The Journal of Systems and Software 214 (2024) 112068

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Grammar-based test suite construction using coverage-directed algorithms over LR-graphs[☆]

Christoff Rossouw, Bernd Fischer[*]

Division of Computer Science Stellenbosch University, Stellenbosch, South Africa

ARTICLE INFO

Keywords:
Software testing and debugging
Grammars and context-free languages
Test case generation
Push-down automata

ABSTRACT

In grammar-based testing, the test suites that drive the system under test are typically constructed from a given context-free grammar through a set of derivations that jointly satisfy some coverage criterion. In this paper, we describe and evaluate a new algorithm that instead constructs test suites from a set of valid paths that cover all edges in a labeled directed graph corresponding to an LR-automaton that accepts the language of the grammar. Vertices in this graph correspond to states in the LR-automaton; two vertices are connected by an edge iff the top of the LR-automaton's stack can change from one state to the other, either by shifting a terminal or non-terminal symbol (push edges), or by reducing with a grammar rule (pop edges). The algorithm constructs a unique reduction path for each pop edge in the graph. These reduction paths are recursively embedded into each other, and any unresolved non-terminal push edges are substituted by shortest derivations for the non-terminal symbol. The algorithm can work with different types of LR-automata, including LR(0)- and LR(1)-automata, and can successfully generate a test suite from an LR-graph even if the underlying LR-automaton construction leads to shift/reduce or reduce/reduce conflicts. The algorithm only constructs valid paths over the LR-graphs that correspond to sentences in the language and thus generates only positive tests. We therefore also describe mutations to the positive paths that are guaranteed to generate negative tests without needing any further verification by an oracle.

Our algorithm is substantially more efficient than an earlier algorithm that explores LR-graphs with two consecutive breadth-first graph traversals and our experimental evaluation shows that it scales to large production-quality grammars. It is robust against random choices made to resolve ambiguity in the construction of the tests, while the code coverage of the different test suite variants is relatively uniform. Finally, our evaluation shows that the negative test suites constructed by path mutation identify more faults in a set of student grammars than those constructed by rule mutation.

1. Introduction

Bugs in safety-critical software can have catastrophic consequences. Parsers are not generally considered to be safety-critical, but parser bugs can still have severe effects, since parsers are used in many systems trying to recognize structured input. For example, Cloudflare, a large cloud network solution provider, discovered in 2017 a vulnerability where a private key of one of their internal servers was leaked through corrupt HTML pages that were produced by their HTTP rewrite service (Graham-Cumming, 2017). The leak was caused by a hand-written parser that allowed malformed HTML elements like `< script type =` at the end of an HTML page. This led to a buffer overflow during the on-the-fly page modification, which in turn led to data outside the buffer being written into the resulting HTML page.

Testing parsers and other grammarware (Klint et al., 2005) for such errors typically requires large test suites to drive the system under test (SUT). In grammar-based testing, these test suites are constructed from a context-free grammar (CFG) (Lämmel, 2001), which serves as model for the input and can thus be different from the CFG actually embodied in the SUT. Most grammar-based test suite construction algorithms take a *generative* view and work directly on the grammar: they take the grammar as input and construct a set of derivations according to some coverage criterion, generating the actual tests as byproducts. Random algorithms (Lämmel and Schulte, 2006; Yang et al., 2011; Livinskii et al., 2020; Soremekun et al., 2021) use different strategies to select the non-terminal occurrences and productions for the derivation steps, while systematic algorithms (Purdom, 1972; Lämmel, 2001; Zelenov and Zelenova, 2005; Fischer et al., 2011; Havrikov and Zeller, 2019;

☆ Editor: Laurence Duchien.
* Corresponding author.
E-mail addresses: 20723806@sun.ac.za (C. Rossouw), bfischer@sun.ac.za (B. Fischer).

https://doi.org/10.1016/j.jss.2024.112068
Received 5 March 2023; Received in revised form 25 February 2024; Accepted 18 April 2024
Available online 25 April 2024

van Heerden et al., 2020) construct derivations that jointly satisfy different criteria.

In this paper, we take an *analytic* view where rules are interpreted as reductions. We work with a graph that represents a push-down automaton (PDA) accepting the language generated by the grammar, rather than directly with the grammar, and use this to drive our test suite construction. Our goal is to create test suites that cover all possible state transitions in the PDA, i.e., all possible pairs of states that can follow each other as the top elements of the PDA's stack. This allows us in particular to exploit the different contexts of rule applications, which are encoded in the PDA.

More specifically, we construct test suites from sets of valid paths that cover all edges in a labeled directed graph $LR_G^{\rightarrow}$ corresponding to an LR-automaton LR_G^* that accepts the language $L(G)$. Vertices in $LR_G^{\rightarrow}$ correspond to states in LR_G^*; two vertices are connected by an edge iff LR_G^* can transition between the states when shifting a terminal or non-terminal symbol (*push edges*), or when reducing with a grammar rule (*pop edges*). The terminal labels along each valid accepting path from the graph's source to its sink then form a sentence $w \in L(G)$ that can be used as a positive test case. We discuss the specifics of the LR-graph construction in more detail in Section 3.

We originally (Rossouw and Fischer, 2020) developed an algorithm that "grows" the paths from source to sink, using two consecutive breadth-first graph traversal phases. The *flooding* phase first covers the edges of the graph in a layer-by-layer fashion, and so constructs paths that correspond to prefixes of valid sentences in $L(G)$. The following *path completion* phase then completes each prefix from the flooding phase with the shortest postfix to construct valid accepting paths over $LR_G^{\rightarrow}$. However, the number of prefixes pending in the BFS queue grows very quickly, due to the presence of *stubborn* edges that need to be traversed repeatedly, and the algorithm does not scale to larger grammars, despite several optimizations we attempted. We give in Section 4 a simplified description of this breadth-first search algorithm (LR-BFS).

In this paper we thus develop a new algorithm that keeps the main goal of ensuring all edges of the graph are covered but takes a different approach. It constructs *reduction paths* that relate to the rules in the CFG and consist of the edges that encode the stack actions of a rule's execution. The structure of $LR_G^{\rightarrow}$ guarantees that there is a unique reduction path for each pop edge in the graph. These reduction paths are then recursively embedded into each other to form a shallowest embedding, and any unresolved non-terminal push edges are substituted by shortest derivations. This again results in valid accepting paths over $LR_G^{\rightarrow}$, from which positive test cases are then extracted. Section 5 provides further details on this pop-edge coverage (PEC) algorithm.

