Static Test Case Prioritization Strategies for Grammar-Based Testing

Moeketsi Raselimo*†, Lars Grunske*, Bernd Fischer†
*Humboldt-Universität zu Berlin, Germany
{raselimm, grunske}@informatik.hu-berlin.de
†Stellenbosch University, South Africa
bfischer@sun.ac.za

***Abstract*—Grammar-based test case generators can quickly produce large test suites that structurally and systematically cover the input space of a system under test. However, running these large test suites requires large computation resources. In this paper, we investigate whether simple test case prioritization strategies based on statically determined grammar-related properties (e.g., token-length of input, novel rule coverage, and relative frequency of rule coverage), can detect faults in the system under test faster. Our preliminary results indicate that different test execution orderings from these strategies have an effect on the fault detection rate. Their performance varies across the different test suites, but they generally perform better than a simple random ordering of test executions.**

I. Introduction

Context-free grammars are widely used to describe the structure of objects in complex domains, such as internal and external file formats (e.g., pdf readers), generic interchange formats (e.g., XML, JSON, etc.), language specifications which are usually inputs to compiler-compiler tools, and of course programming languages. Grammar-based testing exploits the nature of grammars to automatically derive inputs to exercise a system under test. However, the simplicity and speed with which these test inputs can be generated often leads to very large test suites whose execution requires substantial resources; for example, grammar-based fuzzers routinely generate millions of test inputs [1].

Test case prioritization (TCP) strategies try to reduce the resource requirements by *reordering* the test suites such that the tests most likely to detect faults are executed first [2, 3, 4, 5, 6, 7, 8, 9]. TCP is typically applied in regression testing, using information collected from the first execution round (e.g., coverage, fault detection), and information about the changes in the SUT. Most of the TCP techniques are indeed coverage-driven and require white- or grey-box access to the SUT, with only a few black-box techniques (such as model-based strategies) proposed [10, 11, 12, 13]. Since most grammar-based testing methods only use the model that captures the SUT's input space, without much knowledge of its internal source code, it is not clear how ideas from TCP techniques transfer to the domain of grammar-based testing.

In this paper, we describe and compare static test case prioritization strategies specifically designed for grammar-based testing. Our strategies use grammar-related properties of the tests that can easily be collected during the derivation process, such as their length in tokens, the set of rules used in their derivation, or the relative frequency of these rules.

We evaluate the efficacy of our test prioritization strategies using test suites derived systematically to satisfy certain grammar coverage goals and randomized derivations. We preliminarily evaluate these strategies on parsers derived from grammars with seeded faults. Our experimental results indicate that the baseline strategies, random ordering and test size, perform the worst on a systematic test suite, while the novel rule coverage finds the faults the fastest, and the rule frequency based strategies find the last fault quickly than novel rule coverage. For longer and deeper randomly generated tests, however, results become less discriminatory because of the smaller size of our SUTs. These random derivations seem to favour test size, as longer test executions uncover more faults earlier, but the random ordering still performs poorly.

The grammar-based oriented TCP strategies offer several benefits. First, they can speed up fault detection; this is obviously useful to correct SUT faults faster, but can also be useful when the SUT's applied input grammar is incorrect or incomplete, i.e., in model hardening. Second, they can be used to reduce the fuzzing load—grammar-based fuzzers can utilize these strategies to generate diverse inputs. Third, our results show that the strategies approach the maximum number of faults detectable faster than random orders. This can be exploited to develop up-front test suite reduction strategies.

The rest of the paper is organized as follows, Section II provides background material and discusses how this paper relates to existing studies. We describe domain specific test prioritization strategies in Section III. We provide preliminary evaluation in Section IV, we then conclude, and provide plans for the extension of this work.

II. Background and Related Work

In this Section, we provide some background material necessary to understand the work, and discuss related work drawn from existing empirical studies.

***Grammar Notation*.** A context-free grammar (or simply grammar, CFG) is a four-tuple $G = (N, T, P, S)$, where N, T, P, are disjoint sets of non-terminals, terminals, and grammar productions, respectively, and and $S \in N$ denotes the start symbol. In examples, we use $italics$ and

bold typewriter font for non-terminal and terminal symbols, respectively; we use `normal typewriter` font for structured tokens with different instances such as identifiers.

Derivations. We use $\alpha A\beta \Rightarrow \alpha\gamma\beta$ to denote that $\alpha A\beta$ *produces* $\alpha\gamma\beta$ by application of the rule $A \rightarrow \gamma$ and use $\Rightarrow^*$ for its reflexive-transitive closure. We call a phrase α a *sentential form* if $S \Rightarrow^* \alpha$. The *yield* of α is the set of all words that can be derived from it, i.e., $\text{yield}(\alpha) = \{w \in T^* \mid \alpha \Rightarrow^* w\}$. The *language* $L(G)$ *generated by a grammar* G is the yield of its start symbol, i.e., $L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$. $w \in L(G)$ is also called a *sentence*.

Test Suites. A *test suite* comprises a set of test inputs for a system under test (SUT) and their corresponding outcome. A test *passes* if it produces the expected output, and *fails* otherwise.

