The Journal of Systems and Software 215 (2024) 112067

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

# Spectrum-based rule- and item-level localization of faults in context-free grammars☆

Moeketsi Raselimo[1], Bernd Fischer *,[1]

*Stellenbosch University, Stellenbosch, South Africa*

ARTICLE INFO

*Keywords:*
Software testing and debugging
Grammars and context-free languages

ABSTRACT

We describe and evaluate spectrum-based methods aimed at finding faults in context-free grammars. In their basic form, they take as input a test suite and a parser for the grammar that is modified to collect grammar spectra (i.e., the sets of grammar elements used in attempts to parse the individual test cases), and return as output a ranked list of suspicious elements. We define grammar spectra suitable for localizing faults on the level of the grammar rules (i.e., rule spectra) and the rules' individual symbols (i.e., item spectra), respectively. We show how both types of grammar spectra can be collected by both LL and LR parsers, and how the JavaCC, ANTLR, and CUP parser generators can be modified and used to automate the collection of the grammar spectra. We also show how grammar spectra can be synthesized directly from test cases derived from a grammar, and how such synthetic spectra can be used to localize differences between a grammar and a black-box system under test.

We first evaluate our approach over a large number of medium-sized single fault grammars, which we constructed by fault seeding from a common origin grammar. At the rule level, it ranks the rules containing the seeded faults within the top five rules in about 40%–70% of the cases, depending on the applied parsing technique, test suite, and ranking metric, and pinpoints them (i.e., correctly identifies them as unique most suspicious rule) in about 10%–30% of the cases, with significantly better results for the synthetic spectra. At the item level, our approach remains remarkably effective despite the larger number of possible locations, provided it is coupled with a simple tie-breaking strategy that prefers items with the right-most designated position over other items from the same rules in a tie. It typically ranks the seeded faults within the top five positions in about 30%–60% of the cases, and pinpoints them in about 15%–40% of the cases. This specialized item-level localization also significantly outperforms a simplistic extension of the rule-level localization, where all positions within a rule are given the same score.

We further evaluate our approach over grammars that contain real faults. We show that an iterative method can be used to localize and manually remove one by one multiple faults in grammars submitted by students enrolled in various compiler engineering courses; in most iterations, the top-ranked rule already contains an error, and no error is ranked outside the top five ranked rules. We finally apply our approach to a large open-source SQLite grammar and show where this version deviates from the language accepted by the actual SQLite system.

## 1. Introduction

Grammars are software, and can contain bugs like any other software. Testing can show the presence of bugs, but does not give any direct information about bug locations. Software fault localization (de Souza et al., 2016; Wong et al., 2016) builds on testing and tries to automatically identify likely bug locations. *Spectrum-based fault localization* (SFL) (Renieris and Reiss, 2003; Jones and Harrold, 2005; Abreu et al., 2006; Wong et al., 2014; Naish et al., 2011) executes the system under test (SUT) over a given test suite and records a *program spectrum*, a representation of the execution information for the SUT's individual program elements; SFL methods typically rely on *method coverage* or *statement coverage*, i.e., record whether a method has been called or not resp. whether a statement has been executed or not. From the spectrum, they then compute a *suspiciousness score* for each program element,

☆ Editor: Laurence Duchien.

* Corresponding author.
*E-mail addresses:* 22374604@sun.ac.za (M. Raselimo), bfischer@sun.ac.za (B. Fischer).

[1] Both authors have contributed equally to the paper.

https://doi.org/10.1016/j.jss.2024.112067
Received 8 March 2023; Received in revised form 15 March 2024; Accepted 18 April 2024
Available online 21 April 2024

which can be seen as the likelihood that that element contains a bug. The elements are then sorted by decreasing score so that the most likely bug locations are ranked first.

In this paper, we describe and evaluate spectrum-based methods to localize faults in a context-free grammar. We first consider localization at the level of grammar rules. We view a rule to be possibly faulty if it is applied in a derivation of a string that is accepted by a parser for the grammar but is outside the "true" language (which may be different), or vice versa, if it is applied in a partial derivation of a string that is rejected by the parser, but that is within the true language. This view fits well with spectrum-based fault localization: we keep the established framework in place and only replace the concept of "called methods" by that of "applied rules". We therefore introduce *rule spectra* to summarize which the grammar rules have been (partially) applied in an attempt to parse an input string. We show how rule spectra can be collected for both LL and LR parsers. One technical problem here is to identify the rules that have been applied only partially when an LR parser encounters a syntax error and thus does not execute the reduction that marks the completion of the rule application. We recover the missing rules from the items contained in the states that are on the parser stack when it encounters a syntax error.

We then refine our method to localize errors more precisely, at the level of the individual symbols in a rule. Our basic idea here is to use spectra over *items* (i.e., rules with designated positions) for localization. This exploits the fact that the designated position marks the boundary between the part of a rule that has already been processed successfully and the part that still needs to be processed; hence, we can assume that the error is at the symbol following the designated position. We show how item spectra can be collected for both LL and LR parsers; in both cases, items are collected (explicitly or implicitly) on shift operations. For LL parsers, we add the corresponding item whenever the parser consumes a token or returns from a parse function call. For LR parsers, we propose two different methods. In the first method, we also extend the extraction of rule spectra and add all other items corresponding to a rule on reduction with this rule; on encountering a syntax error, we extract and add the relevant items from the states on the parse stack. This amounts to a backwards-looking reconstruction of the items as they are shifted onto the stack. In the second, forward-looking, method we add all items associated with a state whenever that state is pushed onto the stack; note that this yields larger spectra than the backwards-looking method, but this apparent loss of precision does not necessarily translate into a worse localization performance, because the metrics are based on the spectral differences and compute a quotient between passing and failing counts.

For the construction of the grammar spectra the parser must provide information about which rules it has applied in the parsing process. Parsers generated by ANTLR (2018) contain some extensions to provide this support, but parsers generated by CUP (2014) or JavaCC (2020) do not. We have therefore extended CUP and JavaCC themselves to generate parsers with the required logging. However, we also show how our approach can be "flipped" and can be used with black-box parsers that cannot be extended to collect spectra. More specifically, we construct *synthetic* grammar spectra directly from test cases derived from a grammar, and use these to localize differences between the grammar and the language accepted by the SUT.

We qualitatively and quantitatively evaluate our approach under a number of different fault models and scenarios, including the use of various parsing techniques, test suites, and ranking metrics. We first use a large number of medium-sized single fault grammars, which are constructed from a common grammar by fault seeding. Note that fault seeding is widely used in SFL evaluation (e.g., Abreu, 2009; Wen et al., 2011) because it produces a large number of faulty subjects with known error locations. Our rule-level localization method typically ranks the rules containing the seeded faults within the top five grammar rules for about 40%–70% of the grammars, depending on the applied parsing technique, test suite, and ranking metric. It pinpoints them (i.e., correctly identifies them as unique most suspicious rule) in about 10%–30% of the cases. On average, it ranks the faulty rules within about 25% of all rules, and in less than 15% for a very large test suite containing both positive and negative test cases. Our method pinpoints far fewer of the seeded faults down to the exact symbol position, or even ranks them within the top five positions, due to the much larger number of possible locations and corresponding larger ties (i.e., groups of equally suspicious locations). However, a simple tie-breaking strategy that prefers the item with the right-most designated position over all items derived from the same rule in a tie proves remarkably effective: it typically ranks the seeded faults within the top five positions in about 30%–60% of the cases, and pinpoints them in about 15%–40% of the cases. On average, it ranks the seeded faults within about 10%–20% of all positions. The specialized item-level localization also significantly outperforms a simplistic extension of the rule-level localization, where all positions within a rule are given the same score.

Second, we analyze grammars submitted by students enrolled in compiler engineering courses. Such grammars contain *real* and often *multiple* faults; however, SFL is based on a single fault assumption, and SFL methods can be misled by interactions between multiple faults (Abreu et al., 2009; Xue and Namin, 2013). We demonstrate that our method remains effective in this more difficult situation. We use an iterative "one-bug-at-a-time" (OBA) technique originally proposed by Jones et al. (2002) in the context of fault localization in programs to localize and manually remove multiple faults one by one; in most iterations, the top-ranked rule already contains a fault, and no fault is ranked outside the top five ranked rules.

Finally, we address the *scalabilty* of our approach, and use it to identify four locations in an open-source SQLite grammar (Kiers, 2016) where it deviates from the language accepted by the actual SQLite system (SQLite, 2021). Here, we construct synthetic grammar spectra directly from the test cases derived from the grammar, and use the SQLite system as a black box to collect the required pass/fail information.

Our approach works at a higher abstraction level than generic SFL approaches and returns fault locations in domain-specific terms (i.e., rules or symbols rather than methods or statements). This yields several benefits. First, it can increase the localization precision because it discards all aspects of the parser's internal bookkeeping and error handling code that could impact the localization using generic program spectra. Second, it can also be meaningfully applied when there is no direct representation of the individual rules as executable code; this is typically the case for LR parsers that use table-driven implementations. Third, it can simplify subsequent repair attempts – grammar writers can use the results directly and do not need to manually map between the parser's implementation and the grammar elements.

*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 Sections 3 and 4, we define the various notions of grammar spectra that are at the core of our work and illustrate them with worked examples. In Section 5, we describe our implementation of grammar spectrum extraction for the ANTLR, CUP, and JavaCC parser generators. In Section 6, we discuss the research questions that underly our experimental evaluation. In Section 7, we evaluate our methods for rule-level and item-level fault localization, respectively, over grammars with seeded single faults, and give observations on the effects of different parsing techniques, test suites, and ranking metrics on the effectiveness of our methods. In Section 8, we evaluate the rule-level localization over grammars submitted by students enrolled in compiler engineering courses, which contain real faults. In Section 9, we demonstrate that our approach also works for large grammars and for large black box systems under test (i.e., SQLite). The experimental results show that our approach can in many cases identify faults in a CFG precisely, even in the presence of

multiple, real faults; however, in Section 10, we discuss threats to the validity of generalizing these results. In Sections 11 and 12, we discuss related work and conclude with suggestions for future work.

This paper builds on ideas we introduced in our SLE 2019 paper (Raselimo and Fischer, 2019). The major conceptual extension over that paper is the definition of item spectra for a more precise fault localization at the level of the individual symbols in a rule (see Section 4). This extension is technically intricate but it significantly improves over a simplistic extension of rule-level localization to positions (see Table 6 and RQ4) and opens up new applications of our approach, in particular in grammar repair (Raselimo and Fischer, 2021). We also show how grammar spectra can be synthesized directly from test cases derived from a grammar (see Section 5.3), and how such synthetic spectra can be used to localize differences between a grammar and a black-box system under test. We further provide an extended evaluation using the JavaCC parser generator, demonstrating that our approach works over a wider range of parsing technologies, and show that it scales to a large SQL grammar as well.

In summary, in this paper we make the following main contributions:

- we present spectrum-based methods to localize faults in a context-free grammar, at the level of rules and at the level of the individual symbols in the rules;
- we describe an implementation of our methods that works with the JavaCC, ANTLR, and CUP parser generators;
- we demonstrate the effectiveness of our approach over grammars with seeded faults as well as student submissions from compiler engineering courses that contain multiple real faults;
- we demonstrate that our approach scales to large grammars;
- we show how our approach can be applied to grammars for black-box systems as well; and
- we evaluate the effects of different ranking metrics, test suites, and applied parsing techniques on the localization results.

In combination with the underlying SLE 2019 paper (Raselimo and Fischer, 2019), our work represents the first approach to localize faults in a context-free grammar.

## 2. Background and notation

*Grammars and meta-variables.* A *context-free grammar* (CFG) or simply *grammar* is a four-tuple $G = (N, T, P, S)$ with $N \cap T = \emptyset$, $V = N \cup T$, $P \subset N \times V^*$, and $S \in N$. We call $S$ the *start symbol* and use $A, B, C, \ldots$ for *non-terminals* in $N$, $a, b, c, \ldots$ for *terminals* or *tokens* in $T$, $X, Y, Z$ for *grammar symbols* in $V$, $p, q, r$ for *productions* or *rules* in $P$, $u, v, w, x, y, z$ for *strings* over $T^*$, and $\alpha, \beta, \gamma, \ldots, \omega$ for *phrases* over $V^*$, with $\varepsilon$ for the empty phrase and $|\alpha|$ for the length of $\alpha$. In more complex examples, we also use *italics* and **`bold typewriter`** font for non-terminal and terminal symbols, respectively; we use `normal typewriter` font for structured tokens with different instances such as identifiers.

We also write $A \to \gamma$ for a rule $(A, \gamma) \in P$ and use $P_A = \{A \to \gamma \in P\}$ to denote the set of all rules for $A$; for consistency, we define $P_a = \emptyset$. We call $\text{left}(R) = R \cup \{B \to \beta \in P \mid A \to B\alpha \in R\}$ the *left expansion* of $R \subseteq P$ and use $\text{closure}(R)$ to denote the closure of $R$ under left expansion. Note that this mirrors the itemset closure operation used in the construction of LR parsers.

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

*Derivations and generated language.* A *derivation* $\Rightarrow_G \subseteq V^* \times V^*$ *over* $G$ relates phrases 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 $A \to \gamma \in P$. We write $\Rightarrow$ if the grammar is clear from the context and $\Rightarrow_R$ if $A \to \gamma \in R \subseteq P$. We use $\Rightarrow^*$ for the reflexive-transitive closure. We call a phrase $\alpha$ a *sentential form* if $S \Rightarrow^* \alpha$ and a *sentence* if it is also a string, i.e., $\alpha \in T^*$. For a derivation $\Delta = \alpha_0 \Rightarrow_{p_1} \alpha_1 \Rightarrow_{p_2} \ldots \Rightarrow_{p_n} \alpha_n$, we use $\text{rules}(\Delta) = \bigcup_i \{p_i\}$ to denote the set of applied rules.

**Table 1**
SFL ranking metrics.

| Ranking metric | $score(e)$ |
|---|---|
| Tarantula (Jones and Harrold, 2005) | $\left(\frac{ef(e)}{ef(e)+nf(e)}\right) / \left(\frac{ef(e)}{ef(e)+nf(e)} + \frac{ep(e)}{ep(e)+np(e)}\right)$ |
| Ochiai (Ochiai, 1957) | $\frac{ef(e)}{\sqrt{(ef(e)+nf(e))(ef(e)+ep(e))}}$ |
| Jaccard (Chen et al., 2002) | $\frac{ef(e)}{ef(e)+nf(e)+ep(e)}$ |
| DStar (Wong et al., 2014) | $\frac{ef(e)^n}{nf(e)+ep(e)}$ |

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

*Prefixes and bounded derivations.* We call $u$ a *viable* $k$*-prefix* of a string $w = uv$ if $|u| \leq k$ and $S \Rightarrow^* uv'$ for a $v' \in T^*$, and denote this by $u \preceq_k w$. We call a viable $k$-prefix $u \preceq_k w$ *maximal* if there is no $a \in T$ such that $ua \preceq_{k+1} w$. Hence, $w \preceq_{|w|} w$ iff $w \in L(G)$ and, conversely, if the maximal viable prefix $u$ has length $k < |w|$ then $w$ has a syntax error that can be detected at position $k + 1$.

A derivation $\Delta = S \Rightarrow^* \omega$ is *$k$-prefix bounded for* $w$ if (i) $\omega \Rightarrow^* w$ and (ii) for any derivation step $\alpha A \beta \Rightarrow \alpha\gamma\beta$ in $\Delta$ we have $\alpha \Rightarrow^* u$ and $\alpha A \beta \Rightarrow^* uv = w$ implies (a) $|u| < k$ or (b) $|u| = k$ and $u\beta \Rightarrow^* w$. We denote this by $S \overset{k}{\underset{w}{\Rightarrow}}{}^* \omega$. A $k$-prefix bounded derivation for $w$ therefore never expands a non-terminal symbol whose yield in $w$ will ultimately start only beyond a prefix of length $k$. $S \overset{k}{\underset{w}{\Rightarrow}}{}^* \omega$ is *maximal* if $\omega = u\alpha$ for $|u| = k$, i.e., if we have applied all rules in the $k$-prefix. Note that for any $w = uv \notin L(G)$ with maximal viable $k$-prefix $u$, there exist a not necessarily unique $w' = uv' \in L(G)$ and corresponding maximal $k$-prefix bounded derivation $S \overset{k}{\underset{w'}{\Rightarrow}}{}^* uX\alpha$ for $w'$. We call any such $\Delta$ a maximally viable $k$-prefix bounded derivation for $w$ with *frontier* $X$ and define its *frontier rules* as $\text{closure}(P_X)$. We call the implied $v'$ its right completion. The frontier rules thus describe how the correct prefix $u$ could be completed via a frontier $X$. If the grammar $G$ is assumed to be correct, they can be seen as the specification for an error correction for the erroneous input $w$; however, if $G$ is incorrect, the frontier rules or the rules applied to derive $u$ must contain a fault.

*Test suites.* A *test suite* is a set of SUT inputs and corresponding expected outputs; note that the expected output can also be a specific system error, e.g., in reaction to an illegal input. The SUT *passes* a test if it produces the expected output for the given input. In our case, test inputs are strings $w \in T^*$ and expected outputs are either "accept" or "reject". We could in principle try to prevent the mis-classification of applied rules, and so increase the precision of the fault localization, by using more details in the expected outputs (e.g., error locations) but we do not do this here because it is difficult to implement as these details may depend on internal aspects of the parser (e.g., lookahead size, verbosity, or error correction strategy). We call a test case *positive* if its expected output is accept (i.e., the input is syntactically correct) and *negative* otherwise.

*Grammar-based test suite construction.* For our experimental evaluation we use different test suites that are generated from a "golden" grammar that describes the intended language. For the construction of positive tests we use a generic cover algorithm (van Heerden et al., 2020) with different *grammar coverage criteria* (Lämmel, 2001). The algorithm follows the similar approaches by Fischer et al. (2011) and Havrikov and Zeller (2019). Its basic idea is to (i) iterate over all symbols $X \in V$, (ii) embed $X$, i.e., compute a minimal derivation $S \Rightarrow^* \alpha X \omega$, (iii) cover

$X$, i.e., compute a set of minimal derivations $X \Rightarrow^* \gamma$ that conform to the criterion, and (iv) convert the sentential form into a sentence, i.e., compute a minimal derivation $\alpha\gamma\omega \Rightarrow^* w$ where each non-terminal $A$ in $\alpha\gamma\omega$ is replaced by its minimal yield $w_A$. Note that this algorithm is by construction biased towards short tests because it uses of minimal derivations in all steps.

We use the standard coverage criteria *rule* and *cdrc* (Lämmel, 2001) as well as their extension to $k$-step derivations (also called *k-path coverage* Havrikov and Zeller, 2019) as arguments to the cover algorithm. In addition, we also use *derivable pair coverage* (van Heerden et al., 2020), which can be seen as a fixpoint version of $k$-step coverage, since it covers the shortest derivation between any two symbols irrespective of its length, *adjacent pair coverage*, which ensures that any pairs $X, Y \in V$ with $Y \in \text{follow}(X)$ are covered, and *full cdrc*, a breadth-first version of *cdrc* (van Heerden et al., 2020).

For the construction of negative tests, we use the token- and rule-mutation algorithms by Raselimo et al. (2019). Both algorithms guarantee that the generated tests contain exactly one syntax error.

*Failure, error, fault.* In software engineering, the informal notion of a "bug" is deconstructed into three different concepts (IEEE Std 610.12-1990, 1990).[2] A *failure* occurs when 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 can cause an error in the system when it is executed. Note that errors do not necessarily manifest themselves as observable failures. *Fault localization* tries to identify the unknown fault from an observed failure.

*Program spectra.* A *program spectrum* is a representation of the execution information for the SUT's individual program elements; most SFL methods use method or statement coverage, i.e., record whether a method (resp. statement) has been called (resp. executed) in a given test or not. The coverage information for each test is then correlated with the corresponding test outcomes. Spectra can thus be interpreted as two binary relations $\sim_{✓}, \sim_{✗} : \mathcal{S} \times \mathcal{T}$ between the elements $\mathcal{S}$ of the SUT and the test suite $\mathcal{T}$, where $e_i \sim_{✓} t_j$ holds iff $e_i$ is exercised in the execution of the passing test $t_j$ (and similarly for $e_i \sim_{✗} t_k$ for a failing test $t_k$). They are typically visualized as matrices, as for example shown in Table 2.

From the spectra, we compute four basic counts for each individual program element $e$: $ep(e)$ (resp. $ef(e)$) are the number of passed (resp. failed) tests in which $e$ is executed, while $np(e)$ (resp. $nf(e)$) are the number of passed (resp. failed) tests in which $e$ is *not* executed. Given the number of passed tests $tp$ (resp. failed tests $tf$) in the test suite, we have $ep(e) + np(e) = tp$ and $ef(e) + nf(e) = tf$ for each $e$.

