms*, which can be parenthesized alternations, literals, or references (which are essentially nonterminals on the right side of a production). In Figure 1, an *AddExpr* nonterminal expands into either a *MultExpr*, or a concatenation of another *AddExpr*, an operator (plus or minus), and a *MultExpr*.

Optional *quantifiers* allow to express that an atom can be repeated zero or once (?), once or more (+), or zero or more ($\star$) times; $\{n,m\}$ indicates at least n repetitions and at most m repetitions. In Figure 1, *DecDigits* is a non-empty sequence of *DecDigit* literals.

In the remainder of this paper we will collectively refer to references and literals as *symbols*. Additionally, we assume that all grammars have exactly one *start symbol*: a nonterminal which no reference refers to. In Figure 1 the start symbol is *Expr*, while in Figure 4 it is *Grammar*.

B. Graph Representation

A grammar can also be seen as a directed graph whose nodes represent parts of the grammar from Figure 4. Not unlike a tree, this graph has a "root" at the start symbol of the grammar, and we will refer to nodes that are connected to a given node as its *children*. The children correspond to the derivations available from the part of the grammar their parent node corresponds to. Unlike a tree, however, a grammar graph can have cycles because some nonterminals can be used in multiple derivations. To illustrate this concept Figure 5 shows an excerpt of the graph representation of the expression grammar from Figure 1. A graph is derived from the productions of a grammar by applying the following rules:

1) Each nonterminal is replaced by the graph representation of the right hand side of its production.
2) For each alternation a synthetic node ⊕ is created. Its children are the corresponding concatenations.
3) Analogously, concatenations become synthetic ~⃝ nodes.
4) Each quantifier becomes a node (+, ∗, ?, `{n,m}`) whose child is the graph representation of the atom the quantifier is attached to.
5) Literals become nodes with no children.
6) References to nonterminals become nodes that have as their child the subgraph created by the nonterminal they refer to. In Figure 5 these connections are represented by dashed lines so that the underlying tree-like structure remains clearly visible.
7) All reference and literal nodes (referred to as *symbolic nodes*) are assigned a unique id by numbering equally named elements. This is necessary to distinguish between equal nodes in different contexts. For example, consider the literal `"+"` which occurs both inside the *AddExpr* and *UnaryExpr* productions—in the graph these should be different nodes, hence they get two different ids: `"+"`$_0$ and `"+"`$_1$.

For space reasons, Figure 5 omits synthetic ⊕ nodes which would be the only children of references. Otherwise $\mathit{AddExpr}_0$, $\mathit{MultExpr}_0$, and $\mathit{UnaryExpr}_0$ would each have one such node as their only child.

C. Derivation Trees

In this work we use the classical definition of derivation trees as given in literature [6]. In short, a derivation tree represents the structure of a string according to a given grammar. Additionally, we say that each node in the tree is of the type of its corresponding node in the grammar graph. Further, a tree node has numbered *slots* which are said to be filled with their children. Due to the nature of our grammars, the root of a derivation tree always has exactly one child slot. Figure 6 shows the derivation tree for the string “x+42” according to the grammar given in Figure 1.

III. Grammar Coverage Criteria

When producing strings from a grammar, we want to *cover* individual features of the grammar. The intuition of coverage is that if some feature of the grammar is never exercised during testing, neither will the code be executed that processes this very feature; and if code is not executed, it cannot reveal faults during testing. In Figure 1, for instance, if the generated inputs miss out the `"++"` operator, we will not be able to exercise the associated code. In this section, we introduce *coverage criteria* as used in this paper.

A. Symbol Coverage

We start with the (very basic) symbol coverage, stating that each symbol (i.e. reference or literal) in the grammar should be encountered at least once:

Definition 1 (Symbol Coverage): A set of strings achieves *symbol coverage* if each symbolic node in the graph representation of the grammar occurs at least once in the set of their derivation trees.

This definition assumes a (constructive) derivation tree; however, if the grammar is used for *parsing* inputs, we can obtain an equivalent parse tree.

The associated *symbol coverage metric* then becomes

$$\mathit{symbol\ coverage} = \frac{\#\mathit{symbolic\ nodes\ in\ derivation\ tree}}{\#\mathit{symbolic\ nodes\ in\ grammar\ graph}}$$

How much symbol coverage does the derivation in Figure 6 achieve? The full grammar consists of 40 symbolic nodes of which the given derivation covers 13, resulting in a symbol coverage of 32.5%. Note that nodes with the same id are counted as only one node.

B. k-Path Coverage

As the next step in coverage metrics, we introduce the notion of *context:* We would like to cover symbols not only individually, but also in the context of other symbols. This is based on the intuition that some input elements have different meaning in different contexts, and programs may thus process them differently. For arithmetic expressions (Figure 1), for instance, it may be useful to test various combinations of *AddExpr* expressions within *MultExpr* expressions and vice versa because we could then cover program features like applying distributive laws.

In principle, one would like to see *all combinations* of symbols covered; however, this quickly leads to a combinatorial explosion. We thus introduce the notion of *k-path coverage,* which mandates that all unique *symbol paths* up to a length of k be covered.

Definition 2 (k-path): A k-path is a sequence of nodes connected in the direction of the edges with exactly k symbolic nodes.

For example, walking along the left edges in the graph in Figure 5 starting from its root, we end up with the 5-path $\mathit{Expr}_0 \rightarrow \mathit{AddExpr}_0 \rightarrow \mathit{MultExpr}_0 \rightarrow \mathit{UnaryExpr}_0 \rightarrow$ `"+"`$_0$. Note that k-paths need not start at the root and thus the value of k does not limit the depth of a tree. In fact, a tree must have at least depth k to contain k-paths.

In the following definition we leverage the fact that the definition of k-paths applies to both grammar graphs and derivation trees.