In grammar-based testing, a test suite TS can comprise both positive tests $TS^+ \in L$ and negative tests $TS^- \notin L$, with the corresponding expected outputs *accept* and *reject*, but for simplicity we will work only with positive tests.

Grammar-based Test Suite Construction. In grammar-based testing, test inputs are typically constructed by deriving words in the grammar's language (although there are also approaches to systematically construct negative tests that are not in the language [14, 15]). There are two principal approaches to deriving words, *coverage-based* and *randomized*.

Coverage-based approaches construct a set of derivations that together satisfy a structurally defined criterion; for example, rule coverage ensures that every rule in P is used in the derivation of at least one test in the test suite (see Figure 1 for an example). More complex criteria (e.g., context-dependant rule coverage [16], which takes into account the different contexts in which a rule can be applied, or derivability coverage [17], which ensures that paths between all connected pairs of symbols are covered) induce larger test suites with more complex tests. For our evaluation, we use context-dependant derivability coverage (*cderiv*), which ensures that the derivability paths are started in a context corresponding to each occurrence of a non-terminal symbol on the right-hand side of each rule. We use the generic cover algorithm [18] to derive the test suites. This algorithm constructs a separate derivation for each coverage goal, leading to larger test suites of smaller tests than related algorithms that try to satisfy multiple coverage goals at the same time [19, 20], but all coverage-based approaches induce some level of bias because they necessarily need to use some rules more often than others (see Table I).

Several other algorithms exist that yield sufficiently detailed test suites that strike the right balance between syntactic regularity and variation, e.g., *k-path* coverage [20] or automata-based methods [21, 22].

Randomized approaches iteratively construct derivations, by randomly choosing a non-terminal occurrence in the sentential form and expanding it with a randomly chosen rule [23]. Different implementations use different strategies for choosing non-terminal occurrences and rules, and for completing the sentential form into a word, in order to ensure termination. Our implementation follows a breadth-first strategy, with the depth k as a control parameter; when k is reached, it replaces the unexpanded non-terminals in the sentential form with their minimal yield.

Grammar-based fuzzers (e.g., LangFuzz [24] and IFuzzer [25]) mostly use random sentence generation techniques, and often exploit a given corpus to extract seed code fragments [24, 25, 26]. Nautilus [27] exploits grey-box access to the SUT to provide feedback to the sentence generation.

Test Case Selection and Prioritization. Test prioritization strategies assign priorities to test cases in a test suite to determine the order of execution of each test case; high priority tests get preference. These strategies employ different criteria (e.g., coverage, time, cost, etc) to ensure orderings that achieve certain requirements. See [28, 29, 30] for detailed overviews.

Test case selection and prioritization techniques are usually applied during regression testing [31], which is a quality assurance activity that provides confidence that code changes and the resulting evolution steps do not break the current functionality of the system. The goal of regression testing is to uncover behavioural changes from one software version to the next. These changes are known as regressions. Since test suites grow during the evolution of software systems, large projects cannot re-run the complete test suite for every system build. As a result, the goal of a search process is to minimise the test suite [32, 9], select the most suitable test cases [33], or to prioritize the test cases [2, 3, 4, 5, 6, 7, 8, 9] with the aim of having higher chances to uncover regression errors in a limited testing time budget [31].

As used in other empirical studies [4, 34] , we evaluate the efficiency of our test case prioritization strategies using the standard *average percentage of fault detected* (APFD). Higher APFD indicates better fault detection rate. It is typically computed as

$$APFD = 1 - \frac{\sum_{i=1}^{m} F_i(w)}{m \times |TS|} + \frac{1}{2 \times |TS|}$$

where $F_i(w)$ is the function that returns the index of the first test case $w \in TS$ that detects fault i, m is the total number of faults detectable under TS, and $|TS|$ denotes the size of the test suite TS.

We also evaluate the effectiveness of of each strategy, by computing the number T_{max} of tests in TS it takes to find all detectable faults.

III. Test Case Prioritization Strategies

In this section, we propose four different grammar-based test case prioritization strategies. We use the grammar and test suite shown in Figure 1 to illustrate these strategies. We consider each top-level alternative as a separate rule, so that rule 4 refers to $\textit{block} \rightarrow$ **{** *stmts* **}** and not to $\textit{decl} \rightarrow$ **var** `id` **:** *type*. Table I shows the size of each of the *rule* tests, as well as the actually used rules.

Random Ordering. As baseline for our evaluation, we use a simple random ordering of the the test cases.