*Ranking metrics.* From the basic counts, SFL methods compute a *suspiciousness score* for each program element; elements that have a higher score are seen as more likely to contain a bug and are ranked higher. The methods differ in the formulas (which are traditionally called *ranking metrics*, even though they are not proper metrics) used for the score computation. In our evaluation, we use four different ranking metrics (see Table 1 for their definitions) that are widely used in SFL. 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 a maximally suspicious element.

If the test suite lacks a *failing* test, the metrics may become undefined or result in equal rankings for all elements; if the test suite lacks a *passing* test, the metrics may become undefined or rank elements based solely on their occurrence count. We therefore assume here that test suites indeed contain at least one failing and one passing test.

[2] This slightly differs from the "defect - infection - failure" model by Zeller (2009). The Zeller model is, however, not as widely used.

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

(a) An example grammar $G_{\mathcal{T}oy}$.

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

(b) Test suite satisfying *rule*-coverage for $G_{\mathcal{T}oy}$.

**Fig. 1.** An example grammar $G_{\mathcal{T}oy}$ (top) and its corresponding *rule* test suite (bottom).

The metrics sometimes assign the same score to different elements. Specifically, for ranking, we need to resolve such *ties* and assign a well-defined rank to all tied elements. 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. A more pessimistic variant uses the lowest possible rank that is consistent with the scores; this corresponds to a worst-case estimate of the number of elements to be inspected.

## 3. Rule-level fault localization

In this section, we illustrate and formalize fault localization at the level of grammar rules; we introduce a more fine-grained localization at the level of the rules' individual symbols in Section 4. In the following, we assume that the SUT is a CFG $G = (N, T, P, S)$, which we assume to be implemented faithfully in an executable parser (since we are trying to localize faults in the grammar, not in the parser's implementation).

### 3.1. Worked example

We illustrate our method with a worked example centered on the toy grammar $G_{\mathcal{T}oy}$ shown in Fig. 1(a), and the test suite shown in Fig. 1(b) that satisfies *rule*-coverage. Note that $G_{\mathcal{T}oy}$ is not LL($k$), but in a form that is suitable for LR parsers. We assume that the grammar developer has made mistakes in formulating some statement rules in

**Table 2**
Rule spectra, suspiciousness scores, and ranks for the faulty grammar version $G'_{Toy}$ and *rule* test suite. Rules are denoted by the non-terminal name and the index of the corresponding alternative. ✓(resp. ✗) indicates execution in a passing (resp. failing) test cases shown in Fig. 1(b); (✗) indicates that the rule is added to the spectrum only through the closure of the frontier symbol. Ranks are only shown for rules with non-zero scores; ties are indicated by a preceding "=". Faulty rules are shown in bold.

| *Rule* | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | *ep* | *np* | *ef* | *nf* | Tarantula | | Ochiai | | Jaccard | | DStar | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *prog*:1 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | 14 | 0 | 2 | 0 | 0.50 | 10 | 0.35 | =8 | 0.13 | 10 | 0.29 | 8 |
| *block*:1 | | | | | | | | | | ✓ | | | | (✗) | | | 1 | 13 | 1 | 1 | 0.88 | =2 | 0.50 | =3 | 0.33 | =2 | 0.50 | =3 |
| *block*:2 | | | | | | | | | | | ✓ | ✓ | ✓ | (✗) | | | 3 | 11 | 1 | 1 | 0.70 | 6 | 0.35 | =8 | 0.20 | =6 | 0.25 | 9 |
| *block*:3 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | | ✓ | | | | | ✗ | ✓ | | 8 | 6 | 2 | 0 | 0.64 | =7 | 0.45 | 5 | 0.20 | =6 | 0.50 | =3 |
| *block*:4 | | | | | | | | | | | | | | (✗) | ✓ | ✓ | 2 | 12 | 1 | 1 | 0.78 | 4 | 0.41 | 7 | 0.25 | 5 | 0.33 | 7 |
| *decls*:1 | | | | | | | | | | | ✓ | | | | | | 1 | 13 | 0 | 2 | 0.00 | – | 0.00 | – | 0.00 | – | 0.00 | – |
| *decls*:2 | | | | | | | | | | ✓ | | ✓ | ✓ | | | | 3 | 11 | 0 | 2 | 0.00 | – | 0.00 | – | 0.00 | – | 0.00 | – |
| *decl*:1 | | | | | | | | | | ✓ | ✓ | ✓ | ✓ | | | | 4 | 10 | 0 | 2 | 0.00 | – | 0.00 | – | 0.00 | – | 0.00 | – |
| *type*:1 | | | | | | | | | | ✓ | ✓ | ✓ | | | | | 3 | 11 | 0 | 2 | 0.00 | – | 0.00 | – | 0.00 | – | 0.00 | – |
| *type*:2 | | | | | | | | | | | | | ✓ | | | | 1 | 13 | 0 | 2 | 0.00 | – | 0.00 | – | 0.00 | – | 0.00 | – |
| *stmts*:1 | | | | | | | | ✓ | | | | | | | | | 1 | 13 | 0 | 2 | 0.00 | – | 0.00 | – | 0.00 | – | 0.00 | – |
| *stmts*:2 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | | ✓ | ✓ | | | | ✗ | ✓ | | 9 | 5 | 2 | 0 | 0.61 | 9 | 0.43 | 6 | 0.18 | 8 | 0.44 | 6 |
| *stmt*:1 | | | | | | ✗ | ✓ | ✓ | ✓ | ✓ | | | | | | | 4 | 10 | 1 | 1 | 0.64 | =7 | 0.32 | 10 | 0.17 | 9 | 0.20 | 10 |
| **stmt:2** | | | | | | ✗ | ✓ | | | | | | | | | | **1** | **13** | **1** | **1** | **0.88** | **=2** | **0.50** | **=3** | **0.33** | **=2** | **0.50** | **=3** |
| **stmt:3** | | | | | | | | | | | | | | ✗ | | | **0** | **14** | **1** | **1** | **1.00** | **1** | **0.71** | **1** | **0.50** | **1** | **1.00** | **1** |
| *stmt*:4 | ✓ | ✓ | ✓ | ✓ | ✓ | | | | | | | | | | | | 5 | 9 | 0 | 2 | 0.00 | – | 0.00 | – | 0.00 | – | 0.00 | – |
| *stmt*:5 | | | | | | | | | | | | | | | ✓ | | 1 | 13 | 0 | 2 | 0.00 | – | 0.00 | – | 0.00 | – | 0.00 | – |
| *expr*:1 | | | | ✓ | | | | | | | | | | | | | 1 | 13 | 0 | 2 | 0.00 | – | 0.00 | – | 0.00 | – | 0.00 | – |
| *expr*:2 | | ✓ | | | | | | | | | | | | | | | 1 | 13 | 0 | 2 | 0.00 | – | 0.00 | – | 0.00 | – | 0.00 | – |
| *expr*:3 | ✓ | | | | | | | | | | | | | | | | 1 | 13 | 0 | 2 | 0.00 | – | 0.00 | – | 0.00 | – | 0.00 | – |
| *expr*:4 | ✓ | ✓ | ✓ | ✓ | | ✗ | ✓ | | | | | | | ✗ | | | 5 | 9 | 2 | 0 | 0.74 | 5 | 0.53 | 2 | 0.29 | 4 | 0.80 | 2 |
| *expr*:5 | | | | | ✓ | | | | | | | | | | | | 1 | 13 | 0 | 2 | 0.00 | – | 0.00 | – | 0.00 | – | 0.00 | – |

$G_{Toy}$, requiring the `else`-branch to be present and restricting the body of `while`-loops to be blocks:

*stmt* → `sleep`
| `if` *expr* `then` *stmt* `else` *stmt*
| `while` *expr* `do` *block*
| `id` = *expr*
| *block*

We call this faulty version $G'_{Toy}$.

We create a parser for $G'_{Toy}$ and run it over the test suite in Fig. 1(b) to collect the grammar spectra shown in Table 2. Rules are denoted by the non-terminal name and the index of the corresponding alternative. ✓(resp. ✗) indicates execution of a rule in a passing (resp. failing) test cases; (✗) indicates that a rule is added to the spectrum only through the closure of the frontier symbol. The faulty rules are shown in bold. We finally compute the scores according to the four ranking metrics shown in Table 1 and rank the rules; note that ties can also result from different execution counts, as the example of *block*:1 and *block*:3 under DStar shows. Ranks are only shown for rules with non-zero scores.

All four metrics pinpoint the faulty `while`-rule (i.e., *stmt*:3) as the unique most suspicious rule. This is hardly surprising because the rule is only applied in one test case and that one is failing. The second fault is more difficult to localize because the faulty rule is executed in both failing and passing test cases. Tarantula and Jaccard rank it second while Ochiai and DStar rank it third, in both cases behind the rule *expr* → `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., 11.4% of all rules) before we find both faults using Tarantula or Jaccard, 3.5 rules (or 15.9%) using Ochiai, and 4 rules (or 18.2%) using DStar.

### 3.2. Rule spectra

We can informally define a *rule spectrum* as the set of all rules $R \subseteq P$ that are applied when a test $w$ in the test suite is parsed. However, which rules are actually applied depends on the nature of the parser, in particular when it rejects $w$. We therefore first formalize our intuition in terms of generic derivations, and then concretize it in Sections 5.1 and 5.2 for LL and LR parsers.

For accepted tests the formal definition of rule spectra directly follows our intuition; note that a string $w \in L(G)$ can induce multiple rule spectra if $G$ is ambiguous.

**Definition 3.1** (*Positive Rule Spectrum*). If $\Delta = S \Rightarrow_{p_1} \alpha_1 \Rightarrow_{p_2} \alpha_2 \Rightarrow_{p_3} \cdots \Rightarrow_{p_n} \alpha_n = w$, then $R = \bigcup_i \{p_i\} = \text{rules}(\Delta)$ is called a *positive rule spectrum* for $w$.

Note that $\Delta = S \Rightarrow^* w$ is a maximally viable $|w|$-prefix bounded derivation (see Section 2) for $w$, since $w \in L(G)$. We can use this observation as starting point for the definition of spectra in cases where $w \notin L(G)$: we consider all rules of a maximally viable $k$-prefix bounded derivation $\Delta$ for $w$, i.e., all rules that have been applied to the left of the error position. However, we must also consider the frontier rules because the error could be either in the rule in $\Delta$ that introduced the frontier, or in any of $\Delta$'s frontier rules themselves. Consider for example the situation where we are trying to parse $w =$ `program a={while a do sleep;}.` (i.e., test case #14) with the faulty version $G'_{Toy}$. This fails at `sleep` because $G'_{Toy}$ expects *body* to be a *block* and `sleep` is a *stmt*. The maximal viable prefix $u$ of $w$ is thus $u =$ `program a={while a do`, with $v =$ `{};}.` a possible right completion so that $uv \in L(G'_{Toy})$. One (and in this case the only possible) maximal $k$-prefix bounded derivation using $G'_{Toy}$ with $k = 7$ for $w$ is

$\Delta = prog \Rightarrow_{prog:1}$ `program id=`*block*`.`
$\Rightarrow_{block:3}$ `program id={`*stmts*`}.`
$\Rightarrow_{stmts:2}$ `program id={`*stmt*`}.`
$\Rightarrow_{stmt:3}$ `program id={ while` *expr* `do` *block*`}.`
$\Rightarrow_{expr:4}$ `program id={ while id do` *block*`}.`

with the frontier symbol *block* and rules($\Delta$) ={*prog*:1, *block*:3, *stmts*:2, *stmt*:3, *expr*:4}. We can presume that the fault is in *stmt*:3 (where it was indeed introduced) but with the information at hand we cannot rule out that any of the *block*-rules is at fault — for example, *block*:4 could have been meant to be of the form *block* → `sleep;`. This requires us to include all possible frontier rules in the (negative) spectra. Hence, we get the following formal definition:

**Definition 3.2** (*Negative Rule Spectrum*). Let $w = uv \notin L(G)$ with maximal viable $k$-prefix $u$, and $S \Rightarrow_{p_1} \alpha_1 \Rightarrow_{p_2} \alpha_2 \Rightarrow_{p_3} \cdots \Rightarrow_{p_n} uX\alpha$ be a maximally viable $k$-prefix bounded derivation for $w$ with frontier $X$. Then $R = \bigcup_i \{p_i\} \cup \text{closure}(P_X)$ is called a *negative rule spectrum* for $w$.

Note that this is a "loose" definition in the sense that a negative test $w \notin L(G)$ may induce several different negative spectra, since there can be different maximally viable $k$-prefix bounded derivations for $w$, with different right completions and frontiers. However, an LL(1)-parser will exploit only one of these derivations and thus determinstically produce only a single negative spectrum. Note also that Definition 3.2 can be modified to subsume the "positive" and "negative" definitions, but we keep the cases apart for simplicity.

**Table 3**
Item spectra, suspiciousness scores, ranks, and resolved ranks for the faulty grammar version $G'_{Toy}$ and *rule* test suite. Items are denoted by the non-terminal name, the index of the corresponding alternative, and the index of the designated position, with zero denoting the position before the right-hand side of the rule. ✓, ✗, and (✗) are defined as in Table 2. Entries are only shown for items with non-zero scores. The standard ranking is shown on the left side of rank column, the resolved ranking using the *k-max* tie breaking strategy described in Section 4.4 on the right side; ties are indicated by a preceding "=". Items corresponding to the fault locations are shown in bold.

| *Item* | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | *ep* | *np* | *ef* | *nf* | Tarantula | | | Ochiai | | | Jaccard | | | DStar | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *prog*:1:0 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | 14 | 0 | 2 | 0 | 0.50 | =19 | | 0.35 | =16 | | 0.13 | =19 | | 0.29 | =16 | |
| *prog*:1:1 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | 14 | 0 | 2 | 0 | 0.50 | =19 | | 0.35 | =16 | | 0.13 | =19 | | 0.29 | =16 | |
| *prog*:1:2 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | 14 | 0 | 2 | 0 | 0.50 | =19 | | 0.35 | =16 | | 0.13 | =19 | | 0.29 | =16 | |
| *prog*:1:3 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | 14 | 0 | 2 | 0 | 0.50 | =19 | 9 | 0.35 | =16 | =7 | 0.13 | =19 | 9 | 0.29 | =16 | 7 |
| *block*:1:0 | | | | | | | | | | ✓ | | | | (✗) | | | 1 | 13 | 1 | 1 | 0.88 | =5 | =2 | 0.50 | =7 | =3 | 0.33 | =5 | =2 | 0.50 | =7 | =3 |
| *block*:2:0 | | | | | | | | | | | ✓ | ✓ | ✓ | (✗) | | | 3 | 11 | 1 | 1 | 0.70 | 14 | 6 | 0.35 | =16 | =7 | 0.20 | =14 | =6 | 0.25 | 20 | 8 |
| *block*:3:0 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | | ✓ | | | | | ✗ | ✓ | | 8 | 6 | 2 | 0 | 0.64 | =15 | | 0.45 | =13 | | 0.20 | =14 | | 0.50 | =7 | |
| *block*:3:1 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | | ✓ | | | | | ✗ | ✓ | | 8 | 6 | 2 | 0 | 0.64 | =15 | =7 | 0.45 | =13 | 5 | 0.20 | =14 | =6 | 0.50 | =7 | =3 |
| *block*:4:0 | | | | | | | | | | | | | | (✗) | ✓ | ✓ | 2 | 12 | 1 | 1 | 0.78 | 11 | 4 | 0.41 | 15 | 6 | 0.25 | 13 | 5 | 0.33 | 15 | 6 |
| *stmts*:2:0 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | | ✓ | | | | | ✗ | ✓ | | 8 | 6 | 2 | 0 | 0.44 | 24 | 11 | 0.22 | 24 | 11 | 0.09 | 24 | 11 | 0.10 | 24 | 11 |
| *stmts*:2:1 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | | ✓ | | | | | | ✓ | | 8 | 6 | 1 | 1 | 0.47 | 23 | 10 | 0.24 | 23 | 10 | 0.10 | 23 | 10 | 0.11 | 23 | 10 |
| *stmt*:1:0 | | | | | | ✗ | ✓ | ✓ | ✓ | ✓ | | | | | | | 4 | 10 | 1 | 1 | 0.64 | =15 | | 0.32 | =21 | | 0.17 | =17 | | 0.20 | =21 | |
| *stmt*:1:1 | | | | | | ✗ | ✓ | ✓ | ✓ | ✓ | | | | | | | 4 | 10 | 1 | 1 | 0.64 | =15 | =7 | 0.32 | =21 | 9 | 0.17 | =17 | 8 | 0.20 | =21 | 9 |
| *stmt*:2:0 | | | | | | ✗ | ✓ | | | | | | | | | | 1 | 13 | 1 | 1 | 0.88 | =5 | | 0.50 | =7 | | 0.33 | =5 | | 0.50 | =7 | |
| *stmt*:2:1 | | | | | | ✗ | ✓ | | | | | | | | | | 1 | 13 | 1 | 1 | 0.88 | =5 | | 0.50 | =7 | | 0.33 | =5 | | 0.50 | =7 | |
| *stmt*:2:2 | | | | | | ✗ | ✓ | | | | | | | | | | 1 | 13 | 1 | 1 | 0.88 | =5 | | 0.50 | =7 | | 0.33 | =5 | | 0.50 | =7 | |
| *stmt*:2:3 | | | | | | ✗ | ✓ | | | | | | | | | | 1 | 13 | 1 | 1 | 0.88 | =5 | | 0.50 | =7 | | 0.33 | =5 | | 0.50 | =7 | |
| **stmt:2:4** | | | | | | ✗ | ✓ | | | | | | | | | | **1** | **13** | **1** | **1** | **0.88** | **=5** | **=2** | **0.50** | **=7** | **=3** | **0.33** | **=5** | **=2** | **0.50** | **=7** | **=3** |
| *stmt*:3:0 | | | | | | | | | | | | | | ✗ | | | 0 | 14 | 1 | 1 | 1.00 | =1 | | 0.71 | =1 | | 0.50 | =1 | | 1.00 | =1 | |
| *stmt*:3:1 | | | | | | | | | | | | | | ✗ | | | 0 | 14 | 1 | 1 | 1.00 | =1 | | 0.71 | =1 | | 0.50 | =1 | | 1.00 | =1 | |
| *stmt*:3:2 | | | | | | | | | | | | | | ✗ | | | 0 | 14 | 1 | 1 | 1.00 | =1 | | 0.71 | =1 | | 0.50 | =1 | | 1.00 | =1 | |
| **stmt:3:3** | | | | | | | | | | | | | | ✗ | | | **0** | **14** | **1** | **1** | **1.00** | **=1** | **1** | **0.71** | **=1** | **1** | **0.50** | **=1** | **1** | **1.00** | **=1** | **1** |
| *expr*:4:0 | ✓ | ✓ | ✓ | ✓ | | ✗ | | ✓ | | | | | | ✗ | | | 5 | 9 | 2 | 0 | 0.74 | =12 | | 0.53 | =5 | | 0.29 | =11 | | 0.80 | =5 | |
| *expr*:4:1 | ✓ | ✓ | ✓ | ✓ | | ✗ | | ✓ | | | | | | ✗ | | | 5 | 9 | 2 | 0 | 0.74 | =12 | 5 | 0.53 | =5 | 2 | 0.29 | =11 | 4 | 0.80 | =5 | 2 |

## 4. Item-level fault localization

Localization with rule spectra allows us to identify faulty rules in a grammar; however, we still have to inspect the individual symbols on the right-hand side of the rules to identify the actual fault location. This can involve substantial effort since the rules can be long; for example, the BNF version of the SQLite grammar (Kiers, 2016) has more than twenty rules that each contain six or more symbols, and the length of the longest rule is sixteen.

We therefore refine our method to localize errors more precisely, at the level of the individual symbols in a rule. Our basic idea here is to use spectra over *items* rather than over rules for the localization. This exploits the fact that the designated position marks the boundary between the part of a rule that has already been processed successfully and the part that still needs to be processed; hence, we assume that the error is at the symbol following the designated position.

Furthermore, the development of item-level localization is a necessary step for automatic repair (Raselimo and Fischer, 2021). Item-level localization naturally makes the search space easier to navigate than rule-level localization. This follows a similar trend to automated program repair algorithms (Monperrus, 2018; Gazzola et al., 2019; Ghanbari et al., 2019), which use fine-grained statement-level fault localization results, despite several spectrum-based fault localization studies (de Souza et al., 2016; Wong et al., 2016) demonstrating that SFL works better with method-level spectra than with statement-level spectra.

In this section, we again illustrate our method with a worked example, based on the same grammar $G'_{Toy}$ used in the previous section. We then give formal definitions of item spectra and also introduce a domain-specific tie breaking strategy.