Definition 3 (k-Path Coverage): A set of strings achieves *k-path coverage* if each k-path in the grammar graph occurs at least once in the set of their derivation trees.

The appropriate coverage metric is again defined by the fraction of possible paths covered.

The case $k = 1$ is special: *1-path coverage* is equivalent to the symbol coverage as defined above.

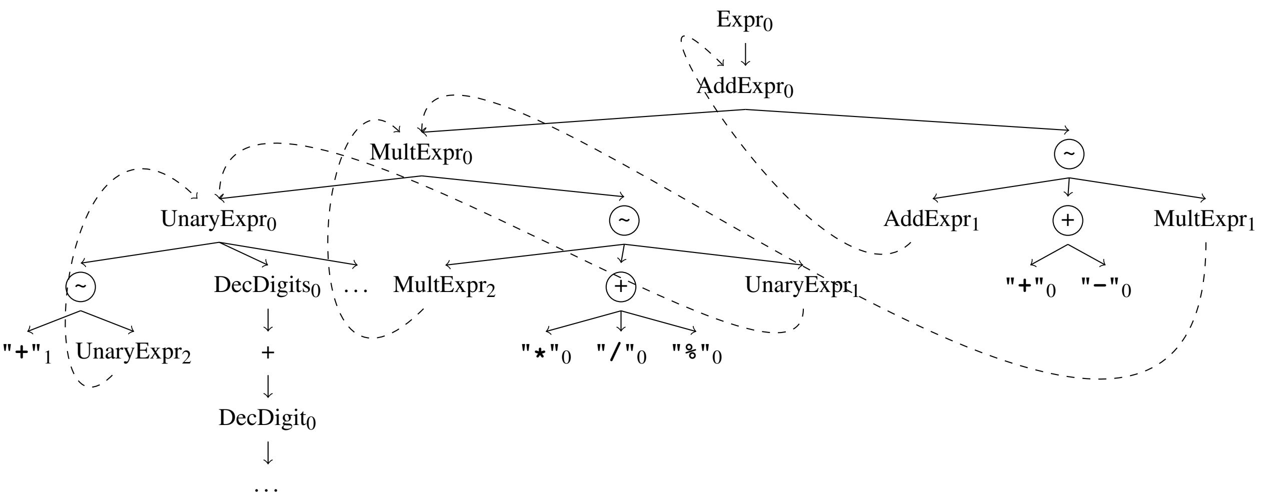

Figure 5: Grammar from Figure 1 represented as a graph (excerpt)

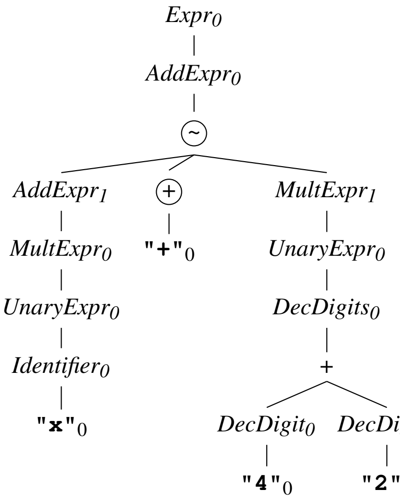

Figure 6: Derivation tree representing the string **"x+42"** according to the grammar from Figure 1

Let us assume $k = 2$ and see which 2-paths are covered by the derivation tree in Figure 6:

1) $\mathit{Expr}_0 \rightarrow \mathit{AddExpr}_0$
2) $\mathit{AddExpr}_0 \rightarrow \mathit{AddExpr}_1$
3) $\mathit{AddExpr}_0 \rightarrow$ "+"$_0$
4) $\mathit{AddExpr}_0 \rightarrow \mathit{MultExpr}_1$
5) $\mathit{AddExpr}_1 \rightarrow \mathit{MultExpr}_0$
6) $\mathit{MultExpr}_0 \rightarrow \mathit{UnaryExpr}_0$
7) $\mathit{UnaryExpr}_0 \rightarrow \mathit{Identifier}_0$
8) $\mathit{Identifier}_0 \rightarrow$ "x"$_0$
9) $\mathit{MultExpr}_1 \rightarrow \mathit{UnaryExpr}_0$
10) $\mathit{UnaryExpr}_0 \rightarrow \mathit{DecDigits}_0$
11) $\mathit{DecDigits}_0 \rightarrow \mathit{DecDigit}_0$
12) $\mathit{DecDigit}_0 \rightarrow$ "4"$_0$
13) $\mathit{DecDigit}_0 \rightarrow$ "2"$_0$

To find all the paths to be covered, we can use the grammar and determine all distinct sequences of symbols with a length of k. For Figure 1 and $k = 2$ we have 126 paths; these include, for example, encountering *Identifier* in the context of all possible unary expressions – a condition which Figure 6 does not satisfy. For $k = 3$ we already have 526 paths to cover, of which the same tree only covers 12.

Note that the number of paths grows exponentially with k. For the grammar in Figure 1, $k = 4$ already gives us 2333 paths of symbols to be covered; and $k = 5$ yields 10247 paths.

IV. Coverage-Driven Generation

Because we can enumerate all possible k-paths, we can also construct an algorithm that systematically produces a forest of derivation trees that cover them. Our algorithm for doing just that is given in Figure 7.

Given k, a grammar, and a depth limit the algorithm begins by storing all k-paths available into a list P and shuffling it. It then iterates over this list generating a derivation tree to cover each k-path.

For each such tree, the algorithm maintains a list Q of child slots that still need to be expanded. Initially, it only holds the only child slot of the start symbol.

The loop in Line 9 expands the slots from Q forming an ever growing tree-front. Having found a node n which is the first node of the currently targeted path p that has not yet been used in a derivation, in Line 11 we select and remove from Q a slot a that enables reaching n. If all nodes in p have already been used, the choice does not matter and a slot is chosen at random.

