Spectrum-Based Fault Localization for Context-Free Grammars

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

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

Abstract

We describe and evaluate the first spectrum-based fault localization method aimed at finding faulty rules in a context-free grammar. It takes as input a test suite and a modified parser for the grammar that can collect grammar spectra, i.e., the sets of rules used in attempts to parse the individual test cases, and returns as output a ranked list of suspicious rules. We show how grammar spectra can be collected for both LL and LR parsers, and how the ANTLR and CUP parser generators can be modified and used to automate the collection of the grammar spectra. We evaluate our method over grammars with seeded faults as well as real world grammars and student grammars submitted in compiler engineering courses that contain real faults. The results show that our method ranks the seeded faults within the top five rules in more than half of the cases and can pinpoint them in 10%–40% of the cases. On average, it ranks the faults at around 25% of all rules, and better than 15% for a very large test suite. It also allowed us to identify deviations and faults in the real world and student grammars.

CCS Concepts • **Software and its engineering** → **Parsers**; *Syntax*; *Software testing and debugging*; • **Theory of computation** → *Grammars and context-free languages.*

Keywords Spectrum-based fault localization.

ACM Reference Format:
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 '19), October 20–22, 2019, Athens, Greece.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3357766.3359538

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SLE '19, October 20–22, 2019, Athens, Greece
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6981-7/19/10…$15.00
https://doi.org/10.1145/3357766.3359538

1 Introduction

Grammars are software, and can contain bugs like any other software. Testing can be used to demonstrate the presence of bugs in grammars (or any other software), but does not directly give any further information about their location. Software fault localization techniques [13, 53] build on testing and try to automatically identify likely bug locations. *Spectrum-based fault localization* (SFL) methods [3, 23, 36, 43, 52] execute the unit under test (UUT) over a given test suite and record a *program spectrum*, a representation of the execution information for the UUT's individual program elements; most SFL methods use *binary statement coverage*, i.e., record whether a statement has been executed or not. From the spectrum they then compute a *suspiciousness score* for each program element; the methods differ in the details of the score computation but elements with a higher score are seen as more likely to contain a bug.

Hence, we describe and evaluate the first method to localize faulty rules in a context-free grammar. We view a rule to be possibly faulty if it is applied in a derivation of a word that is accepted by a parser for the grammar but is outside the "true" language (which may have a different grammar), or vice versa, if it is applied in a partial derivation of a word that is rejected by the parser but that is within the true language. This view lends itself readily to a spectrum-based fault localization method: we only need to replace the concept of "executed statements" by that of "used rules", but can keep the remaining established framework in place. We therefore introduce the notion of *grammar spectra* which summarize which of the grammar rules have been (partially) applied in an attempt to parse an input. We show how grammar spectra can be collected for both LL and LR parsers. The main technical challenge is to identify partially applied rules in cases where an LR parser encounters a syntax error and thus fails to execute the reduction steps that mark the completion of rule application. We show that the missing rules can be recovered from the kernel items contained in the states that are on the parser stack when it encounters a syntax error.

The collection of the grammar spectra requires runtime support from the parser, which needs to log the rules that are applied. Parsers generated by ANTLR [2] already provide this support through some extensions, but parsers generated by CUP [1] do not. We have therefore extended CUP itself to generate parsers with the required logging. We then used parsers generated by ANTLR and CUP from grammars with both seeded and real faults to evaluate our method; the latter grammars include different real-world grammars for the Pascal language as well as student submissions in several compiler engineering courses. The evaluation results show that our method can identify grammar bugs with a high precision: it ranks the seeded faults within the top five rules in more than half of the cases and can pinpoint them (i.e., uniquely rank them as most suspicious) in 10%–40% of the cases. On average, it ranks the faults at around 25% of all rules, and better than 15% for a large test suite. It also allowed us to identify deviations and faults in the real world and student grammars, which both contain multiple real faults.

Our approach works at a higher level of abstraction than generic SFL approaches and returns fault locations in domain-specific terms (i.e., rules rather than statements). This has several advantages. First, it simplifies any subsequent repair attempts—grammar writers can directly use our results and do not need to manually trace back from the parser's implementation to the grammar's rules. Second, it increases the localization precision because it discards all aspects of the parser's internal bookkeeping and error handling code that could impact the localization process if generic program spectra were used. Third, it can also be meaningfully applied when the parser uses a table-driven implementation and there is no direct representation of the individual rules as executable code; this is typically the case for LR parsers.

Outline and contributions. In Section 2, we fix the basic grammar notations that we use in this paper and give the necessary background on spectrum-based fault localization. In Section 3, we define the notion of grammar spectrum that is at the core of our work and illustrate it with a worked example. In Section 4, we describe our implementation of grammar spectrum extraction for the ANTLR and CUP parser generators. In Section 5, we evaluate our method over grammars with seeded faults as well as real world grammars and submitted student grammars that contain real faults. The results show that our method can in many cases identify a faulty rule precisely, even in the presence of multiple, real faults. In Sections 6 and 7, we discuss related work and conclude with suggestions for future work.

In summary, we make the following contributions in this paper: (*i*) we present the first method to localize faulty rules in a context-free grammar; (*ii*) we describe an implementation of our method using the ANTLR and CUP parser generators; and (*iii*) we demonstrate the effectiveness of our method over grammars with seeded faults as well as real-world grammars and student submissions from compiler engineering courses that contain multiple real faults.

2 Background and Notation

Grammars and meta-variables. A *context-free grammar* (or simply *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 call N its *non-terminal symbols*, T its *terminal symbols* or *tokens*, P its *production rules* (or simply *productions* or *rules*), and S its *start symbol*. We also write $A \longrightarrow \gamma$ for a rule $(A, \gamma) \in P$. We follow the notation in [4, Section 4.2.2] and use the meta-variables $A, B, C, \ldots$ for non-terminals, $a, b, c, \ldots$ for terminals, X, Y, Z for *grammar symbols* in $V = N \cup T$, $u, v, \ldots$ for strings of terminals or *words*, $\alpha, \beta, \gamma, \ldots$ for strings of grammar symbols or *sentential forms*, with ϵ denoting the empty string, and $p, q, \ldots$ for rules. We use $|\alpha|$ to denote the length of a string. We assume that P contains only one rule $S \longrightarrow \alpha$, and that S does not occur in any other rule.

Items. An *item* is a rule $A \longrightarrow \alpha \bullet \beta$ with a designated position (denoted by $\bullet$) on its right-hand side. An item is called *kernel item* if $\alpha \neq \epsilon$ or $\alpha = \epsilon$ and $A = S$. We use $P^\bullet$ to denote the set of all items, i.e., all rules with all designated positions.

Derivations and generated language. A *derivation over G* $\Rightarrow_G \subseteq V^* \times V^*$ relates sentential forms according to G. We use $\alpha A \beta \Rightarrow_G \alpha\gamma\beta$ to denote that $\alpha A \beta$ *produces* (or *derives*) $\alpha\gamma\beta$ by application of the rule $A \longrightarrow \gamma \in P$. We write $\Rightarrow$ if the grammar is clear from the context and $\Rightarrow_R$ if $A \longrightarrow \gamma \in R \subseteq P$. We use $\Rightarrow^*$ for the reflexive-transitive closure.

The *yield* of a sentential form α is the set of all words that can be derived from it, i.e., $\text{yield}(\alpha) = \{w \in T^* \mid \alpha \Rightarrow^* w\}$. α is *nullable* if $\epsilon \in \text{yield}(\alpha)$. The *language $L(G)$ generated by a grammar G* is the yield of its start symbol, i.e., $L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$.

A derivation $S \Rightarrow^* \alpha A \beta \Rightarrow \alpha\gamma\beta \Rightarrow^* w$ is *k-prefix bounded for w* if for any derivation step we have either $\alpha \in T^*$ and $|\alpha| < k$ or $\gamma = \mu\nu$, μ is not nullable, $w = uv$ and $|u| \leq k$, such that $\alpha\mu \Rightarrow^* u$ and $\nu\beta \Rightarrow^* v$. Intuitively, this means that a k-prefix bounded derivation for w never expands a non-terminal symbol whose yield in w will ultimately start only beyond a prefix of length k. We will later use this concept to rule out derivations past a syntax error.

Test suites. A *test suite* consists of a list of UUT inputs and corresponding expected outputs (which can also be a specific system error, e.g., for illegal inputs). The UUT *passes* a test if it produces the expected output for the given input. In our case, test inputs are words $w \in T^*$, expected outputs are either "accept" or "reject". More detailed expected outputs (e.g., error locations) could prevent the mis-classification of applied rules, and so increase the precision of the fault localization, but are difficult to implement because they may depend on internal aspects of the parser (e.g., error correction strategy).

Grammar-based test suite construction. For our experimental evaluation we use several test suites that satisfy different *grammar coverage criteria* [26] such as *rule* or *cdrc* coverage. We use a generic coverage algorithm that follows the approach of Fischer et al. [16] to construct these test suites. The algorithm constructs a minimal embedding for each *coverage target* (i.e., the instantiation of the criterion for a given symbol) and merges identical test satisfying different targets.