The PEC algorithm can, like the LR-BFS algorithm, successfully generate test suites even if the LR-automaton induces shift/reduce or reduce/reduce conflicts because the LR-graph is constructed with edges for all options in a conflict—it simply solves for each pop edge independently. Hence, there is no need to attempt to resolve any conflict. This enables the algorithm to work over any LR-automaton construction, independent of the grammar, and our implementation (see Section 7) supports LR(0) and LR(1)-automata. Obviously, the different LR-automaton constructions can result in different test suites as the automaton's structure impacts the construction of the LR-graph and thus the test suite; in particular, the large LR(1)-automata induce disproportionally larger test suites. However, the LR-graph induced by the grammar is only a *model* of the SUT's input, and we can thus use *any construction* to test *any SUT*, independently of the technique the SUT uses to parse the inputs; in particular, we have used LR(0)-graphs to successfully test SUTs relying on both LL(1) and LALR(1) parsers.

The PEC algorithm, like the LR-BFS algorithm, only explores valid paths over the LR-graphs and thus generates only positive tests. We therefore also formalize in Section 6 three different *edge* (symbol deletion, replacement, and insertion) and *stack* (top-of-stack deletion, replacement, and insertion) mutations to the positive paths that are guaranteed to generate negative tests or *counterexamples* (i.e., inputs that the SUT is expected to reject) without needing any further verification.

Our experimental evaluation focuses on the PEC algorithm, since the LR-BFS algorithm does not scale to more complex grammars. It answers the following research questions:

RQ1. Can the PEC algorithm generate test suites for complex grammars?

RQ2. How does the size of the test suites produced by the PEC algorithm vary with the choice of the LR-automaton?

RQ3. How do the PEC and generic cover algorithm (instantiated with different coverage criteria) compare in terms of the test suite sizes and the induced SUT code coverage, respectively?

RQ4. How well do the different positive and negative test suites reveal errors in the language accepted by a grammar?

The results show that the PEC-algorithm indeed scales to complex, production-quality grammars, such as for the SQL and Go languages. It also produces test suites that more effectively cover deeply nested grammar structures than the generic cover algorithm (van Heerden et al., 2020) with coverage criteria that produce test suites of a similar size, and is generally also more robust against random choices made to resolve ambiguity in the construction of the tests, while the code coverage of the different test suite variants over different SUTs (including SQLite and GCCGo) is more uniform. Finally, it shows that the positive and negative test suites constructed by the PEC algorithm proposed here identify all faults in a set of student grammars, while the test suites constructed by the generic cover algorithm miss several faults. We give a detailed discussion of the results in Section 8.

Relation to previously published work. This paper is a revised and substantially extended version of our conference paper published at SLE 2020 (Rossouw and Fischer, 2020). The main extensions are as follows:

- We introduce in Section 5 a new algorithm that meets the same coverage criteria as the simple LR-BFS algorithm we originally presented but runs in linear time over the number of pop edges in the LR-graph.
- We describe in Section 6 a modification of our original edge mutation algorithm for negative test suite construction that can handle LR-graphs over grammars with conflicts, and introduce a stack mutation algorithm which constructs more complex negative tests than purely local edge mutations.
- We extensively evaluate in Section 8 the PEC algorithm over a range of grammars and systems under test and demonstrate the effect of different LR-automata constructions on the size of the generated test suites; we also show its application for the localization of faults in student grammars.

We originally introduced the concept of LR-graphs and the LR-BFS algorithm in the SLE 2020 paper; the current paper revises and substantially expands the exposition, and simplifies the presentation of the algorithm.

2. Notation and background

Grammars. A *context-free grammar* (or simply *grammar*) is a four-tuple $G = (N, T, P, S)$, where N and T, with $N \cap T = \emptyset$, are the sets of *non-terminal* and *terminal* symbols, respectively, $P \subset N \times (N \cup T)^*$ is the set of *productions*, and $S \in N$ is the *start symbol*. We use the meta-variables $A, B, C, \ldots$ for non-terminals, $a, b, c, \ldots$ for terminals, X, Y, Z for *grammar symbols* in $N \cup T$, w, x, y, z for strings of terminals or *sequences*, $\alpha, \beta, \gamma, \ldots$ for strings of grammar symbols or *phrases*, with $|\cdot|$ denoting the length of a string and ϵ denoting the empty string. We write $A \rightarrow \gamma$ for $(A, \gamma) \in P$ and use $\$ \notin T$ to denote the end of the input.

In examples, we also use *italics* and `typewriter` font for non-terminal and terminal symbols, respectively. By convention, we consider the non-terminal on the left-hand side of the first rule as start symbol.

Derivations. We use $\alpha A\omega \Rightarrow \alpha\gamma\omega$ to denote that $\alpha A\omega$ *produces* or *derives* $\alpha\gamma\omega$ by application of the rule $A \to \gamma \in P$, with $\Rightarrow^k$ denoting its k-fold repetition and $\Rightarrow^*$ its reflexive-transitive closure. A *sentential form* is a phrase α with $S \Rightarrow^* \alpha$, a *sentence* is a sequence w with $S \Rightarrow^* w$. We also use a *simultaneous derivation* relation $\Rrightarrow$, where $X_1 \dots X_n \Rrightarrow \gamma_1 \dots \gamma_n$ if $X_i \to \gamma_i \in P$ for all $X_i \in N$ and $\gamma_i = X_i$ for all $X_i \in T$ (van Heerden et al., 2020).

The *yield* of a phrase α is the set of all sequences that can be derived from it, i.e., $\text{yield}(\alpha) = \{w \in T^* \mid \alpha \Rightarrow^* w\}$. We assume that $\text{yield}(\alpha) \neq \emptyset$ for all α. The *language* $L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$ *generated by a grammar* G is the yield of its start symbol, i.e., the set of all sentences.

An *embedding of* X *in* A is a phrase $\alpha X\omega$ with $A \Rightarrow^* \alpha X\omega \Rightarrow^* w$. It is a *shortest yield embedding* if for all $A \Rightarrow^* \beta X\gamma \Rightarrow^* v$ we have $|w| \leq |v|$. It is a *shallowest embedding* if $S \Rightarrow^k \alpha X\omega$, and there is no $i < k$ such that $S \Rightarrow^i \beta X\gamma$; hence, the length of a shallowest embedding corresponds to the height of a minimal derivation tree with the root A and a leaf X (i.e., a terminal or un-expanded non-terminal). By abuse of notation, we also call the left-to-right sequence of the leaf symbols in such a tree a *shallowest yield* of X. We use $\Rightarrow^*_\leq$ and $\Rightarrow^*_\sqsubseteq$ to denote shortest yield and shallowest derivations, respectively.