Starting at Line 12 when selecting among available alternatives for the chosen slot a, we again select the one getting us to cover p in the least number of derivation steps. If we already covered p, one of two strategies is applied depending on the current tree depth: If there are nodes which lead to producing k-paths which fit in the given depth limit, choose one of the least often used, otherwise choose the node with the smallest possible derivation to complete the current tree r as quickly as possible.

```
1:  procedure ConstructKPath(k, grammar, depth limit)
2:      forest ← {}
3:      P ← { all k-paths in grammar }
4:      P ← shuffled P
5:      while P ≠ {} do
6:          p ← remove next k-path from P
7:          r ← the start symbol from grammar
8:          Q ← { child slot of r }
9:          while Q ≠ {} do
10:             n ← next uncovered node in p, or ✗ if all covered
11:             a ← a slot removed from Q with shortest derivation path to n if it is not ✗, otherwise any.
12:             if n is not ✗ then
13:                 m ← expansion of a closest to n
14:             else
15:                 C ← expansions of a that fit in the depth limit
16:                 if C is empty then
17:                     m ← shortest expansion of a
18:                 else
19:                     m ← element of C with fewest k-paths used so far
20:             Fill slot a with m
21:             Remove from P all k-paths ending in m found in r
22:             for each admissible child slot b of m do
23:                 Q ← Q ∪ {b}
24:         forest ← forest ∪ {r}
25:     return forest
```

Figure 7: k-Path Algorithm

Since we have seen in Section III that the number of k-paths can grow exponentially with k we would like to avoid having to generate an entire derivation tree for each k-path. In Line 21 we therefore keep track of k-paths that we happened to cover while "on the way" to p and remove them from P. This dramatically reduces the number of trees generated.

We then add to Q the slots of the chosen expansion m, such that the loop in Line 9 completes the current tree in r, which will then be added to the *forest*, which the algorithm returns after all k-paths in P are covered.

The algorithm avoids boundless growth by construction: it always takes the shortest derivation route until the targeted path is covered and closes off peripheral subtrees within the given depth limit if possible, or with the shortest available derivations otherwise.

As an example, consider the derivation of a tree when the next goal to cover is the 3-path $Expr_0 \rightarrow AddExpr_0 \rightarrow$ `"+"`$_0$ from Figure 5. At this point this path is removed from the set of the not yet reached targets in Line 6 and stored as p. Then a new tree root node r is created with type $Expr_0$. In Line 10 n becomes $AddExpr_0$ and a is filled with an instance of $AddExpr_0$ as it is the only option at this point. In the next iteration of the loop n becomes `"+"`$_0$ and a is the $\oplus$ node as it is the only child of $AddExpr_0$ (we left this node out in Figure 5 due to space reasons). The next important step is when beginning in Line 12 a decision is made regarding which of the two children of $\oplus$ are admissible in this context since only one of them can be expanded in the current tree. Again, this decision is guided by the shortest path to the currently targeted `"+"`$_0$ node—in this case the $\textcircled{\sim}$ node gets expanded and its three children $AddExpr_1$, $\oplus$, and $MultExpr_1$ are then added to Q. After the `"+"`$_0$ is added to the tree in the next iteration, p will become ✗ and the algorithm will close off the two remaining nodes $AddExpr_1$ and $MultExpr_1$, while also keeping track of 3-paths covered while doing so and removing them from P.

Note that in our example, we will have additionally covered the 3-paths $Expr_0 \rightarrow AddExpr_0 \rightarrow AddExpr_1$ and $Expr_0 \rightarrow AddExpr_0 \rightarrow MultExpr_1$ (and many more on the way down such as $MultExpr_1 \rightarrow MultExpr_0 \rightarrow UnaryExpr_0$) while only actively trying to cover the path to `"+"`$_0$.

V. Implementation

Our *tribble* prototype comes in about 1,200 lines of code written in the Scala programming language and only requires a Java 8 or later runtime to work. The grammars it takes as input are written using our specifically designed language, which is a subset of Scala itself.

In fact, the grammar in Figure 1 only needs minor adjustments to be readable by *tribble*: All nonterminals and references need to be prefixed with ′, the → must be replaced by `:=`, and productions are delimited by a comma instead of a semicolon. Concatenations must be explicitly indicated by ~, and quantifiers are encoded as calls to the `.rep(n,m)` method. Additionally, regular expression shorthands are supported by means of the `.regex` method that can be called on strings.

As an example, consider the following small grammar: `'S := 'A.rep(2,5) ~ "(b|c)*".regex, 'A := "a"`. This grammar produces strings that start with two to five "a"s, optionally followed by any number of "b"s and "c"s.

Our grammar description language being a valid subset of Scala enables users of *tribble* to profit from syntax highlighting available in all IDEs that support Scala at no development cost to us.

VI. Evaluation

We want to study the effects of systematically covering k-paths compared to grammar-based fuzz testing. We compare our approach against the state-of-the-art grammar-based input generator *Grammarinator* [2], which expects its grammars to be in the format defined by the ANTLR parser generator [7]. For our experiments we selected the grammars of popular and well known languages from the popular GitHub repository hosting a variety of grammars in ANTLR format [8]–[11] and manually translated them into the format required by *tribble* as consistently as possible, i.e. only changing their notation. We carry out our experimental investigation on open source projects listed in the leftmost column of Table I. Their selection consists of the most popular results on Google Search among open source projects consuming the selected formats.

For the JSON language all our subjects are parsers, except for some notable exceptions: *jackson-databind*, *genson*, *gson*, *fastjson* additionally allow data-binding for automatic (de-) serialization 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.

