Breaking Parsers: Mutation-Based Generation of Programs with Guaranteed Syntax Errors

Moeketsi Raselimo
University of Stellenbosch
Stellenbosch, South Africa
22374604@sun.ac.za

Jan Taljaard
University of Stellenbosch
Stellenbosch, South Africa
18509193@sun.ac.za

Bernd Fischer
University of Stellenbosch
Stellenbosch, South Africa
bfischer@cs.sun.ac.za

Abstract

Grammar-based test case generation has focused almost exclusively on generating syntactically correct programs (i.e., positive tests) from a context-free reference grammar but a positive test suite cannot detect when the unit under test accepts words outside the language (i.e., false positives). Here, we investigate the converse problem and describe two mutation-based approaches for generating programs with guaranteed syntax errors (i.e., negative tests). Word mutation systematically modifies positive tests by deleting, inserting, substituting, and transposing tokens in such a way that at least one impossible token pair emerges. Rule mutation applies such operations to the symbols of the right-hand sides of productions in such a way that each derivation that uses the mutated rule yields a word outside the language.

CCS Concepts • **Software and its engineering** → **Parsers**; *Syntax*; *Software testing and debugging*.

Keywords Mutation testing.

ACM Reference Format:
Moeketsi Raselimo, Jan Taljaard, and Bernd Fischer. 2019. 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 '19), October 20–22, 2019, Athens, Greece.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3357766.3359542

1 Introduction

Following Purdom's seminal paper [11], many different approaches and algorithms for grammar-based test case generation have been investigated, ranging from systematic sentence generation to satisfy various coverage criteria [7, 13] to controlled random sentence generation [8]. However, except for the work by Zelenov and Zelenova [13], existing work focuses on generating syntactically correct programs from a context-free reference grammar G_{ref}, i.e., *positive test cases*. This is not sufficient because a positive test suite cannot detect when the *unit under test* (UUT) U_{test} accepts words outside the language, i.e., it cannot detect *false positives*.

This can easily happen, even if the grammar G_{test} that is implemented by U_{test} is "almost correct", i.e., structurally very similar to G_{ref}. Consider for example the grammar G_{toy}

prog → **module** *prio* `id` = *block* .
prio → `[` `num` `]`
block → **begin** (*decl* `;`)* (*stmt* `;`)* **end**
decl → **var** `id` : *type*
type → **bool** | **int**
stmt → **if** *expr* **then** *stmt* (**else** *stmt*)? |
while *expr* **do** *stmt* | `id` = *expr* | *block*
expr → *expr* = *expr* | *expr* + *expr* | `(` *expr* `)` | `id` | `num`

and assume that U_{test} implements the rule

prog → **module** (`[` *expr* `]`)? `id` = *block* .

instead. Since any positive test generated from G_{toy} starts with "**module** `[` `num` `]` `id`", U_{test} never executes the branch where the optional priority specification is not present, and the error remains undetected. Moreover, since `num` matches *expr*, the use of the wrong symbol remains undetected as well. Similarly, a wrong implementation of the *block*-rule

block → **begin** ((*decl* `;`) | (*stmt* `;`))* **end**

remains also undetected: this allows declarations and statements to be mixed arbitrarily, but positive tests derived from G_{toy} will never have a declaration directly after a statement.

Here we therefore address the problem of generating programs with guaranteed syntax errors, i.e., *negative test cases*. This problem is far from trivial: G_{toy} contains already 22 tokens and the shortest word in $\mathcal{L}(G_{toy})$ has a length of 9, so that we need to enumerate 54875873536 test cases to find out that the UUT does for example not expect the final full stop. This renders impractical approaches that systematically enumerate or randomly generate token strings and use an Earley [3] or similar parser derived from the reference grammar as oracle to identify the negative test cases.

In our opinion, negative test suites should have a number of properties. (i) In order to provide more assurance, test suites should *systematically cover* the fault space, up to some

SLE '19, October 20–22, 2019, Athens, Greece

ACM ISBN 978-1-4503-6981-7/19/10…$15.00
https://doi.org/10.1145/3357766.3359542

criterion. (ii) In order to reduce the testing efforts, test suites should be *irredundant* (i.e., not contain tests that can be removed without loosing coverage) and *interreduced* (i.e., not contain tests that can be replaced by a shorter test without loosing coverage). (iii) In order to simplify bug fixing, individual tests should contain a *single, well-defined fault* that can be related easily to the reference grammar.