Failure, error, fault. The informal notion of a "bug" can be deconstructed into three different concepts [51]. A *failure* is a situation where the system's observed output deviates from the correct output, an *error* is an internal system state that may lead to a failure, and a *fault* is a code fragment which causes an error in the system when it is executed. Note that errors do not necessarily manifest themselves as observable failures. *Fault localization* is an attempt to identify the unknown position of the fault from an observed failure.

Program spectra. A *program spectrum* is a representation of the execution information for the UUT's individual program elements; most SFL methods use (binary) statement coverage, i.e., record whether a statement has been executed for a given test or not. The spectra for the individual tests are then correlated with the test outcomes and aggregated into four basic counts for each individual program element e: $ep(e)$ resp. $ep(e)$ are the number of passed resp. failed tests in which e is executed, while $np(e)$ resp. $np(e)$ are the number of passed resp. failed tests in which e is *not* executed. Note that these counts are related to each other via the number of passed tests tp (resp. failed tests tf) in the test suite, i.e., $ep(e) + np(e) = tp$ and $ef(e) + nf(e) = tf$ for each e.

Ranking metrics. SFL methods use the basic counts to compute for each program element a *suspiciousness score*; elements that are ranked higher (i.e., have a higher score) are seen as more likely to contain a bug. The methods (which are traditionally called *ranking metrics*, even though they are not proper metrics) differ in the formulas used for the score computation. In this paper, we use four ranking metrics that are widely used in SFL. Table 1 shows their score definitions. Note that Tarantula is the only metric that uses the number of passed tests $np(e)$ in which an element e is *not* executed. Note also that DStar is parameterized over the exponent n; here, we use the most common value $n = 2$. DStar becomes undefined for an element e if it is executed *only* in failing test cases. We assign a maximal score in this case, since we consider e to be the most suspicious element.

The metrics become undefined or degenerate and rank all elements equally if the test suite does not contain at least one *failing* test; similarly, the metrics become undefined or simply rank the elements by occurrence count if the test suite does not contain at least one *passing* test. We therefore assume in our work that test suites indeed contain at least one failing and one passing test.

Ranking metrics can assign the same score to different elements. For ranking purposes, we need to resolve such *ties* and assign a well-defined rank to all tied elements. Here we use the mid-point of the range of elements with the same score; the assigned rank then indicates how many elements a user is expected to inspect before they find the fault if elements with the same score are inspected in random order.

Table 1. SFL ranking metrics

Ranking metric	$score(e)$
Tarantula [23]	$\frac{\frac{ef(e)}{ef(e)+nf(e)}}{\frac{ef(e)}{ef(e)+nf(e)}+\frac{ep(e)}{ep(e)+np(e)}}$
Ochiai [38]	$\frac{ef(e)}{\sqrt{(ef(e)+nf(e))(ef(e)+ep(e))}}$
Jaccard [12]	$\frac{ef(e)}{ef(e)+nf(e)+ep(e)}$
DStar [52]	$\frac{ef(e)^n}{nf(e)+ep(e)}$

A more pessimistic variant uses the lowest possible rank that is consistent with the scores; this would result in a worst-case estimate of the number of elements to be inspected.

3 Grammar Spectra

We can informally define a *grammar spectrum* as the set of all rules $R \subseteq P$ that are applied when a word w in the test suite is successfully parsed. But which rules should be taken as applied when the parser rejects w? Since this depends on the nature of the applied parser, we first formalize our intuition in terms of arbitrary derivations and then concretize it for LL and LR parsers. In the following we assume that $G = (N, T, P, S)$ is the UUT, so $w \in L(G)$ (resp. $w \notin L(G)$) only means that the parser accepts (resp. rejects) w; each outcome can be associated with a passing or a failing test.

The formal definition of grammar spectra for accepted words directly follows our intuition.

Definition 1 (positive grammar spectrum)**.** *If* $S \Rightarrow_{p_1} \alpha_1 \Rightarrow_{p_2} \alpha_2 \Rightarrow_{p_3} \cdots \Rightarrow_{p_n} \alpha_n = w$*, then* $R = \bigcup_i p_i$ *is called a* positive grammar spectrum *for* w*.*

If the parser rejects w, then we construct the spectrum from all rules that have been applied to the left of the error position; more precisely, we look at the productions used in a k-prefix bounded derivation.

Definition 2 (negative grammar spectrum)**.** *Assume* $w = uav \notin L(G)$ *with* $|u| = k$ *maximal such that there exists* $w' = uv' \in L(G)$*. Let* $S \Rightarrow_{p_1} \alpha_1 \Rightarrow_{p_2} \alpha_2 \Rightarrow_{p_3} \cdots \Rightarrow_{p_n} u\alpha \Rightarrow^* uv'$ *be a* k*-prefix bounded derivation for* w'*. Then* $R = \bigcup_i p_i$ *is called a* negative grammar spectrum *for* w*.*

Any partial derivation that completely consumes the unique longest valid prefix u of w but does not apply any rules beyond the location of the syntax error (i.e., a) induces a negative spectrum. However, since the continuation v' is not unique, w may induce several negative spectra. We can make Definition 2 more precise by considering the union of all spectra for all possible continuations v'. Note that a strict interpretation of Definition 2 does not allow any error

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

Figure 1. Example grammar G_{toy}

```
program x = { x = (x); }.
program x = { x = x + x; }.
program x = { x = x; }.
program x = { x = x = x; }.
program x = { x = 0; }.
program x = { if x then sleep; }.
program x = { if x then sleep else sleep; }.
program x = { sleep; }.
program x = { var x : bool; }.
program x = { var x : int; }.
program x = { while x do sleep; }.
program x = { { }; }.
program x = { }.
```

Figure 2. Test suite for G_{toy} satisfying *rule*-coverage

corrections by the parser; however, in our experimental evaluation we will also consider a relaxed variant that includes error corrections.

Running example. We illustrate our method with a worked example based on the toy grammar G_{toy} shown in Figure 1 and a corresponding test suite satisfying *rule*-coverage, as shown in Figure 2.

We assume that the grammar developer has made mistakes in both the **`if`**- and **`while`**-rule,

$$
\begin{array}{ll}
\textit{stmt} \rightarrow \ldots & \mid \texttt{if}\ \textit{expr}\ \texttt{then}\ \textit{stmt}\ \texttt{else}\ \textit{stmt} \\
 & \mid \texttt{while}\ \textit{expr}\ \texttt{do}\ \textit{block} \mid \ldots
\end{array}
$$

requiring the **`else`**-branch to be present and restricting the body of **`while`**-loops to be blocks.

We then create a parser for this faulty version G'_{toy} and run it over the test suite to collect the grammar spectra shown in Table 2. We finally compute the scores according to the four ranking metrics shown in Table 1 and rank the rules; note how different execution counts can lead to the same scores (see for example *stmt*:1 and *stmt*:3 for Tarantula), resulting in ties.

All four metrics rank the faulty **`while`**-rule (i.e., *stmt*:3) top. Tarantula and DStar pinpoint it as the unique most suspicious rule, while Ochiai and Jaccard rank it as tied first together with the correct **`sleep`**-rule (i.e., *stmt*:1). The second fault is more difficult to localize because the faulty rule is executed in both failing and passing test cases. Tarantula and DStar rank it as tied second together with the **`sleep`**-rule (although for different reasons). Jaccard ranks it third, while Ochiai ranks it only fourth, even behind the rule $\textit{expr} \rightarrow \texttt{id}$ that is applied in most derivations.

If we inspect the rules in rank order and resolve ties by picking rules arbitrarily, we have on average to look at 2.5 rules (i.e., 16.7% of all rules) before we find both faults using Tarantula or DStar, 3 rules (or 20%) using Jaccard, and 4 rules (or 26.7%) using Ochiai.

4 Implementation

In order to collect the grammar spectra, we need to modify the parser to log which rules it has applied. The nature of the modifications depends on the general parsing technology and the specifics of the parser; here we describe the modifications we made to the ANTLR and CUP parser generators.

4.1 Recursive-descent Parsers: ANTLR

Since LL parsers build the derivation top-down, left-to-right, every step $\alpha_i \Rightarrow_{p_i} \alpha_{i+1}$ adds the corresponding rule p_i to the spectrum, whether the derivation ends in success or not. In a simple recursive-descent parser, each rule is implemented by its own parsing function, and each step corresponds to a call to one of these functions. Hence, a grammar spectrum is a set of parse functions entered at least once in a derivation. This definition is applicable for both valid and invalid words. The only difference is that for an invalid word (i.e., $w' \notin \mathcal{L}(G)$), there is at least one function which was entered and never exited successfully.