Our subjects for the Markdown format concern themselves with rendering their inputs into HTML outputs for displaying in a web browser.

Since most of the subjects are libraries, they require a test harness to execute test cases. 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.

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 [12] provided by the authors of *Grammarinator*. Further, we set the parameter `--cooldown` to 0.9 and add a `simple_space_transformer` as described in the tool's paper. 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.

A. Code Coverage

Table I 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 [13] tool to gather coverage data by means of offline bytecode instrumentation. The columns labeled as k-path show the average branch coverage achieved with files generated by the k-path algorithm with the given value of k. The k-gram columns show the average branch coverage for runs of *Grammarinator* having the same number of files as runs of k-path with the given k. E.g. if an invocation of 2-path produced a set of 10 files, the corresponding 2-gram run would also consist of 10 files. Note that for each k the values in Table I represent the average of 50 such corresponding pairs rounded to four decimals. For each pair, the bigger entry is given in bold font for easy comparison at a glance. To investigate if these average values, in fact, do represent the average performance of both approaches, we performed a statistical significance analysis using the two-sided Mann–Whitney U test [14] as implemented in the Python SciPy library [15]. In Table I, the significantly different entries (all but six) are shown in bold. Further, the subjects listed in Table I are grouped by the grammar describing the language of their inputs: JSON, CSV, URL, and Markdown.

Table I shows that for $k = 1$ the coverage achieved by inputs generated by the k-path algorithm roughly equates the one achieved by *Grammarinator* 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 meaningful combinations and nesting of JSON arrays and objects, which might trigger 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 coverage is much higher than that achieved by 1-path. To produce these additional combinations, however, more inputs needed to be generated by 2-path (see Table III) and so *Grammarinator* also has a higher generation budget. Still, for *Grammarinator* this does not always lead to an increase in coverage. For an example consider the coverage for the subject *argo*, whose coverage is 0.4116 for 1-gram and only 0.3963 for 2-gram.

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

Increasing the context depth k to 5 improves the achieved coverage over the previous configurations, but this time, *Grammarinator* is beginning to catch up again. This is due to contexts deeper than a certain threshold not necessarily corresponding to explicit variations in the executed code anymore. For a JSON parser for example, there would not be much difference between seeing a doubly or triply nested array in terms of control flow. However, there could still be some notion of context encoded in the flow of data instead. For example there could be a counter 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.

For $k = 2$ and $k = 3$, tribble *covers more branches than* Grammarinator *on 22/24 and 23/24 subjects.*

The advantage of *tribble* over *Grammarinator* can be 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.

B. Defect Detection

When generating test inputs, one must not forget why we test in the first place. During our experiments, we found a number of exceptions thrown by our test subjects; as all of these are triggered by system inputs, they all indicate internal errors. We filter out those exception classes that are defined inside the subjects' packages assuming they represent expected user-facing error behavior. The results are summarized in Table II: For each subject in which exceptions could be triggered, the exception class name, its origin, as well as its detection rate is given for both approaches. The detection rate indicates 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 stacktrace 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 its implementation of input generation 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 $k = 1$ configuration k-path is able to trigger three exceptions exclusively: Two `NullPointerExceptions` and a `StringIndexOutOfBoundsException`, none of which should ever be allowed to be thrown into user code as they all indicate fatal errors of the internal state.

With increasing k the detection rate increases for both approaches, but it does so for *tribble* more reliably: 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 are unique to *tribble*, and the remaining 12 were found by both.

> *Compared to* Grammarinator,
> tribble *found more unique exceptions.*

C. Threats to Validity

Our evaluation is subject to threats to validity.

- In terms of *external validity,* we have examined 24 subjects and four grammars, covering a variety of input and implementation features; while the results are consistent, we cannot claim generality across all programs and inputs.
- In terms of *internal validity,* we have taken great care in validating our findings, notably by using well-established tools for computing code coverage, and validating grammar coverage during construction. In addition, our tool chain from raw data to paper is fully automated, avoiding the risk of human error; all data and tools are available for external replication and validation.
- Regarding *construct validity,* the code coverage metrics as well as the linear regression techniques to establish correlations are well-established textbook techniques.

VII. Beyond Text Inputs

The general principle embodied in our approach, namely to systematically search and explore yet uncovered combinations, is not limited to grammars and text inputs alone. Actually, as we will demonstrate in this section, our approach can be used to solve coverage issues in a *large variety of settings.* All it takes is to encode the respective problem as an input language.

```
Configuration → OperatingSystem;
OperatingSystem → "linux-" LinuxDB
      | "windows-" WindowsDB;
LinuxDB → "mysql-" LinuxServer;
WindowsDB → "mssql-" WindowsServer
      | "mysql-" WindowsServer;
LinuxServer → "apache";
WindowsServer → "apache" | "iis";
```

Figure 8: A grammar for configuration testing

A. Configurations

Let us take a look at the following *configuration testing problem.* We have built a Web server application that is supposed to run on a variety of configurations. We do have two *operating systems* **"linux"** and **"windows"**; two *databases* **"mysql"** and **"mssql"**; and two *web servers* **"apache"** and **"iis"**. The **"mssql"** and **"iis"** components require **"windows"** as operating system; **"apache"** and **"mysql"** run on both.

The set of possible configurations can be expressed as a grammar that produces a triple *os–database–server*, for instance **"windows-mysql-apache"**. The grammar in Figure 8 encodes the possible configurations.

In *configuration testing,* the typical problem is that there is a combinatorial explosion of possible configurations. Hence, as always in testing, one wants to determine a good sample of configurations that are to be tested. An obvious minimum is that each component (each OS, each database, each server) should be covered at least once; a reasonable compromise is *pairwise testing,* meaning that every combination of two components should be covered at least once.