$$\begin{array}{lll}
\textit{prog} & \rightarrow & \textbf{program}\ \texttt{id}\ \texttt{=}\ \textit{block}\ \texttt{.} \\
\textit{block} & \rightarrow & \texttt{\{}\ \textit{decls stmts}\ \texttt{\}} \mid \texttt{\{}\ \textit{decls}\ \texttt{\}} \mid \texttt{\{}\ \textit{stmts}\ \texttt{\}} \mid \texttt{\{\}} \\
\textit{decls} & \rightarrow & \textit{decl}\ \texttt{;}\ \textit{decls} \mid \textit{decl}\ \texttt{;} \\
\textit{decl} & \rightarrow & \textbf{var}\ \texttt{id}\ \texttt{:}\ \textit{type} \\
\textit{type} & \rightarrow & \textbf{bool} \mid \textbf{int} \\
\textit{stmts} & \rightarrow & \textit{stmt}\ \texttt{;}\ \textit{stmts} \mid \textit{stmt}\ \texttt{;} \\
\textit{stmt} & \rightarrow & \textbf{sleep} \\
 & \mid & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt} \\
 & \mid & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt}\ \textbf{else}\ \textit{stmt} \\
 & \mid & \textbf{while}\ \textit{expr}\ \textbf{do}\ \textit{stmt} \\
 & \mid & \texttt{id}\ \texttt{=}\ \textit{expr} \\
 & \mid & \textit{block} \\
\textit{expr} & \rightarrow & \textit{expr}\ \texttt{=}\ \textit{expr} \mid \textit{expr}\ \texttt{+}\ \textit{expr} \mid \texttt{(}\ \textit{expr}\ \texttt{)} \mid \texttt{id} \mid \texttt{num}
\end{array}$$

```
1  program x = {x = (x);}.
2  program x = {x = x + x;}.
3  program x = {x = x;}.
4  program x = {x = x = x;}.
5  program x = {x = 0;}.
6  program x = {if x then sleep;}.
7  program x = {if x then sleep else sleep;}.
8  program x = {sleep; sleep;}.
9  program x = {sleep;}.
10 program x = {var x:bool; sleep;}.
11 program x = {var x:bool; var x bool;}.
12 program x = {var x:bool;}.
13 program x = {var x:int;}.
14 program x = {while x do sleep;}.
15 program x = {{};}.
16 program x = {}.
```

Fig. 1: An example grammar G_{Toy} (top) and its corresponding *rule* coverage test suite (bottom).

test	size	used rules
1	12	{1, 4, 12, 16, 21, 22}
2	12	{1, 4, 12, 16, 20, 22}
3	10	{1, 4, 12, 16, 22}
4	12	{1, 4, 12, 16, 19, 22}
5	10	{1, 4, 12, 16, 23}
6	11	{1, 4, 12, 13, 17, 22}
7	13	{1, 4, 12, 14, 17, 22}
8	10	{1, 4, 11, 12, 17}
9	8	{1, 4, 12, 17}
10	13	{1, 2, 7, 8, 9, 12, 17}
11	16	{1, 3, 6, 7, 8, 9}
12	11	{1, 3, 7, 8, 9}
13	11	{1, 3, 7, 8, 10}
14	11	{1, 4, 12, 15, 17, 22}
15	9	{1, 4, 5, 12, 18}
16	6	{1, 5}

TABLE I: Derivation data for *rule* test suite shown in Figure 1. Size is measured in tokens.

Test Size. Smaller test cases (e.g., #16, #15, and #9) are better for fault localization because they typically execute a smaller part of the SUT, but on the other hand, larger test cases (#7, #10, and #11) are better for fault detection, because they (presumably) execute a larger part of the SUT. We therefore order the test suite in decreasing size of the test cases, measured in the number of tokens, and resolve ties randomly.

Rule Novelty. The central tenet of grammar-based testing is that test cases derived using different combinations of rules exercise different parts of the SUT. We can try to maximize this diversity by prioritizing test cases that use rules not yet covered by test cases executed earlier. This process is started with one of the test cases using the highest number of rules.

In our example, we first pick test case #10, which uses the rules 1, 2, 7, 8, 9, 12, and 17, although it was originally derived only to target rule 2; note also that this is not the largest test measured by the number of tokens. Test cases #1, #2, and #4 then all use four yet uncovered rules; we randomly pick #1, which covers rules 4, 16, 21, and 22. Test cases #11 and #15 then cover rules 3 and 6 resp. 5 and 18. Finally, test cases #2, #4, #5, #6, #7, #8, #13, and #14 each covers one of the remaining rules. The remaining four test cases (#3, #9, #12, and #16) do not use any uncovered rules; we could therefore drop them to achieve a test suite reduction, or simply execute them in any random order.

Rule Rarity. We can exploit the central tenet of grammar-based testing in a different way, by prioritizing test cases that use "rare" rules, i.e., rules that are only used in the derivations of very few test cases. In the extreme, we should execute test cases first that use a rule not used by any other test case—if the SUT contains an error related to such a rule, those test cases are the only ones that can possibly identify it.

Given a test case $w \in TS$ with used rules R_w we define the *rule occurrence vector* $\mathbf{r}(w) \in \{0,1\}^{|P|}$ such that $\mathbf{r}_i(w) = 1$ iff $p_i \in R_w$. Each component $\mathbf{r}_i$ thus that indicates whether p_i is used in the derivation of w or not. From the rule occurrence vectors we define the *rule probability vector* $\mathbf{p}(TS)$ for a test suite TS such that $\mathbf{p} = (\Sigma_{w \in TS}\mathbf{r}(w)) \,/\, |TS|$ (where Σ is repeated vector addition). Each component $\mathbf{p}_i$ thus describes the fraction of test cases in which the rule p_i has been used during the construction of TS. In our example, we get $\mathbf{p}_1 = 1.0$ (because every derivation starts with the rule $\textit{prog} \rightarrow \textbf{program}\ \texttt{id = } \textit{block}\ \texttt{.}$) but $\mathbf{p}_2 = 0.0625$ (because only test case #10 contains both declarations and statements.