In a simple recursive-descent parser, spectrum collection is thus a simple logging task that can be implemented easily, for example using aspect-oriented programming. However, ANTLR generates adaptive LL(*) parsers that use unbounded look-ahead, which complicates the structure of the parse functions. It provides runtime support to automate collection of grammar spectra through tree walkers but this only works when it actually completes the parse and builds a tree. ANTLR's error recovery strategy allows it to do so in most cases, but this means that rules used after any error recovery will be mis-classified in the spectrum.

We therefore turned off error recovery, forcing the parser to bail out without returning a parse tree when it encounters the first syntax error. We then used aspect-oriented programming to track all calls to ANTLR's internal `enterOuterAltNum` method that sets the rule and alternative fields in the tree. In this way, we can (in principle) extract spectra conforming to Definitions 1 and 2. In practice, however, we encountered two problems that can cause the extracted spectra to be wrong. First, ANTLR's powerful adaptive LL(*) parsing mechanism can cause it to raise a syntax error (typically `no viable alternative`) without actually entering the parse function for the corresponding rule. Second, ANTLR's tracking of rule applications is wrong (i.e., the call to `enterOuterAltNum` is missing, see the open issue #2222) for grammars that contain left-recursive rules.

Table 2. Grammar spectra, suspiciousness scores, and ranks for the faulty grammar version G'_{toy} and *rule* test suite; ✓(resp. ✗) indicates execution in a passing (resp. failing) test cases. Ranks are only shown for rules with non-zero scores; ties are indicated by a preceding "=". Scores in italics indicate rules ranked ahead of or tied with any faulty rule. Faulty rules are shown in bold.

rule	1	2	3	4	5	6	7	8	9	10	11	12	13	*ep*	*np*	*ef*	*nf*	Tarantula		Ochiai		Jaccard		DStar	
prog	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✗	✓	✓	11	0	2	0	0.50	=5	0.39	=5	0.15	=5	0.36	=5
block	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✗	✓	✓	11	0	2	0	0.50	=5	0.39	=5	0.15	=5	0.36	=5
decl									✓	✓				2	9	0	0	0.00	-	0.00	-	0.00	-	0.00	-
type:1										✓				1	10	0	0	0.00	-	0.00	-	0.00	-	0.00	-
type:2									✓					1	10	0	0	0.00	-	0.00	-	0.00	-	0.00	-
stmt:1						✗	✓	✓			✗			2	9	2	0	*0.85*	=2	*0.71*	=1	*0.50*	=1	*2.00*	=2
stmt:2						✗	✓							**1**	**10**	**1**	**1**	**0.85**	**=2**	**0.50**	**4**	**0.33**	**3**	**2.00**	**=2**
stmt:3											✗			**0**	**11**	**1**	**1**	**1.00**	**1**	**0.71**	**=1**	**0.50**	**=1**	**4.00**	**1**
stmt:4	✓	✓	✓	✓	✓									5	6	0	2	0.00	-	0.00	-	0.00	-	0.00	-
stmt:5												✓		1	10	0	2	0.00	-	0.00	-	0.00	-	0.00	-
expr:1				✓										1	10	0	2	0.00	-	0.00	-	0.00	-	0.00	-
expr:2		✓												1	10	0	2	0.00	-	0.00	-	0.00	-	0.00	-
expr:3	✓													1	10	0	2	0.00	-	0.00	-	0.00	-	0.00	-
expr:4	✓	✓	✓	✓		✗	✓				✗			5	6	2	0	0.79	4	*0.53*	3	0.29	4	0.80	4
expr:5					✓									1	10	0	2	0.00	-	0.00	-	0.00	-	0.00	-

state	corresponding kernel items
2	$prog \longrightarrow$ `program` $\bullet$ `id` = *block* .
3	$prog \longrightarrow$ `program` `id` $\bullet$ = *block* .
4	$prog \longrightarrow$ `program` `id` = $\bullet$ *block* .
5	$block \longrightarrow$ `{` $\bullet$ $((decl\,;)^*\,(stmt\,;)^*)$ `}`
8	$stmt \longrightarrow$ `while` $\bullet$ *expr* `do` *stmt*
50	$stmt \longrightarrow$ `while` *expr* $\bullet$ `do` *stmt* $expr \longrightarrow expr \bullet = expr$ $expr \longrightarrow expr \bullet + expr$
51	$stmt \longrightarrow$ `while` *expr* `do` $\bullet$ *stmt*

Figure 3. CUP parse stack on syntax error.

4.2 Table-driven LR Parsers: CUP

In bottom-up parsing, application of a rule is carried out by two main operations, shift and reduce. For a valid word $w \in L(G)$ we can rely simply on the reduce operation to extract the spectrum, since the reduction concludes the rule application. Since CUP does not provide the required logging capabilities, we added this to the table interpreter.

For negative spectra, we use the logging extension in the reduce operation to capture the fully applied rules to the left of the syntax error but we also need to capture the partially applied rules. In LR parsing, these are reflected in the states that are on the parser stack when it encounters an error: each state represents a set of items $\{A_i \longrightarrow \alpha_i \bullet \beta_i\}$, each kernel item $A_i \longrightarrow \alpha_i \bullet \beta_i$ with $\alpha_i \neq \epsilon$ represents a partially applied rule, and the yield of each β_i describes the prefixes of possible continuations v' (see Definition 2). We could even re-construct the k-prefix bounded derivation from the rules in the kernel items and the logged successful rule applications, but we would need to log more details to obtain the right order. We added a simple stack traversal to the table interpreter that replaces the normal error handling routine which may modify the stack. We collect the rules in the kernel items in each state by analyzing CUP's output when it builds the parse tables.

Figure 3 shows CUP's parse stack when it uses the modified grammar G'_{toy} to parse the test program

```
program x = { while x do sleep }.
```

and encounters the syntax error at `sleep`. The stack traversal gives us the (partial) spectrum {*prog:1, block:1, stmt:4, expr:1,expr:2*}. In addition, we get *expr:4* as result of a successful reduce operation.

5 Evaluation

We evaluate our method over grammars with seeded faults as well as real world grammars and student grammars submitted in compiler engineering courses that contain real faults. In this section we present the details of our evaluation.

5.1 Fault Seeding

Experimental setup. In a first series of experiments we used fault seeding to evaluate the efficacy of our method for different parsing techniques, test suites, and ranking metrics. We used the grammar of a small programming language called SIMPL as basis for these experiments. SIMPL was originally designed for use in a second-year computer architecture course at our university, where students were given an LL(1) grammar for SIMPL in EBNF format, and had to manually implement a recursive-descent parser. We derived grammars for ANTLR (v4) and CUP (v0.11b) from this version. The ANTLR version required left-factorization. It contains 84 rules, 42 non-terminal symbols, and 47 terminal symbols. The CUP version required the elimination of the EBNF extensions. It contains 80 rules, 32 non-terminal symbols, and 47 terminal symbols.

We then mutated the grammars by blindly applying individual symbol edit operations (deletion, insertion, substitution, and transposition) at every position on the right-hand side of every rule of the grammars. We discarded all grammar mutants that do not allow the parser generator to produce a parser (e.g., by introducing indirect left-recursion in an ANTLR grammar). This leaves us with 32274 mutants for ANTLR and 26628 mutants for CUP.

We then executed each mutant on five different test suites derived from the original EBNF form of the SIMPL grammar. The first two test suites, *rule* and *cdrc*, contain only passing test cases. They are constructed according to the rule and cdrc coverage criteria [26], respectively, and contain 43 and 86 test cases, respectively. Note that *rule* is a proper subset of *cdrc*. The test suite *cdrc+nlr* contains the 86 positive test from *cdrc* and 2522 negative tests that are constructed using an implementation of Zelenov's negative LR algorithm [57]. *large* is very large, varied test suite that contains 2964 positive tests and 32157 negative tests. The positive tests are constructed according to four different coverage criteria (*cdrc2*, *step6*, *deriv*, and *pair*) we developed to produce diverse test suites. The negative tests are constructed using token mutation over the *rule* test suite, and using mutation of the rules themselves [42]. *instructor* refers to the test suite the instructor used to grade the student submissions. It comprises 20 (syntactically) positive and 61 negative tests.

A grammar mutant is killed by a test suite if the parser fails on at least one test case; however, we consider a mutant *not* killed if the parser fails on all test cases, because the metrics then become undefined or degenerate as discussed in Section 2. For ANTLR, we also considered a mutant as not killed if it requires the application of a left-recursive rule, because the computed grammar spectra are known to be wrong (see the discussion in Section 4.1).

For each grammar mutant killed by the test suite we ordered the rules by the scores produced by each of the ranking metrics and computed the mutated rule's predicted rank. We resolved ties using the middle rank, as discussed in Section 2.

Results. Figure 4 shows the results of these fault seeding experiments as a series of boxplots. Each boxplot summarizes the ranks predicted by the corresponding metric for the mutated (i.e., faulty) rules, given a specific parsing method and test suite. The boxes show the Q3/Q1 interquartile range of the ranks, i.e., the upper end of the box corresponds to the 75th percentile (i.e., in 75% of the cases the faulty rule is ranked better than the indicated value) while its lower end corresponds to the 25% percentile. The median is indicated by a dotted line across the box. The"whiskers" extend from the 5th to the 95th percentile. Table 3 contains more details.