### 4.1. Worked example

We illustrate the item-level localization with the same example as in Section 3; specifically, we assume the same faulty grammar under test $G'_{Toy}$, with the faults in the `while`- and `if`-rules, but since we are now trying to locate the position of one or more offending symbols on the right-hand side of a rule, we use the two items

*stmt* → `if` *expr* `then` *stmt* • `else` *stmt*

*stmt* → `while` *expr* `do` • *block*

to represent the faults.

Table 3 shows the detailed results for the item spectra collected according to Definitions 4.1 and 4.2. Items are denoted by the non-terminal name, the index of the corresponding alternative, as shown in Fig. 1, and the index of the designated position (with zero denoting the position before the right-hand side of the rule). Note that, due to the large spectrum size (81 elements, compared to 15 elements for rule spectrum), entries are shown only for the 25 items which have been executed in at least one of the two failing test cases 6 and 24, and thus have non-zero suspiciousness scores. For each metric, Table 3 shows two different rankings, the standard ranking and a resolved ranking where ties are resolved using the *k-max* tie breaking mechanism described in Section 4.4. In both cases, ties are indicated by a preceding "=". Items corresponding to the fault locations are shown in bold.

*Results.* In Table 3, we see that all four metrics assign the highest suspiciousness score to four items from the `while`-rule but fail to separate the actual fault position *stmt*:3:3 from the three preceding items in the rule and so produce a four-way tie. As in the case of the rule-level localization, the second fault is harder to localize, and all four metrics again tie the actual fault position *stmt*:2:4 with the preceding items in the rule. Tarantula and Jaccard have it further tied with one of the *block*-items at rank 5, while Ochiai and DStar rank it behind two *expr*:4-items at rank 7, in a 6-way and 8-way tie, respectively. Note that the items from a rule are not necessarily all scored identically. In particular, the item *stmt*:2:5 (i.e., *stmt* → `if` *expr* `then` *stmt* `else` • *stmt*) is scored zero by all metrics; this is a strong indication that the fault is located to the left of its designated position. However, note also that many items from the same rule are indeed scored identically, and that the ties are therefore much longer than in rule-level localization.

If we inspect the items in rank order and resolve ties by picking rules arbitrarily, we have on average to look at 7.5 positions (i.e., 9.3% of all items) in 3 rules before we find both faults using Tarantula and Jaccard, 9.5 positions (11.7%) in 4 rules using Ochiai and 10.5 positions (13.0%) in 6 rules using DStar. Note that the number of rules involved changes with the different orders in which the tied positions are inspected. Note also that by deriving a rule rank from the highest ranked item for each rule, the items induce the same rule-level ranking as the rule-level localization (cf. Table 2), although that is not guaranteed in general.

### 4.2. Plain item spectra

In our first approach, we still define the spectra over maximal derivations; it is based on a more or less straightforward adaptation of the corresponding definitions of rule spectra. We can therefore informally (as we did for rule spectra in Section 3.2) define an *item spectrum* for the string $w$ in the test suite as the set of positions within rules $R^\bullet \subseteq P^\bullet$ that are processed successfully when $w$ is parsed. For accepted strings, the item spectrum simply includes *all items* from each applied rule.

| Derivation | Extracted items |
|---|---|
| $\Delta = prog$ | |
| $\Rightarrow_{prog:1}$ **program** id = *block* . | {*prog*:1:0, *prog*:1:1, *prog*:1:2, *prog*:1:3} |
| $\Rightarrow_{block:3}$ **program** id = { *stmts* }. | {*block*:3:0, *block*:3:1} |
| $\Rightarrow_{stmts:2}$ **program** id = { *stmt* }. | {*stmts*:2:0} |
| $\Rightarrow_{stmt:3}$ **program** id = { **while** *expr* **do** *block* }. | {*stmt*:3:0, *stmt*:3:1} |
| $\Rightarrow_{expr:4}$ **program** id = { **while** id **do** *block* }. | {*expr*:4:0, *expr*:4:1, *stmt*:3:2, *stmt*:3:3, *block*:1:0, *block*:2:0, *block*:3:0, *block*:4:0} |

**Fig. 2.** Example construction of negative item spectrum.

**Definition 4.1** (*Positive Item Spectrum*)**.** If $S \Rightarrow_{p_1} \alpha_1 \Rightarrow_{p_2} \alpha_2 \Rightarrow_{p_3} \cdots \Rightarrow_{p_n} \alpha_n = w$, then $R^\bullet = \bigcup_i \text{items}(p_i)$ is called a *positive item spectrum* for $w$.

For rejected strings, the adaptation is slightly more complex than in the positive case. More specifically, we include in the spectrum only items from rules in derivation steps whose yield up to the designated position occurs before the syntax error. As in the case of the rule spectra (see Definition 3.2), we must take the frontier rules into account; however, here we only add the corresponding non-kernel items.

**Definition 4.2** (*Negative Item Spectrum*)**.** Let $w = uv \notin L(G)$ with maximal viable $k$-prefix $u$, and $\Delta = S \Rightarrow_{p_1} \alpha_1 \Rightarrow_{p_2} \alpha_2 \Rightarrow_{p_3} \cdots \Rightarrow_{p_n} uX\alpha$ be a maximally viable $k$-prefix bounded derivation for $w$ with frontier $X$. Then

$$R^\bullet = \{p^\bullet \mid \alpha A\beta \Rightarrow_p \alpha\gamma\beta \in \Delta,\ p^\bullet = A \to \mu \bullet \nu \in \text{items}(p),\ \exists x \in T^* \cdot \alpha\mu x \Rightarrow^* u\} \cup \text{closure}(\{X \to \bullet\gamma \mid X \to \gamma \in P\})$$

is called a *negative item spectrum* for $w$.

Fig. 2 illustrates the negative item spectrum construction for the same maximal prefix-bounded derivation $\Delta$ as in Section 3.2. It shows on the left the individual derivation steps and on the right the corresponding extracted items; the spectrum $R^\bullet$ is the union of all these sets.

### 4.3. Shift item spectra

The plain item spectra we introduced in the previous section derive the spectra from the individual derivation for each test case. Alternatively, we can try to extract a single spectrum from *all possible derivations* that consume prefixes of the maximal viable prefix.

**Definition 4.3** (*Shift Item Spectrum*)**.** Let $u$ be the maximal viable $k$-prefix of $w$. Then

$$R^\bullet = \bigcup_{w'=uv \in L(G)} \bigcup_{\substack{u' \preceq_i u,\ i \le k \\ u'}} \{p^\bullet \mid \Delta = S \Rightarrow^*_{u'} u'\omega, \alpha A\beta \Rightarrow_p \alpha\gamma\beta \in \Delta,\ p^\bullet = A \to \mu \bullet \nu \in \text{items}(p), \exists x \in T^* \cdot \alpha\mu x \Rightarrow^* u'\}$$

is called the *shift item spectrum* for $w$.

Definition 4.3 formulates this intuition. It considers all right completions $v$ of the maximal viable prefix $u$, and then all maximal prefix-bounded derivations of the prefixes $u'$ of $u$ from these derivations, and‘ extracts the items of all applied rules. Note that this definition does not explicitly consider the frontier rules (cf. Definition 4.2) because they are implied by the different right completions $v$. Note also that Definition 4.3 gives larger and denser spectra than Definition 4.1 if $w \in L(G)$ because it considers all possible completions of valid prefixes, while the plain positive item spectra only consider items for successful rule applications.

*Worked example.* Table 4 shows the corresponding results for shift item spectra (see Definition 4.3). It is easy to see that these are indeed both larger (i.e., have more non-zero suspiciousness scores, at 38 compared to 25) and denser (i.e., items are associated with more test cases) than the plain item spectra. However, this apparent loss of precision does not necessarily translate into a worse localization performance, because the metrics are based on the spectral difference between passing and failing tests.

In fact, in this case, the results *improve* slightly for some metrics. All four metrics now assign the highest suspiciousness scores to only three items from the **`while`**-rule (i.e., *stmt*:3:1, *stmt*:3:2 and *stmt*:3:3). Tarantula and Jaccard both rank the second fault (i.e., *stmt*:2:4) tied fourth, with only three other items from the **`if`**-rule in the tie. Ochiai ranks it tied fifth, but with a much wider tie comprising 11 items from 6 rules, while DStar struggles and ranks it tied twelfth. Overall, we have on average to look at 5.5 positions (i.e., 6.8% of all positions) in 2 rules before we find both faults using Tarantula or Jaccard, 10 positions (12.3%) in 7 rules using Ochiai, and 13.5 positions (16.7%) in 7 rules using DStar.

### 4.4. Specialized tie breaking strategy

Since item spectra are larger and denser than rule spectra, ties are more common and longer than in rule-level localization. Tables 3 and 4 clearly illustrate this. Our challenge is to reduce the sizes of the ties, and ideally to rank the faulty items uniquely at the top of the tied group. Here, we can take advantage of contextual information such as the type of test suites used or even the structure of the grammar under test to break ties on the fly. In the worked example, we used a test suite with positive tests only, so we can resolve ties between items from the same rule in favor of the item with the largest designated position (and could in fact even drop items with subsumed designated positions entirely from consideration). This improves the ranking and reduces the average wasted effort, as the resolved ranks in Tables 3 and 4 show.

The basic idea of tie breaking using the *k-max* strategy is to resolve ties in favor of the item from a grammar rule $r$ with the larger designated position (i.e., further to the right of the rule) over other items from the same rule $r$ with smaller designated positions. If there exists a tie from items of different grammar rules, *k-max* picks from each rule the item with the highest position. The other items are dropped altogether.[3] Generally, the *k-max* strategy creates two ranks from a set of tied items from different rules, with items with larger designated positions ranked higher.

Table 3 illustrates the benefits of *k-max* tie breaking strategy for plain item spectra. It resolves the respective ties over both faulty rules’ items in favor of the actual fault locations; this allows all four

[3] Note that dropping items may in principle also drop the actual faults. Our implementation therefore uses a variant where the subsumed items are simply ranked directly below the identified maximal items.

**Table 4**
Shift item spectra, suspiciousness scores, ranks, and resolved ranks for the faulty grammar version $G'_{toy}$ and *rule* test suite. We use the same layout as in Table 3.

| *Item* | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | *ep* | *np* | *ef* | *nf* | Tarantula | | | Ochiai | | | Jaccard | | | DStar | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *prog*:1:0 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | 14 | 0 | 2 | 0 | 0.50 | =18 | | 0.35 | =16 | | 0.13 | =17 | | 0.29 | =16 | |
| *prog*:1:1 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | 14 | 0 | 2 | 0 | 0.50 | =18 | | 0.35 | =16 | | 0.13 | =17 | | 0.29 | =16 | |
| *prog*:1:2 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | 14 | 0 | 2 | 0 | 0.50 | =18 | | 0.35 | =16 | | 0.13 | =17 | | 0.29 | =16 | |
| *prog*:1:3 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | 14 | 0 | 2 | 0 | 0.50 | =18 | =9 | 0.35 | =16 | =8 | 0.13 | =17 | =9 | 0.29 | =16 | =8 |
| *block*:1:0 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | 14 | 0 | 2 | 0 | 0.50 | =18 | | 0.35 | =16 | | 0.13 | =17 | | 0.29 | =16 | |
| *block*:1:1 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | 14 | 0 | 2 | 0 | 0.50 | =18 | =9 | 0.35 | =16 | =8 | 0.13 | =17 | =9 | 0.29 | =16 | =8 |
| *block*:2:0 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | 14 | 0 | 2 | 0 | 0.50 | =18 | | 0.35 | =16 | | 0.13 | =17 | | 0.29 | =16 | |
| *block*:2:1 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | 14 | 0 | 2 | 0 | 0.50 | =18 | =9 | 0.35 | =16 | =8 | 0.13 | =17 | =9 | 0.29 | =16 | =8 |
| *block*:3:0 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | 14 | 0 | 2 | 0 | 0.50 | =18 | | 0.35 | =16 | | 0.13 | =17 | | 0.29 | =16 | |
| *block*:3:1 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | 14 | 0 | 2 | 0 | 0.50 | =18 | =9 | 0.35 | =16 | =8 | 0.13 | =17 | =9 | 0.29 | =16 | =8 |
| *block*:4:0 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | 14 | 0 | 2 | 0 | 0.50 | =18 | | 0.35 | =16 | | 0.13 | =17 | | 0.29 | =16 | |
| *block*:4:1 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | 14 | 0 | 2 | 0 | 0.50 | =18 | =9 | 0.35 | =16 | =8 | 0.13 | =17 | =9 | 0.29 | =16 | =8 |
| *decls*:1:0 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | 14 | 0 | 2 | 0 | 0.50 | =18 | =9 | 0.35 | =16 | =8 | 0.13 | =17 | =9 | 0.29 | =16 | =8 |
| *decls*:2:0 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | 14 | 0 | 2 | 0 | 0.50 | =18 | =9 | 0.35 | =16 | =8 | 0.13 | =17 | =9 | 0.29 | =16 | =8 |
| *decl*:1:0 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | 14 | 0 | 2 | 0 | 0.50 | =18 | =9 | 0.35 | =16 | =8 | 0.13 | =17 | =9 | 0.29 | =16 | =8 |
| *stmts*:1:0 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | 14 | 0 | 2 | 0 | 0.50 | =18 | =9 | 0.35 | =16 | =8 | 0.13 | =17 | =9 | 0.29 | =16 | =8 |
| *stmts*:2:0 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | 14 | 0 | 2 | 0 | 0.50 | =18 | =9 | 0.35 | =16 | =8 | 0.13 | =17 | =9 | 0.29 | =16 | =8 |
| *stmt*:1:0 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | 14 | 0 | 2 | 0 | 0.50 | =18 | =9 | 0.35 | =16 | =8 | 0.13 | =17 | =9 | 0.29 | =16 | =8 |
| *stmt*:1:1 | | | | | | ✗ | ✓ | ✓ | ✓ | ✓ | | | | | | | 4 | 10 | 1 | 1 | 0.64 | 17 | 8 | 0.32 | 37 | 21 | 0.17 | 16 | 8 | 0.20 | 37 | 21 |
| *stmt*:2:0 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | 14 | 0 | 2 | 0 | 0.50 | =18 | =9 | 0.35 | =16 | =8 | 0.13 | =17 | =9 | 0.29 | =16 | =8 |
| *stmt*:2:1 | | | | | | ✗ | ✓ | | | | | | | | | | 1 | 13 | 1 | 1 | 0.88 | =4 | | 0.50 | =5 | | 0.33 | =4 | | 0.50 | =12 | |
| *stmt*:2:2 | | | | | | ✗ | ✓ | | | | | | | | | | 1 | 13 | 1 | 1 | 0.88 | =4 | | 0.50 | =5 | | 0.33 | =4 | | 0.50 | =12 | |
| *stmt*:2:3 | | | | | | ✗ | ✓ | | | | | | | | | | 1 | 13 | 1 | 1 | 0.88 | =4 | | 0.50 | =5 | | 0.33 | =4 | | 0.50 | =12 | |
| **stmt:2:4** | | | | | | ✗ | ✓ | | | | | | | | | | **1** | **13** | **1** | **1** | **0.88** | **=4** | **2** | **0.50** | **=5** | **=3** | **0.33** | **=4** | **2** | **0.50** | **=12** | **7** |
| *stmt*:3:0 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | 14 | 0 | 2 | 0 | 0.50 | =18 | =9 | 0.35 | =16 | =8 | 0.13 | =17 | =9 | 0.29 | =16 | =8 |
| *stmt*:3:1 | | | | | | | | | | | | | | ✗ | | | 0 | 14 | 1 | 1 | 1.00 | =1 | | 0.71 | =1 | | 0.50 | =1 | | 1.00 | =1 | |
| *stmt*:3:2 | | | | | | | | | | | | | | ✗ | | | 0 | 14 | 1 | 1 | 1.00 | =1 | | 0.71 | =1 | | 0.50 | =1 | | 1.00 | =1 | |
| **stmt:3:3** | | | | | | | | | | | | | | ✗ | | | **0** | **14** | **1** | **1** | **1.00** | **=1** | **1** | **0.71** | =1 | **1** | **0.50** | **=1** | **1** | **1.00** | **=1** | **1** |
| *stmt*:4:0 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | 14 | 0 | 2 | 0 | 0.50 | =18 | =9 | 0.35 | =16 | =8 | 0.13 | =17 | =9 | 0.29 | =16 | =8 |
| *stmt*:5:0 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | ✓ | 14 | 0 | 2 | 0 | 0.50 | =18 | =9 | 0.35 | =16 | =8 | 0.13 | =17 | =9 | 0.29 | =16 | =8 |
| *expr*:1:0 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | | | | | | | ✗ | | | 6 | 8 | 2 | 0 | 0.70 | =9 | | 0.50 | =5 | | 0.25 | =9 | | 0.67 | =5 | |
| *expr*:1:1 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | | | | | | | ✗ | | | 6 | 8 | 2 | 0 | 0.70 | =9 | =4 | 0.50 | =5 | =3 | 0.25 | =9 | =4 | 0.67 | =5 | =3 |
| *expr*:2:0 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | | | | | | | ✗ | | | 6 | 8 | 2 | 0 | 0.70 | =9 | | 0.50 | =5 | | 0.25 | =9 | | 0.67 | =5 | |
| *expr*:2:1 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | | | | | | | ✗ | | | 6 | 8 | 2 | 0 | 0.70 | =9 | =4 | 0.50 | =5 | =3 | 0.25 | =9 | =4 | 0.67 | =5 | =3 |
| *expr*:3:0 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | | | | | | | ✗ | | | 6 | 8 | 2 | 0 | 0.70 | =9 | =4 | 0.50 | =5 | =3 | 0.25 | =9 | =4 | 0.67 | =5 | =3 |
| *expr*:4:0 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | | | | | | | ✗ | | | 6 | 8 | 2 | 0 | 0.70 | =9 | | 0.50 | =5 | | 0.25 | =9 | | 0.67 | =5 | |
| *expr*:4:1 | ✓ | ✓ | ✓ | ✓ | | ✗ | ✓ | | | | | | | ✗ | | | 5 | 9 | 2 | 0 | 0.74 | 8 | 3 | 0.53 | 4 | 2 | 0.29 | 8 | 3 | 0.80 | 4 | 2 |
| *expr*:5:0 | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | ✓ | | | | | | | ✗ | | | 6 | 8 | 2 | 0 | 0.70 | =9 | =4 | 0.50 | =5 | =3 | 0.25 | =9 | =4 | 0.67 | =5 | =3 |

metrics to pinpoint one of the faults (i.e., identify it as the single top-ranked item). However, ties over items from different rules remain unresolved (e.g., using Tarantula *block*:1:0 and *stmt*:2:4 remain tied). If we again inspect the items in rank order and resolve the remaining ties arbitrarily, we now have to look only at 2.5 positions (i.e., 3.1% of all positions) in 3 rules before we find both faults using Tarantula or Jaccard, 3.5 positions (4.3%) in 4 rules using Ochiai and 4 positions (4.9%) in 5 rules using DStar.

In the case of shift item spectra (see Table 4), the tie resolution strategy is even more effective, and allows Tarantula and Jaccard to pinpoint both faults. However, we have to look at 5 positions (6.2%) in 5 rules using Ochiai, and 7 positions (8.6%) in 7 rules using DStar to identify the faults.

## 5. Implementation

In order to collect the grammar spectra, we typically have to modify the parser to log which rules it has applied. The specific modifications vary with the general parsing technology and the specific parser; here we describe the modifications we made to the JavaCC, ANTLR, and CUP parser generators in order to generate parsers with the required logging extensions. We also sketch how synthetic spectra can be constructed directly from test cases.

### 5.1. Spectra for recursive-descent LL parsers

*Rule spectra.* Since LL parsers build the maximally viable derivation $\Delta$ top-down, left-to-right, every expansion step $\alpha_i \Rightarrow_{p_i} \alpha_{i+1} \in \Delta$ adds the corresponding rule $p_i$ to the spectrum, whether the derivation process succeeds or not. In a recursive-descent parser, each rule is implemented by its own parsing function, and each derivation step corresponds to a call to one of these functions. Hence, a rule-level spectrum includes the set of parse functions entered at least once by the parser. This holds for both valid and invalid strings, but for an invalid string (i.e., $w' \notin L(G)$), there is at least one parse function which was entered but not exited successfully. For an invalid string, this set of parse functions does not necessarily include the frontier rules. Consider for example an LL(1)-grammar with the rules $A \to \alpha B \gamma$ and $B \to \beta_1 \mid \ldots \mid \beta_n$, and a maximally viable derivation $S \overset{k}{\Rightarrow}{}^{*}_{w} uA\gamma$ for $w = uav$ with $a \notin \text{first}(B)$. The parse function for $A$ checks $a$ against all $b \in \text{first}(\beta_i)$ before calling the parse function for the respective alternative of $B$, but since $a \notin \text{first}(B)$, none of them will actually be called and added to the spectrum. We must therefore modify the parser's error handling routines to add the corresponding frontier rules explicitly. Note also that an LL(1) parser only explores a single maximally viable derivation $\Delta$ for each $w \notin L(G)$ and we therefore only get a single negative spectrum. For LL($k$) parsers with $k > 1$, the derivation may not even be maximal because the parser may detect the syntax error before actually reaching the error location. The collected rules may thus underapproximate the negative spectra, especially if the definition of frontier rules is not adapted properly to sequences of grammar symbols.