We propose two systematic, mutation-based approaches to generate test suites with these properties. In Section 2, we describe a *word mutation* approach that modifies each word in a given positive test suite, by deleting, inserting, substituting, and transposing tokens. The core insight to make this approach work is that a word $\alpha ab\beta$ cannot be in the language if b is not in the follow set of a. In Section 3, we transfer the ideas from word mutation to *rule mutation*, and systematically modify the rules of the grammar so that every derivation that uses such a modified rule yields a word that is not in the language. We apply similar mutation operators to the symbols of the rules, under conditions that ensure that the yields from the (unchanged) contexts of a mutated location and from the mutated symbol itself always contain an impossible token pair. We implemented both approaches and give some initial experimental results in Section 4.

Basic notation. A *grammar* is a four-tuple $G = (N, T, P, S)$ with $N \cap T = \emptyset$, $P \subset N \times (N \cup T)^*$, and $S \in N$. We follow the use of meta-variables by Aho et al. [1, Section 4.2.2], and use $A \to \alpha \bullet \beta$ to denote an *item*, with its designated position indicated by $\bullet$, and use $P^\bullet$ for the set of all items of G.

We use the established concepts such as derivation, yield, or nullable in their usual meaning. We extend the definitions of first and last so that map to sets of symbols rather than terminals. The *precede set* (resp. *follow set*) of a symbol X are defined as $\text{precede}(X) = \{Y \mid S \Rightarrow^* \alpha YX\beta\}$ and $\text{follow}(X) = \{Y \mid S \Rightarrow^* \alpha XY\beta\}$. We call (X, Y) a *poisoned pair* iff X and Y cannot occur next to each other in any derivation from the start symbol or, equivalently, iff $X \notin \text{precede}(Y)$, or iff $Y \notin \text{follow}(X)$, and use $PP(G)$ to denote the set of all poisoned pairs of a grammar G. Note that any token sequence that contains a poisoned pair must be an invalid word.

Test suites. A *test suite* is a set of individual test cases; each *test case* consists of the *test input data* $\vec{x}$ and the *expected output* y. The UUT u is executed over $\vec{x}$ and its output $u(\vec{x})$ is compared against the expected output, resulting in either a *pass* or a *fail* verdict. In our context, the test input is a string x of terminals and the expected output is either *accept* (if $x \in \mathcal{L}(G)$) or *reject* (if $x \notin \mathcal{L}(G)$). $TS^+ \subset \mathcal{L}(G)$ is called a *positive test suite* and $TS^- \subset \overline{\mathcal{L}(G)}$ a *negative test suite*.

Related work. Harm and Lämmel [5] first proposed to use mutation to create negative test cases, but gave no algorithm. Offutt et al. [10] mutated the grammar rules, but gave no conditions to show that the generated test cases are indeed syntactically invalid. Köroglu and Wotawa [6] followed a similar approach but used a CYK-parser derived from a second grammar as oracle. Zelenov and Zelenova [13] gave negative coverage criteria for LL and LR parsers. Their NLL algorithm mutates words in a way that is guaranteed to result in failing test cases, but unlike the work proposed here, it does not mutate the grammar rules themselves (so it does not allow non-terminal mutations), and it does not support symbol deletions. They also gave an algorithm that derives negative test cases by covering paths in the LR automaton. However, we were unable to re-implement this algorithms from their description, and can thus not compare in detail.

2 Word Mutation

The basic idea of negative test suite generation by *word mutation* is to systematically modify words $w \in \mathcal{L}(G)$ into words $w' \notin \mathcal{L}(G)$. This idea can be implemented by applying a family M of *mutation operators* to each test in a given positive test suite TS^+, i.e., $TS^- = \{m(w) \mid m \in M, w \in TS^+\}$.

The two basic insights of our approach to word mutation are (i) that the basic string editing operations *insertion*, *deletion*, *substitution*, and *transposition* (which are also used to compute the Damerau-Levenshtein string edit distance [2, 9]) are a suitable family of mutation operators, and (ii) that we can guarantee that the mutants are negative test cases by ensuring that the mutation produces a poisoned pair.