If we run the k-path algorithm with $k = 1$ on the grammar in Figure 8, the algorithm ensures that each alternative is taken at least once, resulting in the set **"linux-mysql-apache"**, **"windows-mssql-apache"**, **"windows-mysql-iis"**. This is a minimal set that gives us component coverage.

If we set $k = 2$, we ensure that every *pair* of alternatives is taken at least once. This additionally gets us the configuration **"windows-mssql-iis"**. (Note that the combination of **"mysql"** and **"apache"** is already covered in the **"linux"** variant, above.) These four now cover all pairs, as in pairwise testing. Setting $k = 3$ would get us all *triples,* adding **"windows-mysql-apache"** to the set. In all cases, the k-path algorithm produces a minimal set of productions that satisfies the given coverage goal.

Obviously, there can always be configuration constraints that cannot be expressed by a grammar (if they are decidable at all). But if a grammar does the job, then the k-path algorithm is a good choice for satisfying common configuration testing criteria.

Table I: Average Branch Coverage

Subject	1-path	1-gram	2-path	2-gram	3-path	3-gram	5-path	5-gram
argo [16]	**0.4116**	0.4000	**0.4187**	0.3963	**0.4197**	0.4092	**0.4242**	0.4187
fastjson [17]	0.0364	**0.0376**	**0.0404**	0.0374	**0.0413**	0.0388	**0.0431**	0.0414
genson [18]	0.0842	**0.0866**	**0.0886**	0.0864	**0.0902**	0.0883	**0.0916**	0.0905
gson [19]	0.2080	**0.2215**	**0.2264**	0.2213	**0.2294**	0.2266	0.2352	0.2371
jackson-databind [20]	0.0886	**0.0926**	**0.0932**	0.0924	0.0938	0.0935	0.0940	**0.0952**
json-flattener [21]	0.5039	**0.6246**	**0.6828**	0.6235	**0.7127**	0.6609	**0.7809**	0.7475
json-java [22]	0.1093	**0.1377**	**0.1457**	0.1310	**0.1661**	0.1441	**0.1890**	0.1733
json-simple [23]	0.4427	**0.4695**	**0.4870**	0.4662	**0.4931**	0.4836	**0.5093**	0.5062
json-simple-cliftonlabs [24]	0.3355	0.3325	**0.3445**	0.3301	**0.3446**	0.3410	0.3478	**0.3546**
minimal-json [25]	**0.4158**	0.3970	**0.4267**	0.3936	**0.4163**	0.4054	0.4174	0.4166
pojo [26]	0.1246	**0.1414**	**0.1428**	0.1423	0.1597	0.1433	**0.2112**	0.1462
commons-csv [27]	**0.3828**	0.3773	**0.3903**	0.3772	**0.3984**	0.3770	**0.4034**	0.3799
jackson-dataformat-csv [28]	**0.1665**	0.1527	**0.1666**	0.1523	**0.1700**	0.1532	**0.1777**	0.1573
jcsv [29]	**0.3287**	0.3167	**0.3337**	0.3152	**0.3374**	0.3201	**0.3400**	0.3270
sfm-csv [30]	0.0628	**0.0686**	0.0664	**0.0686**	0.0675	**0.0686**	0.0682	**0.0686**
simplecsv [31]	**0.3472**	0.3377	**0.3482**	0.3368	**0.3481**	0.3395	**0.3489**	0.3439
super-csv [32]	**0.1560**	0.1433	**0.1589**	0.1423	**0.1646**	0.1439	**0.1646**	0.1471
autolink [33]	**0.4514**	0.2861	**0.4673**	0.2861	**0.5716**	0.2859	**0.6265**	0.2889
galimatias [34]	**0.0879**	0.0343	**0.0897**	0.0341	**0.0875**	0.0346	**0.0873**	0.0356
jurl [35]	0.6790	**0.6854**	0.6807	**0.6872**	0.6933	0.6904	**0.7095**	0.7012
url-detector [36]	**0.4057**	0.3273	**0.4083**	0.3244	**0.4188**	0.3342	**0.4352**	0.3458
commonmark [37]	**0.6678**	0.6253	**0.6991**	0.6322	**0.7183**	0.6419	**0.7335**	0.6634
markdown4j [38]	0.6772	**0.6817**	**0.7094**	0.6851	**0.7162**	0.6931	**0.7313**	0.7129
txtmark [39]	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 averaged over 50 runs.
Bold values indicate significantly higher values according to the Mann–Whitney U test [14]. ($p < 0.005$)