While the details change with the applied parsing technology and ranking metric, and the underlying test suite, Figure 4 and Table 3 show overall positive results. On average, the metrics rank the faulty rules at ∼25% of all rules (i.e., within the top 20 of 80 rules), with better results for the *large* test suite (∼15%) and worse results for the *instructor* test suite (∼35%). The median is typically at 2.5%–5%, and so much smaller than the mean. Hence in more than half of the cases the metrics rank the faulty rule within the top five rules, and in 10%–40% of the cases they correctly pinpoint it.

While we have not statistically analyzed the results in detail, a few observations can be made. First, fault localization works better for CUP than for ANTLR: for CUP we universally achieve lower mean and median values, independent of the test suite and the ranking metric, and typically pinpoint a higher fraction of the observed faults (with Tarantula the only metric under which results for ANTLR are sometimes better than those for CUP).

Second, ANTLR's error correction introduces noise into the spectra that compromises the quality of the fault localization. ANTLR with bail-out on error uniformly produces better results than ANTLR* with error correction, although the differences are smaller than between ANTLR and CUP.

Third, without detailed statistical analysis there is no clear winner visible between Ochiai, Jaccard, and DStar, but all three seem to outperform Tarantula, except for the *large* test suite, where Tarantula produces the tightest interquartile range and the lowest mean (although not the lowest median nor the highest fraction of top-ranked faults).

Fourth, the localization performance depends strongly on the size and variance of the test suite. The difference of the results between the *rule* and *cdrc* test suites that contain very similar positive test cases is marginal, despite the fact that *cdrc* includes *rule*. In contrast, both of them induce substantially better results than the manually constructed *instructor* test suite whose size is between both of them. This also indicates that it is hard to manually construct test suites that are well suited for fault localization.

Finally, our simple pass/fail oracle seems to be too simple for negative test cases, as adding the *nlr* tests to the *cdrc* test suite degrades the localization performance.

Threats to validity. In addition to the usual concerns about construct (i.e., implementation and data collection errors) and statistical conclusion validity, we see several threats to the validity of generalizing our observations above beyond the experimental setup, e.g., to other ranking metrics, parsing methods, grammars, or test suites. Our fault seeding experiments are based on a single grammar; since test suite construction, mutant construction, and spectrum collection all depend on the structure of the grammar, different grammars may yield different results. We encountered this when we used an ANTLR version that was not left-factorized, which triggered the rule tracking issues described in Section 4.1 and led to incomplete spectra that distorted the results. Our fault seeding also includes mutations at the first symbol of a rule, which may produce non-LL(1) mutants that also trigger the rule tracking issues and so distort results; preliminary analysis has shown that the localization performance can differ by 15 rules (i.e., close to 20 percentage points) between mutations at the first and at other symbols. We will conduct further experiments to address this threat.

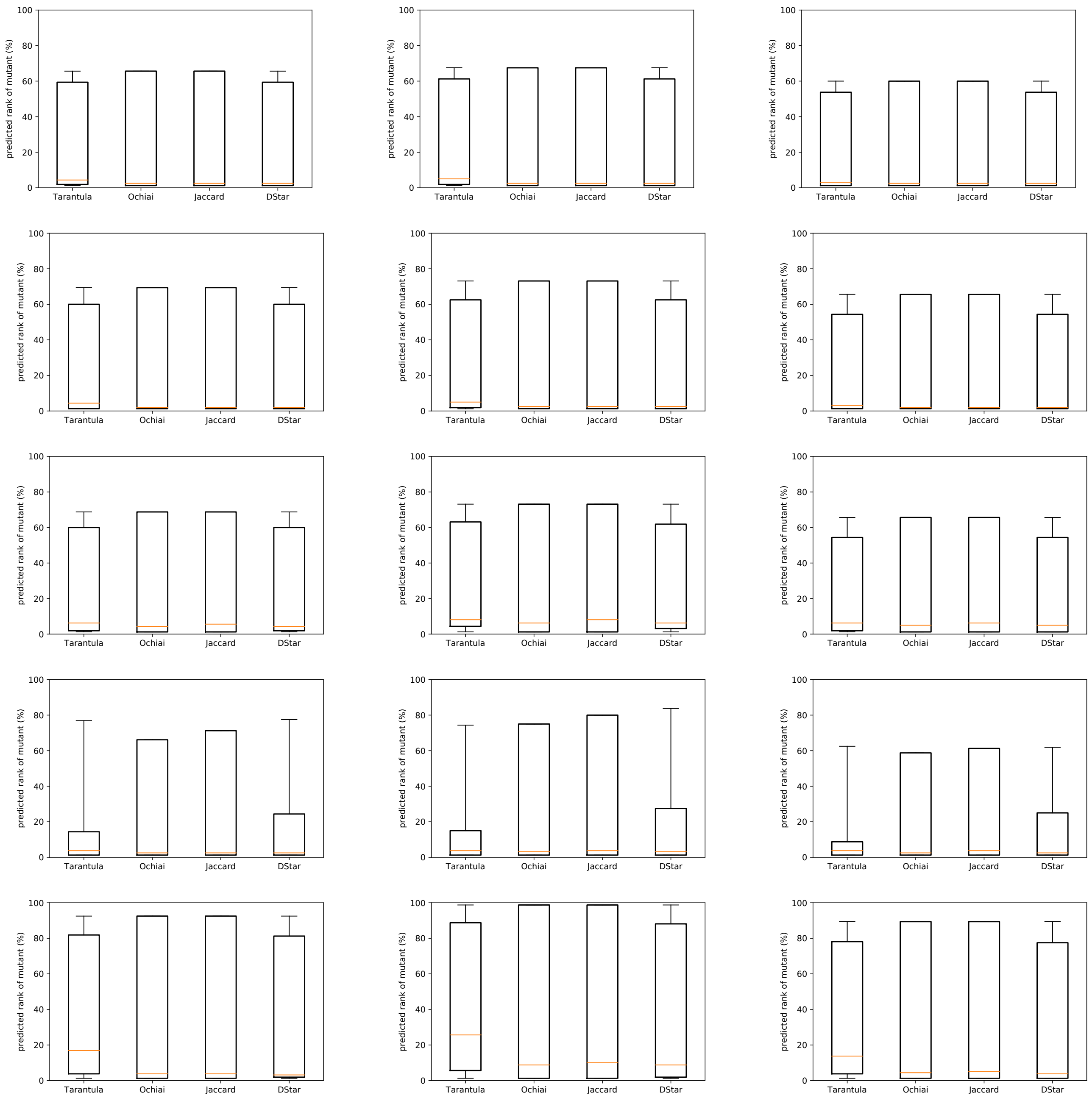

Figure 4. Results of fault seeding experiments over SIMPL grammar. Columns show results for different parsers, left to right: ANTLR (without error correction), ANTLR* (with default error correction) and CUP. Rows show results for different test suites, top to bottom: *rule* (43 positive tests), *cdrc* (86 positive tests), *cdrc+nlr* (86 positive tests, 2522 negative tests), *large* (2964 positive tests, 32157 negative tests), *instructor* (20 positive tests, 41 negative tests). Table 3 contains more details.

Gopinath et al. [17] have shown that mutants are not syntactically close to real faults, but there is evidence that they are nevertheless a valid substitute in many software engineering applications, including fault localization [24]. However, grammar mutations as we have used here have not been investigated, and other mutation operations (e.g., adding epsilon-productions or deleting entire rules) may yield different results. Hence, even though our localization experiments with student grammars (see Section 5.3) show similar results, care should be taken in generalizing the results above.

The experiments have shown that the localization performance depends on the test suites and may thus not generalize, despite the differences in the test suites we have used. The

Table 3. Detailed results of fault seeding experiments over SIMPL grammars. $\tilde{x}$ and $\bar{x}$ denote the median and mean rank, respectively, of the seeded fault. #1 denotes the number of cases where the metric ranked the seeded fault as most suspicious.