Proposition 1 (Damerau-Levenshtein mutations)**.** *Let G be a grammar. Then:*

1. (token deletion) *If $uabcv \in \mathcal{L}(G)$ and $(a, c) \in PP(G)$, then $uacv \notin \mathcal{L}(G)$.*
2. (token insertion) *If $uacv \in \mathcal{L}(G)$, $b \in T$, and either $(a, b) \in PP(G)$ or $(b, c) \in PP(G)$, then $uabcv \notin \mathcal{L}(G)$.*
3. (token substitution) *If $uabcv \in \mathcal{L}(G)$, $d \in T$, and either $(a, d) \in PP(G)$ or $(d, c) \in PP(G)$, then $uadcv \notin \mathcal{L}(G)$.*
4. (token transposition) *If $uabcdv \in \mathcal{L}(G)$, and either $(a, c) \in PP(G)$ or $(c, b) \in PP(G)$ or $(b, d) \in PP(G)$, then $uacbdv \notin \mathcal{L}(G)$.*

For any positive test suite TS^+ we denote by $DL(TS^+)$ the negative test suite that results from applying to all words from TS^+ all token mutations at all positions that satisfy the conditions of Proposition 1; we also call this set the *Damerau-Levenshtein mutants* of TS^+. We can construct $TS^-_{DL(TS^+)}$ with a simple *token-stream fuzzing* algorithm: we iterate token-by-token over each word in TS^+ and check whether the conditions of Proposition 1 are satisfied at the current position; if so, we output the corresponding mutant.

Consider for example the following grammar G_{arith} for arithmetic expressions:

$$E \to E \texttt{*} E \mid E \texttt{/} E \mid E \texttt{+} E \mid E \texttt{-} E \mid \texttt{(}\, E \,\texttt{)} \mid \texttt{-?num} \mid \texttt{id}$$

Note that the minus sign is not part of the `num` token but a unary operator that is only applicable to `num`'s; consequently, `x*-1` is a valid word, but `x*+1` or `x*-y` are not.

We have follow(`(`) = {`(`, `-`, num, id} (with the same for `*`, `/`, `+`, and `-`), and follow(`)`) = {`*`, `/`, `+`, `-`, `)`} (with the same for num and id). Now consider the valid word `x*1` $\in TS^+$. Since `+` $\notin$ follow(`*`) (and hence (`+`, `*`) $\in PP(G_{arith})$), we know that inserting a `+` after the `*` produces a syntax error. Similarly, since (id, num) $\in PP(G_{arith})$, we know that deleting the `*` produces a syntax error as well.

The conditions stated in Proposition 1 are sufficient for a syntax error, but not necessary. The main limitation comes from the fact that the conditions only check the local context, and do not take the derivation into account. For example, since an id can follow a `-`, the positive test `x*-1` is not mutated into `x*-y` $\notin \mathcal{L}(G_{arith})$. Similarly, `(x)` is mutated into `(x)(` (because `(` $\notin$ follow(`)`)), but not into `(x))`.

3 Rule Mutation

The basic idea of negative test suite generation by *rule mutation* is to systematically modify the rules of the grammar so that every derivation that uses such a modified rule yields a word $w' \notin \mathcal{L}(G)$. We can adapt our word mutation approach to the rule context, i.e., (i) we can apply string edit operations to the rules, and (ii) we must ensure that *any* sentence derived via a mutated rule produces a poisoned pair. Hence, we need to ensure that the yields from the (unchanged) contexts of a mutated location and from the mutated symbol itself always contain a poisoned pair. Note that we cannot delete or insert any nullable symbol B to mutate a rule $p = A \to \gamma$ into $p' = A \to \gamma'$ because we would get a word $w \in \mathcal{L}(G)$ derivable via p that is also derivable via p'.

We first define two functions that compute the *left* (resp. *right*) *set* of an item. These sets comprise the symbols that can occur immediately to the left (resp. right) of the designated position (where the mutation operator will be applied). Hence, the left set of an item $A \to \alpha \bullet \beta$ contains all tokens that can occur at the end of α and, if α is nullable, all tokens that in other contexts can occur left of A.