*Item spectra.* An extension to record item spectra is straightforward: we only need to map the actions (i.e., matching tokens and recursive calls to other parse functions) taken in a body of a parse function that implements the rule back to their positional occurrence on the right hand-side of the corresponding rule.

*JavaCC.* In principle, spectrum collection is a simple logging task that can be implemented easily. For JavaCC, which generates straightforward recursive-descent LL($k$) parsers from a given grammar specification, we found it indeed relatively easy to modify the generator source code itself, and to add code for (both rule- and item-level) grammar spectra extraction task as a side effect to parser generation. This allows us full control and management of individual rule alternative resolution. JavaCC offers advanced top-down parsing features like localized lookahead configurations, i.e., users can explicitly set a value of $k > 1$ for portions of a grammar that are not within the LL(1) parsing capabilities, thereby making the generated parser LL($k$) for only those portions. Our evaluation, however, focuses on the default LL(1) configuration.

*ANTLR.* ANTLR, in contrast, generates adaptive LL(*) parsers that use unbounded lookahead, which complicates the structure of the parse functions, and thus in turn the spectra extraction. ANTLR provides runtime support to automate collection of grammar spectra through tree walkers (via generated listener and visitor interfaces). However, this only works when ANTLR actually builds a parse 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 misclassified in the spectrum.[4]

As an alternative, 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 (Kiczales et al., 1997) 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 3.1 and 3.2. In practice, however, we encountered two problems that can cause the extracted spectra to be wrong. First, ANTLR's adaptive LL(*) parsing mechanism can cause it to raise a syntax error with unbounded lookahead (typically a `no viable alternative` error) without actually entering the parse function for the corresponding rule, so that frontier rules may be missing. Second, ANTLR's tracking of rule applications is wrong[5] for grammars that contain left-recursive rules.

The adaptive LL(*) parsing mechanism also makes it difficult to track which tokens have been seen; our attempts at an extension to item spectra collection were brittle and unreliable, therefore, we do not use ANTLR in our evaluation of item-level localization.

### 5.2. Spectra for table-driven LR parsers using CUP

*Rule spectra.* In table-driven LR parsing, there are no parse functions that could be tracked. Instead, a small parser core interprets the LR tables and maintains an explicit state stack, where each state represents a set of items $\{A_i \to \alpha_i \bullet \beta_i\}$. The application of a rule is then carried out by the two main operations on the stack, shift and reduce. For a valid string, we can rely simply on the reduce operation to extract the rule spectrum, since all rule applications end successfully with a reduction. For invalid strings, we use the reduce operation to capture the rules applied fully to the left of the error position, i.e., at the viable prefix, but we also need to capture the partially applied rules and the frontier rules. These are both reflected in the states that remain on the stack when the parser encounters an error. The frontier rules are by construction given by the *non-kernel* items of the state at the top of the stack, while each kernel item $A_i \to \alpha_i \bullet \beta_i$ at the top of the stack represents a partially applied rule, with the yield of each $\beta_i$ describing the prefixes of a possible right continuation.

Fig. 3 shows CUP's parse stack when it uses the faulty grammar $G'_{Toy}$ to parse the test case `program a = { while a do sleep; }.` and encounters the syntax error at `sleep`. We get *expr*:4 as result of a complete rule application at the reduce operation in state 25. The non-kernel items at the top of stack give us the frontier rules {*block*:1, *block*:2, *block*:3, *block*:4} while the single kernel item gives us *stmt*:3.

Further partially applied rules are associated with the kernel items from states further down on the stack; we therefore traverse the stack and extract these. We do not extract any rules that are associated with non-kernel items only because these rules have arguably not been applied even partially, as the designated position is at the begin of the rule. In the example, we get the rules {*prog*:1, *block*:1, *block*:2, *block*:3, *block*:4, *stmt*:3, *expr*:1, *expr*:2} from this stack traversal; note that the four *block*-rules are logged again.

Note also that in this example, some rules are extracted from kernel items $A \to \alpha \bullet \beta$ even though they cannot be applied in a *maximally* viable $k$-prefix bounded derivation. Consider for example the two *expr*-rules in state 50. Here, our implementation is only an approximation of Definition 3.2, but we show in our evaluation that this does not necessarily lead to poor localization performance. The extraction of rule spectra that matches the definition precisely would require some extra bookkeeping, as it requires further stack unwinding to filter out items that cannot be applied in viable $k$-prefix bounded derivation. We leave this for future work.

Since CUP does not provide the required logging capabilities, we modified the table interpreter accordingly. For the rule spectra and the plain item spectra, 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 items in each state by analyzing CUP's output when it builds the parse tables.

*Item spectra.* Plain item spectra logging as per Definitions 4.1 and 4.2 in closely follows the approach for rule spectra logging described above; we do not need to map items left on the stack to their corresponding rules, but simply extract them "as they are". For every successful reduction, we log all items of the corresponding reduced rule. The definition of shift item spectra is easy to operationalize in LR-parsers: since the parse stack represents a viable prefix, the spectra are composed of the (kernel and non-kernel) items in the states that are pushed on the stack.

### 5.3. Synthetic rule spectra

Sections 5.1 and 5.2 focused on the traditional and perhaps most common use case of grammar development, that of developing grammars to use as inputs to compiler-compiler tools. However, applications such as grammar-based testing and fuzzing take a more general view. Here, the grammar is not implemented directly by the SUT but serves as an abstract model for the input domain of a system under test. In such applications, it is often hard to extend the SUT to produce spectra as it is processing the inputs, e.g., because no standalone parser can be extracted.

However, we can construct *synthetic (rule) spectra* through automatic test suite generation, as long as we have an oracle $\mathcal{O}$ (typically the SUT itself) that can answer *membership queries*, i.e., decide whether a generated test case is in the target language $\mathcal{L} = L(\mathcal{O})$. More specifically, for each positive test case $w$ with $\Delta = S \Rightarrow^* w$ and $w \in L(\mathcal{O})$ (resp. $w \notin L(\mathcal{O})$), we have $p \sim_{\checkmark} w$ (resp. $p \sim_{\times} w$) for all $p \in \text{rules}(\Delta)$. For negative test cases we rely on the rule mutation algorithm by Raselimo et al. (2019), and add the applied mutated rule to the spectrum, i.e., if $\Delta = S \Rightarrow^* \alpha A \beta$, $r = A \to \gamma \in P$ with a mutation $r' = A \to \gamma'$, $\Delta' = \alpha\gamma'\beta \Rightarrow^* w$, and $w \in L(\mathcal{O})$, then we have $p \sim_{\times} w$ for all $p \in \text{rules}(\Delta) \cup \text{rules}(\Delta') \cup \{r\}$ (and similarly for passing negative tests $w \notin L(\mathcal{O})$).

## 6. Research questions

We evaluate our approaches under a number of different scenarios, including the use of different parsing techniques, test suites, and ranking metrics. Overall, we are trying to answer six main research questions discussed in more detail below.

In the first set of experiments, we evaluate our method over a large number of medium-sized single fault grammars, which are constructed from a common grammar by fault seeding. Note that fault seeding is widely used in SFL evaluation (e.g., Abreu, 2009; Wen et al., 2011) because it produces a large number of faulty subjects with known error locations. This is the largest set of experiments in our evaluation. We use this set of experiments to answer four of the six research questions.

First, as a baseline, we investigate the effectiveness of rule-level fault localization. We analyze in detail whether the different applied parsing techniques, test suites, and ranking metrics have an effect on the effectiveness of the technique.

**RQ1** How effective are fault localization techniques based on rule spectra in identifying seeded single faults in grammars?

Then, we investigate the effectiveness of synthetic spectra.

[4] Note that this requires the compilation option `-DcontextSuperClass=RuleContextWithAltNum` in order to get the right alternative for a matched rule. The call to `getAltNumber()` returns the default value 0 otherwise.

[5] I.e., the call to `enterOuterAltNum` is missing, see the open issue #2222.

| state | corresponding kernel items | corresponding non-kernel items |
|---|---|---|
| 2 | *prog* → **program** • `id` = *block* `.` | |
| 3 | *prog* → **program** `id` • = *block* `.` | |
| 4 | *prog* → **program** `id` = • *block* `.` | *block* → • `{` *decls stmts* `}`<br>*block* → • `{` *decls* `}`<br>*block* → • `{` *stmts* `}`<br>*block* → • `{ }` |
| 5 | *block* → `{` • *decls stmts* `}`<br>*block* → `{` • *decls* `}`<br>*block* → `{` • *stmts* `}`<br>*block* → `{` • `}` | *decls* → • *decls decl* `;`<br>*decls* → • *decl* `;`<br>*decl* → • **bool**<br>*decl* → • **int**<br>*stmts* → • *stmts stmt* `;`<br>*stmts* → • *stmt* `;`<br>*stmt* → • **sleep**<br>*stmt* → • **if** *expr* **then** *stmt* **else** *stmt*<br>*stmt* → • **while** *expr* **do** *block*<br>*stmt* → • `id` = *expr*<br>*stmt* → • *block*<br>*block* → • `{` *decls stmts* `}`<br>*block* → • `{` *decls* `}`<br>*block* → • `{` *stmts* `}`<br>*block* → • `{ }` |
| 8 | *stmt* → **while** • *expr* **do** *block* | *expr* → • *expr* = *expr*<br>*expr* → • *expr* + *expr*<br>*expr* → • `(` *expr* `)`<br>*expr* → • `id`<br>*expr* → • `num` |
| *25* | *expr* → `id` • | |
| 50 | *stmt* → **while** *expr* • **do** *block*<br>*expr* → *expr* • = *expr*<br>*expr* → *expr* • + *expr* | |
| 51 | *stmt* → **while** *expr* **do** • *block* | *block* → • `{` *decls stmts* `}`<br>*block* → • `{` *decls* `}`<br>*block* → • `{` *stmts* `}`<br>*block* → • `{ }` |

**Fig. 3.** CUP parse stack when encountering the syntax error while parsing the test case `program a = {while a do sleep;}.` with $G'_{Toy}$. State 25 (shown in gray) is popped off the stack after reduction.

**RQ2** How effective are fault localization techniques based on synthetic spectra in identifying seeded single faults?

Next, we evaluate whether our method remains effective when we switch to item spectra for a more precise localization of faults at the level of individual symbols, and whether the switch from rule spectra does indeed improve its accuracy.

**RQ3** How effective are fault localization techniques based on item spectra in identifying seeded single faults in grammars at the level of individual symbols?

**RQ4** Does the use of item spectra improve the localization accuracy?

In the second set of experiments, we look at grammars submitted by students enrolled in compiler engineering courses. Such grammars contain *real* and often *multiple* faults; however, SFL is based on a single fault assumption, and SFL methods can be misled by interactions between multiple faults (Abreu et al., 2009; Xue and Namin, 2013). We investigate whether this is the case here.

**RQ5** How effective are fault localization techniques in identifying *real* faults in grammars that possibly contain multiple faults?

In the final experiment, we address the *scalability* of our approach by applying it to a large, production-quality grammar, specifically the ANTLR4 SQLite grammar.

**RQ6** Does our approach remain effective for large grammars?

## 7. Localization for seeded faults

### 7.1. Experimental setup

*Base grammars.* We used the grammar of a small artificial programming language called SIMPL as the basis for these experiments. SIMPL was originally designed for a second-year computer architecture course at Stellenbosch University, where students were given an LL(1) grammar for SIMPL in EBNF format, and had to implement a recursive-descent parser. We manually eliminated the EBNF operators for this grammar by adding new BNF rules, in order to simplify the mutation process.

For ANTLR (v4.7.2), we left-factorized the BNF version and eliminated left-recursive rules to minimize the effect of its adaptive LL(*)

parsing mechanism, which can lead to imprecise spectra (see the discussion in Section 5.1). The resulting grammar contains 84 rules, 42 non-terminal symbols, and 47 terminal symbols.

JavaCC (v7.0.5) also requires left-factorization and left-recursion elimination; we used the ANTLR version as a starting point, but used slightly different representation of inner alternatives. This version contains 93 rules, 49 non-terminal symbols, and 47 terminal symbols, giving rise to 242 items.

CUP (v0.11b) requires the elimination of the EBNF extensions; this version was developed independently by a student assistant directly from the EBNF version. It contains 80 rules, 32 non-terminal symbols, and 47 terminal symbols, giving rise to 258 items.

The parsers generated from the three respective baseline grammars pass all tests in the different test suites (see below).

*Mutation operators.* We 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 also used two more complex mutation operators that take the entire rules into account rather than individual grammar symbols. More specifically, we introduce an $\epsilon$-production to every non-nullable non-terminal, and we blindly delete alternatives for every rule. We only applied a single mutation to derive each mutant, to ensure that each mutant contains at most one fault. We discarded all grammar mutants where the parser generator fails to produce a parser (e.g., by introducing indirect left-recursion in an ANTLR grammar). This leaves us with 30930 mutants for ANTLR, 36497 mutants for JavaCC, and 26548 mutants for CUP.

*Test suites.* We then executed each mutant on four different test suites derived from the original EBNF form of the SIMPL grammar. Note that our test suite generation tool gtestr internally eliminates those EBNF operator as well. The first two test suites, *rule* and *cdrc*, contain only positive test cases. They are constructed according to the rule and cdrc coverage criteria (Lämmel, 2001), respectively, and contain 43 and 86 test cases, respectively. Note that *rule* is a proper subset of *cdrc*. *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 ($bfs_2$, $step_6$, and derivable and adjacent pair coverage, respectively) developed by van Heerden et al. (2020) 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 (Raselimo et al., 2019). *instructor* refers to the test suite the instructor used to grade the student submissions. It comprises 20 positive and 61 negative tests.

*Mutant selection.* All widely used SFL metrics require that the test suite contains at least one passing and one failing test in order to properly localize faults (cf. Section 2). For any given test suite, we therefore only select those mutants as basis for our evaluation where this property holds. Moreover, for ANTLR we do not select mutants where failing tests require the application of a left-recursive rule, because the computed grammar spectra are imprecise (see the discussion in Section 5.1). In analogy to the terminology used in mutation testing, we call the selected mutants *killed*.

*Spectrum extraction and ranking.* We extracted the rule-level spectra for all target systems as described in Section 5. Note CUP's rule-level spectra approximate Definition 3.2 and that ANTLR's adaptive LL(*) parsing mechanism interferes with the spectrum extraction, as described in Section 5.1. We used both methods discussed there, and denote the version without error recover in the following by ANTLR*.

For JavaCC we followed the approach described in Section 5.1 to extract item-level spectra; we denote this version by $\text{JavaCC}_{\texttt{item}}$. We did not extract item-level spectra for ANTLR, due to the stability issues discussed in Section 5.1. For CUP, we extracted both plain item spectra ($\text{CUP}_{\texttt{plain}}$) and shift item spectra ($\text{CUP}_{\texttt{shift}}$).

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. In the rule-level spectra, we resolved ties using the middle rank, as discussed in Section 2. In the symbol-level spectra, we also used the *k-max* tie breaking strategy described in Section 4.4.

We finally use an item-level extension of the rule spectra where, given a rule $r$ with assigned suspiciousness score $s$, we replace $r$ by a set of all possible items $r^\bullet$ that can be derived from $r$ and assign $s$ to each item of $r$. For example, consider the worked example in Section 4.1: the rule $stmt \rightarrow \texttt{sleep}$ from the grammar $G'_{foy}$ which has a Tarantula score of 0.64 is replaced by its two items $stmt \rightarrow \bullet\texttt{sleep}$ and $stmt \rightarrow \texttt{sleep}\bullet$, which both get the Tarantula score of 0.64. We then re-rank these reconstructed items using the mid-rank tie breaking strategy.

Note that we used the location at which we applied the mutation operation as "true" fault location. However, as described in Section 7.2, the proper "blame assignment" is not always as clear, in particular when the mutations impact the first-sets of rules. This can impact the quality of the predictions.

*Synthetic spectra.* We used the SIMPL grammar mutants containing individual symbol mutations (but not the more complex rule-level mutations) for ANTLR to generate *rule*, *cdrc*, and *large* test suites from each mutant in order to answer RQ2. The ANTLR grammar from which these mutants were derived acts as the ground-truth grammar (i.e., oracle $\mathcal{O}$) and determines the outcome of each test case derived from these mutants. Our grammar-based test case generator tool gtestr contains a converter that translates ANTLR grammars to their equivalent gtestr grammars. We discarded mutants with unreachable non-terminal symbols and those that contain symbols for which gtestr cannot compute the yield. This leaves us with 27894 mutants.

### 7.2. Rule-level fault localization of seeded faults (RQ1)

Fig. 4 shows the rule-level results of the fault seeding experiments in 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 5 contains more details.

While the details change with the applied parsing technology and ranking metric, and the underlying test suite, the boxplots and Table 5 show overall positive results. On average, the metrics rank the faulty rules at ~22% of all rules, with better results for the *large* test suite (~15%) and worse results for the *instructor* test suite (~31%). The median is typically at 2.5%–5% (except Tarantula under *instructor* at ~17%), 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. Furthermore, in 10%–40% of the cases they correctly pinpoint it, in about 15%–60% the fault is localized within the top three rules and in up to 65% of the cases the fault is found within the top five rules.

We can make a few high-level observations. First, fault localization works better for JavaCC than for both ANTLR and CUP: for JavaCC 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 with mixed results that are sometimes better for CUP than the other tools). The Top1/Top3/Top5 values also seem to be in favor of JavaCC, with ANTLR giving us slightly lower values across the board.

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

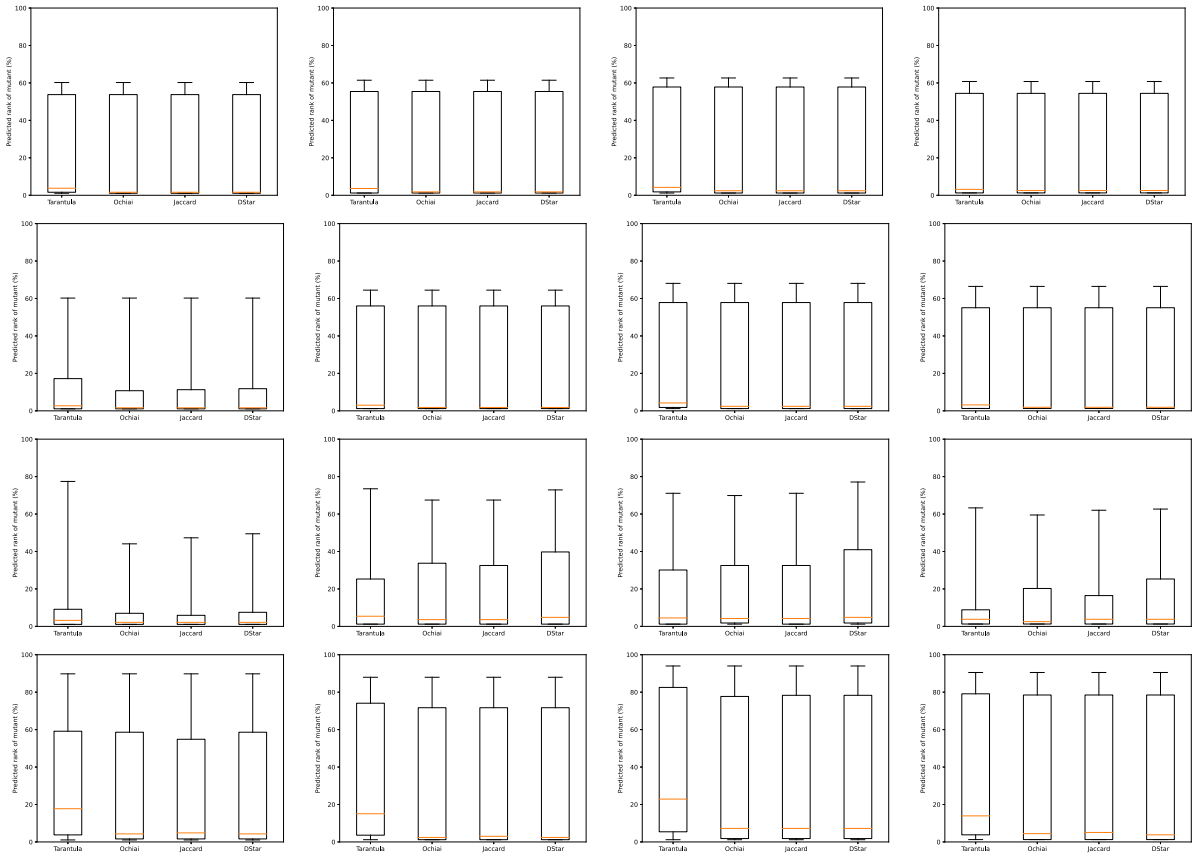



**Fig. 4.** Rule-level results of fault seeding experiments over SIMPL grammar. Columns show results for different parsers, left to right: JavaCC, ANTLR (without error correction), ANTLR* (with default error correction), and CUP. Rows show results for different test suites, top to bottom: *rule* (43 positive tests), *cdrc* (86 positive tests), *large* (2964 positive tests, 32157 negative tests), *instructor* (20 positive tests, 41 negative tests). The boxes show the Q3/Q1 interquartile range of the ranks, thee median is indicated by a dotted line across the box, whiskers extend from the 5th to the 95th percentile. Table 5 contains more details.