			Tarantula			Ochiai			Jaccard			DStar		
		killed	$\tilde{x}$	$\bar{x}$	*#1*	$\tilde{x}$	$\bar{x}$	*#1*	$\tilde{x}$	$\bar{x}$	*#1*	$\tilde{x}$	$\bar{x}$	*#1*
ANTLR	*rule*	25630	4.4%	25.7%	6313	2.5%	24.9%	7257	2.5%	24.9%	7260	2.5%	24.9%	7251
	cdrc	25789	4.4%	25.9%	6649	1.9%	25.3%	7368	1.9%	25.4%	7369	1.9%	25.4%	7360
	cdrc+nlr	27675	6.2%	26.7%	5596	4.4%	25.2%	6418	5.6%	26.1%	5684	4.4%	25.0%	6806
	large	29265	3.8%	15.1%	9420	2.5%	14.8%	9368	2.5%	14.7%	9496	2.5%	16.9%	9006
	instructor	27557	16.9%	36.3%	3806	3.8%	32.3%	6674	3.8%	32.5%	6493	3.1%	32.3%	6668
ANTLR*	*rule*	25594	5.0%	25.7%	5812	2.5%	24.6%	6886	2.5%	24.6%	6887	2.5%	24.6%	6866
	cdrc	25753	5.0%	25.9%	5994	2.5%	24.9%	7016	2.5%	25.0%	7016	2.5%	25.0%	7005
	cdrc+nlr	27637	8.1%	29.3%	2518	6.3%	26.5%	3320	8.1%	28.0%	2703	6.3%	26.2%	3356
	large	29227	3.8%	15.8%	9101	3.1%	16.5%	8123	3.8%	16.4%	8376	3.1%	18.8%	7801
	instructor	27521	25.6%	42.7%	2170	8.8%	34.6%	4625	10.0%	34.8%	4538	8.8%	34.6%	4938
CUP	*rule*	23650	3.1%	23.9%	7855	2.5%	23.2%	9992	2.5%	23.3%	9992	2.5%	23.3%	9992
	cdrc	25429	3.1%	24.2%	8445	2.5%	23.6%	10590	2.5%	23.6%	10588	2.5%	23.6%	10586
	cdrc+nlr	25429	6.3%	25.7%	5718	5.0%	24.4%	6561	6.3%	24.8%	5913	5.0%	24.4%	6590
	large	26308	3.8%	11.7%	8773	2.5%	13.8%	10156	3.8%	14.0%	9512	2.5%	15.7%	10276
	instructor	25537	13.8%	35.1%	2688	4.4%	32.3%	6913	5.0%	32.7%	6525	3.8%	32.2%	7457

large test suite contains tests that are constructed based on the same principle as the mutants (i.e., rule mutation) and may thus overestimate performance.

5.2 Real-world Grammars

In a second experiment, we compared two real-world Pascal grammars with each other. These grammars have been derived from different original sources, and even though they both describe the "Pascal" language they contain differences, as shown in previous work [31]. Due to the considerable freedom the CFG formalism allows grammar developers, these differences are difficult to spot; even a different terminal set does not necessarily indicate differences (e.g., one of the grammars defines specific terminal symbols for the basic types such as `BOOLEAN` while the other subsumes them under identifiers). The purpose of this experiment is therefore to evaluate whether our method can localize differences in real-world grammars and so improve on the state-of-the-art, which can only find counterexamples [31] or approximately match non-terminals [16].

Experimental setup. In this experiment we used essentially the same Pascal grammars as Madhavan et al. [31]. More specifically, we used the YACC grammar from ftp://ftp.iecc.com/pub/file/pascal-grammar, which we converted into CUP format, and the ANTLR (v4) grammar from github.com/antlr/grammars-v4/blob/master/pascal/pascal.g4, which was converted from the ANTLR (v3) version used by Madhavan et al. [31]. Table 4 summarizes the characteristics of these grammars.

Since neither of the two grammars can be considered as true version, we followed a differential testing approach, i.e., we first used the BNF-variant of one grammar as "golden" version or source to generate a test suite satisfying *cdrc* coverage, which we then used to localize "faults" (more precisely, deviations) in the other grammar. We finally switched roles and ran the same experimental setup in the other direction.

In each direction, we used an iterative process to identify deviations. In each step, we used the Tarantula metric to compute suspiciousness scores for all rules from the current test suite, starting with the full *cdrc* test suite. We manually identified a clearly visible cut-off value in the scores of the highest ranked rules (typically the rules tied at top rank), and picked one rule $A \longrightarrow \alpha$ above the cut-off. We computed its "suspicious closure", i.e., the set of all rules $B \longrightarrow \beta$ where either $A \in \beta$ or $B \in \alpha$ that are also ranked above the cut-off. We then selected all failing test cases in which any of these rules was applied, and used the *cdrc* test targets (i.e., the rule in whose right-hand side the occurrence of a non-terminal is replaced as well as the rule that is used in the replacement) to identify the corresponding fragment in the source grammar. We manually inspected the two fragments to identify the cause of the deviation. We finally removed the involved failing test cases from the test set and repeated this process until Tarantula did not clearly identify any further suspicious rules or the test set became empty.

Table 4. Characteristics of Pascal grammars (original and BNF versions) and test suites

Type	$\|T\|$	$\|N\|$	$\|P\|$	$\|N_{bnf}\|$	$\|P_{bnf}\|$	$\|cdrc\|$
CUP	61	79	176	121	219	342
ANTLR	71	97	156	210	322	194

Results: CUP → ANTLR. Using the CUP grammar as source and the ANTLR grammar as target gave us an initial set of 51 failing tests out of a total of 342 tests. In the first iteration, Tarantula scored 14 rules tied at top rank and another three rules above a cut-off value of 0.99. We picked the rule defining `formalParamList`, which yields a suspicious closure group comprising the following nine rules (note that we shortened some non-terminal names for layout reasons; we also show in parentheses the Tarantula score and the number of passing and failing tests for each rule):

```
procedureAndFunctionDeclarationPart:  (0.99: 1/21)
  procedureOrFunctionDeclaration SEMI;
procedureOrFunctionDeclaration :
  procedureDeclaration                (0.99: 1/16)
| functionDeclaration;                (1.00: 0/16)
procedureDeclaration:                 (0.99: 1/16)
```

```
  PROCEDURE id (formalParamList)? SEMI block;
functionDeclaration :                (1.00: 0/5)
  FUNCTION id (formalParamList)?
  COLON resultType SEMI block;
resultType: typeId;                  (1.00: 0/1)
formalParamList:                     (1.00: 0/16)
  LPAR formalParamSection
  (SEMI formalParamSection)* RPAR;
formalParamSection:
  paramGroup                         (1.00: 0/9)
| VAR paramGroup                     (1.00: 0/2)
| FUNCTION paramGroup                (1.00: 0/7)
| PROCEDURE paramGroup;              (1.00: 0/2)
paramGroup: idList COLON typeId;     (1.00: 0/16)
```

The rules in this group are applied in 16 different failing test cases. One of them is

```
  PROGRAM A0;
    PROCEDURE A0(FUNCTION A0; FUNCTION A0); A0;
  BEGIN
  END.
```

which results from three different *cdrc*-targets[1]

$$\begin{aligned}&\mathit{formal\_params} \rightarrow \textbf{(}\,\mathit{formal\_p\_sects}\,\textbf{)}\\ &\quad @_1\ \mathit{formal\_p\_sects} \rightarrow \mathit{formal\_p\_sects}\ \textbf{;}\ \mathit{formal\_p\_sect}\\ &\mathit{formal\_p\_sects} \rightarrow \mathit{formal\_p\_sects}\ \textbf{;}\ \mathit{formal\_p\_sect}\\ &\quad @_0\ \mathit{formal\_p\_sects} \rightarrow \mathit{formal\_p\_sect}\\ &\mathit{formal\_p\_sects} \rightarrow \mathit{formal\_p\_sects}\ \textbf{;}\ \mathit{formal\_p\_sect}\\ &\quad @_2\ \mathit{formal\_p\_sect} \rightarrow \mathit{func\_heading}\end{aligned}$$

and so traces through to four rules. Tracing through all of the 16 tests gives us a corresponding fragment of the CUP source grammar comprising of 16 rules.

```
proc_heading ::= PROCEDURE newident formal_params;
func_heading ::= FUNCTION  newident function_form;
function_form ::= | formal_params COLON ident;
formal_params ::= | LPAR formal_p_sects RPAR;
formal_p_sects ::=
  formal_p_sects SEMI formal_p_sect
| formal_p_sect;
formal_p_sect ::=
  param_group | VAR param_group
| proc_heading | func_heading;
param_group ::= newident_list COLON paramtype;
paramtype ::=
  ident
| ARRAY LBRAC index_specs RBRAC OF paramtype
| PACKED ARRAY LBRAC index_spec RBRAC OF ident;
```

A visual comparison of the two fragments then uncovers two differences. First, the CUP definition for param_group uses paramtype after the COLON, which includes arrays and packed arrays, while the corresponding ANTLR definition paramGroup uses typeId, which only allows identifiers or the pre-defined type names such as CHAR (which are subsumed by identifiers in CUP). Second, the CUP definition for formal_p_sect uses proc_heading and func_heading, which allows anonymous types for formal parameters of procedure resp. function type, while the ANTLR grammar requires explicitly defined types that are referenced by their identifier.