Definition 2 (left set, right set)**.** *The functions* left, right : $P^\bullet \to \mathcal{P}(T)$ *are defined by*

$$\text{left}(A \to \alpha \bullet \beta) = \begin{cases} (\text{last}(\alpha) \cup \text{precede}(A)) \cap T & \text{if } \alpha \text{ nullable} \\ (\text{last}(\alpha) \cap T & \text{otherwise} \end{cases}$$

and

$$\text{right}(A \to \alpha \bullet \beta) = \begin{cases} (\text{first}(\beta) \cup \text{follow}(A)) \cap T & \text{if } \beta \text{ nullable} \\ (\text{first}(\beta)) \cap T & \text{otherwise} \end{cases}$$

We can then formulate conditions under which we allow a symbol mutation. The idea here is to check for "boundary incursions" over the designated position of the modified item, i.e., to check whether any token that can follow (precede) the left (right) set of the modified item is also in its right (left) set. If that is not the case, then any yield must contain a poisoned pair straddling the designated position, and hence cannot be part of a word in the language.

Definition 3 (symbol deletion mutation)**.** *Let* $p = A \to \alpha \bullet X\beta$ *be an item in* $P^\bullet$*. If either*

$$\text{follow}(\text{left}(A \to \alpha \bullet \beta)) \cap \text{right}(A \to \alpha \bullet \beta) = \emptyset$$

or

$$\text{left}(A \to \alpha \bullet \beta) \cap \text{precede}(\text{right}(A \to \alpha \bullet \beta)) = \emptyset$$

then the deletion of X *from* p *at the designated position yields the* mutated production $p' = A \to \alpha\beta$.

Definition 4 (symbol insertion mutation)**.** *Let* $p = A \to \alpha \bullet \beta$ *be an item in* $P^\bullet$*, and* $X \in V$ *be a symbol. If either*

$$\text{follow}(\text{left}(A \to \alpha \bullet X\beta)) \cap \text{right}(A \to \alpha \bullet X\beta) = \emptyset$$

or

$$\text{left}(A \to \alpha \bullet X\beta)) \cap \text{precede}(\text{right}(A \to \alpha \bullet X\beta) = \emptyset$$

then the insertion of X *into* p *at the designated position yields the* mutated production $p' = A \to \alpha X\beta$.

If X is nullable, then both conditions of Definitions 3 and 4 are false by construction. Hence, we never delete or insert a nullable symbol.

Definition 5 (symbol substitution mutation)**.** *Let* $p = A \to \alpha \bullet X\beta$ *be an item in* $P^\bullet$*, and* $Y \in V$*. If either*

$$\text{follow}(\text{left}(A \to \alpha \bullet Y\beta)) \cap \text{right}(A \to \alpha \bullet Y\beta) = \emptyset$$

or

$$\text{left}(A \to \alpha \bullet Y\beta) \cap \text{precede}(\text{right}(A \to \alpha \bullet Y\beta)) = \emptyset$$

then the substitution of X *by* Y *in* p *at the designated position yields the* mutated production $p' = A \to \alpha Y\beta$.

We can easily extend the rule mutations by a symbol transposition; we simply need to translate the conditions of Proposition 1 into left/right and precede/follow terms.

We use the notation $p \rightsquigarrow p'$ to denote that a production p' has been constructed from p by mutation with any of the mutation operations above. We then get the following correctness theorem, stated here without proof due to space restrictions.

Proposition 6 (Correctness of rule mutation)**.** *Let* G *be a grammar,* $A \to \gamma \in P$*,* $S \Rightarrow^* \alpha A\beta$*, and* $A \to \gamma \rightsquigarrow A \to \gamma'$*. Then for all* $w \in T^*$ *such that* $\alpha\gamma'\beta \Rightarrow^* w$*, we have* $w \notin \mathcal{L}(G)$.