**Table 5**
Detailed rule-level 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, #3 and #5 denote the number of cases where the seeded fault was ranked in the Top 3 and Top 5, respectively. The first four blocks contain the main results to answer RQ1, the next three blocks demonstrate the results for position-one mutants. The final block contains the results for synthetic spectra used to answer RQ2.

| | | Killed | Tarantula | | | | | Ochiai | | | | | Jaccard | | | | | DStar | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | $\tilde{x}$ | $\bar{x}$ | *#1* | *#3* | *#5* | $\tilde{x}$ | $\bar{x}$ | *#1* | *#3* | *#5* | $\tilde{x}$ | $\bar{x}$ | *#1* | *#3* | *#5* | $\tilde{x}$ | $\bar{x}$ | *#1* | *#3* | *#5* |
| ANTLR | *rule* | 24 385 | 3.6 | 22.7 | 6 235 | 12 656 | 13 653 | 1.8 | 21.8 | 7 166 | 14 890 | 15 081 | 1.8 | 21.8 | 7 169 | 14 891 | 15 082 | 1.8 | 21.8 | 7 160 | 14 880 | 15 077 |
| (30930) | *cdrc* | 24 545 | 3.0 | 22.8 | 6 562 | 12 732 | 13 824 | 1.8 | 22.1 | 7 276 | 15 041 | 15 281 | 1.8 | 22.2 | 7 277 | 15 041 | 15 279 | 1.8 | 22.2 | 7 270 | 15 028 | 15 173 |
| | *large* | 30 629 | 5.4 | 19.1 | 8 384 | 13 225 | 16 359 | 3.6 | 19.1 | 8 811 | 15 397 | 17 065 | 3.6 | 18.9 | 8 701 | 15 791 | 17 732 | 4.8 | 21.5 | 8 564 | 14 882 | 16 499 |
| | *instr* | 26 282 | 15.1 | 32.9 | 4 090 | 7 601 | 10 390 | 2.4 | 28.6 | 7 062 | 14 143 | 14 930 | 3.0 | 28.8 | 6 878 | 13 934 | 14 723 | 2.4 | 28.6 | 7 058 | 14 472 | 14 929 |
| ANTLR* | *rule* | 24 349 | 4.2 | 22.7 | 5 718 | 11 284 | 13 618 | 2.4 | 21.4 | 6 780 | 15 199 | 15 442 | 2.4 | 21.4 | 6 781 | 15 200 | 15 394 | 2.4 | 21.4 | 6 760 | 15 148 | 15 378 |
| | *cdrc* | 24 508 | 4.2 | 22.7 | 5 902 | 11 530 | 13 928 | 2.4 | 21.7 | 6 909 | 15 334 | 15 678 | 2.4 | 21.7 | 6 909 | 15 320 | 15 625 | 2.4 | 21.7 | 6 900 | 15 316 | 15 549 |
| | *large* | 30 590 | 4.5 | 19.2 | 8 598 | 13 873 | 18 008 | 4.2 | 19.7 | 7 354 | 14 405 | 17 684 | 4.2 | 19.4 | 7 861 | 14 138 | 18 023 | 4.8 | 22.0 | 7 140 | 13 865 | 16 940 |
| | *instr* | 26 211 | 22.9 | 39.7 | 2 170 | 4 837 | 6 897 | 7.2 | 31.0 | 4 631 | 10 360 | 11 973 | 7.2 | 31.3 | 4 543 | 10 048 | 11 574 | 7.2 | 30.9 | 4 956 | 10 895 | 11 980 |
| JavaCC | *rule* | 28 463 | 3.8 | 17.9 | 6 987 | 14 012 | 17 761 | 1.6 | 16.5 | 8 659 | 19 350 | 19 774 | 1.6 | 16.5 | 8 657 | 19 348 | 19 770 | 1.6 | 16.6 | 8 646 | 19 217 | 19 695 |
| (36497) | *cdrc* | 29 138 | 2.7 | 16.5 | 8 673 | 15 614 | 19 021 | 1.6 | 15.2 | 10 401 | 20 135 | 20 755 | 1.6 | 15.2 | 10 401 | 20 134 | 20 754 | 1.6 | 15.5 | 9 817 | 19 494 | 20 200 |
| | *large* | 32 383 | 3.2 | 12.5 | 10 523 | 16 857 | 19 474 | 2.2 | 9.0 | 12 128 | 20 842 | 23 338 | 2.2 | 8.6 | 12 343 | 20 996 | 23 830 | 2.2 | 9.7 | 12 048 | 21 041 | 23 347 |
| | *instr* | 30 864 | 17.7 | 32.4 | 2 448 | 7 035 | 9 711 | 4.3 | 27.8 | 6 378 | 14 240 | 16 162 | 4.8 | 26.9 | 6 274 | 14 015 | 16 083 | 4.3 | 27.7 | 6 350 | 14 198 | 16 117 |
| CUP | *rule* | 23 564 | 3.2 | 24.4 | 7 789 | 12 385 | 13 400 | 2.5 | 23.7 | 9 909 | 13 881 | 13 925 | 2.5 | 23.7 | 9 909 | 13 882 | 13 925 | 2.5 | 23.7 | 9 909 | 13 870 | 13 913 |
| (26548) | *cdrc* | 25 336 | 3.2 | 24.6 | 8 378 | 13 502 | 14 588 | 1.9 | 24.0 | 10 502 | 14 919 | 14 972 | 1.9 | 24.0 | 10 502 | 14 921 | 14 972 | 1.9 | 24.0 | 10 500 | 14 908 | 14 954 |
| | *large* | 26 487 | 3.8 | 11.9 | 8 653 | 15 505 | 18 706 | 2.5 | 14.0 | 10 132 | 15 030 | 16 617 | 3.8 | 14.2 | 9 359 | 14 947 | 17 667 | 3.8 | 15.9 | 10 279 | 14 603 | 15 979 |
| | *instr* | 25 459 | 13.9 | 35.6 | 2 702 | 7 239 | 9 829 | 4.4 | 32.8 | 6 890 | 11 497 | 13 217 | 5.1 | 33.2 | 6 505 | 10 646 | 13 030 | 3.8 | 32.7 | 7 429 | 13 001 | 13 858 |
| ANTLR | *rule* | – | 55.4 | 48.8 | 145 | 1 095 | 1 297 | 55.4 | 49.3 | 170 | 875 | 1 042 | 55.4 | 49.3 | 173 | 876 | 1 043 | 55.4 | 49.3 | 164 | 868 | 1 038 |
| | *cdrc* | – | 56.5 | 49.9 | 206 | 1 028 | 1 241 | 56.5 | 50.3 | 207 | 906 | 1 088 | 56.5 | 50.4 | 208 | 906 | 1,086 | 56.5 | 50.5 | 201 | 893 | 980 |
| | *large* | – | 8.3 | 23.5 | 3 003 | 4 529 | 5 679 | 31.0 | 32.7 | 1 547 | 2 460 | 2 907 | 22.0 | 31.4 | 1 691 | 2 946 | 3 656 | 38.1 | 38.7 | 1 095 | 1 894 | 2 327 |
| | *instr* | – | 79.8 | 62.0 | 307 | 1 042 | 1 414 | 79.8 | 63.3 | 387 | 1 176 | 1 236 | 79.8 | 63.3 | 380 | 1 175 | 1 240 | 79.8 | 63.8 | 369 | 1 139 | 1 182 |
| JAVACC | *rule* | – | 53.8 | 37.8 | 940 | 2 511 | 3 143 | 53.8 | 37.2 | 1 706 | 3 041 | 3 361 | 53.8 | 37.2 | 1 704 | 3 039 | 3 357 | 53.8 | 37.4 | 1 693 | 2 908 | 3 282 |
| | *cdrc* | – | 54.8 | 34.5 | 2 231 | 3 437 | 4 037 | 54.8 | 34.1 | 2 793 | 3 677 | 3 987 | 54.8 | 34.1 | 2 793 | 3 677 | 3 987 | 54.8 | 34.7 | 2 209 | 3 036 | 3 432 |
| | *large* | – | 4.3 | 15.5 | 4 172 | 6 163 | 7 393 | 6.5 | 17.5 | 4 321 | 5 879 | 6 492 | 3.2 | 14.7 | 4 900 | 6 921 | 7 693 | 8.1 | 19.5 | 4 122 | 5 661 | 6 141 |
| | *instr* | – | 59.1 | 53.6 | 179 | 1 235 | 1 794 | 61.8 | 52.4 | 947 | 2 311 | 2 737 | 61.8 | 52.2 | 920 | 2 290 | 2 595 | 62.4 | 52.5 | 914 | 2 244 | 2 651 |
| CUP | *rule* | – | 54.4 | 53.6 | 305 | 394 | 435 | 54.4 | 53.7 | 296 | 387 | 420 | 54.4 | 53.7 | 296 | 388 | 420 | 54.4 | 53.7 | 296 | 379 | 418 |
| | *cdrc* | – | 55.1 | 55.2 | 342 | 429 | 458 | 55.1 | 55.3 | 327 | 404 | 445 | 55.1 | 55.3 | 327 | 406 | 445 | 55.1 | 55.3 | 327 | 401 | 433 |
| | *large* | – | 6.3 | 19.9 | 2 210 | 3 885 | 4 656 | 27.2 | 29.8 | 1 067 | 1 587 | 1 955 | 19.0 | 28.0 | 1 242 | 2 567 | 3 248 | 31.6 | 34.3 | 813 | 1 022 | 1 251 |
| | *instr* | – | 81.0 | 72.9 | 263 | 436 | 501 | 81.0 | 73.5 | 324 | 435 | 461 | 81.0 | 73.6 | 327 | 436 | 460 | 81.0 | 73.7 | 322 | 431 | 456 |
| Synthetic | *rule* | 23 563 | 1.8 | 5.1 | 9 894 | 14 805 | 20 197 | 1.2 | 2.0 | 13 012 | 22 772 | 23 085 | 1.2 | 2.0 | 13 012 | 22 772 | 23 085 | 1.2 | 1.9 | 13 014 | 22 780 | 23 105 |
| (27894) | *cdrc* | 25 756 | 1.8 | 4.7 | 10 761 | 17 284 | 22 141 | 1.2 | 2.1 | 13 582 | 24 189 | 24 978 | 1.2 | 2.1 | 13 567 | 24 170 | 24 964 | 1.2 | 2.1 | 13 583 | 24 212 | 24 995 |
| | *large* | 27 538 | 2.4 | 9.0 | 12 866 | 19 083 | 22 058 | 1.2 | 2.8 | 16 482 | 24 187 | 25 296 | 1.2 | 3.3 | 15 766 | 22 972 | 24 727 | 1.2 | 2.7 | 16 625 | 24 223 | 25 320 |

**Table 6**
Detailed item-level results of fault seeding experiments over SIMPL grammars. The left-most column gives the tie resolution mechanism (see Section 4.4); *rule extension* denotes the item-level extension of rule-level spectra (see Section 7.1). See Table 5 for numbers of generated and killed mutants, respectively, and for further notation details.

| | | | Tarantula | | | | | Ochiai | | | | | Jaccard | | | | | DStar | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | $\bar{x}$ | $\tilde{x}$ | *#1* | *#3* | *#5* | $\bar{x}$ | $\tilde{x}$ | *#1* | *#3* | *#5* | $\bar{x}$ | $\tilde{x}$ | *#1* | *#3* | *#5* | $\bar{x}$ | $\tilde{x}$ | *#1* | *#3* | *#5* |
| Middle rank | $\text{JavaCC}_{\text{item}}$ | *rule* | 2.9 | 20.2 | 128 | 9 941 | 13 559 | 1.4 | 19.5 | 211 | 13 645 | 16 456 | 1.4 | 19.5 | 212 | 13 645 | 16 456 | 1.4 | 19.6 | 211 | 13 577 | 16 382 |
| | | *cdrc* | 2.7 | 20.5 | 186 | 10 386 | 13 841 | 1.4 | 19.9 | 268 | 14 000 | 17 058 | 1.4 | 19.9 | 269 | 14 001 | 17 058 | 1.4 | 19.9 | 259 | 13 882 | 16 933 |
| | | *large* | 1.7 | 8.3 | 5 228 | 14 757 | 18 016 | 0.8 | 7.5 | 10 162 | 20 786 | 22 220 | 0.8 | 7.0 | 9 919 | 20 924 | 22 762 | 0.6 | 8.5 | 10 205 | 21 142 | 22 176 |
| | | *instr* | 8.5 | 26.7 | 807 | 5 686 | 7 980 | 1.2 | 23.6 | 4 613 | 16 578 | 18 508 | 1.2 | 23.7 | 4 591 | 16 307 | 18 243 | 1.2 | 23.7 | 4 600 | 16 559 | 18 570 |
| | $\text{CUP}_{\text{plain}}$ | *rule* | 3.9 | 14.2 | 3 412 | 7 367 | 8 790 | 3.3 | 13.2 | 4 083 | 8 596 | 10 035 | 3.3 | 13.2 | 4 083 | 8 595 | 10 033 | 3.3 | 13.1 | 4 083 | 8 596 | 10 035 |
| | | *cdrc* | 3.9 | 13.9 | 3 799 | 7 910 | 9 811 | 2.9 | 12.8 | 4 498 | 9 191 | 11 397 | 3.1 | 12.9 | 4 498 | 9 189 | 10 970 | 2.9 | 12.8 | 4 498 | 9 191 | 11 397 |
| | | *large* | 5.2 | 13.5 | 1 718 | 6 371 | 8 707 | 2.5 | 10.1 | 6 047 | 9 611 | 12 545 | 3.9 | 12.1 | 5 957 | 9 006 | 11 380 | 1.9 | 10.5 | 6 670 | 10 513 | 13 482 |
| | | *instr* | 11.4 | 23.9 | 1 459 | 3 286 | 5 269 | 4.7 | 16.6 | 4 367 | 8 232 | 10 272 | 4.7 | 18.4 | 4 364 | 8 003 | 10 203 | 3.9 | 16.1 | 4 428 | 8 429 | 10 445 |
| | $\text{CUP}_{\text{shift}}$ | *rule* | 2.1 | 20.0 | 3 126 | 8 874 | 11 572 | 1.6 | 19.3 | 3 469 | 10 843 | 13 139 | 1.6 | 19.3 | 3 469 | 10 843 | 13 138 | 1.6 | 19.3 | 3 469 | 10 843 | 13 139 |
| | | *cdrc* | 2.1 | 20.1 | 3 954 | 9 975 | 12 624 | 1.4 | 19.4 | 4511 | 12 079 | 14 499 | 1.4 | 19.5 | 4 511 | 12 079 | 14 498 | 1.4 | 19.5 | 4 511 | 12 077 | 14 498 |
| | | *large* | 4.3 | 13.8 | 2 082 | 7 669 | 10 214 | 1.9 | 12.5 | 8 842 | 12 304 | 13 615 | 2.3 | 12.8 | 8 086 | 10 678 | 12 824 | 1.6 | 12.9 | 10 151 | 12 633 | 13 584 |
| | | *instr* | 12.2 | 29.8 | 1 173 | 4 021 | 5 353 | 3.3 | 26.4 | 4 029 | 9 250 | 11 323 | 3.5 | 26.4 | 4 032 | 9 165 | 11 013 | 3.1 | 26.4 | 4 056 | 9 535 | 11 558 |
| *k-max* | $\text{JavaCC}_{\text{item}}$ | *rule* | 1.9 | 19.6 | 5 919 | 12 082 | 15 553 | 0.8 | 19.1 | 6 935 | 16 848 | 17 461 | 0.8 | 19.1 | 6 936 | 16 848 | 17 460 | 0.8 | 19.2 | 6 935 | 16 780 | 17 399 |
| | | *cdrc* | 1.7 | 19.9 | 6 247 | 12 687 | 15 980 | 0.8 | 19.5 | 7 575 | 17 265 | 17 886 | 0.8 | 19.5 | 7 576 | 17 266 | 17 886 | 0.8 | 19.5 | 7 566 | 17 167 | 17 770 |
| | | *large* | 1.7 | 8.2 | 7 691 | 15 142 | 18 232 | 0.6 | 7.4 | 13 883 | 20 916 | 22 182 | 0.6 | 6.9 | 13 586 | 20 992 | 22 695 | 0.6 | 8.4 | 14 046 | 21 144 | 22 140 |
| | | *instr* | 7.6 | 26.3 | 2 856 | 7 017 | 8 959 | 0.8 | 23.3 | 9 714 | 17 654 | 18 510 | 0.8 | 23.4 | 9 683 | 17 285 | 18 147 | 0.8 | 23.4 | 9 712 | 17 589 | 18 413 |
| | $\text{CUP}_{\text{plain}}$ | *rule* | 2.1 | 13.4 | 5 057 | 8 361 | 11 505 | 1.4 | 12.6 | 6 007 | 9 394 | 13 345 | 1.4 | 12.7 | 6 007 | 9 393 | 13 343 | 1.4 | 12.6 | 6 007 | 9 394 | 13 345 |
| | | *cdrc* | 1.7 | 13.1 | 5 512 | 9 235 | 12 811 | 1.4 | 12.2 | 6 499 | 10 337 | 14 923 | 1.4 | 12.3 | 6 499 | 10 335 | 14 496 | 1.4 | 12.3 | 6 499 | 10 337 | 14 923 |
| | | *large* | 5.2 | 13.5 | 1 801 | 6 427 | 8 711 | 2.5 | 10.1 | 6 303 | 9 740 | 12 566 | 3.9 | 12.1 | 6 199 | 9 054 | 11 404 | 1.9 | 10.5 | 6 941 | 10 520 | 13 494 |
| | | *instr* | 10.9 | 23.6 | 2 022 | 5 011 | 6 867 | 4.1 | 16.4 | 5 660 | 9 694 | 10 877 | 4.1 | 18.2 | 5 655 | 9 580 | 10 810 | 3.7 | 15.9 | 5 725 | 9 889 | 11 049 |
| | $\text{CUP}_{\text{shift}}$ | *rule* | 1.0 | 19.1 | 5 447 | 13 257 | 14 034 | 0.8 | 18.8 | 5 930 | 14 431 | 14 997 | 0.8 | 18.8 | 5 930 | 14 431 | 14 996 | 0.8 | 18.8 | 5 930 | 14 431 | 14 997 |
| | | *cdrc* | 0.8 | 19.2 | 7 999 | 14 314 | 15 295 | 0.8 | 19.0 | 8 799 | 15 488 | 16 041 | 0.8 | 19.0 | 8 799 | 15 486 | 16 038 | 0.8 | 19.0 | 8 799 | 15 485 | 16 037 |
| | | *large* | 4.1 | 13.6 | 2 168 | 7 926 | 10 360 | 1.6 | 12.2 | 9 119 | 12 449 | 13 784 | 2.3 | 12.7 | 8 341 | 10 891 | 12 882 | 1.6 | 12.8 | 10 601 | 12 684 | 13 656 |
| | | *instr* | 12.0 | 29.4 | 1 935 | 5 438 | 7 521 | 3.1 | 26.1 | 5 734 | 10 845 | 12 191 | 3.1 | 26.1 | 5 732 | 10 764 | 11 880 | 2.7 | 26.1 | 5 763 | 11 133 | 12 425 |
| *Rule extension* | JavaCC | *rule* | 3.7 | 17.9 | 14 | 6 475 | 10 182 | 2.1 | 16.5 | 23 | 8 608 | 14 460 | 2.1 | 16.5 | 23 | 8 608 | 14 460 | 2.1 | 16.5 | 23 | 8 593 | 14 448 |
| | | *cdrc* | 3.1 | 16.4 | 25 | 8 271 | 11 787 | 1.9 | 15.2 | 33 | 10 298 | 15 537 | 1.9 | 15.2 | 33 | 10 298 | 15 536 | 2.1 | 15.5 | 29 | 9 712 | 14 941 |
| | | *large* | 3.1 | 12.1 | 37 | 9 873 | 13 241 | 2.1 | 8.9 | 46 | 11 437 | 16 550 | 2.1 | 8.4 | 48 | 11 552 | 16 498 | 2.1 | 9.7 | 45 | 11 304 | 16 484 |
| | | *instr* | 17.1 | 32.0 | 7 | 1 796 | 4 258 | 4.5 | 27.3 | 16 | 5 494 | 11 568 | 4.5 | 26.6 | 16 | 5 381 | 11 080 | 4.5 | 27.2 | 16 | 5 543 | 11 611 |
| | CUP | *rule* | 4.5 | 24.8 | 3 | 4 889 | 6 784 | 3.5 | 24.2 | 4 | 6 202 | 8 990 | 3.5 | 24.2 | 4 | 6 202 | 8 990 | 3.5 | 24.2 | 4 | 6 202 | 8 990 |
| | | *cdrc* | 4.5 | 25.1 | 6 | 5 465 | 7 469 | 3.3 | 24.5 | 7 | 6 785 | 9 522 | 3.3 | 24.5 | 7 | 6 785 | 9 522 | 3.3 | 24.5 | 7 | 6 783 | 9 519 |
| | | *large* | 3.1 | 11.5 | 34 | 6 321 | 10 325 | 3.7 | 14.9 | 38 | 6971 | 11 098 | 3.5 | 14.8 | 37 | 6 446 | 10 654 | 3.7 | 17.1 | 37 | 6 992 | 11 090 |
| | | *instr* | 15.1 | 35.9 | 25 | 2 023 | 3 216 | 7.6 | 33.3 | 34 | 5 438 | 7 919 | 7.8 | 33.7 | 33 | 5 051 | 7 536 | 6.4 | 33.1 | 33 | 5 683 | 8 408 |

correction, although the differences are typically smaller than those to JavaCC and CUP.