Table II: Exception Detection Rates

Subject	Exception	Location	Detection Rate							
			1-path	1-gram	2-path	2-gram	3-path	3-gram	5-path	5-gram
argo [16]	argo.saj.InvalidSyntax	argo...InvalidSyntaxRuntime$3:60	0%	**76%**	0%	**76%**	0%	**84%**	0%	**100%**
genson [18]	java.lang.NullPointer	com...genson.stream.JsonWriter:414	100%	100%	100%	100%	100%	100%	100%	100%
json-flattener [21]	...json.ParseException	com...wnameless...Flattener:122	0%	**76%**	0%	**76%**	0%	**84%**	0%	**100%**
	java.lang.NullPointer	com...wnameless...Unflattener:393	88%	**90%**	**94%**	88%	**100%**	94%	100%	100%
		com...wnameless...Unflattener:409	**4%**	0%	6%	6%	**10%**	4%	22%	**26%**
		location unknown	0%	0%	0%	0%	0%	0%	**2%**	0%
pojo [26]	StringIndexOutOfBounds	org.jsonschema...NameHelper:46	98%	**100%**	100%	100%	100%	100%	100%	100%
commons-csv [27]	java.io.IOException	org.apache.commons.csv.Lexer:281	100%	100%	100%	100%	100%	100%	100%	100%
		org.apache.commons.csv.Lexer:288	100%	100%	100%	100%	100%	100%	100%	100%
jackson-csv [28]	java.io.CharConversion	com.fasterxml...CsvDecoder:429	0%	0%	0%	0%	**2%**	0%	**4%**	0%
		com.faster...ParserBootstrapper:383	0%	0%	0%	0%	**2%**	0%	**4%**	0%
jcsv [29]	java.lang.IllegalState	com.googlecode...TokenizerImpl:73	**100%**	30%	**100%**	22%	**100%**	46%	**100%**	78%
sfm-csv [30]	java.lang.IllegalState	org...$NoColumnCsvWriterDSL:449	100%	100%	100%	100%	100%	100%	100%	100%
super-csv [32]	java.lang.NullPointer	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%
galimatias [34]	java.net.MalformedURL	io.mola.galimatias.URL:527	**100%**	0%	**100%**	0%	**100%**	0%	**100%**	0%
	java.net.URISyntax	io.mola.galimatias.URL:509	**92%**	0%	**94%**	0%	**96%**	0%	**96%**	0%
jurl [35]	StringIndexOutOfBounds	com.anthony...PercentEncoder:176	**100%**	0%	**100%**	0%	**100%**	0%	**100%**	0%
markdown4j [38]	StringIndexOutOfBounds	org...Markdown4jProcessor:53	30%	**100%**	100%	100%	100%	100%	100%	100%
txtmark [39]	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**.

Table III: Grammars and Sizes

Grammar	Rules	Average # of files generated			
		$k = 1$	$k = 2$	$k = 3$	$k = 5$
JSON [8]	17	40	35	58	201
CSV [10]	12	42	38	51	221
URL [10]	27	43	45	72	552
Markdown [11]	236	653	980	1,880	11,409

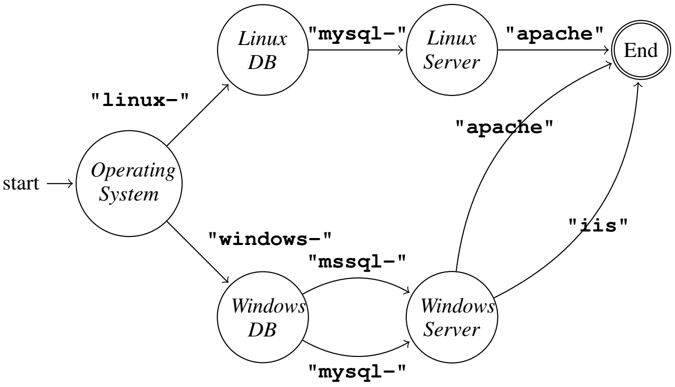

Figure 9: The grammar from Figure 8 as a finite state automaton producing valid configurations

B. Graphical User Interfaces

The astute reader may have noticed that the language in Figure 8 is *regular;* that is, it encodes a *finite-state automaton* in which the states are given by the nonterminal symbols, and the transitions are given by the terminal symbols. Indeed, the grammar is an *embedding* of the finite state automaton shown in Figure 9 which produces the same configurations.

Since context-free languages are a superset of regular languages, *any finite-state model can be encoded into a grammar*—and the k-path algorithm will produce a minimal set that covers states ($k = 1$), transitions ($k = 2$), or sequences of $k - 1$ transitions.

This property is beneficial in any domain where the testing domain would typically be encoded via finite state models. When testing a GUI, one models the application as going through different states of a process, where the transitions between states are triggered by user interactions.

The model in Figure 10, for instance, represents possible interactions in a subset of a shopping site. We assume two actions `click(ui-element)`, which clicks on the given `ui-element`, and `fill(ui-element, text)`, which fills `text` into the text field given by `ui-element`. The finite state automaton thus represents the sequence of possible interactions.

Again, we can embed this automaton into an equivalent grammar, as shown in Figure 11. The advantage of the grammar is that we can easily include productions for text parts such as *Name*, *City*, or *CreditCardNumber*; these could again be arbitrary context-free languages.

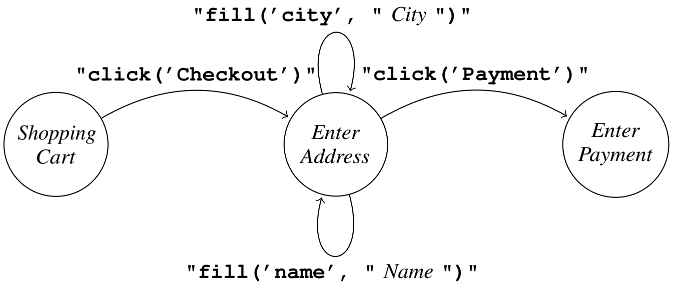

Figure 10: A FSA for a shopping GUI (excerpt)

```
ShoppingCart → "click('Checkout')" EnterAddress;
EnterAddress → "fill('city', '" City "')" EnterAddress
     | "fill('name', '" Name "')" EnterAddress
     | "click('Payment', '" EnterPayment "')";
EnterPayment → "select('Credit Card')"
      "fill('number', '" CreditCardNumber "')" ...;
Name → "Walter White" | "Gretchen Schwartz" | ...;
City → "Albuquerque" | "Amarillo" | ...;
```

Figure 11: A grammar for GUI testing (excerpt)

If we feed this grammar into the k-path algorithm with $k = 1$, the algorithm again produces a minimal set of interactions that cover the alternatives, starting with the single sequence

```
click('Checkout')
fill('city', 'Albuquerque')
fill('name', 'Walter White')
click('Payment')
```

In the next steps, k-path would go and cover more names and cities. As it automatically determines the shortest path to get there, it would again "click" through *Checkout* and fill the next (uncovered) name and city alternatives (**"Gretchen Schwartz"** and **"Amarillo"**). Likewise, if *CreditCardNumber* were defined as a series of digits, k-path would cover one digit after another. And with $k = 2$ or higher, the algorithm covers combinations of all these as well. By modeling GUI exploration as a grammar, and by using the k-path algorithm to systematically cover it, we can thus integrate textual interaction and graphical interaction in a single model, allowing us to test and explore deep user interfaces.