4 Implementation and Evaluation

4.1 Implementation

We have implemented both approaches in approx. 3000 lines of Prolog code. Our implementation first reads in an EBNF grammar, eliminates the EBNF operators, and computes the standard grammar predicates over the resulting plain BNF grammar. Word mutation is implemented as a straightforward Prolog-version of the token-stream fuzzing algorithm sketched in Section 2. We use a generic coverage algorithm that follows the approach of Fischer et al. [4] as basis for rule mutation. The algorithm constructs a minimal embedding for each *coverage target*, i.e., the instantiation of the coverage criterion for a given symbol. We simply need to modify the

Table 1. Characteristics of test suites for benchmark and application grammars

Grammar	$\|N\|$	$\|T\|$	$\|P\|$		$DL(sym)$		$DL(rule)$		$DL(cdrc)$		$DL(pll)$		total_{DL}	*rule-mut*		total	overlap
G_{arith}	8	10	15	<0.1	236 (8)	<0.1	236 (8)	<0.1	11260 (190)	0.1	749 (20)	<0.1	11410	174	<0.1	11416	96.6%
G_{toy}	17	24	28	0.2	6729 (11)	0.1	7276 (12)	0.1	95232 (126)	0.9	9952 (16)	0.1	95232	2447	0.5	95975	69.6%
alan-14	49	48	93	0.6	28640 (39)	0.2	33771 (45)	0.2	164398 (186)	1.5	42119 (58)	0.3	171518	10211	3.8	173859	77.1%
alan-16	50	48	95	0.7	29442 (40)	0.2	34534 (46)	0.2	145426 (169)	1.0	47753 (65)	0.3	153757	9531	3.6	155652	80.1%
simpl-13	46	47	88	0.6	27017 (38)	0.1	31218 (43)	0.2	139331 (166)	1.0	36135 (50)	0.2	143049	8984	3.3	144959	78.7%
simpl-15	49	48	94	0.6	30393 (41)	0.2	35008 (46)	0.2	145547 (168)	1.3	44166 (58)	0.3	150598	10139	3.6	152769	78.6%
dot	30	17	50	0.2	2162 (14)	<0.1	3073 (19)	<0.1	6071 (33)	<0.1	4232 (26)	<0.1	7202	771	0.5	7360	79.5%
email	44	97	192	0.9	7857 (101)	0.1	15039 (149)	0.2	225697 (1145)	4.9	56987 (602)	1.0	260276	653	226.0	260449	73.5%
Modula-2	243	85	384	28.3	200669 (111)	2.7	244376 (132)	3.7	742292 (375)	12.4	537601 (263)	10.4	1024798	67503	187.2	1058950	64.2%

coverage target for rule coverage: instead of covering the rules, we cover all rule mutations.

Our implementation can also produce a detailed explanation of the syntax error, or more precisely, its cause. This is based on the applied operation and position of the mutation, and can be used to construct detailed oracles.

4.2 Benchmark and Application Grammars

Table 1 summarizes the characteristics of the test suites generated for a variety of benchmark and application grammars. For each grammar, we give the size of the terminal, non-terminal, and rule set, respectively. We derive the mutants from four different positive test suites that respectively satisfy symbol, rule [11], and a variant of context-dependent rule coverage (CDRC) [7], as well as positive LL coverage (PLL) [13]. We give the number of DL mutants derived for each of the positive test suites (whose sizes are given in brackets). total_{DL} denotes the total number of unique DL mutants derived from all four positive test suites. *rule-mut* gives the results for rule mutation. The final two columns give the total number of unique negative test cases (i.e., both word and rule mutants), and the fraction of rule mutants that are also produced by word mutation (denoted as overlap).

All times are given in seconds and were measured on a standard laptop with a 2.6Ghz Intel i7-6600 dual core CPU and 8GB memory, running under Windows 10.

Running examples. G_{arith} is very simple, so the generated test suites are unsurprisingly small and most rule mutations can also be produced by word mutations, except for the insertion and replacement of non-terminal symbols that produce longer token sequences. For example, we get the test `x(y)z` from the replacement of the `*` in the first alternative by the non-terminal representing the fifth alternative in the BNF version of the grammar. For G_{toy}, rule mutation creates tests that uncover the first two errors discussed in the introduction, but not the last one, due to role of the semicolon as terminator for both declarations and statements.

Coursework grammars. The next block contains the results for four simple, Pascal-like languages that have been used in a second-year systems programming course. The languages are more complex than our running example G_{toy} and the number of generated test cases is uniformly larger.

We evaluated the instructor-implemented parsers against the generated negative test suites. They uncovered a similar error in all four parsers: they all accept further tokens after the closing **`end`** of the start rule. We also used a similarly complex different language in a compiler engineering course; here, the negative test suites uncovered false positives in *all* 13 student submissions, all relating to some deliberately unclear formulations in the textual language description.