Third, the difference between Ochiai, Jaccard, and DStar is negligible, but all three outperform Tarantula. The only exception is 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). This follows the observation that Tarantula does not get particularly overwhelmed by a high count of failing tests compared to the other three metrics, which under such scenarios typically assign the highest rank to the non-faulty rules (mostly dominating ones, e.g., the start rule) executed in most failing tests.

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 is marginal, despite the fact that that they contain very similar test cases and, in fact, *cdrc* even 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. *large* has the highest fraction of localized faults compared to other test suites, and the smallest mean (but typically not median) values.

We can thus answer our first research question.

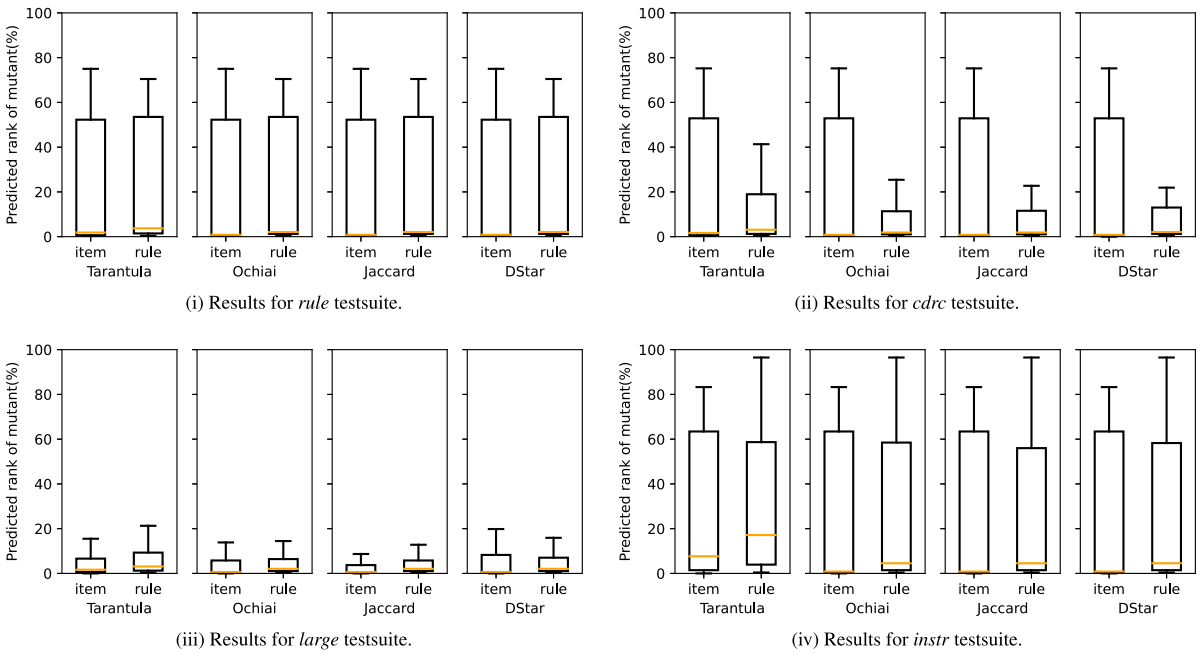
**RQ1**: Our fault localization based on rule spectra is effective in identifying faults in fault-seeded grammars. In more than half of the cases, the fault is localized within the top three rules. In about 10%–30% of the cases, the fault is uniquely identified as the most suspicious rule. We observe also that Ochiai, Jaccard and DStar, by and large, produce identical rankings, and outperform Tarantula.

*Results for position-one mutants.* The application of a rule, no matter the parsing technology used, heavily relies on lookahead symbols. These lookahead tokens are derived from the first and follow sets, at least in the case of our target LL(1) and LALR(1) parsers. Therefore, it becomes a challenge to correctly localize faults where the first symbol in a rule's right-hand side has been mutated (so called position-one mutants) as the mutated rules may never be executed. The results from the second block of Table 5 demonstrate how hard it indeed is to correctly localize these mutants. Specifically, in subject grammars in which the mutation operators have been applied to the first position on the right hand-side of a rule, the median rank of the faults range between 50%–82%, i.e., the localization performance can differ by 20 percentage points between mutants at the first and other symbols. Moreover, the average ranks are lower than median ranks, which means there are only a few cases where the fault is localized in fewer than half of the rules. The *large* test suite is an exception, with lower median values (less than 27%) because the generation of the negative test cases contained in *large* in part follow similar mutation strategies used to seed faults in the subject grammars.

However, we found that errors in such positions are relatively rare in practice, perhaps due to the large effect they have on the behavior of the parser, which causes grammar developers to realize these errors quickly.

### 7.3. Synthetic rule-level localization of seeded faults (RQ2)

The third block in Table 5 shows the results of our approach when using synthetic spectra derived directly from the test cases constructed from a grammar under test. We see that our approach remains effective and, in fact, produces even better results. First, on average, the faulty rule is ranked in ∼5% of the rules, with a slightly worse figure (9.0%) for Tarantula under the *large* test suite. The median range of 1.2%–2.8% means that in half of the cases, we only need to look within the




(i) Results for *rule* testsuite.

(ii) Results for *cdrc* testsuite.

(iii) Results for *large* testsuite.

(iv) Results for *instr* testsuite.

**Fig. 5.** Item-level localization results over fault-seeded SIMPL grammars using JavaCC$_{\texttt{item}}$. *item* refers to originally extracted spectra using the *k-max* tie resolution strategy, *rule* to the item-level extension of the rule spectra (see Section 7.1). See Fig. 4 for more details.

top three rules to find the faulty rule. Second, interestingly, the fault is uniquely localized in at least 40% of the cases and in ∼85% of the cases the prediction is within the top five rules. Third, as before, Tarantula performs worse than the three other metrics, with higher median and mean ranks and lower Top1/Top3/Top5 values. Finally, we do not run into the first position mutants issues here because all the rules are effectively and fully reduced during the generation of individual test cases.

**RQ2**: Our localization based on synthetic rule spectra is more effective in identifying single faults in mutants than using grammar spectra extracted from parsers. The fault is found within the top five rules in almost all cases.

### 7.4. Item-level localization of seeded faults (RQ3)

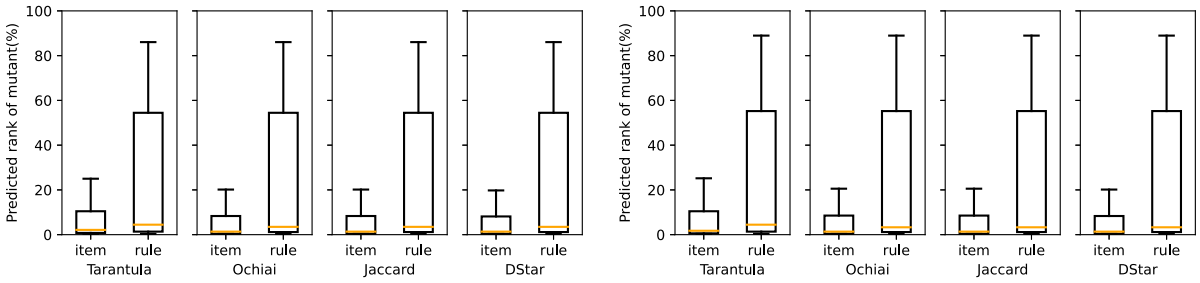
Table 6 presents our experimental results for the item-level localization in three blocks. The first block shows the results using the default ranking assigned to items based on the suspiciousness scores computed by the different metrics. Tied items, i.e., items with the same suspiciousness scores, are assigned a rank using the mid-rank tie breaking mechanism. The second block summarizes results using the *k-max* tie breaking strategy (see Section 4.4). In both cases, CUP$_{\texttt{shift}}$ refers to the results based on shift item spectra (see Definition 4.3) while CUP$_{\texttt{plain}}$ refers to the results for plain item spectra (see Definitions 4.1 and 4.2).

The third block enables a fair comparison between item- and rule-level localization. This comparison is based on the fact that when given a correctly predicted fault using rule-level fault localization, we still need to look (in the worst case) at all the symbols at the right-hand side of a faulty rule to find the offending symbol(s). On average, we need to inspect half of the symbols in the rule to identify the exact fault location. We can therefore extend the rule-level spectra to item-level spectra, as described in Section 7.1.

*Experimental results.* We first focus on the configuration without specialized tie breaking, as shown in the first block of Table 6. We observe the following results. First, as in the rule-level localization, Ochiai, Jaccard, and DStar outperform Tarantula, here even for all test suites and parsing technologies: they give lower mean and median values, and identify more faults. For the smaller test suites (*rule*, *cdrc*, and *instr*), the differences between Ochiai, Jaccard, and DStar are marginal; for the *large* test suite, Jaccard slightly outperforms Ochiai and DStar for JavaCC but underperforms for both versions of CUP. Second, the mixed test suite *large* yields better results than the other three test suites. Using *large*, we are able to uniquely localize the seeded fault in 6%–32% of the cases, and in 24%–65% and 32%–70% of the cases the fault is localized in the Top3 and Top5 of the ranked items, respectively. Third, the choice of the parsing technology does have an effect on fault localization. With the exception of low Top1 values, item-level localization appears to be more effective in JavaCC$_{\texttt{item}}$ than in both CUP$_{\texttt{plain}}$ and CUP$_{\texttt{shift}}$. With JavaCC$_{\texttt{item}}$, the fault is typically located at a median rank of 0.6%–8.5%, hence, in more than half of the cases the fault is correctly predicted within the top five items. Another interesting insight, and perhaps hardly surprising, is that we cannot tell apart effectiveness of fault localization based on plain and shift item spectra in the LR case. In particular, CUP$_{\texttt{shift}}$ finds more faults within the top five items but has slightly worse mean ranks across the board than CUP$_{\texttt{plain}}$.

*Tie breaking.* The second block of Table 6 summarizes the fault localization results, where we break ties using *k-max* strategy that picks the item with the highest position among tied items from the same rule. In general, in most cases we see an increase in effectiveness — the median and mean ranks are improved and Top1/Top3/Top5 numbers increase substantially; in particular, we see up to 30× increase in the number of seeded faults that are pinpointed exactly (i.e., Top1) when we are using small *rule* and *cdrc* test suites, and still a 2× increase for *instr*. The relative performance of the different metrics, however, remains largely unaffected.

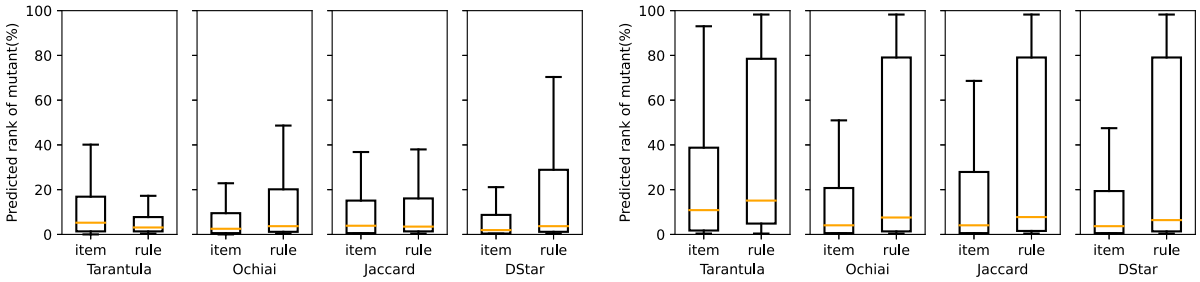
**RQ3**: Our fault localization method based on item spectra remains effective in identifying single faults in grammars with seeded faults. The tie breaking mechanism that prefers the item with the highest position over other items from the same rule improves the results substantially.




(i) Results for *rule* testsuite.




(ii) Results for *cdrc* testsuite.

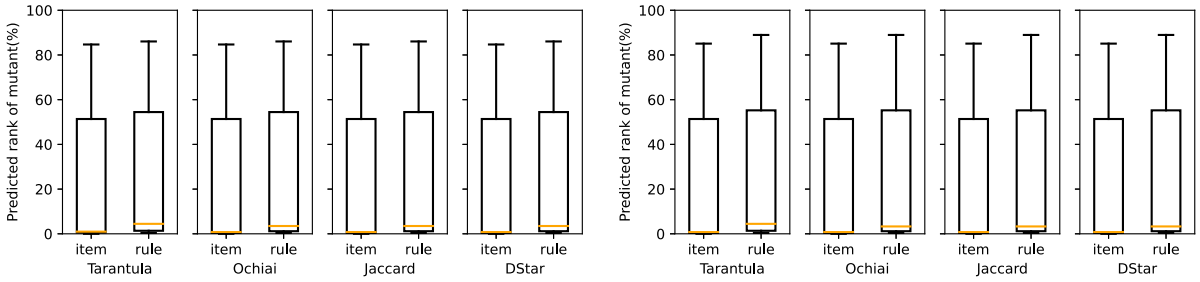



(iii) Results for *large* testsuite.




(iv) Results for *instr* testsuite.

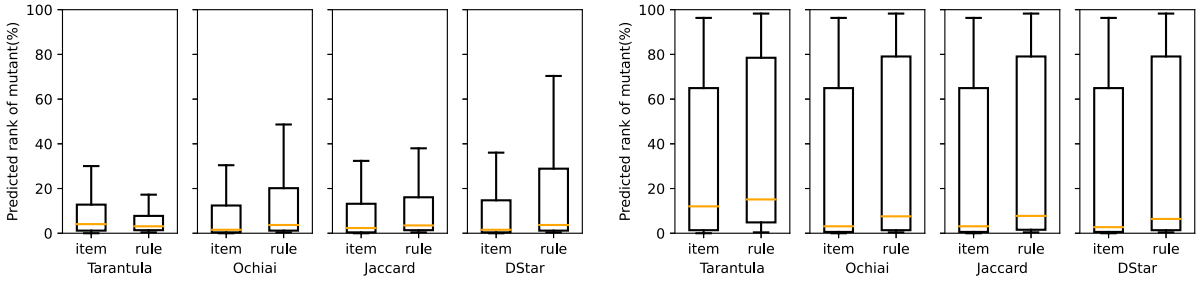
**Fig. 6.** Item-level localization results over fault-seeded SIMPL grammars using $\text{CUP}_{\texttt{plain}}$. See Fig. 5 for more details.




(i) Results for *rule* testsuite.




(ii) Results for *cdrc* testsuite.




(iii) Results for *large* testsuite.




(iv) Results for *instr* testsuite.

**Fig. 7.** Item-level localization results over fault-seeded SIMPL grammars using $\text{CUP}_{\texttt{shift}}$. See Fig. 5 for more details.

### 7.5. Rule- vs item-level fault localization (RQ4)

Figs. 5, 6, and 7 summarize the comparison of item- and rule-level fault localization as a series of paired box plots. Each pair contains ranks from item spectra and ranks from rule spectra computed using each ranking metric over each test suite on each of the two parsing mechanisms. We derive the comparison from the second block (*k-max*) and the third block (*rule extension*) from Table 6.

While results differ with the applied parsing mechanism and the underlying test suite, it is easily observable that the *k-max* strategy performs better than a simple extension of rule-level localization. This is made evident by better median (0.6%–10.9% vs 2.1%–17.2%) and mean ranks. However, note that for JavaCC the rule extension gives more uniform results, with better 95th and 75th percentile values.

**RQ4**: Item-level localization with the specialized tie breaking mechanism outperforms the simplistic extension of rule-level localization where all positions within a rule are assigned the same score.

## 8. Localization of real faults (RQ5)

In order to see how well our method performs over grammars with multiple real faults we used grammars students submitted in

**Table 7**
Results of iterative fault localization in student grammars and manual repair. #fail shows the number of failing test cases in an iteration and rank shows rank of the manually repaired rule.

| # | Language | Type | Iteration 1 | | Iteration 2 | | Iteration 3 | | Iteration 4 | | Iteration 5 | | Iteration 6 | | Iteration 7 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | #Fail | Rank | #Fail | Rank | #Fail | Rank | #Fail | Rank | #Fail | Rank | #Fail | Rank | #Fail | Rank |
| 1 | SIMPL | CUP | 557 | 1.5 | 254 | 1 | 131 | 1 | 98 | 1 | | | | | | |
| 2 | SIMPL | CUP | 206 | 2 | 95 | 2 | | | | | | | | | | |
| 3 | SIMPL | CUP | 498 | 1 | 40 | 1 | | | | | | | | | | |
| 4 | SIMPL | CUP | 305 | 5 | 48 | 1 | | | | | | | | | | |
| 5 | SIMPL | CUP | 854 | 3 | 854 | 1 | 295 | 1 | 139 | 1 | 48 | 2 | 19 | 1 | 5 | 1 |
| 6 | SIMPL | CUP | 48 | 1 | | | | | | | | | | | | |
| 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 | 4.5 | 1 | 1.5 | | | | | | | | | | |
| 16 | SIMPL | ANTLR | 361 | 3 | 46 | 1 | | | | | | | | | | |
| 17 | SIMPL | ANTLR | 396 | 1.5 | 117 | 2 | 81 | 2 | 47 | 1 | 1 | 1.5 | | | | |
| 18 | SIMPL | ANTLR | 46 | 2 | | | | | | | | | | | | |
| 19 | SIMPL | ANTLR | 356 | 1 | 233 | 2 | 1 | 1 | | | | | | | | |
| 20 | SIMPL | ANTLR | 1 | 1 | | | | | | | | | | | | |

assignments of various compiler engineering courses. Unsurprisingly, these grammars contain many errors.

*Experimental setup.* We used two languages, SIMPL (which we also used for the fault seeding experiments in Section 7), and Blaise, another artificial teaching language of similar syntactic complexity: the instructor's EBNF version of the Blaise 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 7. 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 at the University of Southampton and at Stellenbosch University. In one assignment, the students were given the same EBNF as in the computer architecture course (in fact, most students had already been exposed to SIMPL in that course), and were asked in two different assignments to use ANTLR and a LALR(1) parser generator of their choice, respectively, to develop parsers for SIMPL. We randomly picked ten ANTLR submissions, from which we discarded two that passed all tests and one that did not produce a compilable parser. We picked all ten CUP submissions, from which we discarded three that passed all tests and one that passed none. For Blaise, the students were given a textual language description and a small set of short example programs and asked to develop an ANTLR grammar and parser for the language. We randomly picked nine Blaise grammars from 110 submissions, from which we also discarded two that pass all tests. This left us with 20 subject grammars.

We then followed an iterative one-bug-at-a-time (OBA) debugging technique (Zakari et al., 2020) where we focus our attention on the first discovered fault, fix this fault and then re-localize. In each iteration, 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" languages to identify and repair faulty rules. In each iteration, we only repaired the top-ranked faulty rule; note that we made repairs in the lexer as well. After each repair, we repeated the process, until the grammar under test passed all test cases.