We can then define two different TCP strategies.

Rarest rule: We rearrange the test suite in ascending order of the lowest probability of the rules used in the derivation of each test case, i.e., we prioritize tests that use the rarest rules. We can formalize this order by computing for each test case w the score

$$\|(\mathbf{1} - \mathbf{p}(w)) \odot \mathbf{r}(w)\|_\infty$$

where the Hadamard product $\mathbf{z} = \mathbf{x} \odot \mathbf{y}$ is the component-wise multiplication of $\mathbf{x}$ and $\mathbf{y}$, i.e., $\mathbf{z}_i = \mathbf{x}_i\,\mathbf{y}_i$, and the max norm $\|\mathbf{x}\|_\infty$ is (the absolute value of) the largest component of a vector $\mathbf{x}$. Note that we are using $\mathbf{1} - \mathbf{p}(w)$ because we want to maximize rarity, and that the Hadamard product with $\mathbf{r}(w)$ "projects out" rules that are not used by w.

In our example, 12 out of the 16 test cases exclusively use one rule each; this is not surprising because the test suite has been constructed to satisfy rule coverage with the smallest possible test cases, but it means that this strategy can degenerate into a version of random ordering that is biased against large tests.

TS	$\|TS\|$	$\#killed$	$\overline{\|w\|}$	s^2	$\overline{\|R\|}$	s^2
cderiv	792	4763	13.15	4.44	21.9	3.61
$random_{12}$	1000	4965	63.45	44.82	39.10	17.10
$random_{20}$	1000	4965	145.25	146.49	46.44	22.27

TABLE II: Characteristics of test suites used in our evaluation. $|TS|$ is the size of each individual test suite. Note that for random tests suites, we generate five instances of 1000 tests each. $\#killed$ denotes the total number of faults found by each test suite. $\overline{|w|}$ and $\overline{|R|}$ denote the average number of tokens and rules used, respectively, and s^2 are the respective sample variances.

Rarest average rule: We can alternatively rearrange the test suite by the average rarity of the rules used by each test case. We can achieve this by changing from the max norm to the standard Euclidean norm $\|\cdot\|_2$, and re-normalizing this by the norm of the occurrence vector, to prevent bias towards tests that use only few rules. Hence, we compute for each test case w the score

$$\|(\mathbf{1} - \mathbf{p}(w)) \odot \mathbf{r}(w)\|_2 \,/\, \|\mathbf{r}(w)\|_2$$

IV. Evaluation

Experimental Setup. For our experiments, we used a grammar for a small language called SIMPL as our base grammar. SIMPL is used in the second year computer architecture course at Stellenbosch University to introduce students to low level programming concepts such as memory management, intermediate code representation, code generation, etc. This base grammar has 41 non-terminals, 43 terminals, and 83 rules or productions.

We then derived an equivalent CUP grammar from the base grammar. CUP is a popular parser generator that implements the LALR parsing algorithm and emits parsers written in Java. We mutated the CUP grammar by applying simple string edit operations (i.e., deletion, insertion, and substitution) on the right-hand side of every production. This gave us a total of 33300 grammars from which we randomly selected 5000. We then generated parsers for each of the selected mutated grammars.

We executed each parser derived from mutated grammars over test suites generated from the base grammar. The first test suite *cderiv* (see Section II for details) contains 792 positive test cases. The last two sets of test suites are randomly generated with depth 12 and 20. We denote these by $random_{12}$ and $random_{20}$. The maximum acyclic derivation length in our base grammar is 12, so the $random_{12}$ test suite ensures that every grammar symbol $X \in V$ can occur in the derivation of a test case. We also wanted to see the effect of longer and deeper derivations with $random_{20}$. Note that, to account for random effects, we generate five sets for each random test suite of the size of 1000 each. Table II contains more characteristics of the test suites used to evaluate our test case ordering approaches.

Although some strategies such as *size*, *rarest rule*, and *rarest rule average* are deterministic by construction, they are not from free from tie manifestations. We resolve these ties randomly, and therefore run five repetitions of each test ordering strategy over our subject parsers.

We measure the rate at which test executions ordered by different strategies find faults, i.e., kill the mutants in our subject parsers. A good strategy finds all (detectable) faults faster than a random ordering. However, in our evaluation, not all faults are detectable, i.e., not all mutants are killed by the test suites, largely because of weaknesses in the test suites, equivalent mutations, or execution time-outs on unstable parsers whose underlying mutations introduced parsing conflicts.

Experimental Results. Figures 2 to 4 summarize our results in a series of plots. Each plot shows the rate at which faults are found by the different TCP strategies, given a specific test suite. The top resp. bottom curve in each plot shows the highest resp. lowest number of faults found after i test executions across five repetition runs, while the dark blue curve in the middle shows the average number of faults found in these five runs. In each figure, the bottom right plot shows the comparison of five test selection strategies using the average number of faults found. Table III contains additional details.