LR-automata. Our approach is based on a graph representation of the PDA accepting $L(G)$. There are several PDA variants (e.g., SLR, LALR, GLR etc. (DeRemer, 1971; Harris, 1987; Tomita, 1985)), which differ in the method of resolving conflicts via lookahead or breadth-first search. Our approach is based on an abstract formalization of LR-automata and is thus independent of the specific variant.

More specifically, an *LR-automaton over a grammar* $G = (N, T, P, S)$ is a five-tuple $LR^*_G = (Q, \delta, \rho, q_{init}, q_{acc})$, where Q is a finite set of *states*, $\delta \subseteq Q \times (T \cup N \cup \{\$\}) \times Q$ is the *transition relation*, $\rho \subseteq Q \times T \times P$ is the *reduction relation*, $q_{init} \in Q$ is the *initial* or *start state*, and $q_{acc} \in Q$ is the *final* or *accepting state*.

A *configuration* $C = (\vec{q}, w\$)$ of an LR-automaton consists of the *stack* $\vec{q} \in Q^*$ and the remaining input $w \in T^*$, which must be followed by the end-of-input symbol \$. The automaton can transition between two configurations either by *shifting*, consuming the next input symbol and pushing a successor state onto the stack, i.e., $(\vec{q}\, q, aw\$) \overset{a}{\Longrightarrow} (\vec{q}\, q\, q', w\$)$ where $\delta(q, a, q')$, or by *reducing* with a rule, popping a number of states off and pushing a successor state back onto the stack, i.e., $(\vec{q}\, q\, q_1 \dots q_n, aw\$) \overset{A}{\Longrightarrow} (\vec{q}\, q\, q', aw\$)$ where $\rho(q, a, A \to \alpha)$, $|\alpha| = n$, and $\delta(q, A, q')$. LR_G accepts a sequence w iff $(q_{init}, w\$) \Longrightarrow^* (q, \$)$ where $\delta(q, \$, q_{acc})$. We assume that LR_G accepts w iff $w \in L(G)$.

Note that this formulation of LR-automata is *non-deterministic* and allows for both *shift/reduce* (i.e., $\delta(q, a, q')$ and $\rho(q, a, A \to \alpha)$), and *reduce/reduce* (i.e., $\rho(q, a, A \to \alpha)$ and $\rho(q, a, B \to \beta)$) conflicts. However, we assume that the transition relation is deterministic on terminal symbols (i.e., $\delta(q, a, q')$ and $\delta(q, a, q'')$ imply $q' = q''$), so that there are no shift/shift conflicts in the LR-automaton.

Note also that our formulation of LR-automata slightly differs from the usual (deterministic) implementation with ACTION- and GOTO-tables (Aho et al., 2007). In particular, we only represent reduce-actions explicitly, through the reduction relation, while we encode shift- and accept-actions in the transition relation. This makes it easier to handle conflicts, and to extract the LR-graphs from the representation.

Grammar-based testing. A *test suite* is a set of individual *test cases* comprising the *test input data* $\vec{x}$ and the *expected output* y. The *system under test* (SUT) is executed over $\vec{x}$ and its output $SUT(\vec{x})$ is compared against the expected output, resulting in either a *pass* or a *fail* verdict.

In grammar-based testing, the test input is a sequence $w \in T^*$ and the expected output is either *accept* (for *positive* tests $w \in L(G)$) or *reject* (for *negative* tests $w \notin L(G)$). Note that we do not compare the required and actual error locations, so that the verdict is already *pass* if the program rejects a negative test case for any syntax error.

Our goal is to produce test suites that exercise an arbitrary SUT which processes inputs described by G. Normally, G is the grammar for a programming language, and the SUT is a parser or compiler; however, G can be the grammar for any input language, and the SUT can be any program that contains a input reader for $L(G)$, independent of its implementation technique. Usually, the tests are produced in textual form, but the SUT may not even necessarily "read" the inputs described by G; for example, G may describe the structure of user interactions through the SUT's GUI, or the calling conventions of a complex API implemented in a library.

Algorithm 1: Generic cover algorithm

```
   input : A CFG $G = (N, T, P, S)$
   input : A coverage criterion $C$
   input : A minimal derivation relation $\Rightarrow^*_\leq$
   output: A test suite $TS$ over $G$
 1 $TS \leftarrow \emptyset$
 2 for $X \in V$ do
 3     compute $S \Rightarrow^*_\leq \alpha X\omega$
 4     for $\theta \in C(X)$ do
 5         compute $\alpha\theta\omega \Rightarrow^*_\leq w$
 6         $TS$.add($w$)
 7 return $TS$
 8 // coverage criteria
 9 $\text{rule}(X) \;\hat{=}\; \{\alpha \mid X \to \alpha \in P\}$
10 $\text{cdrc}(X) \;\hat{=}\; \{\alpha\gamma\omega \mid X \to \alpha Y\omega \in P, Y \to \gamma \in P\}$
11 $\text{step}_k(X) \;\hat{=}\; \{\alpha Y\omega \mid X \Rightarrow^k_\leq \alpha Y\omega, Y \in V\}$
12 $\text{pll}(X) \;\hat{=}\; \{a\omega \mid X \Rightarrow^*_\leq a\omega\}$
13 $\text{deriv}(X) \;\hat{=}\; \{\alpha Y\omega \mid X \Rightarrow^*_\leq \alpha Y\omega, Y \in V\}$
14 $\text{bfs}_k(X) \;\hat{=}\; \{\alpha Y\omega \mid X \Rrightarrow^k \alpha Y\omega, Y \in V\}$
```

Generic cover algorithm. For the construction of positive tests we use a generic cover algorithm (van Heerden et al., 2020) with different *grammar coverage criteria* (Lämmel, 2001). Algorithm 1 shows the algorithm and the formal definition of the criteria we use in our evaluation. The algorithm follows the similar approaches by Fischer et al. (2011) and Havrikov and Zeller (2019). Its basic idea is to

1. iterate over all symbols $X \in V$,
2. embed X, i.e., compute a minimal derivation $S \Rightarrow^*_\leq \alpha X\omega$ wrt. the derivation relation $\Rightarrow^*_\leq$ given as argument,
3. cover X, i.e., compute a set of minimal derivations $X \Rightarrow^*_\leq \gamma$ that conform to the criterion,
4. convert the sentential form into a sentence, i.e., compute a minimal derivation $\alpha\gamma\omega \Rightarrow^*_\leq w$ where each non-terminal A in $\alpha\gamma\omega$ is replaced by its minimal yield w_A, and
5. merge identical test satisfying different targets.