C. Reactive Systems

Interaction with systems need not be limited to user interfaces, but just as well extends to reactive systems. As an example, consider a *mail server* using the SMTP protocol [40], [41]. When sending a mail to a mail server, one sends a series of commands as in

```
HELO relay.example.com
MAIL FROM:<bob@example.com>
RCPT TO:<alice@example.com>
DATA
<mail contents>
.
```

Each of these commands has a specific syntax, and all of these are formally defined as part of the SMTP standard—as a grammar. If one combines the above embedding of states and the grammars for individual commands into one big grammar, the k-path algorithm again would attempt to cover all commands, all elements of commands, and all states; and if $k > 1$, also all sequences thereof—and of course, all of this fully automatically.

VIII. Related Work

A. Fuzzing and Test Generation

tribble is a language-based test generator ("fuzzer"), relying on a *language specification* to produce syntactically valid inputs that are set to cover as many program behaviors as possible. The usage of language models as *producers* was introduced in 1970 by Hanford in his *syntax machine* [42]. Now as then, such producers are mainly used for testing compilers and interpreters: CSmith [43] produces syntactically correct C programs, and LANGFUZZ [1] uses a JavaScript grammar to parse, recombine, and mutate existing inputs while maintaining syntactic validity. Grammar-based whitebox fuzzing [44], combining grammar-based fuzzing with symbolic testing, has saved Microsoft millions of dollars in testing.

None of the above approaches aim for grammar coverage. Our results, however, indicate that any of these approaches could be extended with k-path coverage at low cost, guiding test generation without having to execute the system under test and yielding additional coverage.

B. Grammar Coverage and Production

The concept of grammar coverage was invented in 1972 with Purdom's *sentence generator* [5], which improved over Hanson's syntax machine by systematically covering all production rules. In 2001, Lämmel introduced context-dependent rule coverage for grammars [45]. Given a set of words, this "non-trivial coverage notion" (Lämmel) determines for a rule whether a set of words covers all contexts the rule can occur in. The *Geno* tool by Lämmel and Schulte [46] also gave developers control over which rules should be exhaustively recombined, and to which depth. Our k-path approach generalizes over such context notions and recombinations by introducing context depth as a single parameter. It is inspired by similar metrics both on code and finite-state models. On code, its closest equivalent is the LCSAJ metric, covering all sequences of branches up to a given length [47]. On finite-state models, its equivalent is subsequences of transitions of length k, i.e. all-states, all-transitions, all-transition-pairs, etc. [48].

C. Grammarware: Past and Future

Despite their versatility both as parsers and producers of program inputs, grammars play only a minor role in software testing. In 2005, Klint et al. [49] noted a lack of best practices for "grammarware", that is, grammars and grammar-dependent software, as well as a lack of metrics and other quality notions for testing. In the context of test generation, we find that a multitude of input languages can be easily described using grammars, and that grammar-based testing reaches well into domains not traditionally associated with formal grammars—file formats, network protocols, command languages, and more. Recent advances in grammar inference [50]–[52] suggest that grammars may be inferred from programs or sample inputs, promising to make grammar-based fuzzing easier than ever.

IX. Conclusion

As a formalism for describing the languages of programs and data, grammars have shown their usefulness again and again. This paper introduces a coverage metric that is easy to measure and hence allows for simple assessment of the quality of test cases. Even more important, grammar coverage is easy to achieve, using the definitions and coverage-driven generation algorithm introduced in this paper. Our evaluation shows that both our metric and algorithm improve the state of the art in test generation. If one wants to exercise program features to reveal defects, starting with covering the input features should be a cost-effective starting point.

In our own future work, we will focus on the following:

Code coverage feedback. Besides grammar coverage, *code coverage* may serve as a driver for decisions during test generation. We are extending *tribble* with search-based test generation, where an evolutionary algorithm makes use of the grammar structure to mutate and recombine inputs in an evolving population. The challenge here is how to integrate the two: Should one go for grammar coverage or code coverage first?

Grammar coverage as predictor for code coverage. Our results show that aiming for grammar coverage is helpful for achieving code coverage. One would assume that there would be strong *correlations* between individual language elements and some code in the program under test—notably, the code that processes such elements, but also any code that depends on them. Such correlations could turn grammar coverage into a *predictor* of code coverage.

Advanced formats. By construction, context-free grammars cannot capture situations in which some input fragment depends on context previously processed; examples include definitions and uses of identifiers or other references, checksums and other consistency checks in binary formats, or program states induced by previous inputs. We are working on extending *tribble* with a *constraint language* to express such context-dependent features.

Probabilistic testing. Grammar productions may be associated with *probabilities,* allowing to control how frequently individual productions would take place. We plan to extend *tribble* and its coverage criteria to include such probabilistic testing.

We are committed to reproducible and extensible science. Hence, a replication packages including *tribble* and all evaluation data from this paper is available as open source at

https://github.com/havrikov/covering-input-structure

References

[1] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX, 2012, pp. 445–458. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[2] R. Hodován, Á. Kiss, and T. 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.