While the details change with different test suites and strategies, the plots show overall positive results. First we observe that, for the shorter systematically generated tests, the *rarest rule* strategy has the highest APFD ($\sim$87%) and achieves the maximum kill rate much quicker than all other strategies, i.e., with this strategy, we only need to execute about 14% of tests to find all faults. The *rule novelty* strategy finds most faults faster than *random* but struggles to find the last faults, and barely outperforms *random* in terms of AFPD. *size* and *rarest rule average* show a worse AFPD performance than *random*, although *size* finds the last fault faster.

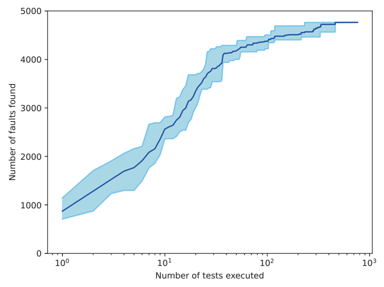
Second, for the longer and and more complex derivations in both $random_{12}$ and $random_{20}$ test suites, the results become better (both in terms of AFPD and number of tests required to kill all mutants) but also less discriminatory, as all the strategies find all killable mutants within fewer than 10% of the total test executions. This is possibly a consequence of the (small) size of our base grammar. Running the largest tests first (i.e., *size*) finds (the presence of) faults fastest, provided the tests are large enough, but unfortunately this does not help with finding the location of these faults.

Third, we see that the *random* strategy has the largest variance across the board, while the *rarest average rule* strategy shows less variance in both $random_{12}$ and $random_{20}$.

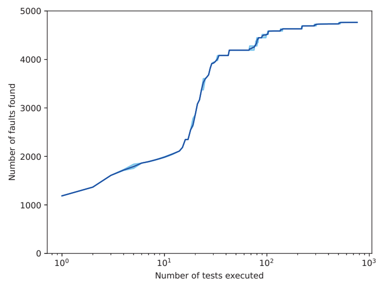
Overall we see that the *random* and (in some cases) *rarest average rule* strategies perform relatively poorly, that the *rarest rule* strategy outperforms all other strategies in terms of APFD and T_{max} scores, that the *rule novelty* strategy by and large finds faults the fastest, and that *size* is effective only for complex test cases.

V. Threats to validity

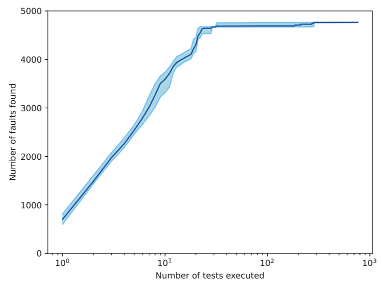
The main threats to validity can be attributed to our evaluation, In particular, we only considered SUTs derived from a single grammar that describes a small teaching language, and



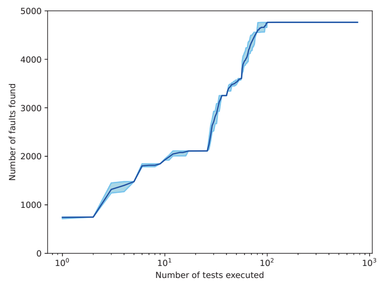
(a) Random.



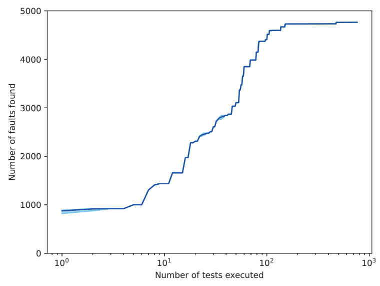
(b) Size.



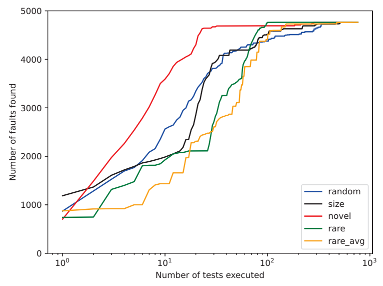
(c) Rule novelty.



(d) Rarest.

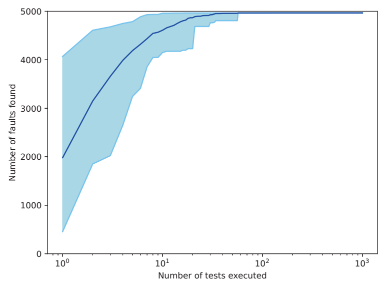


(e) Rarest average.

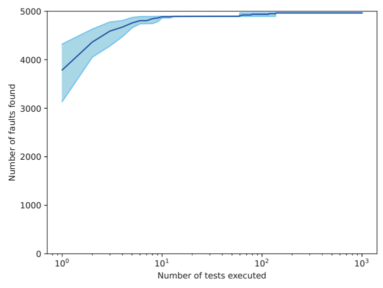


(f) Summary.

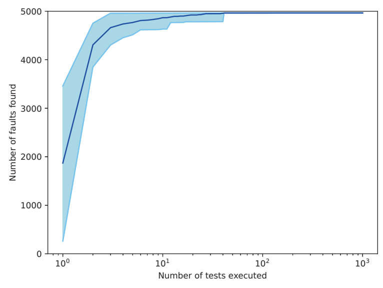
Fig. 2: A summary of results of different test case ordering over *cderiv*.