*Experimental results.* Table 7 summarizes the results of our evaluation over student grammars. For each iteration, we show the number of test cases failed in the respective grammar version, and the rank of the rule that we identified as faulty and repaired in the next iteration. Empty cells indicate that a previous iteration's 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 of the grammars the number of failed test cases decreases with each repair; the exception is grammar #8, where the repair of the second iteration triggers more failing test cases. This 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. In other cases, we could identify similar lexical errors via the rules.

In Blaise, most faults related to the structure of the formal and actual function parameter lists. These cannot be empty, but the textual language specification was vague about this, and many students chose a wrong structure. The other fault classes include:

1. the interaction between the parser and lexer, which some students did not handle well, especially in cases involving unary and binary MINUS operators, with the latter subsumed by ADDOP operators;
2. token issues such as typographical errors and wrong regular expressions (strings in most cases); and
3. a few cases of tool specific issues, e.g., wrong use of EBNF operators in ANTLR.

**RQ5**: Our fault localization remains effective under multiple faults: in all steps of an OBA approach, the repaired rule is within the top five rules, and even was the top ranked rule in more than half of the cases.

## 9. Localization for large black-box systems (RQ6)

To address questions related to scalability of our approach, we try to identify parts of a public SQLite grammar that is known to deviate from the language accepted by the actual SQLite system. We retrieved the ANTLR4 SQLite grammar from https://shorturl.at/fhsBG. The BNF version of the grammar that has been used to generate test queries has 440 rules, 181 non-terminals and 170 terminals. This shows that it is a fairly large grammar, at almost 5× the size of the SIMPL grammars we used in Sections 7.2 and 7.4. The black-box system that we used is the sqlite3 Python module (v2.6.0) that is essentially

```
1 model :- create_table_stmt | create_virtual_table_stmt .
2 %%% set the allowed table names
3 table_name :- 'STAFF' | 'DEPARTMENTS' | 'ORDERS'.
4 %%% set the allowed column names
5 column_name :- 'Name' | 'Department' | 'Number'.
6 %% set database name
7 databasename :- 'databasename' .
```

**Fig. 8.** Example model that generates tests from `CREATE TABLE` statements, with hard-coded allowed values for table, column and database names.

an API wrapper for a runtime SQLite library (v3.22.0) written in the C programming language. We then wrote a simple adaptor that creates a database connection, executes generated queries and logs each execution outcome.

We assume that the adaptor provides, on executing each query, the syntactic pass/fail information. Here, we consider a test case to fail if the system detects any syntax errors in the input and to pass if the query executes successfully or if it throws exceptions that lie deeper in the system, beyond the syntax analysis stage. With the above framework established, it appears straight-forward to directly invoke our "flipped" version of our method that uses synthetic grammar spectra derived directly from test cases to identify deviations. However, it proved impractical to blindly generate and run queries on the system, despite our sole interest in exercising the parser. In particular, the system complains of early stage errors such as `incomplete input`; perhaps more importantly, since the runtime system implements a relaxed phase distinction, execution can stop due to a semantically ill-formed query before it could complete parsing. Lack of a standalone parser also means that we cannot directly exercise these generated queries.

To tackle the aforementioned limitations and handle some of the preconditions, we provide our test generator with predefined and fixed table and column names during query generation. Fortunately, the language accepted by the SQLite system is composed of different types of statements, which can be seen as sub-languages that define, query, and manipulate tables and data in different ways. For example, the symbol *sql_stmt* (which is reached directly from the start symbol *program*) below is a union of entry points to these sub-languages.

$$\begin{aligned} &\mathit{program} \rightarrow \mathit{sql\_stmt}\ (\texttt{;}\mathit{sql\_stmt})^* \\ &\mathit{sql\_stmt} \rightarrow \ldots \mid \mathit{create\_table\_stmt} \mid \mathit{create\_trigger\_stmt} \mid \ldots \end{aligned}$$

Our generator grammar defines over 20 alternatives for the *sql_stmt*-rule. This allows us to model the system effectively by overriding the start rule and start derivations from each sub-language entry-point. We also fix values for rules *table_name*, *column_name* and *database_name* by setting allowed names. For example, Fig. 8 shows a simplistic code snippet written in the Prolog programming language which our test suite generation tool uses as input that sets the start rule to `CREATE TABLE` related statements (line #1). The values to set table, column, and database names are given in lines #3, #5, and #7, respectively. This separation also allows us to handle some dependencies between statement types, e.g., the `DROP TRIGGER` statement requires a successful execution of the `CREATE TRIGGER` statement.

We therefore wrote a series of models (13 in total) like the one shown in Fig. 8. The one in the figure targets the statements related to `CREATE TABLE` and uses the *create_table_stmt* and *create_virtual_stmt* rules exclusively to generate tests. The other twelve models generate tests from the remaining statements. Unlike the model in Fig. 8, they assume the prior successful creation of tables, and like the first model, they are equipped with valid table and column names. Dependent statement types are handled by chaining up their corresponding rules with a semicolon separator. For example, *create_trigger_stmt* is always followed by optional sequence `;` *drop_trigger_stmt*, i.e.,

$$\mathit{model} \rightarrow \mathit{create\_trigger\_stmt}\ (\texttt{;}\ \mathit{drop\_trigger\_stmt})?$$

Note that *drop_trigger_stmt* can also be used independently with an optional `IF EXISTS` clause to avoid a corresponding dependency-related exception to be thrown. However, this was less intuitive as it required meddling with the grammar in order to enforce the clause to always be present in the `DROP TRIGGER` statements and to other similar highly dependent statements.

We are aware that this exploitation of the structure of the SQLite grammar targets certain parts of the system and does not exercise all grammar rules. However, we still managed to cover a large portion of the grammar. For example, spectra from the model that tests create table related statements are composed of 274 rules that are applied in the generation of the *deriv* and $bfs_2$ test suites. The highest number of applied rules in generation of the test suite by any model is 371 out of the total of 440 rules.

We then followed a multi-stage fault localization approach where we, in each stage, use each model to orchestrate the generation of the test suite, which we then use to localize deviations for the language accepted by the SQLite system. In each stage, we then used the same OBA technique as in Section 8, where we again focus our attention on the first discovered deviation, manually fix this deviation, and then re-generate the tests to re-test the SQLite system. In each iteration, we used the Tarantula metric to calculate suspiciousness scores for all grammar rules. This choice of ranking metric is based on the observation that Tarantula seemed to produce more stable rankings under a high number of test failures. We examined these rules in their order of suspiciousness, starting with the most suspicious rule, and identified their corresponding syntactic description as per the SQLite official specification available at https://sqlite.org/syntaxdiagrams.html. We then manually inspected the grammar rule and its corresponding description to identify the cause of deviation. We finally repaired the deviation in the grammar rule and repeated this process until no further tests failed.

### 9.1. Experimental results

*Deviation #1.* The first model (see Fig. 8) gave us an initial set of 462 failing tests out of a total of 57656 generated tests. In the first iteration, we made the following observations. First, Tarantula ranked the rule

$$\mathit{expr} \rightarrow \mathit{expr}\ (\texttt{=} \mid \texttt{==} \mid \ldots \mid \texttt{IS} \mid \texttt{IN} \mid \ldots)\ \mathit{expr}$$

as the most suspicious rule. The rule defines the structure of binary operators in SQLite. We consulted the official documentation and the corresponding description for expressions, which revealed that the deviation is in the use of `IN` operator. This operator has to be followed by parenthesized expressions and not by an arbitrary expression, as is allowed by the grammar rule. However, the grammar contains multiple faults (or more precisely, "deviations"). This is made evident by the fact that the above faulty *expr*-rule has not been executed in all failing tests, but only in 456 of those failing tests.

Since we follow the OBA principle, we first fixed the deviation in the `IN` operator. We, in fact, found another *expr*-rule that correctly implemented the `IN` operator as follows:

$$\mathit{expr} \rightarrow \mathit{expr}\ \texttt{NOT}?\ \texttt{IN}\ (\mathit{select\_stmt} \mid \mathit{expr}\ (\texttt{,}\mathit{expr})?)$$

This means that the top-ranked rule $\mathit{expr} \rightarrow \mathit{expr}\,(\ldots \mid \texttt{IN} \mid \ldots)\,\mathit{expr}$ is an over-approximation fault on the correct use of the `IN` operator, which we fixed by simply deleting the faulty alternative of the *expr*-rule that has the `IN` operator.

*Deviation #2.* After the modification, we regenerated the test suites using the same (first) model and repeated the process. In this iteration, the following six tests failed:

```
create table STAFF as values(0) limit 0
create table STAFF
 as with STAFF as (select *) values(0) order by ?
create table STAFF
 as with STAFF as(select *) values(0) limit 0
```

```
create table STAFF as values(0) order by ? limit 0
create table STAFF as values(0) order by ?
create table STAFF
  as with STAFF as(select *) values(0) order by ? limit 0
```

The SQLite system reports the following syntax error messages each for each of the test cases above.

```
near ‘‘limit’’: syntax error
near ‘‘order’’: syntax error
near ‘‘limit’’: syntax error
near ‘‘order’’: syntax error
near ‘‘order’’: syntax error
near ‘‘order’’: syntax error
```

From the error messages, it is not straightforward to see where the cause of the deviation might be; the token ) occurs on the correctly consumed prefix before the offending **limit** and **order**. The *select_stmt*-rule that causes the deviation is ranked sixth (out of a total 440 rules).

*select_stmt* → (**WITH RECURSIVE** *common_table_expression* (, *common_table_expression*)*)?
*select_or_values*
(*compound_operator select_values*)*
(**ORDER BY** *ordering_term* (, *ordering_term*)*)?
(**LIMIT** *expr* ((**OFFSET** | ,) *expr*)?)?
*select_or_vals* → **SELECT** … *result_col* …
**FROM** *table_or_subquery* …
(**WHERE** *expr*)? …
| **VALUES** ( *expr* (, *expr*)* )

This deviation is confirmed by the official documentation for the VALUES clause in a select statement: “*There are some restrictions on the use of a VALUES clause that are not shown on the syntax diagrams*:

- *A VALUES clause cannot be followed by ORDER BY.*
- *A VALUES clause cannot be followed by LIMIT.*”

We fixed this deviation by transforming the rules as follows:

*select_stmt* → … *select_or_vals* …
*select_or_vals* → **SELECT** … *result_col* …
**FROM** *table_or_subquery* …
(**WHERE** *expr*)? …
(**ORDER BY** *ordering_term*(, *ordering_term*)*)?
(**LIMIT** *expr* ((**OFFSET** | ,) *expr*)?)?
| **VALUES** ( *expr* (, *expr*)* )

The transformation pushes down the sequences that capture the **ORDER BY**- and **LIMIT**-clauses to the end of the original first alternative of the *select_or_vals*-rule.

*Deviation #3.* In this iteration, we used a model that starts derivations from rules that define the structure of triggers (in particular, their creation and removal). More specifically, we have the following as the start production,

*model* → *create_trigger_stmt* (; *drop_trigger_stmt*)?

We execute a generated test suite with 60781 test cases from which 1960 fail. A tie between two top ranked rules below already give a good idea of the location of the deviation.

*with_clause* → **WITH RECURSIVE**? *cte_table_name*
**AS** ( *select_stmt* ) …
*cte_table_name* → *table_name*
(( *column_name* ( , *column_name*)*))?

The official documentation outlines syntax restrictions on **INSERT**, **UPDATE**, and **DELETE** statements within triggers: “*Common table expression are not supported for statements inside of triggers.*” The trigger-related rules in the grammar completely ignore this restriction and allow generic **INSERT**, **UPDATE**, and **DELETE** statements inside triggers. The faulty rules *with_clause* and *cte_table_name* are directly derivable from these statements; the latter only ever occurs in the former (i.e., *with_clause*-rule).

In the fix for this deviation, we simply duplicate rules from the three statements and remove the call to *with_clause*.

*Deviation #4.* The above fix did not cater for all failures as we are left with another 1152 failing tests after the modification. Here, all the test failures have the same structure and the system throws similar syntax error messages. Below, we show one of the failing tests

```
create trigger tr1
 delete on STAFF begin select 0 between 0 or 0 and 0; end
```

and its corresponding error message:

```
near ‘‘;’’: syntax error
```

From these failures we can, to some extent, conclude that the interaction among operators **BETWEEN**, **OR**, and **AND** (in that order) is problematic. Fault localization also confirms this with the two *expr*-rules (as shown below) are flagged as the most suspicious and both rules are applied in the derivation of all failing tests (i.e., both have $ef(e)$ and $nf(e)$ counts of 1152 and 0, respectively).

*expr* → …
| *expr* **OR** *expr*
| *expr* **NOT**? **BETWEEN** *expr* **AND** *expr*
| …

Taking a closer look, it seems the parser detects precedence issues between the operators **OR** and **AND** which has a higher precedence than **OR**, due to a parsing conflict between

*expr* → *expr* **AND** *expr*

and

*expr* → *expr* **NOT**? **BETWEEN** *expr* **AND** *expr*

To circumvent this behavior, wrapping parentheses around the *expr*-symbol before the **AND** token in the second *expr* rule seemed to be the most plausible fix. The modified rule is as follows:

*expr* → *expr* **NOT**? **BETWEEN** ( *expr* ) **AND** *expr*

Note that this is not a “proper” fix, but rather a “grammar hack” that does not modify the language, but it resolves all remaining test failures and so demonstrates that there are no further deviations

In this experiment, we see that our approach enables us to identify four deviations in the larger SQLite grammar. The OBA technique also allows us to find the first deviation in each stage with low wasted efforts as we only needed to inspect one rule to find the first deviation, five rules to find the second deviation, two rules to find the third deviation; finally, the cause of the fourth deviation was the top-ranked rule. The other models did not result in any test failures.

In summary, we can therefore answer RQ6 positively:

**RQ6**: The rule-level localization remains effective and scales to large production-quality grammars.

## 10. Threats to validity

Our experimental evaluation is subject the typical concerns regarding construct validity, in particular possible implementation and data collection errors, as well as statistical conclusion validity. However, there are additional challenges in validating the extension of our findings beyond the specific experimental set-up that we used, such as applying them to alternative ranking metrics, parsing methods,

grammars, or test suites. Our fault seeding experiments are based on a single grammar, meaning that variations in grammar structure may yield different outcomes due to dependencies on test suite construction, mutant creation, and spectrum collection. For example, we originally used a grammar version that was not left-factorized, which resulted in the ANTLR rule tracking issues outlined in Section 5.1 and yielded incomplete spectra, which distorted the results. Additionally, our fault seeding incorporates mutations at the initial symbol of a rule, potentially generating non-LL(1) mutants that also trigger the rule tracking issues. A preliminary analysis indicated that localization performance can differ significantly, by approximately 15 rules (i.e., nearly 20 percentage points), between mutations at the first symbol versus others. However, we used a variety of other grammars on other (non-seeded faults) experiments, without any substantially different results, which partially mitigates against this threat.

Gopinath et al. (2014) have demonstrated that mutants are not syntactically close to actual faults, but they remain valid substitutes in numerous software engineering applications, including fault localization (Just et al., 2014). However, with the exception of the work by Bendrissou et al. (2023), grammar mutations have not been investigated systematically and other mutation operations may yield different results. Hence, even though our localization experiments with student grammars (see Section 8) show similar results, care should be taken in generalizing the results above.

The experiments have shown that the localization performance is influenced by the composition of the test suites and may thus not generalize, despite the differences in the test suites we have used. The *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.

In the item-level localization experiments, we introduce more heuristic elements, especially in our handling of ties. The *k-max* tie breaking strategy is modeled on parse behavior over positive tests and while we also got better results for the larger mixed test suite which contain negative tests, *k-max* may not generalize to other grammars, parsing technologies, or other ranking metrics.

Since our evaluation in Sections 8 and 9 relies on subjective assessments 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 certain aspects such as rule selection and cut-off point determination were not fully specified.

## 11. Related work

We do not know any other work that directly shares our goal of identifying faulty rules in a grammar but there is a wide range of related work from different areas.

*Error recovery in parsers.* Traditional error recovery methods for parsers (Diekmann and Tratt, 2020) assume that the grammar is correct and the input incorrect, and modify the input or the parser state, not the grammar. Both their aims and techniques are therefore different from our work, and we do not consider such methods here.

*Spectrum-based fault localization.* Many different methods (e.g., static analysis, model-based reasoning, or deep learning) have been used in software fault localization; Wong et al. (2016) give a good survey of the entire field. We focus on spectrum-based methods only (de Souza et al., 2016; Wong et al., 2016).

More than 30 different metrics have been proposed in the literature (Naish et al., 2011; de Souza et al., 2016; Wong et al., 2016), with some originally developed for problems in other domains such as botany (Ochiai, 1957) or information retrieval. Many metrics produce identical rankings (Naish et al., 2011; Debroy and Wong, 2011). Theoretical studies (Xie et al., 2013) trying to identify optimal metrics have not been been successful in practice (Le et al., 2013). We use four of the most widely used metrics that have also performed well in other experimental evaluations (Abreu et al., 2006; Le et al., 2013): Tarantula (Jones and Harrold, 2005), Ochiai (1957), Jaccard (Chen et al., 2002), and DStar (Wong et al., 2014). Tarantula is the only of these metrics that takes program entities into account that are *not* executed in passing test cases (see Table 1); however, experiments (Abreu et al., 2006; Wong et al., 2014) have shown that Ochiai, Jaccard, and DStar are more effective for software fault localization. In an experimental comparison of different metrics over seven small or medium-sized programs (with a size of 20-124 basic blocks) from the widely used Siemens benchmark suite, (Abreu et al., 2006) report average ranks of the faulty statements between 1% and 50% of the respective program sizes, with overall averages of 23% for Tarantula, 22% for Jaccard, and 7% for Ochiai.

Our experiments in Sections 7.2 and 7.4 evaluate the efficacy of our SFL approach using grammars with seeded faults. Fault seeding has been used extensively in the literature (Naish et al., 2011; Wen et al., 2011; Abreu, 2009) although we use different, domain-specific mutation operators. We further successfully evaluate our solution over grammars with multiple faults in Sections 8 and 9 even though it is well known that SFL techniques are based on a single-fault assumption and that their accuracy deteriorates for programs with multiple faults (Abreu et al., 2009; Xue and Namin, 2013).

One of the open questions in SFL is tie resolution. Xu et al. (2011) present an evaluation of three heuristics for breaking ties viz., statement order-, confidence- and data dependency-based strategies. Our item-level fault localization uses a simple strategy that prefers the rightmost item of a rule which can be seen as a domain-specific version of a statement-order based tie breaking strategy. Our attempts to resolve ties by further exploiting the hierarchical structure of the grammar did not produce favorable results. Finally, Steimann et al. (2013) study the threats of validity for SFL. Our work inherits most of the threats of validity outlined in their study.

*Grammar smells and ambiguities.* A system *smell* is a specific system structure that indicates a violation of some fundamental design principles; it does not necessarily indicate a fault, but it can negatively impact the design quality and system's evolution. Stijlaart and Zaytsev (2017) have identified a number of grammar smells that are easily checkable, such as different cloning variants or unexpectedly nullable non-terminal symbols. One specific smell is *ambiguity*, which is undecidable in general (Cantor, 1962), although practical algorithms have been developed for several specific cases (Schröer, 2001; Schmitz, 2007, 2008; Brabrand et al., 2010; Basten, 2010). In particular, Basten (2010) describes an algorithm that 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 (2015) 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 smells are not necessarily faults. Consider for example the traditional "dangling else" problem (Aho et al., 2006). Most LR parsers resolve the ambiguity indicated through shift/reduce conflict by shifting, and so accept the intended language.

*Grammar-based test case generation.* (Purdom, 1972) described the first algorithm to systematically generate test suites from grammars; specifically, the algorithm generates the minimal number of sentences that is necessary to exercise all grammar rules. Malloy and Power (2001) give a more declarative formulation of the algorithm and Celentano et al. (1980) extend it with a minimal and a maximal sentence generation strategy. However, the test suites generated by these algorithms are too small and the individual test cases are too complex and cover too many rules, and are thus not well-suited for use in fault localization.

In contrast, context-dependent rule coverage (Lämmel, 2001) or *cdrc* yields more detailed test suites because it requires each rule to be applied to each non-terminal occurrence in any rule of the grammar. We also use several additional coverage criteria that we recently developed (van Heerden et al., 2020) to produce diverse positive test suites: $bfs_2$, a variant of *cdrc* that induces longer phrases; $step_k$, another variant of *cdrc* that induces deeper derivations; *derivable pair coverage (deriv)*, a variation of Zelenov and Zelenova's *pll* criterion (Zelenov and Zelenova, 2005) that also induces deeper derivations; and *adjacent pair coverage*, a criterion that ensures that all possible pairs in the follow-relation are covered.