[3] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, "Fishing for deep bugs with grammars," in *Proceedings of NDSS 2019*, 2019. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/nautilus-fishing-for-deep-bugs-with-grammars/

[4] J. Wang, B. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *Proceedings of ICSE 2019*, 2019. [Online]. Available: https://2019.icse-conferences.org/event/icse-2019-technical-papers-superion-grammar-aware-greybox-fuzzing

[5] P. Purdom, "A sentence generator for testing parsers," *BIT Numerical Mathematics*, vol. 12, no. 3, pp. 366–375, Sep 1972. [Online]. Available: https://doi.org/10.1007/BF01932308

[6] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[7] T. J. Parr and R. W. Quong, "Antlr: A predicated-ll (k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

[8] T. Bray, "The JavaScript Object Notation (JSON) data interchange format," Internet Requests for Comments, RFC Editor, STD 90, December 2017.

[9] "Url grammar," https://github.com/antlr/grammars-v4/blob/master/url/url.g4, 2018.

[10] "Csv grammar," https://github.com/antlr/grammars-v4/tree/master/csv, 2018.

[11] J. MacFarlane, "Markdown-peg grammar," https://github.com/jgm/peg-markdown/blob/master/markdown_parser.leg, 2013.

[12] "Fuzzinator configs," https://github.com/renatahodovan/fuzzinator-configs, 2019.

[13] M. R. Hoffmann, "JaCoCo, version 0.8.2," https://github.com/jacoco/jacoco, 2018.

[14] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60, 1947. [Online]. Available: http://www.jstor.org/stable/2236101

[15] E. Jones, T. Oliphant, and P. Peterson, "{SciPy}: Open source scientific tools for {Python}," 2014.

[16] "Argo," https://sourceforge.net/projects/argo/, 2018, version 5.4.

[17] "fastjson," https://github.com/alibaba/fastjson, 2018, version 1.2.51.

[18] "Genson," https://github.com/owlike/genson, 2017, version 1.4.

[19] "Gson," https://github.com/google/gson, 2017, version 2.8.5.

[20] "Jackson-databind," https://github.com/FasterXML/jackson-databind, 2018, version 2.9.8.

[21] "json-flattener," https://github.com/wnameless/json-flattener, 2018, version 0.6.0.

[22] "Json-java," https://github.com/stleary/JSON-java, 2017, version 20180813.

[23] "json-simple," https://github.com/fangyidong/json-simple, 2014, version 1.1.1.

[24] "json-simple by Cliftonlabs," https://github.com/cliftonlabs/json-simple, 2018, version 3.0.2.

[25] R. Sternberg, "minimal-json," https://github.com/ralfstx/minimal-json, 2017, version 0.9.5.

[26] https://github.com/joelittlejohn/jsonschema2pojo, 2017, version 1.0.0.

[27] https://commons.apache.org/proper/commons-csv/, 2018, version 1.6.

[28] "Jackson Dataformat CSV," https://github.com/FasterXML/jackson-dataformats-text/tree/master/csv, 2018, version 2.9.8.

[29] "jcsv," https://code.google.com/archive/p/jcsv, 2012, version 1.4.0.

[30] "Simple Flat Mapper," https://github.com/arnaudroger/SimpleFlatMapper, 2018, version 6.1.1.

[31] "simplecsv," https://github.com/quux00/simplecsv, 2018, version 2.1.

[32] "super-csv," https://github.com/super-csv/super-csv, 2016, version 2.4.0.

[33] "autolink-java," https://github.com/robinst/autolink-java, 2018, version 0.9.0.

[34] "galimatias," https://github.com/smola/galimatias, 2018, version 0.2.1.

[35] A. Simon, "jurl," https://github.com/anthonynsimon/jurl, 2018, version v0.3.0.

[36] "Url Detector," https://github.com/linkedin/URL-Detector, 2018, version 0.1.17.

[37] "Atlassian Commonmark," https://github.com/atlassian/commonmark-java, 2017, version 0.11.0.

[38] "Markdown4J," https://github.com/jdcasey/markdown4j, 2016, version 2.2-cj-1.1.

[39] "Txtmark," https://github.com/rjeschke/txtmark, 2017, version 0.13.

[40] "Simple Mail Transfer Protocol," RFC 821, Aug. 1982. [Online]. Available: https://rfc-editor.org/rfc/rfc821.txt

[41] D. J. C. Klensin, "Simple Mail Transfer Protocol," RFC 5321, Oct. 2008. [Online]. Available: https://rfc-editor.org/rfc/rfc5321.txt

[42] K. V. Hanford, "Automatic generation of test cases," *IBM Systems Journal*, vol. 9, no. 4, pp. 242–257, 1970.

[43] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 283–294. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993532

[44] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. New York, NY, USA: ACM, 2008, pp. 206–215. [Online]. Available: http://doi.acm.org/10.1145/1375581.1375607

[45] R. Lämmel, "Grammar testing," in *Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering*, ser. FASE '01. London, UK, UK: Springer-Verlag, 2001, pp. 201–216. [Online]. Available: http://dl.acm.org/citation.cfm?id=645369.651271

[46] R. Lämmel and W. Schulte, "Controllable combinatorial coverage in grammar-based testing," in *Testing of Communicating Systems*, M. Ü. Uyar, A. Y. Duale, and M. A. Fecko, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 19–38.

[47] M. A. Hennell, M. R. Woodward, and D. Hedley, "On program analysis," *Information Processing Letters*, vol. 5, no. 5, pp. 136–140, 1976.

[48] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Software Testing, Verification and Reliability*, vol. 22, no. 5, pp. 297–312, 2012.

[49] P. Klint, R. Lämmel, and C. Verhoef, "Toward an engineering discipline for GRAMMARWARE," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 14, no. 3, pp. 331–380, 2005.

[50] M. Höschele and A. Zeller, "Mining input grammars from dynamic taints," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: ACM, 2016, pp. 720–725. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970321

[51] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing program input grammars," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: ACM, 2017, pp. 95–110. [Online]. Available: http://doi.acm.org/10.1145/3062341.3062349

[52] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. Piscataway, NJ, USA: IEEE Press, 2017, pp. 50–59. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155573