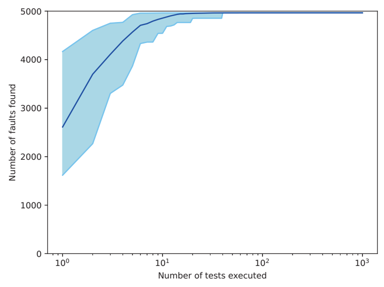
(a) Random.



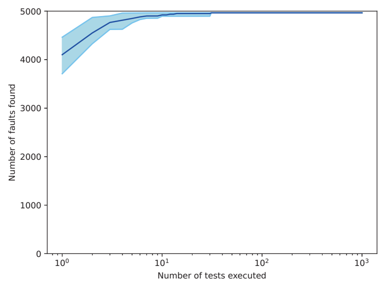
(b) Size.



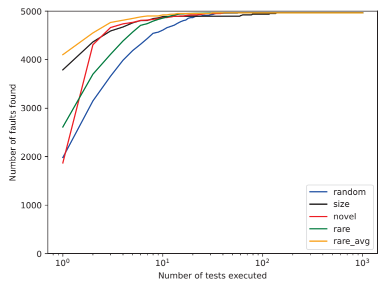
(c) Rule novelty.



(d) Rarest.

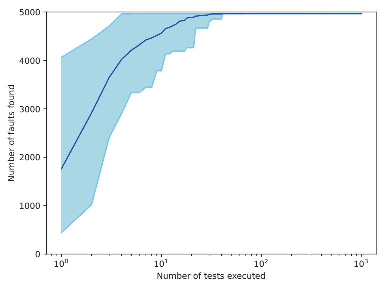


(e) Rarest average.

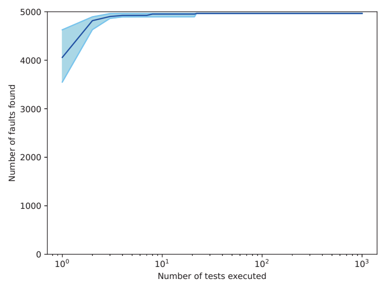


(f) Summary.

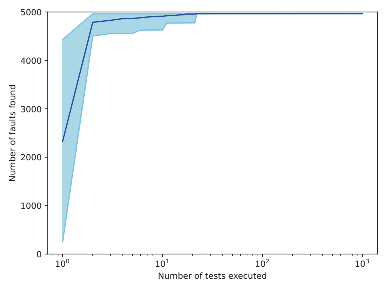
Fig. 3: A summary of results of different test orderings over five instances of random tests at k=12.



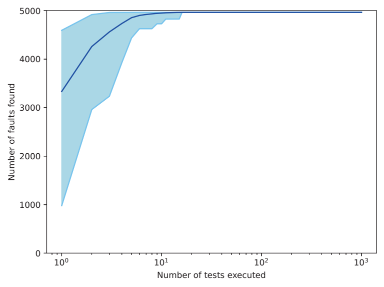
(a) Random.



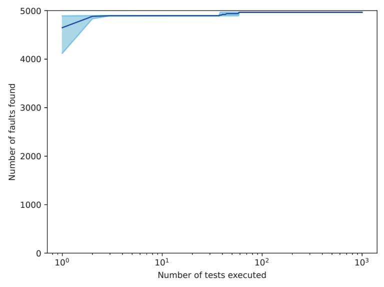
(b) Size.



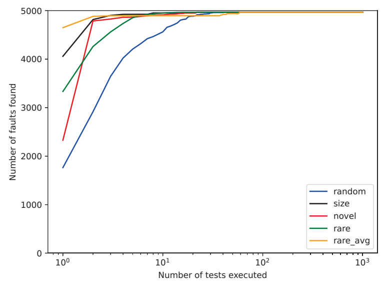
(c) Rule novelty.



(d) Rarest.



(e) Rarest average.



(f) Summary.

Fig. 4: A summary of results of different test orderings over five instances of random tests at k=20.

	cderiv		*random*$_{12}$		*random*$_{20}$	
	T_{max}	*APFD*	T_{max}	*APFD*	T_{max}	*APFD*
random	598.80	63.84	61.20	92.69	28.54	96.55
size	554.17	59.00	90.00	89.61	8.06	99.05
rule novelty	630.20	65.82	52.44	93.80	19.68	97.63
rarest	115.83	86.97	38.50	95.37	9.58	98.86
rarest average	674.00	59.09	14.40	98.27	46.68	94.44

TABLE III: Effectiveness results of each strategy under different test suites using APFD and T_{max}. T_{max} denotes the average number of tests executed until all killable mutants by different test suites are killed by different test case selection strategies.

we only considered injected faults. Our results may therefore not generalize to other SUTs. However, note that fault seeding is widely used in software testing research because it produces many faulty subjects; it is particularly useful when benchmarks with real faults are unavailable. Whether mutants are a good substitute for real faults remains unclear in different testing scenarios, but experiments show a positive correlation between mutant detection and real fault detection [35]. Furthermore, mutation testing has a long history in software testing, specifically, test prioritization studies evaluate their strategies using mutated program versions from the popular Siemens Suite benchmark.