The focus of grammar-based test case generation has mostly been on generating syntactically correct programs (i.e., positive tests) and only little attention has been devoted to the generation of programs with well-defined syntactic errors (i.e., negative tests). We have also developed two algorithms that construct negative test suites using word and rule mutation (Raselimo et al., 2019).

In random sentence generation, introduced by Hanford (1970), rules are randomly selected and applied until a complete sentence is derived. Such approaches often have a large number of control parameters (e.g., rule probabilities, symbol and rule counts, length, depth, and balance restrictions, and many others) that ensure that the derivation process terminates, and that the generated test suites have certain characteristics (Payne, 1978; Bird and Munoz, 1983; Homer and Schooler, 1989; Maurer, 1990, 1992; Lämmel and Schulte, 2006; Hoffman et al., 2011). Such methods have been used to test SQL (Slutz, 1998), C (Yang et al., 2011), and Java (Yoshikawa et al., 2003) processors, and are also applied in some fuzzing tools such as jsfunfuzz (Ruderman, 2007), the CSS grammar fuzzer (Ruderman, 2009), or langfuzz (Holler et al., 2012).

## 12. 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 a spectrum-based approach to localize faults in a context-free grammar where we replaced the concept of "executed statements" by that of "applied grammar elements", but kept the remaining established framework in place.

Our evaluation showed that our approach can localize faults in grammars with a high precision. In a large fault seeding experiment, the rule-level localization ranked the seeded faults within the top five most suspicious rule in more than half of the cases and pinpointed them (i.e., uniquely ranked them as the most suspicious) in 10%–30% of the cases, with significantly better results for our fault localization method based on synthetic rule spectra. Using the same experimental setup, the item-level localization method, coupled with a specialized tie breaking mechanism, ranked the seeded faults within the top five most suspicious positions in about 30%–60% of the cases; on average, it ranked them in about 10%–20% of all positions. It also significantly outperforms a simplistic extension of the rule-level localization, where all positions within a suspicious rule are given the same score. Generally, item-level localization is therefore preferable; the only exception is when the SUT is a black-box system and we need to resort to synthetic spectra.

We were also able to use our fault localization method to identify faults in student submissions that contain real and multiple faults. Finally, the fault localization method based on synthetic rule spectra identified four locations where a large SQLite grammar deviates from a language accepted by a production SQLite system.

*Future work.* We plan to extend our experimental evaluation to include further parsers and languages, but we also see several other interesting areas of future work. First, we can 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 7 suggest that our handling of negative test cases may be overly simplistic; we thus plan to use test suites where the expected outcome includes the error location. Third, we will use specialized spectrum-based fault localization algorithms such as FLITSR (Callaghan and Fischer, 2023) to improve the localization performance in the multi-fault case. Finally, we plan to refine the construction of synthetic spectra. Specifically, we can limit their amount of over-approximation by keeping tack of the positions in the test cases of the yield of each non-terminal, and thus only include rules whose yield starts before the error position. In a similar way, we can also construct synthetic *item* spectra.

## CRediT authorship contribution statement

**Moeketsi Raselimo:** Conceptualization, Data curation, Formal analysis, Investigation, Methodology, Software, Validation, Writing – original draft, Writing – review & editing. **Bernd Fischer:** Conceptualization, Formal analysis, Funding acquisition, Investigation, Validation, Visualization, Writing – original draft, Writing – review & editing, Methodology, Project administration, Resources, Supervision.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Data availability

Data will be made available on request.

## Acknowledgments

The financial assistance of the National Research Foundation (NRF), South Africa under Grants 113364 and SRUG2204234463 towards this research is hereby acknowledged. M. Raselimo was also supported by an SU Consolidoc bursary.

## References

Abreu, R., 2009. Spectrum-based Fault Localization in Embedded Software (Ph.D. thesis). Delft University of Technology, Netherlands, URL: http://resolver.tudelft.nl/uuid:78aa2510-acff-4acb-85ec-15852aa08e5c.

Abreu, R., Zoeteweij, P., van Gemund, A.J.C., 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, pp. 39–46. http://dx.doi.org/10.1109/PRDC.2006.18.

Abreu, R., Zoeteweij, P., van Gemund, A.J.C., 2009. Spectrum-based multiple fault localization. In: ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering. Auckland, New Zealand, November 16-20, 2009, IEEE Computer Society, pp. 88–99. http://dx.doi.org/10.1109/ASE.2009.25.

Aho, A.V., Sethi, M.S.L.R., Ullman, J.D., 2006. Compilers: Principles, Techniques, and Tools, second ed. Addison-Wesley.

ANTLR, 2018. ANTLR 4.7.2. URL: https://www.antlr.org/.

Basten, H.J.S., 2010. Tracking down the origins of ambiguity in context-free grammars. In: Cavalcanti, A., Déharbe, D., Gaudel, M., Woodcock, J. (Eds.), Theoretical Aspects of Computing - ICTAC 2010, 7th International Colloquium, Natal, Rio Grande Do Norte, Brazil, September 1-3, 2010. Proceedings. In: Lecture Notes in Computer Science, vol. 6255, Springer, pp. 76–90. http://dx.doi.org/10.1007/978-3-642-14808-8_6.

Bendrissou, B., Cadar, C., Donaldson, A.F., 2023. Grammar mutation for testing input parsers (Registered report). In: Böhme, M., Noller, Y., Ray, B., Szekeres, L. (Eds.), Proceedings of the 2nd International Fuzzing Workshop. FUZZING 2023, Seattle, WA, USA, 17 July 2023, ACM, pp. 3–11. http://dx.doi.org/10.1145/3605157.3605170.

Bird, D.L., Munoz, C.U., 1983. Automatic generation of random self-checking test cases. IBM Syst. J. 22 (3), 229–245. http://dx.doi.org/10.1147/sj.223.0229.

Brabrand, C., Giegerich, R., Møller, A., 2010. Analyzing ambiguity of context-free grammars. Sci. Comput. Program. 75 (3), 176–191. http://dx.doi.org/10.1016/j.scico.2009.11.002.

Callaghan, D., Fischer, B., 2023. Improving spectrum-based localization of multiple faults by iterative test suite reduction. In: Just, R., Fraser, G. (Eds.), Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2023, Seattle, WA, USA, July 17-21, 2023, ACM, pp. 1445–1457. http://dx.doi.org/10.1145/3597926.3598148.

Cantor, D.G., 1962. On the ambiguity problem of backus systems. J. ACM 9 (4), 477–479. http://dx.doi.org/10.1145/321138.321145.

Celentano, A., Crespi-Reghizzi, S., Vigna, P.D., Ghezzi, C., Granata, G., Savoretti, F., 1980. Compiler testing using a sentence generator. Softw. - Pract. Exp. 10 (11), 897–918. http://dx.doi.org/10.1002/spe.4380101104.

Chen, M.Y., Kiciman, E., Fratkin, E., Fox, A., Brewer, E.A., 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, pp. 595–604. http://dx.doi.org/10.1109/DSN.2002.1029005.

CUP, 2014. CUP 0.11b. URL: http://www2.cs.tum.edu/projects/cup/.

de Souza, H.A., Chaim, M.L., Kon, F., 2016. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. CoRR abs/1607.04347. URL: http://arxiv.org/abs/1607.04347.

Debroy, V., Wong, W.E., 2011. On the equivalence of certain fault localization techniques. In: Chu, W.C., Wong, W.E., Palakal, M.J., Hung, C. (Eds.), Proceedings of the 2011 ACM Symposium on Applied Computing (SAC). TaiChung, Taiwan, March 21 - 24, 2011, ACM, pp. 1457–1463. http://dx.doi.org/10.1145/1982185.1982498.

Diekmann, L., Tratt, L., 2020. Don't panic! Better, fewer, syntax errors for LR parsers. In: Hirschfeld, R., Pape, T. (Eds.), 34th European Conference on Object-Oriented Programming. ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference), In: LIPIcs, vol. 166, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 6:1–6:32. http://dx.doi.org/10.4230/LIPIcs.ECOOP.2020.6.

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

Gazzola, L., Micucci, D., Mariani, L., 2019. Automatic software repair: A survey. IEEE Trans. Softw. Eng. 45 (1), 34–67. http://dx.doi.org/10.1109/TSE.2017.2755013.

Ghanbari, A., Benton, S., Zhang, L., 2019. Practical program repair via bytecode mutation. In: Zhang, D., Møller, A. (Eds.), Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2019, Beijing, China, July 15-19, 2019, ACM, pp. 19–30. http://dx.doi.org/10.1145/3293882.3330559.

Gopinath, R., Jensen, C., Groce, A., 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, pp. 189–200. http://dx.doi.org/10.1109/ISSRE.2014.40.

Hanford, K.V., 1970. Automatic generation of test cases. IBM Syst. J. 9 (4), 242–257. http://dx.doi.org/10.1147/sj.94.0242.

Havrikov, N., Zeller, A., 2019. Systematically covering input structure. In: 34th IEEE/ACM International Conference on Automated Software Engineering. ASE 2019, San Diego, CA, USA, November 11-15, 2019, IEEE, pp. 189–199. http://dx.doi.org/10.1109/ASE.2019.00027.

van Heerden, P., Raselimo, M., Sagonas, K., Fischer, B., 2020. Grammar-based testing for little languages: an experience report with student compilers. In: Lämmel, R., Tratt, L., de Lara, J. (Eds.), Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering. SLE 2020, Virtual Event, USA, November 16-17, 2020, ACM, pp. 253–269. http://dx.doi.org/10.1145/3426425.3426946.

Hoffman, D., Ly-Gagnon, D., Strooper, P.A., Wang, H., 2011. Grammar-based test generation with YouGen. Softw. - Pract. Exp. 41 (4), 427–447. http://dx.doi.org/10.1002/spe.1017.

Holler, C., Herzig, K., Zeller, A., 2012. Fuzzing with code fragments. In: Kohno, T. (Ed.), Proceedings of the 21th USENIX Security Symposium. Bellevue, WA, USA, August 8-10, 2012, USENIX Association, pp. 445–458, URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler.

Homer, W., Schooler, R., 1989. Independent testing of compiler phases using a test case generator. Softw. - Pract. Exp. 19 (1), 53–62. http://dx.doi.org/10.1002/spe.4380190106.

IEEE Std 610.12-1990, 1990. IEEE standard glossary of software engineering terminology. pp. 1–84. http://dx.doi.org/10.1109/IEEESTD.1990.101064.

Isradisaikul, C., Myers, A.C., 2015. Finding counterexamples from parsing conflicts. In: Grove, D., Blackburn, S. (Eds.), Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. Portland, OR, USA, June 15-17, 2015, ACM, pp. 555–564. http://dx.doi.org/10.1145/2737924.2737961.

JavaCC, 2020. JavaCC 7.0.5. URL: https://javacc.github.io/javacc/.

Jones, J.A., Harrold, M.J., 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In: Redmiles, D.F., Ellman, T., Zisman, A. (Eds.), 20th IEEE/ACM International Conference on Automated Software Engineering. ASE 2005, November 7-11, 2005, Long Beach, CA, USA, ACM, pp. 273–282. http://dx.doi.org/10.1145/1101908.1101949.

Jones, J.A., Harrold, M.J., Stasko, J.T., 2002. Visualization of test information to assist fault localization. In: Tracz, W., Young, M., Magee, J. (Eds.), Proceedings of the 24th International Conference on Software Engineering. ICSE 2002, 19-25 May 2002, Orlando, Florida, USA, ACM, pp. 467–477. http://dx.doi.org/10.1145/581339.581397.

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

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J., 1997. Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (Eds.), ECOOP'97 — Object-Oriented Programming. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 220–242. http://dx.doi.org/10.1007/BFb0053381.

Kiers, B., 2016. ANTLR4 grammar for SQLite 3.8.x. URL: https://github.com/bkiers/sqlite-parser.

Lämmel, R., 2001. Grammar testing. In: Hußmann, H. (Ed.), 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. In: Lecture Notes in Computer Science, vol. 2029, Springer, pp. 201–216. http://dx.doi.org/10.1007/3-540-45314-8_15.

Lämmel, R., Schulte, W., 2006. Controllable combinatorial coverage in grammar-based testing. In: Uyar, M.U., Duale, A.Y., Fecko, M.A. (Eds.), Testing of Communicating Systems, 18th IFIP TC6/WG6.1 International Conference, TestCom 2006, New York, NY, USA, May 16-18, 2006, Proceedings. In: Lecture Notes in Computer Science, vol. 3964, Springer, pp. 19–38. http://dx.doi.org/10.1007/11754008_2.

Le, T.B., Thung, F., Lo, D., 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, pp. 380–383. http://dx.doi.org/10.1109/ICSM.2013.52.

Malloy, B.A., Power, J.F., 2001. An interpretation of purdom's algorithm for automatic generation of test cases. In: 1st ACIS Annual International Conference on Computer and Information Science. URL: http://eprints.maynoothuniversity.ie/6434/.

Maurer, P.M., 1990. Generating test data with enhanced context-free grammars. IEEE Softw. 7 (4), 50–55. http://dx.doi.org/10.1109/52.56422.

Maurer, P.M., 1992. The design and implementation of a grammar-based data generator. Softw. - Pract. Exp. 22 (3), 223–244. http://dx.doi.org/10.1002/spe.4380220303.

Monperrus, M., 2018. Automatic software repair: A bibliography. ACM Comput. Surv. 51 (1), 17:1–17:24. http://dx.doi.org/10.1145/3105906.

Naish, L., Lee, H.J., Ramamohanarao, K., 2011. A model for spectra-based software diagnosis. ACM Trans. Softw. Eng. Methodol. 20 (3), 11:1–11:32. http://dx.doi.org/10.1145/2000791.2000795.

Ochiai, A., 1957. Zoogeographical studies on the soleoid fishes found in Japan and its neighhouring regions-II. Bull. Jpn. Soc. Sci. Fish. 22 (9), 526–530. http://dx.doi.org/10.2331/suisan.22.526.

Payne, A.J., 1978. A formalised technique for expressing compiler exercisers. SIGPLAN Not. 13 (1), 59–69. http://dx.doi.org/10.1145/953428.953435, URL: http://doi.acm.org/10.1145/953428.953435.

Purdom, P., 1972. A sentence generator for testing parsers. BIT 366–375. http://dx.doi.org/10.1007/BF01932308.

Raselimo, M., Fischer, B., 2019. Spectrum-based fault localization for context-free grammars. In: Nierstrasz, O., Gray, J., d. S. Oliveira, B.C. (Eds.), Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering. SLE 2019, Athens, Greece, October 20-22, 2019, ACM, pp. 15–28. http://dx.doi.org/10.1145/3357766.3359538.

Raselimo, M., Fischer, B., 2021. Automatic grammar repair. In: Visser, E., Kolovos, D.S., Söderberg, E. (Eds.), SLE '21: 14th ACM SIGPLAN International Conference on Software Language Engineering. Chicago, IL, USA, October 17 - 18, 2021, ACM, pp. 126–142. http://dx.doi.org/10.1145/3486608.3486910.

Raselimo, M., Taljaard, J., Fischer, B., 2019. Breaking parsers: mutation-based generation of programs with guaranteed syntax errors. In: Nierstrasz, O., Gray, J., d. S. Oliveira, B.C. (Eds.), Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering. SLE 2019, Athens, Greece, October 20-22, 2019, ACM, pp. 83–87. http://dx.doi.org/10.1145/3357766.3359542.

Renieris, M., Reiss, S.P., 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, pp. 30–39. http://dx.doi.org/10.1109/ASE.2003.1240292.

Ruderman, J., 2007. Introducing jsfunfuzz. URL: http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/.

Ruderman, J., 2009. CSS grammar fuzzer. URL: http://www.squarefree.com/2009/03/16/css-grammar-fuzzer/.

Schmitz, S., 2007. Conservative ambiguity detection in context-free grammars. In: Arge, L., Cachin, C., Jurdzinski, T., Tarlecki, A. (Eds.), Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings. In: Lecture Notes in Computer Science, vol. 4596, Springer, pp. 692–703. http://dx.doi.org/10.1007/978-3-540-73420-8_60.

Schmitz, S., 2008. An experimental ambiguity detection tool. Electron. Notes Theor. Comput. Sci. 203 (2), 69–84. http://dx.doi.org/10.1016/j.entcs.2008.03.045.

Schröer, F.W., 2001. AMBER, an ambiguity checker for context-free grammars. URL: http://accent.compilertools.net/Amber.html.

Slutz, D.R., 1998. Massive stochastic testing of SQL. In: Gupta, A., Shmueli, O., Widom, J. (Eds.), VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases. August 24-27, 1998, New York City, New York, USA, Morgan Kaufmann, pp. 618–622, URL: http://www.vldb.org/conf/1998/p618.pdf.

SQLite, 2021. SQLite. URL: https://sqlite.org.

Steimann, F., Frenkel, M., Abreu, R., 2013. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In: Pezzè, M., Harman, M. (Eds.), International Symposium on Software Testing and Analysis. ISSTA '13, Lugano, Switzerland, July 15-20, 2013, ACM, pp. 314–324. http://dx.doi.org/10.1145/2483760.2483767.

Stijlaart, M., Zaytsev, V., 2017. Towards a taxonomy of grammar smells. In: Combemale, B., Mernik, M., Rumpe, B. (Eds.), Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering. SLE 2017, Vancouver, BC, Canada, October 23-24, 2017, ACM, pp. 43–54. http://dx.doi.org/10.1145/3136014.3136035.

Wen, W., Li, B., Sun, X., Li, J., 2011. Program slicing spectrum-based software fault localization. In: Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering. SEKE'2011, Eden Roc Renaissance, Miami Beach, USA, July 7-9, 2011, Knowledge Systems Institute Graduate School, pp. 213–218.

Wong, W.E., Debroy, V., Gao, R., Li, Y., 2014. The DStar method for effective software fault localization. IEEE Trans. Reliab. 63 (1), 290–308. http://dx.doi.org/10.1109/TR.2013.2285319.

Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F., 2016. A survey on software fault localization. IEEE Trans. Software Eng. 42 (8), 707–740. http://dx.doi.org/10.1109/TSE.2016.2521368.

Xie, X., Chen, T.Y., Kuo, F., Xu, B., 2013. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Trans. Softw. Eng. Methodol. 22 (4), 31:1–31:40. http://dx.doi.org/10.1145/2522920.2522924.

Xu, X., Debroy, V., Wong, W.E., Guo, D., 2011. Ties within fault localization rankings: Exposing and addressing the problem. Int. J. Softw. Eng. Knowl. Eng. 21 (6), 803–827. http://dx.doi.org/10.1142/S0218194011005505.

Xue, X., Namin, A.S., 2013. How significant is the effect of fault interactions on coverage-based fault localizations? In: 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement. Baltimore, Maryland, USA, October 10-11, 2013, IEEE Computer Society, pp. 113–122. http://dx.doi.org/10.1109/ESEM.2013.22.

Yang, X., Chen, Y., Eide, E., Regehr, J., 2011. Finding and understanding bugs in c compilers. In: Hall, M.W., Padua, D.A. (Eds.), Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2011, San Jose, CA, USA, June 4-8, 2011, ACM, pp. 283–294. http://dx.doi.org/10.1145/1993498.1993532, URL: http://doi.acm.org/10.1145/1993498.1993532.

Yoshikawa, T., Shimura, K., Ozawa, T., 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, p. 20. http://dx.doi.org/10.1109/QSIC.2003.1319081.

Zakari, A., Lee, S.P., Abreu, R., Ahmed, B.H., Rasheed, R.A., 2020. Multiple fault localization of software programs: A systematic literature review. Inf. Softw. Technol. 124, 106312. http://dx.doi.org/10.1016/j.infsof.2020.106312.

Zelenov, S.V., Zelenova, S.A., 2005. Generation of positive and negative tests for parsers. Program. Comput. Softw. 31 (6), 310–320. http://dx.doi.org/10.1007/s11086-005-0040-6.

Zeller, A., 2009. Why Programs Fail - A Guide to Systematic Debugging, second ed. Academic Press, http://dx.doi.org/10.1016/B978-0-12-374515-6.X0000-7.

**Moeketsi Raselimo** received his Ph.D. in Computer Science from Stellenbosch University in 2023. In his thesis he investigated spectrum-based localization and repair of faults in context-free grammars. He has published several papers on software language engineering. He currently works as postdoctoral research fellow at Humboldt University in Berlin.

**Bernd Fischer** received his Ph.D. in Computer Science from the University of Passau in 2001, and is currently Professor at Stellenbosch University. He has published more than 100 papers in formal methods, software verification, and software testing.