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 (or *k-path coverage* Havrikov and Zeller, 2019) as arguments to the cover algorithm; note that *rule* and *cdrc* are simply the "traditional" names for 1-step and 2-step coverage, respectively. These criteria ensure that all k-fold nested rule applications are covered: for example, *cdrc* ensures that each (applicable) rule is applied in the context of each right-hand side of each rule in the grammar. In addition, we also use *pll coverage* (Zelenov and Zelenova, 2005), which covers a derivation to each symbol in each non-terminal symbol's first-set, *derivable pair coverage* (van Heerden et al., 2020), which can intuitively be thought of as a fixpoint version of k-step coverage, since it covers the shortest derivation between any two symbols irrespective of its length, and *bfs*$_k$, a breadth-first version of k-step coverage that uses simultaneous replacements (van Heerden et al., 2020).

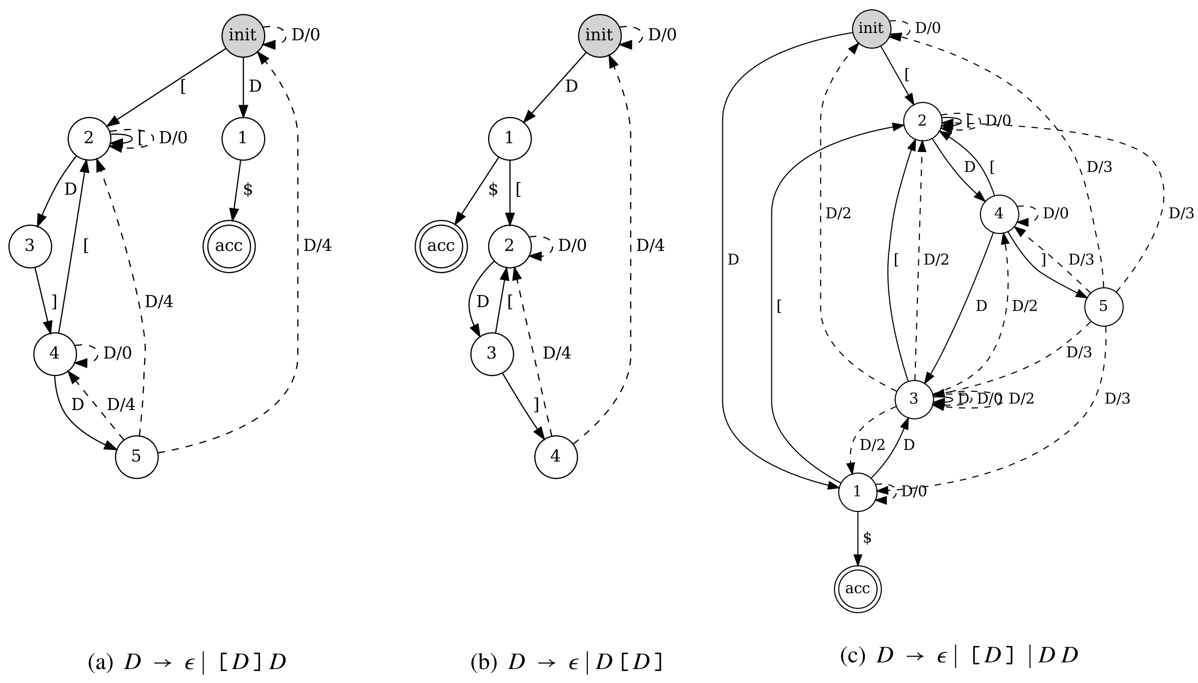

init
D/0
[
D
2
1
D/4
$
acc
3
4
5
]
D/2
D/3
(a) $D \rightarrow \epsilon \mid [D]D$
(b) $D \rightarrow \epsilon \mid D[D]$
(c) $D \rightarrow \epsilon \mid [D] \mid D D$

Fig. 1. Dyck grammar variants and corresponding LR-graphs. These grammars all describe the same language with different locations of recursion in the grammar.

3. LR-graphs

Our goal is to create test suites that execute all possible state transitions in the LR-automaton, i.e., cover all possible pairs of states that can follow each other as the top elements of the automaton's stack. This is straightforward for transitions $(\vec{q}\,q, w\$) \overset{a}{\Longrightarrow} (\vec{q}'\,q', w'\$)$ associated with shift-actions, but for transitions associated with reduce-actions, we need to inspect more than the top element of the stack because the next state on top of the stack (implicitly) depends on states lower down on the stack. We therefore use a graph structure that makes explicit the extra information about the context of rule applications that has implicitly been encoded in the LR-automaton.

We originally introduced the concept of LR-graphs in Rossouw and Fischer (2020); here, we give more details and more explicitly relate the construction to the corresponding LR-automaton.

LR-graphs. Given an LR-automaton $LR^*_G = (Q, \delta, \rho, q_{init}, q_{acc})$ over a grammar $G = (N, T, P, S)$, an *LR-graph* for LR^*_G is a labeled directed graph $LR^{\rightarrow}_G = (Q, E, q_{init}, q_{acc})$ with the automaton's states Q as vertices, the start and accept states q_{init} and q_{acc} as single source and sink vertices, respectively, and a set of edges $E = E_{\rightarrow} \cup E_{\dashrightarrow}$. In figures we show the source vertex in gray and the sink vertex by a double-circle outline.

Edges may be either *push* edges $E_{\rightarrow} \subseteq Q \times (N \cup T \cup \{\$\}) \times Q$ or *pop* edges $E_{\dashrightarrow} \subseteq Q \times (N \times \mathbf{N}) \times Q$. We write push edges as $q \rightarrow_X q'$ with label X and pop edges as $q \dashrightarrow_{A/|\alpha|} q'$ with label $A/|\alpha|$ for a rule $A \rightarrow \alpha \in P$. We write $\rightsquigarrow$ and $\hookrightarrow$ when either a push edge or a pop edge may be used.

Push edges. Push edges correspond to the transition relation in LR^*_G, i.e., $q \rightarrow_X q'$ iff $\delta(q, X, q')$. Push edges labeled with a terminal symbol thus correspond to shift actions in the ACTION-table while push edges labeled with a non-terminal symbol correspond to the GOTO-transitions. Here, q' is the state stored in the tables at the index position (q, X). Note that there can only be one push edge with the same label originating from the same vertex, due to the itemset-construction for the underlying LR-graphs.

Pop edges. One of the main differences between LR-parsers and LR-graphs is how reductions are represented. In an LR-parser a reduction is implemented as a single transition in which a number of items are popped off the stack and a single non-terminal symbol is pushed back onto the stack. In the LR-automaton, we already refine this into the two separate transition and reduction relations. In the LR-graph, we refine the representation even further, into a single pop edge and a number of push edges. Pop edges allow us to explicitly encode into the graph all different top-of-stack contexts in which a reduction can be applied, which in turn allow us to ensure coverage of these contexts by our algorithms.