The mutations used in our evaluation are generated by mutating production rules of the base grammar, however, our conclusions should in principle also hold for code-level mutations of the parser derived from the base grammar.

Our evaluation judgements on the effectiveness of our proposed strategies are based on APFD and the number of test executions required to find the last fault. Other studies, e.g., [29], propose coverage-based metrics to evaluate prioritization strategies. We, however, leave their consideration and comparison of our own prioritization strategies to different existing ones for future work.

We mitigated against the usual human-induced threats by carefully testing our scripts and using well-established tools for generating test suites.

VI. Conclusions and Future Work

Conclusions. In this paper, we described and compared five static test case selection strategies specifically designed for grammar-based testing. Our preliminary results show that the cheap random ordering performs worse than the other four under different test suites. The other baseline strategy which is based on the size of the test suite only ever works well if the underlying test suite contains long tests. The strategy based on rule novelty runs stable across different scenarios; it finds faults much faster across all three test suites, but struggles to find the last fault. Overall, however, we achieved the best results with the *rarest rule* strategy.

Future Work. In addition to conducting more experimental evaluation under different fault models and test suites, we see further interesting directions for future work. First, we plan to design coverage-based strategies to compare effect on SUT rather than mutant kill rates. Second, our results from the *rule novelty* based strategy are promising, we plan to build on this idea and develop and evaluate a much richer strategy based on *novel rule pairs*; considering rule pairs will improve on the short-comings of rule novelty. Finally, we plan to transfer ideas from Ledru et al. [34] to the grammar-based context, and pick the test next that has the biggest string edit distance from those run so far. We plan to lift their character-based strategy to the grammar's tokens.

References

[1] 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.

[2] H. Do, G. Rothermel, and A. Kinneer, "Prioritizing junit test cases: An empirical assessment and cost-benefits analysis," *Empirical Software Engineering*, vol. 11, no. 1, pp. 33–70, 2006.

[3] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel, "The effects of time constraints on test case prioritization: A series of controlled experiments," *IEEE Trans. Software Eng.*, vol. 36, no. 5, pp. 593–617, 2010.

[4] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Trans. Software Eng.*, vol. 28, no. 2, pp. 159–182, 2002.

[5] B. Korel, G. Koutsogiannakis, and L. H. Tahat, "Model-based test prioritization heuristic methods and their evaluation," in *3rd Workshop on Advances in Model Based Testing, A-MOST 2007*. ACM, 2007, pp. 34–43.

[6] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Trans. Software Eng.*, vol. 33, no. 4, pp. 225–237, 2007.

[7] Y. Lu, Y. Lou, S. Cheng, L. Zhang, D. Hao, Y. Zhou, and L. Zhang, "How does regression test prioritization perform in real-world software evolution?" in *Int. Conf. on Software Engineering*, 2016.

[8] A. Marchetto, M. Islam, W. Asghar, A. Susi, and G. Scanniello, "A multi-objective technique to prioritize test cases," *IEEE Trans. Software Eng.*, vol. online first, no. accepted, 2016.

[9] A. Shi, T. Yung, A. Gyori, and D. Marinov, "Comparing and combining test-suite reduction and regression test selection," in *Foundations of Software Engineering, ES-EC/FSE 2015*. ACM, 2015, pp. 237–247.

[10] B. Korel, L. H. Tahat, and M. Harman, "Test prioritization using system models," in *21st IEEE International Conference on Software Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary*. IEEE Computer Society, 2005, pp. 559–568. [Online]. Available: https://doi.org/10.1109/ICSM.2005.87

[11] L. H. Tahat, B. Korel, M. Harman, and H. Ural, "Regression test suite prioritization using system models," *Softw. Test. Verification Reliab.*, vol. 22, no. 7, pp. 481–506, 2012. [Online]. Available: https://doi.org/10.1002/stvr.461

[12] B. Korel and G. Koutsogiannakis, "Experimental comparison of code-based and model-based test prioritization," in *Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1-4, 2009, Workshops Proceedings*. IEEE Computer Society, 2009, pp. 77–84. [Online]. Available: https://doi.org/10.1109/ICSTW.2009.45

[13] L. Tahat, B. Korel, G. Koutsogiannakis, and N. Almasri, "State-based models in regression test suite prioritization," *Softw. Qual. J.*, vol. 25, no. 3, pp. 703–742, 2017. [Online]. Available: https://doi.org/10.1007/s11219-016-9330-x

[14] M. Raselimo, J. Taljaard, and B. Fischer, "Breaking parsers: mutation-based generation of programs with guaranteed syntax errors," in *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2019, Athens, Greece, October 20-22, 2019*, O. Nierstrasz, J. Gray, and B. C. d. S. Oliveira, Eds. ACM, 2019, pp. 83–87. [Online]. Available: https://doi.org/10.1145/3357766.3359542

[15] S. V. Zelenov and S. A. Zelenova, "Automated generation of positive and negative tests for parsers," in *Formal Approaches to Software Testing, 5th International Workshop, FATES 2005, Edinburgh, UK, July 11, 2005, Revised Selected Papers*, ser. Lecture Notes in Computer Science, W. Grieskamp and C. Weise, Eds., vol. 3997. Springer, 2005, pp. 187–202. [Online]. Available: https://doi.org/10.1007/11759744_13