In total, these two differences explain 14 of the 16 failing test cases; we removed these from the test set and continued with the second iteration. Here, Tarantula scored five rules at top rank, and we picked variantPart, which induces a suspicious closure group comprising the following five rules (note that the first alternative for fieldList is not part of the group but simply shown for completeness):

```
fieldList:
  fixedPart (SEMI variantPart)?     (0.59: 4/1)
| variantPart;                      (1.00: 0/10)
variantPart:                        (1.00: 0/10)
  CASE tag OF variant (SEMI variant)*;
tag:
  id COLON typeId                   (1.00: 0/1)
| typeId;                           (1.00: 0/9)
variant:                            (1.00: 0/10)
  constList COLON LPAR fieldList RPAR;
```

Tracing through the *cdrc*-targets of the ten failing test cases gives us again the corresponding CUP grammar fragment (note that only the RECORD-alternative of struct_type is part of the fragment):

```
struct_type ::= ... | RECORD field_list END;
field_list ::=
  fixed_part
| fixed_part SEMI variant_part
| variant_part;
fixed_part ::=
  fixed_part SEMI record_section | record_section;
record_section ::= | newident_list COLON type;
variant_part ::= CASE tag_field OF variants;
tag_field ::= newident COLON ident | ident;
variants ::= variants SEMI variant | variant;
variant ::=
  | case_label_list COLON LPAR field_list RPAR;
```

We can then see the difference: the CUP grammar allows an empty list of variants in the variant_part while the ANTLR grammar requires a non-empty list.

In the third iteration, Tarantula identifies the rules

```
procedureDeclaration:
  PROCEDURE id (formalParamList)? SEMI block
functionDeclaration:
  FUNCTION id (formalParamList)?
  COLON resultType SEMI block;
```

as highly suspicious. The corresponding rules can easily be identified in the CUP grammar. They use a non-terminal body which is defined as block | IDENTIFIER instead of block; comments in the grammar explain that the identifier represents the FORWARD-directive but that is not enforced by the parser.

In the next two iterations, Tarantula identifies problems with the fixed part of records and case list elements that are similar to the problem with variant records identified in the

[1]Here the notation $A \rightarrow \alpha\ @_i\ B \rightarrow \beta$ denotes the the modified rule $A \rightarrow \alpha'$ obtained by replacing the non-terminal B at position i in α by β; *cdrc*-coverage is then equivalent to *rule*-coverage with all such *cdrc*-replacements.

second iteration. For example, CUP allows its `fixed_part` to become empty before the `SEMI` token, and hence admits a type `RECORD ; END` while ANTLR rejects this type. After the fifth iteration, the spectra become too noisy and we cannot see any clearly suspicious rules anymore; we therefore terminate the analysis.

In total, the fault localization allowed us to restrict our attention to 25 (out of 156) ANTLR rules to identify five grammar differences. Tracing through the *cdrc* targets restricted the number or CUP rules we needed to inspect to 45 out of 176 rules.

Results: ANTLR → CUP. We then switched roles and used the ANTLR grammar as source from which we derived the test suite and the CUP grammar as fault localization target. We get an initial set of 47 failing tests out of a total of 194 tests. Using the same iterative approach as above, Tarantula ranks in the first iteration the two rules `type_def` and `type_dcl_part` at the very top. These two rules have been applied in 25 of the failing tests.

```
type_dcl_part ::= TYPE type_defs SEMI;
type_defs ::= type_defs SEMI type_def | type_def;
type_def ::= newident EQ type;
type ::= simple_type
| PACKED struct_type
| struct_type
| CAP ID;
```

Here, the structure of the failing tests such as

```
 PROGRAM id;
   TYPE id = PROCEDURE(FUNCTION id : BOOLEAN);
 BEGIN
 END.
```

gives us already a clue, as they all use function or procedure type declarations. Tracing through the *cdrc*-targets to the the corresponding fragment of the ANTLR source grammar confirms that this has explicit function and procedure type declarations while the CUP grammar only allows them in formal parameter lists (cf. the related discussion above):

```
typeDefinitionPart: TYPE (typeDefinition SEMI)+;
typeDefinition:
  id EQUAL (type | functionType | procedureType);
```

In the later iterations, the fault localization becomes less discriminatory, largely because the ANTLR grammar describes more features of the language than its CUP counterpart (for example, a rudimentary module system using the `UNIT` and `INTERFACE` keywords).

Threats to validity. This experiment is subject to the similar threats to validity as the one described in the previous section; in particular, the results may not generalize to other pairs of grammars or to other ranking metrics. However, as mitigation we used a broadly similar setup in the experiments described in the following section, where we achieved similar results.

Since the setup involves human judgements by the authors, the results are also subject to possible experimenter bias, human error, and human performance variation. We tried to mitigate against this threat by following an experimental protocol over unseen grammars, but this was not fully defined (e.g., rule selection and choice of the cut-off points).

5.3 Student Grammars

In a final set of experiments, we used student submissions (which unsurprisingly contain many errors) to compiler engineering courses at two different universities to see how well our method performs over grammars with multiple real faults.

Experimental setup. We used two languages, SIMPL (which we also used for the fault seeding experiments in Section 5.1), and Blaise, another imperative programming language of similar syntactic complexity: the instructor's version of the grammar has 38 non-terminals, 40 terminals and 75 rules. For SIMPL, we used the same positive test cases as in the *large* test suite in Section 5.1 and the 8668 negative test cases derived using rule mutation. For Blaise, we generated tests using the same mechanism; this comprises 7280 positive and 9119 negative test cases.

Both languages were used in compiler engineering courses. In one course, the students were given the same EBNF as in the computer architecture course (in fact, most students were from a cohort that already used SIMPL in that course), and were asked in two different assignments to use ANTLR and CUP (or a similar LALR(1) parser generator of their choice) to develop parsers for SIMPL. We randomly picked ten ANTLR submissions, from which we discarded three that pass all tests and one that did not produce a compilable parser. We picked all ten CUP submissions, from which we discarded two that pass all tests and one that passes none. For Blaise, the students were given a textual language description and a small set of short example programs. We randomly picked nine Blaise grammars from 110 submissions, from which we also discarded two that pass all tests. This leaves us with 20 subject grammars.

We then followed a similar iterative fault localization process as described in the previous section. In each step, we used the Ochiai metric to compute the suspiciousness scores of the rules. We manually examined the rules in rank order and used our understanding of the true grammars to identify and repair faulty rules. In each step, we only repaired the top-ranked faulty rule; note that we made repairs in the lexer as well. After each repair, we continued with the next iteration, until the grammar under test passed all test cases.

Results. Table 5 summarizes the results of our evaluation over student grammars. For space reasons, we do not show one case that required eleven iterations. For each iteration, we show the number of test cases failed by that grammar version, and the rank of the rule that we identified as faulty and repaired for the next iteration. This rank is based on an optimistic tie-breaking strategy. Empty cells indicate that a previous repair allowed the parser to pass all tests.

While we have no guarantee that we always pick the "right" rule for repair, we can observe for all but one grammar the number of failed test cases reduces after each repair; the exception is #8, where the repair in iteration 2 triggers more failing test cases. Here, the repair can be seen as the first step in a multi-step refactoring that temporarily increases the number of failures, which then drops significantly in the subsequent iterations. Note also that the final repair of #6 has no associated rank because the error was actually in the lexer which returned an identifier token instead of the AND-operator. In other cases, we could identify similar lexical errors via the rules.

The most common issues are around the formal and actual parameter lists of functions. These cannot be empty, but the textual language specification was vague about this, and many students interpreted this differently.

6 Related Work

We are not aware of any other work directly sharing our goal of identifying faulty rules in a grammar but we draw from a wide range of related work in different areas. Note, however, that we co not consider traditional error recovery methods in parsers [15] because they work on the input or the parser state, not on the grammar.

Spectrum-based fault localization. Many different methods (e.g., static analysis, model-based reasoning, or deep learning) have been applied to the problem of software fault localization; Wong et al. [53] give a good survey of the entire field. We focus on spectrum-based methods only [13, 53].

More than 30 different metrics have been proposed for statement ranking [13, 36, 53], which were often originally developed for problems in other domains such as botany [38] or information retrieval. Many metrics produce identical rankings [14, 36]. Theoretical studies [54] trying identify optimal metrics have not been borne out in practice [28]. We use four of the most widely used metrics that have also performed well in other experimental evaluations [3, 28]: Tarantula [23], Ochiai [38], Jaccard [12], and DStar [52]. Tarantula is the only of these metrics that takes program entities into account that are *not* executed in passing test cases (see Table 4); however, experimental results [3, 52] show that Ochiai, Jaccard, and DStar are more effective for software fault localization. Abreu et al. [3] report the following average ranks of the faulty statement over seven programs (with a size of 20-124 basic blocks) from the widely used Siemens benchmark suite: Tarantula 23%, Jaccard 22%, and Ochiai 7%, with individual results varying between 1% and 50%.

Grammar smells. Faults in grammars can manifest themselves in non-terminal symbols that are non-productive or unexpectedly nullable, or in ambiguities that could be resolved unexpectedly by a (deterministic) parser. While non-productivity and nullability are easily checkable, ambiguity is undecidable in general [9], although several practical approximations have been developed [5, 8, 46–48]. Basten's approach [5] identifies rules that are provably not involved in an ambiguity and so helps with localization. LR parser generators typically report any shift/reduce and reduce/reduce conflicts that they encounter; Isradisaikul and Myers [22] produce "unifying counterexamples" for such situations that can help users to debug their grammars.