More specifically, in order to determine the pop edges $q \dashrightarrow_{A/|\alpha|} q'$ corresponding to a reduction with a rule $A \rightarrow \alpha$, we first identify all states q of the LR-automaton that contain the corresponding reduction item $A \rightarrow \alpha\bullet$. We then traverse back from each of these states, using only push edges, and to find all paths with a length of $|\alpha|$. In this way, we calculate all possible prefix path segments for a reduction rule. We then insert a pop edge from the state q at which the reduction rule is applied to the source q' of each prefix path segment. Note that a single reduction rule can thus result in multiple pop edges. Note also that each pop edge is by construction a back edge and thus introduces a cycle. For example, in Fig. 1(a), state 5 is the only state containing the reduction item $D \rightarrow [D]D\,\bullet$. There are three possible top-of-stack contexts which may lead to this reduction, $\langle \dots,\ q_{init},\ q_2\ q_3,\ q_4,\ q_5\rangle$, $\langle \dots,\ q_2,\ q_2\ q_3,\ q_4,\ q_5\rangle$, and $\langle \dots,\ q_4,\ q_2\ q_4,\ q_4,\ q_5\rangle$, which give rise to the three pop edges $q_5 \dashrightarrow_{D/4} q_{init}$, $q_5 \dashrightarrow_{D/4} q_2$, and $q_5 \dashrightarrow_{D/4} q_4$ shown.

Ambiguity and conflicts. One of the key benefits of this construction of pop edges is that it allows us to deal with ambiguity in the input grammar. In Fig. 1(c) multiple reduce/reduce conflicts manifest at state 3. For example, here the reduction transition for the rule $D \rightarrow \epsilon$ is always possible, which conflicts with the reduction transition for the rule $D \rightarrow D\ D$ whenever that is also possible (i.e., when two consecutive D symbols have been pushed). Instead of using methods such as lookaheads to resolve such conflicts, we simply add pop edges for both rules, namely $q_3 \dashrightarrow_{D/0} q_3$ and $q_3 \dashrightarrow_{D/2} q_1$, since the algorithms for edge coverage will ensure both rules are considered by covering both pop edges. Hence, conflicts do not pose an issue to the construction or coverage of the LR-graph, which allows us to use any kind of LR-automaton as base for the construction of the LR-graph.

Lookaheads. Most types of LR-parsers use lookaheads to resolve shift/reduce and reduce/reduce conflicts. If successful, this ensures that the successor state of the chosen reduction has an outgoing transition for the next input symbol, and can thus consume and shift it onto the stack. In our context, lookaheads play no role because we are *constructing* rather than consuming programs, so that we can simply chose the next input symbol as required by the successor states of any of the reductions. Our LR-graph construction thus ignores lookaheads, and constructs pop-edges for all rules, as outlined above.

Paths. A *path* in $LR_G^{\rightarrow}$ is a sequence of edges $\langle e_1, \ldots, e_n \rangle$ connecting consecutive nodes. A path is *valid* if the sequence of its edges satisfies the following conditions that guarantee that the path's vertices describe a valid sequence of states resulting from the transitions of the LR-automaton. First, any terminal push edge $e = q \rightarrow_a q'$ on a path must be the first edge on the path or must be directly preceded by another push edge; it cannot be preceded by a pop edge, since the corresponding pop-operation must be followed in the parser by an appropriate push-operation (with the state specified in the GOTO-table as label), which is represented by a non-terminal push edge. For example, in Fig. 1(a) $q_2 \rightarrow_{[} q_2$ is only valid on a path when it is preceded by $q_{init} \rightarrow_{[} q_2$ or $q_2 \rightarrow_{[} q_2$ and not by $q_2 \dashrightarrow_{D/0} q_2$. Likewise, any non-terminal push edge $e = q \rightarrow_A q'$ must be preceded by a pop edge $q'' \dashrightarrow_{A/|\alpha|} q$. In Fig. 1(a) $q_{init} \rightarrow_D q_1$ is therefore only valid on a path when it is preceded by $q_{init} \dashrightarrow_{D/0} q_{init}$ or by $q_5 \dashrightarrow_{D/4} q_{init}$. Finally, any pop edge with a label $A/|\alpha|$ must be directly preceded by a valid path that results in a top-of-stack context that corresponds to α, so that the pop-operation succeeds. For example, in Fig. 1(a) the pop edge $q_5 \dashrightarrow_{D/4} q_2$ is only valid if the path directly preceding it results in a stack of the form $\langle \ldots,\ q_2,\ q_2,\ q_4,\ q_5 \rangle$. This can only be achieved by sequences of push edges $q_{init} \rightarrow_{[} q_2 \rightarrow_D q_3 \rightarrow_{]} q_4 \rightarrow_D q_5$ or $q_2 \rightarrow_{[} q_2 \rightarrow_D q_3 \rightarrow_{]} q_4 \rightarrow_D q_5$.

Moreover, the pop edge must be followed by an appropriate push edge labeled with the same non-terminal, to reflect the corresponding push operation. We finally write $q \rightsquigarrow_{\leq}^{*} q'$ to denote a shortest valid path from q to q'.

We use $p \circ p'$ to denote the composition of two paths p and p'. If $p = q_1 \rightsquigarrow^* q_n$ and $p' = q_n \rightsquigarrow^* q'_m$ are valid paths, then $p \circ p' = q_1 \rightsquigarrow^* q_n \rightsquigarrow^* q'_m$ is a valid path as well. For $p = q_1 \rightsquigarrow^* q_n$, $p' = q'_1 \rightsquigarrow^* q'_m$, $q_n \neq q'_1$ we define $p \circ p' = q_1 \rightsquigarrow^* q_n \rightarrow_{\epsilon} q'_1 \rightsquigarrow^* q'_m$. Note that the "pseudo edge" $q_n \rightarrow_{\epsilon} q'_1$ is not in E but simply a notational device.

We identify the (labeled) edges with relations over pairs of vertices in the usual way, and write $\rightsquigarrow \circ \hookrightarrow$ to denote the composition of two relations corresponding to $\rightsquigarrow$ and $\hookrightarrow$, $\rightsquigarrow^*$ to denote the corresponding transitive closure. We use $\twoheadrightarrow = \bigcup_{A \in N} (\dashrightarrow_{A/\_} \circ \rightarrow_A)$ to denote the combination of compatible pop and push edges representing reductions.