[16] R. Lämmel, "Grammar testing," in *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, ser. Lecture Notes in Computer Science, H. Hußmann, Ed., vol. 2029. Springer, 2001, pp. 201–216. [Online]. Available: https://doi.org/10.1007/3-540-45314-8_15

[17] P. van Heerden, M. Raselimo, K. Sagonas, and B. 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 Event, USA, November 16-17, 2020*, R. Lämmel, L. Tratt, and J. de Lara, Eds. ACM, 2020, pp. 253–269. [Online]. Available: https://doi.org/10.1145/3426425.3426946

[18] B. Fischer, R. Lämmel, and V. Zaytsev, "Comparison of context-free grammars based on parsing generated test data," in *Software Language Engineering - 4th International Conference, SLE 2011, Braga, Portugal, July 3-4, 2011, Revised Selected Papers*, ser. Lecture Notes in Computer Science, A. M. Sloane and U. Aßmann, Eds., vol. 6940. Springer, 2011, pp. 324–343. [Online]. Available: https://doi.org/10.1007/978-3-642-28830-2_18

[19] P. Purdom, "A sentence generator for testing parsers," *BIT Numerical Mathematics*, vol. 12,

pp. 366–375, Sep. 1972. [Online]. Available: https://doi.org/10.1007/BF01932308

[20] N. Havrikov and A. 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. [Online]. Available: https://doi.org/10.1109/ASE.2019.00027

[21] S. V. Zelenov and S. A. Zelenova, "Generation of positive and negative tests for parsers," *Program. Comput. Softw.*, vol. 31, no. 6, pp. 310–320, 2005. [Online]. Available: https://doi.org/10.1007/s11086-005-0040-6

[22] C. Rossouw and B. Fischer, "Test case generation from context-free grammars using generalized traversal of lr-automata," in *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16-17, 2020*, R. Lämmel, L. Tratt, and J. de Lara, Eds. ACM, 2020, pp. 133–139. [Online]. Available: https://doi.org/10.1145/3426425.3426938

[23] R. Lämmel and W. Schulte, "Controllable combinatorial coverage in grammar-based testing," in *Testing of Communicating Systems, 18th IFIP TC6/WG6.1 International Conference, TestCom 2006, New York, NY, USA, May 16-18, 2006, Proceedings*, ser. Lecture Notes in Computer Science, M. Ü. Uyar, A. Y. Duale, and M. A. Fecko, Eds., vol. 3964. Springer, 2006, pp. 19–38. [Online]. Available: https://doi.org/10.1007/11754008_2

[24] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, T. Kohno, Ed. USENIX Association, 2012, pp. 445–458. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[25] S. Veggalam, S. Rawat, I. Haller, and H. Bos, "Ifuzzer: An evolutionary interpreter fuzzer using genetic programming," in *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I*, ser. Lecture Notes in Computer Science, I. G. Askoxylakis, S. Ioannidis, S. K. Katsikas, and C. A. Meadows, Eds., vol. 9878. Springer, 2016, pp. 581–601. [Online]. Available: https://doi.org/10.1007/978-3-319-45744-4_29

[26] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 2017, pp. 579–594. [Online]. Available: https://doi.org/10.1109/SP.2017.23

[27] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A. Sadeghi, and D. Teuchert, "NAUTILUS: fishing for deep bugs with grammars," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/nautilus-fishing-for-deep-bugs-with-grammars/

[28] R. Mukherjee and K. S. Patnaik, "A survey on different approaches for software test case prioritization," *J. King Saud Univ. Comput. Inf. Sci.*, vol. 33, no. 9, pp. 1041–1054, 2021. [Online]. Available: https://doi.org/10.1016/j.jksuci.2018.09.005

[29] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Trans. Software Eng.*, vol. 33, no. 4, pp. 225–237, 2007. [Online]. Available: https://doi.org/10.1109/TSE.2007.38

[30] H. Singh, L. Singh, and S. Tiwari, "A systematic literature review on test case prioritization techniques," *Int. J. Softw. Innov.*, vol. 10, no. 1, pp. 1–36, 2022. [Online]. Available: https://doi.org/10.4018/ijsi.312263

[31] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Softw. Test., Verif. Reliab.*, vol. 22, no. 2, pp. 67–120, 2012.

[32] G. Rothermel, S. G. Elbaum, A. G. Malishevsky, P. Kallakuri, and X. Qiu, "On test suite composition and cost-effective regression testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 13, no. 3, pp. 277–331, 2004.

[33] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007*. ACM, 2007, pp. 140–150.

[34] Y. Ledru, A. Petrenko, S. Boroday, and N. Mandran, "Prioritizing test cases with string distances," *Autom. Softw. Eng.*, vol. 19, no. 1, pp. 65–95, 2012. [Online]. Available: https://doi.org/10.1007/s10515-011-0093-0

[35] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, S. Cheung, A. Orso, and M. D. Storey, Eds. ACM, 2014, pp. 654–665. [Online]. Available: https://doi.org/10.1145/2635868.2635929