However, none of these approaches can really be seen as fault localization, because the situations that they detect are *grammar smells* rather than necessarily faults. Consider for example the traditional "dangling else" problem [4]. Most LR parsers resolve the ambiguity indicated through shift/reduce conflict by shifting, and so accept the intended language.

Grammar equivalence. Proving the equivalence of the grammar under test to a given "golden" grammar can be seen as an alternative to fault localization, similar to the way proving a program correct is an alternative to testing. CFG equivalence is of course well-known to be undecidable in general, but decision algorithms have been developed for several relevant subclasses, e.g., simple [6, 25], LL(k) [39], or LL-regular grammars [37]. Madhavan et al. [31] describe a system that implements several of these algorithms and can produce counter-examples when it finds that the grammars are not equivalent. Fischer et al. [16] uses systematic test case generation and parsing to identify which non-terminals accept the most similar languages, which can be seen as an approximate, fine-grained equivalence check.

Grammar-based test case generation. Purdom's seminal paper [41] on the systematic generation of test suites from grammars (see Malloy and Power [32] for a modern reformulation) describes an algorithm that generates the minimal number of sentences that is necessary to exercise all grammar rules. Celentano et al. [10] extend Purdom's algorithm, providing both a minimal and a maximal strategy for generating sentences. Such test suites are not well suited for error localization because the generated test suites are too small and the individual test cases are too complex, and cover too many rules.

Laemmel [26] defines context-dependent rule coverage, which requires each rule to be applied to each non-terminal occurrence in the grammar; this yields more detailed test suites and has become a standard coverage criterion. In addition, we use a number of coverage criteria that we recently developed to produce diverse positive test suites: *cdrc2*, a variant of *cdrc* that induces longer words; *step6*, another variant of *cdrc* that induces deeper derivations; *deriv*, a variation of Zelenov and Zelenova's *pll* criterion [57] that also induces deeper derivations; and *pair*, a criterion that ensures that all possible pairs in the follow-relation are covered.

Table 5. Results of fault localization in student grammars

			iteration 1		iteration 2		iteration 3		iteration 4		iteration 5		iteration 6	
#	language	type	#fail	rank	#fail	rank	#fail	rank	#fail	rank	#fail	rank	#fail	rank
1	SIMPL	CUP	557	1	254	1	131	1	98	1				
2	SIMPL	CUP	206	2	95	2								
3	SIMPL	CUP	498	1	40	1								
4	SIMPL	CUP	169	1	46	1								
5	SIMPL	CUP	853	1	378	1	219	1	130	1	37	1	6	1
6	SIMPL	CUP	244	1	121	9	80	✗						
7	Blaise	ANTLR	567	2	4	1	2	1						
8	Blaise	ANTLR	1082	1	535	3	7213	1	358	1	43	1	2	1
9	Blaise	ANTLR	4	3	2	2								
10	Blaise	ANTLR	1068	1	4	2	2	1						
11	Blaise	ANTLR	38	4	3	1								
12	Blaise	ANTLR	654	1	1	1								
13	Blaise	ANTLR	4	2	2	1								
14	SIMPL	ANTLR	555	1	170	1	47	2	1	1				
15	SIMPL	ANTLR	37	5	1	1								
16	SIMPL	ANTLR	361	3	46	1								
17	SIMPL	ANTLR	396	1	117	2	81	2	47	1	1	1		
18	SIMPL	ANTLR	46	2										
19	SIMPL	ANTLR	356	1	233	2	1	1						

Grammar-based test case generation has focused mostly on generating syntactically correct programs (i.e., positive tests), with scant attention paid to generation of programs with well-defined syntactic errors (i.e., negative tests). We use a variant of Zelenov and Zelenova's [57] *nlr* algorithm here. We have also developed two algorithms that construct negative test suites using word and rule mutation [42].

In random sentence generation (which goes back to [18]), rules are randomly selected and applied until a complete sentence is derived. Such approaches typically use a large number of control parameters (e.g., rule probabilities, symbol and rule counts, length, depth, and balance restrictions, and many others) to ensure that the derivation process terminates, and that the generated test suites have certain characteristics [7, 19, 21, 27, 33, 34, 40]. Such methods have been used to test SQL [50], C [55], and Java [56] processors, and are also applied in some fuzzing tools such as jsfunfuzz [44], the CSS grammar fuzzer [45], or langfuzz [20].

Differential compiler testing. Differential testing [35] compares the outputs of two different systems implementing the same specification and flags errors whenever they disagree. It is an appealing technique in domains such as compilers where precise oracles are difficult to construct but multiple systems exist [49]. Since many compiler bugs lurk in the optimizer differential testing with the same compiler but different optimization levels is an easily implementable approach [11]. Le et al. [29] describe another approach that mutates test programs such that the mutants are guaranteed to be semantically equivalent for the given test inputs (and so should produce the same outputs) but are structurally different and may so trigger different compilation paths. The experiment described in Section 5.2 can also be seen as differential testing.

7 Conclusions and Future Work

Grammars can contain bugs like any other software. Testing can demonstrate the presence of bugs in grammars, but does not directly give any further information about their location. In this paper, we described and evaluated the first method to localize faulty rules in a context-free grammar. We proposed a spectrum-based fault localization method where we replaced the concept of "executed statements" by that of "used rules", but kept the remaining established framework in place.

Our evaluation showed that our method can identify grammar bugs with a high precision. In a large fault seeding experiment, it ranked the seeded faults within the top five rules in more than half of the cases and pinpointed them (i.e., uniquely ranked them as most suspicious) in 10%–40% of the cases. On average, it ranked the faults at around 25% of all rules, and better than 15% for a very large test suite. We were also able to identify deviations and faults in real world and student grammars, which contain multiple real faults. Our work therefore improves on the state-of-the-art, which can only find counterexamples [31].

Future work. In addition to extending our experimental evaluation to further parsers and languages, we see several interesting avenues for future work. First, we plan to analyze ANTLR's adaptive LL(*) parsing mechanism in detail to see whether we can extract better spectra for non-LL(k) grammars. Second, the mutation results in Section 5.1 indicate that our handling of negative test cases may be too simplistic; we will use test suites where the expected outcome includes the error location. Third, we will modify the ranking functions to take the structure of the grammar into account, in particular to replace the mid-rank tie breaking strategy. Both steps should increase the precision of the localization. We will also cluster the failed test cases by the ranked rules; this may help users in pinpointing and ultimately repairing the actual fault. Finally, we plan to investigate how approaches to automated program repair [30] can be applied to grammars.

References

[1] 2014. CUP 0.11b. http://www2.cs.tum.edu/projects/cup/

[2] 2018. ANTLR 4.7.2. https://www.antlr.org/

[3] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2006. An Evaluation of Similarity Coefficients for Software Fault Localization. In *12th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2006), 18-20 December, 2006, University of California, Riverside, USA*. IEEE Computer Society, 39–46. https://doi.org/10.1109/PRDC.2006.18

[4] Alfred V. Aho, Monica S. Lam Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (Second Edition)*. Addison-Wesley.

[5] Hendrikus J. S. Basten. 2010. Tracking Down the Origins of Ambiguity in Context-Free Grammars. In *Theoretical Aspects of Computing - ICTAC 2010, 7th International Colloquium, Natal, Rio Grande do Norte, Brazil, September 1-3, 2010. Proceedings (Lecture Notes in Computer Science)*, Ana Cavalcanti, David Déharbe, Marie-Claude Gaudel, and Jim Woodcock (Eds.), Vol. 6255. Springer, 76–90. https://doi.org/10.1007/978-3-642-14808-8_6

[6] Cédric Bastien, Jurek Czyzowicz, Wojciech Fraczak, and Wojciech Rytter. 2006. Prime normal form and equivalence of simple grammars. *Theor. Comput. Sci.* 363, 2 (2006), 124–134. https://doi.org/10.1016/j.tcs.2006.07.021

[7] D. L. Bird and C. U. Munoz. 1983. Automatic generation of random self-checking test cases. *IBM Systems Journal* 22, 3 (1983), 229–245. https://doi.org/10.1147/sj.223.0229

[8] Claus Brabrand, Robert Giegerich, and Anders Møller. 2010. Analyzing ambiguity of context-free grammars. *Sci. Comput. Program.* 75, 3 (2010), 176–191. https://doi.org/10.1016/j.scico.2009.11.002

[9] David G. Cantor. 1962. On The Ambiguity Problem of Backus Systems. *J. ACM* 9, 4 (1962), 477–479. https://doi.org/10.1145/321138.321145

[10] Augusto Celentano, Stefano Crespi-Reghizzi, Pierluigi Della Vigna, Carlo Ghezzi, G. Granata, and Florencia Savoretti. 1980. Compiler Testing using a Sentence Generator. *Softw., Pract. Exper.* 10, 11 (1980), 897–918. https://doi.org/10.1002/spe.4380101104

[11] Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2016. An empirical comparison of compiler testing techniques. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie Williams (Eds.). ACM, 180–190. https://doi.org/10.1145/2884781.2884878