We represent the vertices along a path $p = q_1 \rightsquigarrow_{l_1} q_2 \rightsquigarrow_{l_2} \rightsquigarrow \cdots \rightsquigarrow_{l_{n-1}} q_n$ as $v(p) = q_1 \ldots q_n$. A *word over the path* p is defined as the sequence $w(p) = \langle a_i \mid q_i \rightarrow_{a_i} q_{i+1} \in p,\ a_i \in T \rangle$ of the terminal symbols labeling the push edges in p. We call such a path p *accepting* if p is valid and $v(p) = q_{init} \ldots q_{acc}$; by construction, the sequence of states $v(p)$ in a valid path p reflects a valid sequence of states in LR_G^* and the word over an accepting path is in the language $L(G)$. For example, the path $q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_D q_1 \rightarrow_{[} q_2 \dashrightarrow_{D/0} q_2 \rightarrow_D q_3 \rightarrow_{]} q_4 \dashrightarrow_{D/4} q_{init} \rightarrow_D q_1 \rightarrow_{\$} q_{acc}$ in Fig. 1(b) is an accepting path corresponding to the word [ ]. Finally, we call a path p *nullable* if it contains no terminal push edge, i.e., if $w(p) = \epsilon$.

4. Generating test suites by LR-graph traversal

The LR-BFS algorithm we originally introduced in Rossouw and Fischer (2020) "grows" the paths from source to sink, using two consecutive breadth-first graph traversal phases. The *flooding* phase first covers the edges of the graph in a layer-by-layer fashion, and so generates paths that correspond to prefixes of valid words in $L(G)$. The *path completion* phase then completes these prefixes with the shortest

Algorithm 2: LR-BFS algorithm

```
input  : A CFG $G = (N, T, P, S)$
local  : An LR-graph $LR_G^{\rightarrow} = (Q, E, q_{init}, q_{acc})$
output: A test suite for $G$ covering $E$
fun flooding() =
    $W \leftarrow \langle\langle \bot \rightarrow q_{init} \rangle\rangle$
    $seen\_edges \leftarrow \varnothing$
    while $seen\_edges \neq E$ do
        $p = W.dequeue()$
        for $e$ in $\{(q \rightsquigarrow q') \in E \mid v(p).last = q\}$ do
            if $valid(p \circ e)$ then
                $W.enqueue(p \circ e)$
                $seen\_edges.\cup \{e\}$
    return $W$
fun complete_path(path) =
    $W \leftarrow \langle path \rangle$
    while true do
        $p = W.dequeue()$
        if $p.last = q \rightsquigarrow q_{acc}$ then
            return $p$
        for $e$ in $\{(q' \rightsquigarrow q'') \in E \mid p.last = q \rightsquigarrow q'\}$ do
            if $valid(p \circ e)$ then
                $W.enqueue(p \circ e)$
$LR_G^{\rightarrow} \leftarrow compute\_graph(G)$
$paths \leftarrow \varnothing$
for $prefix \in flooding(LR_G^{\rightarrow})$ do
    $paths.\cup \{complete\_path(LR_G^{\rightarrow}, prefix)\}$
return $\{w(p) \mid p \in paths\}$
```

possible postfix to the sink and so generates valid paths over $LR_G^{\rightarrow}$. Each path results in a word $w \in L(G)$ as a positive test case.

Note that each valid accepting path is necessarily cyclic, because it contains at least one pop edge (i.e., $q \dashrightarrow_{S/|\alpha|} q_{init}$ for a rule $S \rightarrow \alpha$) introducing a back-edge and thus a cycle. However, while each edge can be reached with a finite path, the number of cycles, and the number of copies of each edge, on a path that is necessary to reach any given edge depends on the structure of the LR-graph and thus on the grammar. For example, in Fig. 1(b) the edges $q_2 \rightarrow_D q_3$ and $q_3 \rightarrow_{]} q_4$ must be visited at least twice in order to reach the edge $q_4 \rightarrow_D q_2$. We therefore use breadth-first traversals in both phases of the algorithm which is guaranteed to terminate without cycle detection or avoidance.

Normally, a breadth-first graph traversal uses a queue of nodes, dequeues the current node, and enqueues all of its unseen successors, until the queue becomes empty. The LR-BFS algorithm instead uses a queue of valid paths; in each step, it checks which of the edges going out from the current path's last vertex can be used to validly extend the path, and re-enqueues all extended paths. The exploration continues until all edges are covered for the flooding phase, and until q_{acc} is reached for the path completion phase.

This path replication allows our algorithm to handle conflicts in a way similar to how GLR parsers resolve conflicts (i.e., by parse stack duplication Tomita, 1985): when a conflict is encountered, we do not attempt to choose a specific traversal and instead perform both traversals in parallel. In fact, this similarity runs deeper because the parse stack that is duplicated can be reconstructed from the path as the sequence of vertices (i.e., states) connected by those push edges that are not part of completed cycles.

Algorithm. The LR-BFS algorithm (see Algorithm 2) first constructs the LR-graph from the grammar and then calls the flooding phase to obtain a set of prefixes. Each prefix is then individually completed and the word over the complete accepting path is extracted and added to the test suite.

Both the flooding and path completion phases dequeue a path p and extend it by adding valid edges at its tail. If p's last edge is a pop edge then there is, by construction, a uniquely determined valid push edge with the corresponding non-terminal from the current state that needs to be explored. This push edge completes the reduction action started by the preceding pop edge. If p's last edge is a push edge $q \to q'$ (or p is empty and $q' = q_{init}$) then any push edge $q' \to q''$ from q' is valid, while a pop edge $q' \dashrightarrow_{A/|\alpha|} q''$ is valid if the target q'' is in the current path p and the tail segment starting at q'' contains a sub-path consisting of push edges with labels that match the right-hand side of the rule corresponding to the pop edge. In Fig. 1(c), for a path $q_{init} \dashrightarrow_{D/0} q_{init} \to_D q_1 \dashrightarrow_{D/0} q_1 \to_D q_3$ we can see that only $q_3 \dashrightarrow_{D/0} q_3$ and $q_3 \dashrightarrow_{D/2} q_{init}$ are valid pop edges since the tail segment is $\langle \dots q_{init} \to_D q_1 \to_D q_3 \rangle$.

Stubborn edges. For the LR-graph shown in Fig. 1(b), we can see the layer-by-layer exploration of the flooding phase in Fig. 2. It shows how the queue growth accelerates as more and more layers are explored. We can observe that, from layer 7 onwards, the only edge that remains to be covered is $q_4 \dashrightarrow_{D/4} q_2$, and that the queue grows from 3 to 5 paths until the edge is finally covered in layer 10 of the exploration.