Dot. dot is the grammar for the input language of the dot graph drawing tool. Running the tool over the generated tests uncovered a transcription error in our first version, where we forgot to make the terminal **`id`** optional in the start rule *graph* → **`strict`**? **`graph id`**? **`{`** *stmt_list* **`}`**

Email addresses. email is a grammar for email addresses that is derived from RFC 5322. It specifies the *addr-spec* part of the RFC (e.g., `gray@uab.edu`), not the *address* part, which can contain a display name and the addr-spec in angle brackets (e.g., `"Jeff Gray" <gray@uab.edu>`). It does, however, allow comments as part of either the local part or the domain part (e.g., `gray(Jeff)@uab.edu`).

The grammar specifies the addresses at the (ASCII) character level, which explains the large number of terminals and rules. This in turn leads to large positive test suites (in particular for CDRC), and consequently also to a very large number of word mutants. We evaluated three email validators over the generated test suites and found both false positives and false negatives in all systems.

Modula-2. The last example uses the ISO-standard grammar for Modula-2 from `modula2.org/tutor`. It shows that our approach also scales to production grammars.

5 Conclusions

We introduced two simple and intuitive mutation-based algorithms that systematically generate negative test cases for a given context-free grammar. The tests are compact and have a single, well defined error that can be used for precise oracles. Initial experiments have shown that they uncover grammar faults that positive tests fail to uncover. We see several possible uses for this work, both in grammar engineering (e.g., grammar comparison [4] or fault localization [12]) and in other applications (e.g., automatic grading of student grammars or feedback-directed whitebox fuzzing).

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2007. *Compilers: Principles, Techniques, & Tools* (2nd ed.). Addison-Wesley.

[2] Fred Damerau. 1964. A technique for computer detection and correction of spelling errors. *Commun. ACM* 7, 3 (1964), 171–176.

[3] Jay Earley. 1970. An Efficient Context-Free Parsing Algorithm. *Commun. ACM* 13, 2 (1970), 94–102.

[4] Bernd Fischer, Ralf Lämmel, and Vadim Zaytsev. 2011. 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 (Lecture Notes in Computer Science)*, Anthony M. Sloane and Uwe Aßmann (Eds.), Vol. 6940. Springer, 324–343.

[5] Jörg Harm and Ralf Lämmel. 2000. Two-dimensional Approximation Coverage. *Informatica (Slovenia)* 24, 3 (2000).

[6] Yavuz Köroglu and Franz Wotawa. 2019. Fully automated compiler testing of a reasoning engine via mutated grammar fuzzing. In *Proceedings of the 14th International Workshop on Automation of Software Test, AST@ICSE 2019, May 27, 2019, Montreal, QC, Canada.*, Byoungju Choi, María José Escalona, and Kim Herzig (Eds.). IEEE / ACM, 28–34.

[7] Ralf Lämmel. 2001. 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 (Lecture Notes in Computer Science)*, Heinrich Hußmann (Ed.), Vol. 2029. Springer, 201–216.

[8] Ralf Lämmel and Wolfram Schulte. 2006. 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 (Lecture Notes in Computer Science)*, M. Ümit Uyar, Ali Y. Duale, and Mariusz A. Fecko (Eds.), Vol. 3964. Springer, 19–38.

[9] Vladimir I. Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady* 10, 8 (1966), 707–710.

[10] Jeff Offutt, Paul Ammann, and Lisa Liu. 2006. Mutation testing implements grammar-based testing. In *2nd Workshop on Mutation Analysis (ISSRE Workshops), 2006*. IEEE Computer Society, 12.

[11] Paul Purdom. 1972. A Sentence Generator for Testing Parsers. *BIT* (1972), 366–375.

[12] Moeketsi Raselimo and Bernd Fischer. 2019. Spectrum-Based Fault Localization for Context-Free Grammars. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2019, Athens, Greece, October 21-22, 2019.* This volume.

[13] Sergey V. Zelenov and Sophia A. Zelenova. 2005. 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 (Lecture Notes in Computer Science)*, Wolfgang Grieskamp and Carsten Weise (Eds.), Vol. 3997. Springer, 187–202.