[12] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric A. Brewer. 2002. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *2002 International Conference on Dependable Systems and Networks (DSN 2002), 23-26 June 2002, Bethesda, MD, USA, Proceedings*. IEEE Computer Society, 595–604. https://doi.org/10.1109/DSN.2002.1029005

[13] Higor Amario de Souza, Marcos Lordello Chaim, and Fabio Kon. 2016. Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges. *CoRR* abs/1607.04347 (2016). arXiv:1607.04347 http://arxiv.org/abs/1607.04347

[14] Vidroha Debroy and W. Eric Wong. 2011. On the equivalence of certain fault localization techniques. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), TaiChung, Taiwan, March 21 - 24, 2011*, William C. Chu, W. Eric Wong, Mathew J. Palakal, and Chih-Cheng Hung (Eds.). ACM, 1457–1463. https://doi.org/10.1145/1982185.1982498

[15] Lukas Diekmann and Laurence Tratt. 2018. Reducing Cascading Parsing Errors Through Fast Error Recovery. *CoRR* abs/1804.07133 (2018). arXiv:1804.07133 http://arxiv.org/abs/1804.07133

[16] 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. https://doi.org/10.1007/978-3-642-28830-2_18

[17] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Mutations: How Close are they to Real Faults?. In *25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014, Naples, Italy, November 3-6, 2014*. IEEE Computer Society, 189–200. https://doi.org/10.1109/ISSRE.2014.40

[18] Kenneth V. Hanford. 1970. Automatic Generation of Test Cases. *IBM Systems Journal* 9, 4 (1970), 242–257. https://doi.org/10.1147/sj.94.0242

[19] Daniel Hoffman, David Ly-Gagnon, Paul A. Strooper, and Hong-Yi Wang. 2011. Grammar-based test generation with YouGen. *Softw., Pract. Exper.* 41, 4 (2011), 427–447. https://doi.org/10.1002/spe.1017

[20] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, Tadayoshi Kohno (Ed.). USENIX Association, 445–458. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[21] William Homer and Richard Schooler. 1989. Independent Testing of Compiler Phases Using a Test Case Generator. *Softw., Pract. Exper.* 19, 1 (1989), 53–62. https://doi.org/10.1002/spe.4380190106

[22] Chinawat Isradisaikul and Andrew C. Myers. 2015. Finding counterexamples from parsing conflicts. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Steve Blackburn (Eds.). ACM, 555–564. https://doi.org/10.1145/2737924.2737961

[23] James A. Jones and Mary Jean Harrold. 2005. Empirical evaluation of the Tarantula automatic fault-localization technique. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, David F. Redmiles, Thomas Ellman, and Andrea Zisman (Eds.). ACM, 273–282. https://doi.org/10.1145/1101908.1101949

[24] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. 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*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 654–665. https://doi.org/10.1145/2635868.2635929

[25] A. J. Korenjak and John E. Hopcroft. 1966. Simple Deterministic Languages. In *7th Annual Symposium on Switching and Automata Theory, Berkeley, California, USA, October 23-25, 1966*. IEEE Computer Society, 36–46. https://doi.org/10.1109/SWAT.1966.22

[26] 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. https://doi.org/10.1007/3-540-45314-8_15

[27] 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. https://doi.org/10.1007/11754008_2

[28] Tien-Duy B. Le, Ferdian Thung, and David Lo. 2013. Theory and Practice, Do They Match? A Case with Spectrum-Based Fault Localization. In *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*. IEEE Computer Society, 380–383. https://doi.org/10.1109/ICSM.2013.52

[29] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 216–226. https://doi.org/10.1145/2594291.2594334

[30] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing

55 out of 105 bugs for $8 each. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE Computer Society, 3–13. https://doi.org/10.1109/ICSE.2012.6227211

[31] Ravichandhran Madhavan, Mikaël Mayer, Sumit Gulwani, and Viktor Kuncak. 2015. Automating grammar comparison. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 183–200. https://doi.org/10.1145/2814270.2814304

[32] Brian A. Malloy and James F. Power. 2001. An Interpretation of Purdom's Algorithm for Automatic Generation of Test Cases. In *1st ACIS Annual International Conference on Computer and Information Science*. http://eprints.maynoothuniversity.ie/6434/

[33] Peter M. Maurer. 1990. Generating Test Data with Enhanced Context-Free Grammars. *IEEE Software* 7, 4 (1990), 50–55. https://doi.org/10.1109/52.56422

[34] Peter M. Maurer. 1992. The Design and Implementation of a Grammar-based Data Generator. *Softw., Pract. Exper.* 22, 3 (1992), 223–244. https://doi.org/10.1002/spe.4380220303

[35] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107. http://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf

[36] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.* 20, 3 (2011), 11:1–11:32. https://doi.org/10.1145/2000791.2000795

[37] Anton Nijholt. 1982. The Equivalence Problem for LL- and LR-Regular Grammars. *J. Comput. Syst. Sci.* 24, 2 (1982), 149–161. https://doi.org/10.1016/0022-0000(82)90044-7

[38] Akira Ochiai. 1957. Zoogeographical studies on the soleoid fishes found in Japan and its neighbouring regions-II. *Bulletin of the Japanese Society of Scientific Fisheries* 22, 9 (1957), 526–530. https://doi.org/10.2331/suisan.22.526

[39] Tmima Olshansky and Amir Pnueli. 1977. A Direct Algorithm for Checking Equivalence of LL*(k)* Grammars. *Theor. Comput. Sci.* 4, 3 (1977), 321–349. https://doi.org/10.1016/0304-3975(77)90016-0

[40] A. J. Payne. 1978. A Formalised Technique for Expressing Compiler Exercisers. *SIGPLAN Not.* 13, 1 (Jan. 1978), 59–69. https://doi.org/10.1145/953428.953435

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

[42] 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 2019, Athens, Greece, October 21-22, 2019*. This volume.

[43] Manos Renieris and Steven P. Reiss. 2003. Fault Localization With Nearest Neighbor Queries. In *18th IEEE International Conference on Automated Software Engineering (ASE 2003), 6-10 October 2003, Montreal, Canada*. IEEE Computer Society, 30–39. https://doi.org/10.1109/ASE.2003.1240292

[44] Jesse Ruderman. 2007. Introducing jsfunfuzz. http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/

[45] Jesse Ruderman. 2009. CSS grammar fuzzer. http://www.squarefree.com/2009/03/16/css-grammar-fuzzer/

[46] Sylvain Schmitz. 2007. Conservative Ambiguity Detection in Context-Free Grammars. In *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings (Lecture Notes in Computer Science)*, Lars Arge, Christian Cachin, Tomasz Jurdzinski, and Andrzej Tarlecki (Eds.), Vol. 4596. Springer, 692–703. https://doi.org/10.1007/978-3-540-73420-8_60

[47] Sylvain Schmitz. 2008. An Experimental Ambiguity Detection Tool. *Electr. Notes Theor. Comput. Sci.* 203, 2 (2008), 69–84. https://doi.org/10.1016/j.entcs.2008.03.045

[48] Friedrich Wilhelm Schröer. 2001. AMBER, An Ambiguity Checker for Context-free Grammars. http://accent.compilertools.net/Amber.html

[49] Flash Sheridan. 2007. Practical testing of a C99 compiler using output comparison. *Softw., Pract. Exper.* 37, 14 (2007), 1475–1488. https://doi.org/10.1002/spe.812

[50] Donald R. Slutz. 1998. Massive Stochastic Testing of SQL. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, Ashish Gupta, Oded Shmueli, and Jennifer Widom (Eds.). Morgan Kaufmann, 618–622. http://www.vldb.org/conf/1998/p618.pdf

[51] Ian Sommerville. 2010. *Software Engineering (Ninth Edition)*. Pearson.

[52] W. Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. 2014. The DStar Method for Effective Software Fault Localization. *IEEE Trans. Reliability* 63, 1 (2014), 290–308. https://doi.org/10.1109/TR.2013.2285319

[53] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Trans. Software Eng.* 42, 8 (2016), 707–740. https://doi.org/10.1109/TSE.2016.2521368

[54] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. 2013. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Trans. Softw. Eng. Methodol.* 22, 4 (2013), 31:1–31:40. https://doi.org/10.1145/2522920.2522924

[55] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 283–294. https://doi.org/10.1145/1993498.1993532

[56] Takahide Yoshikawa, Kouya Shimura, and Toshihiro Ozawa. 2003. Random Program Generator for Java JIT Compiler Test System. In *3rd International Conference on Quality Software (QSIC 2003), 6-7 November 2003, Dallas, TX, USA*. IEEE Computer Society, 20. https://doi.org/10.1109/QSIC.2003.1319081

[57] Sergey V. Zelenov and Sophia A. Zelenova. 2005. Generation of Positive and Negative Tests for Parsers. *Programming and Computer Software* 31, 6 (2005), 310–320. https://doi.org/10.1007/s11086-005-0040-6