In order to reduce the number of multiple edge traversals, we modified the original LR-BFS algorithm to use priority queues. For the flooding phase we prioritize edges that have been covered the least number of times; for the path completion phase we prioritize edges that are closer to the accept state. This approach leads to fewer redundant graph exploration attempts which allows the algorithm to explore slightly larger LR-graphs for, e.g., a simple expression grammar. However, as soon as the grammars become even slightly more complex (e.g., by encoding operator precedences in the expression grammar through the introduction of several non-terminals such as *term* and *factor*), the algorithm does not terminate in any reasonable amount of time any longer.

This is due to *stubborn edges* that arise from the structure of the LR-graph. These are pop-edges that are valid only after a long, specific prefix. These prefixes also often consist of edges that have to be covered multiple times in the prefix for the stubborn edge. This means that many other previously explored cycles have to be considered, which leads to rapid, redundant queue growth. We have found these edges in the LR-graphs for many grammars. We have attempted to counteract the queue growth by methods such as bi-directional search and a more intricate edge priority implementation. However, no attempt was effective against stubborn edges for real world grammars. It is this problem that we solve by the algorithm presented in the next section.

5. Generating test suites by solving for pop edges

One of the main drawbacks of the LR-BFS algorithm is that it only considers edges that are immediately adjacent to the node at the tail end of the current path when it explores the graph. This leads to a rapid growth in redundant test cases as the size of the graph increases, because the high branching factor in many nodes leads to a substantial growth in queue size as the required lengths of paths increase. The algorithm we present in this section uses pop-edge coverage to avoid this branching and thus the drawback, and makes the calculation much more efficient.

The *pop-edge coverage* (PEC) algorithm exploits extra information that is by construction contained in the LR-graph. Since each pop edge corresponds to a pop action in a specific top-of-stack context, we know that there is only one configuration of the top $|\alpha|$ elements of the stack for each pop edge $q \dashrightarrow_{A/|\alpha|} q'$.

We therefore introduce the concept of a reduction path for a pop edge, i.e., the path of push edges that correspond to this pop edge's stack constraints, followed by the pop edge itself and its corresponding non-terminal push edge. We embed these reduction paths into each other in a bottom-up fashion to find the shallowest embedding of each reduction path into a valid path in the graph.

Reduction paths. We describe a rule application over the LR-graph using *reduction paths*, i.e., paths of the form

$$r = \underbrace{\iota(p \in E_{\to}^{|\alpha|} \mid \upsilon(p) = q' \ \dots \ q)}_{(i)} \circ \underbrace{(q \dashrightarrow_{A/|\alpha|} q')}_{(ii)} \circ \underbrace{\iota(q' \to_A q'')}_{(iii)}$$

They are uniquely determined by their pop edge $q \dashrightarrow_{A/|\alpha|} q'$ (ii) since the other two components, namely (i) the push path component $p = q' \dots q$ and (iii) the non-terminal push edge $q' \to_A q''$ that follows the pop edge, are unique by construction. More specifically, since each top-of-stack context for a reduction generates a new pop edge, the push path component must be unique as it directly corresponds to the path segment that results in the top-of-stack context for which the pop edge was generated.

Reduction paths capture the required stack context for a pop edge and the resulting reduction. For example, the pop edge $q_4 \dashrightarrow_{D/4} q_{init}$ in Fig. 1(b) induces the reduction path $q_{init} \to_D q_1 \to_{[} q_2 \to_D q_3 \to_{]} q_4 \dashrightarrow_{D/4} q_{init} \to_D q_1$. The reduction path indicates that this pop edge requires the stack to be of the form $\langle q_{init}, \dots, q_{init}, q_1, q_2, q_3, q_4 \rangle$. It also indicates that the last four items will be popped off the stack and q_1 will be pushed back on to the stack. Note that the labels on the push path component match the right-hand side of the rule corresponding to the pop edge (here $D \to D$ [D]).

Note that the reduction path for a pop edge $\dashrightarrow_{A/|\alpha|}$ is not a valid path if $\alpha \notin T^*$, i.e., it contains a non-terminal symbol B, because the respective push edge $\to_B$ is not preceded by a corresponding pop edge $\dashrightarrow_{B/|\beta|}$. We call a valid path p a *valid expansion* of the reduction path r if r's edges are a subsequence of p's edges. We can construct a valid expansion of r by recursively replacing the non-terminal push edges in its push path component by corresponding reduction paths ending in the respective edges, until we get a valid path.

Since the set of all reduction paths covers all pop edges, and all push edges must lead to pop edges so that the accept state can be reached in a valid path, reduction path coverage leads to coverage of all edges.

Algorithm. The algorithm contains two auxiliary functions that are used within the main loop, namely the reduction path construction and the embedding function (see Algorithm 3).

The embedding function *embed* takes a reduction path r and embeds it into other reduction paths r' when there are non-terminal push edges in the push path component of r' that are equal to the final non-terminal push edge in the current reduction path r. This continues until the resulting path is of the form $(q_{init} \rightsquigarrow^* q) \circ p \circ (q' \rightsquigarrow^* q'' \to_{\$} q_{acc})$. This results in a shallowest embedding. Any non-terminal push edge in the resulting path that is not preceded by a pop edge is substituted out by the shortest reduction path that has this non-terminal push edge as its final edge. For example, in Fig. 1(b), the pop edge $q_4 \dashrightarrow_{D/4} q_2$ results in the reduction path $q_2 \to_D q_3 \to_{[} q_2 \to_D q_3 \to_{]} q_4 \dashrightarrow_{D/4} q_2 \to_D q_3$. It may be embedded into the reduction path $q_{init} \to_D q_1 \to_{[} q_2 \to_D q_3 \to_{]} q_4 \dashrightarrow_{D/0} q_{init} \to_D q_1$ since the non-terminal push edge $q_2 \to_D q_3$ is contained in the push path component of this reduction path. This gives the path $q_{init} \to_D q_1 \to_{[} q_2 \to_D q_3 \to_{[} q_2 \to_D q_3 \to_{]} q_4 \dashrightarrow_{D/4} q_2 \to_D q_3 \to_{]} q_4 \dashrightarrow_{D/0} q_{init} \to_D q_1$. The remaining non-terminal push edges that are not preceded by a pop edge, for example $q_{init} \to_D q_1$, are then substituted out to give the valid path $q_{init} \dashrightarrow_{D/0} q_{init} \to_D q_1 \to_{[} q_2 \dashrightarrow_{D/0} q_2 \to_D q_3 \to_{[} q_2 \dashrightarrow_{D/0} q_2 \to_D q_3 \to_{]} q_4 \dashrightarrow_{D/4} q_2 \to_D q_3 \to_{]} q_4 \dashrightarrow_{D/0} q_{init} \to_D q_1 \to_{\$} q_{acc}$ and hence the test case [[]].

The algorithm iterates over the set of all pop edges in the graph and completes the reduction path corresponding to each edge into a valid path by embedding it. The word over this complete path is then added to the test suite.

layer	queue	N_p	test case
0	q_{init}		
1	$q_{init} \dashrightarrow_{D/0} q_{init}$		
2	$q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1$		
3	$q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{\$} q_{acc}$, $q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{[} q_2$		
4	$q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{\$} q_{acc}$, $q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{[} q_2 \dashrightarrow_{D/0} q_2$		
5	$q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{\$} q_{acc}$, $q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{[} q_2 \dashrightarrow_{D/0} q_2 \rightarrow_{D} q_3$		
6	$q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{\$} q_{acc}$, $q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{[} q_2 \dashrightarrow_{D/0} q_2 \rightarrow_{D} q_3 \rightarrow_{[} q_2$, $q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{[} q_2 \dashrightarrow_{D/0} q_2 \rightarrow_{D} q_3 \rightarrow_{]} q_4$		
7	$q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{\$} q_{acc}$, $q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{[} q_2 \dashrightarrow_{D/0} q_2 \rightarrow_{D} q_3 \rightarrow_{[} q_2 \dashrightarrow_{D/0}$, $q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{[} q_2 \dashrightarrow_{D/0} q_2 \rightarrow_{D} q_3 \rightarrow_{]} q_4 \dashrightarrow_{D/4} q_{init}$		
8	$q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{\$} q_{acc}$, $q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{[} q_2 \dashrightarrow_{D/0} q_2 \rightarrow_{D} q_3 \rightarrow_{[} q_2 \dashrightarrow_{D/0}\rightarrow_{D} q_3$, $q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{[} q_2 \dashrightarrow_{D/0} q_2 \rightarrow_{D} q_3 \rightarrow_{]} q_4 \dashrightarrow_{D/4} q_{init} \rightarrow_{D} q_1$		
9	$q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{\$} q_{acc}$, $q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{[} q_2 \dashrightarrow_{D/0} q_2 \rightarrow_{D} q_3 \rightarrow_{[} q_2 \dashrightarrow_{D/0} q_2 \rightarrow_{D} q_3 \rightarrow_{[} q_2$, $q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{[} q_2 \dashrightarrow_{D/0} q_2 \rightarrow_{D} q_3 \rightarrow_{[} q_2 \dashrightarrow_{D/0} q_2 \rightarrow_{D} q_3 \rightarrow_{]} q_4$, $q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{[} q_2 \dashrightarrow_{D/0} q_2 \rightarrow_{D} q_3 \rightarrow_{]} q_4 \dashrightarrow_{D/4} q_{init} \rightarrow_{D} q_1 \rightarrow_{\$} q_{acc}$, $q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{[} q_2 \dashrightarrow_{D/0} q_2 \rightarrow_{D} q_3 \rightarrow_{]} q_4 \dashrightarrow_{D/4} q_{init} \rightarrow_{D} q_1 \rightarrow_{[} q_2$		
10	$q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{\$} q_{acc}$,	1	ϵ
	$q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{[} q_2 \dashrightarrow_{D/0} q_2 \rightarrow_{D} q_3 \rightarrow_{[} q_2 \dashrightarrow_{D/0} q_2 \rightarrow_{D} q_3 \rightarrow_{[} q_2 \dashrightarrow_{D/0} q_2$,	16	[[[ ]]]
	$q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{[} q_2 \dashrightarrow_{D/0} q_2 \rightarrow_{D} q_3 \rightarrow_{[} q_2 \dashrightarrow_{D/0} q_2 \rightarrow_{D} q_3 \rightarrow_{]} q_4 \dashrightarrow_{D/4} q_2$,	4	[[ ]]
	$q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{[} q_2 \dashrightarrow_{D/0} q_2 \rightarrow_{D} q_3 \rightarrow_{]} q_4 \dashrightarrow_{D/4} q_{init} \rightarrow_{D} q_1 \rightarrow_{\$} q_{acc}$,	1	[ ]
	$q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{[} q_2 \dashrightarrow_{D/0} q_2 \rightarrow_{D} q_3 \rightarrow_{]} q_4 \dashrightarrow_{D/4} q_{init} \rightarrow_{D} q_1 \rightarrow_{[} q_2 \dashrightarrow_{D/0} q_2$	4	[ ] [ ]

Fig. 2. Queue growth for flooding phase over LR-graph in Fig. 1(b). N_p is the number of paths explored in path completion following the completion of the flooding phase at layer 10.

pop edge	reduction path	embedding	test case
$q_{init} \dashrightarrow_{D/0} q_{init}$	$q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1$	$q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{\$} q_{acc}$	ϵ
$q_2 \dashrightarrow_{D/0} q_2$	$q_2 \dashrightarrow_{D/0} q_2 \rightarrow_{D} q_3$	$q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{[} q_2 \dashrightarrow_{D/0} q_2 \rightarrow_{D} q_3 \rightarrow_{]} q_4$ $\dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{\$} q_{acc}$	[ ]
$q_4 \dashrightarrow_{D/4} q_2$	$q_2 \rightarrow_{D} q_3 \rightarrow_{[} q_2 \rightarrow_{D} q_3$ $\rightarrow_{]} q_4 \dashrightarrow_{D/4} q_2 \rightarrow_{D} q_3$	$q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{[} q_2 \dashrightarrow_{D/0} q_2 \rightarrow_{D} q_3 \rightarrow_{[} q_2$ $\dashrightarrow_{D/0} q_2 \rightarrow_{D} q_3 \rightarrow_{]} q_4 \dashrightarrow_{D/4} q_2 \rightarrow_{D} q_3 \rightarrow_{]} q_4$ $\dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{\$} q_{acc}$	[[ ]]
$q_4 \dashrightarrow_{D/0} q_{init}$	$q_{init} \rightarrow_{D} q_1 \rightarrow_{[} q_2 \rightarrow_{D} q_3$ $\rightarrow_{]} q_4 \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1$	$q_{init} \dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{[} q_2 \dashrightarrow_{D/0} q_2 \rightarrow_{D} q_3 \rightarrow_{]} q_4$ $\dashrightarrow_{D/0} q_{init} \rightarrow_{D} q_1 \rightarrow_{\$} q_{acc}$	[ ]

Fig. 3. Paths constructed by pop-edge algorithm over LR-graph in Fig. 1(b).

Example. Fig. 3 shows the reduction paths and embeddings found by the PEC algorithm from the LR-graph for the Dyck grammar shown in Fig. 1(b). We can see that the algorithm computes fewer paths than the LR-BFS algorithm (cf. Fig. 2). It is clear that the number of valid paths is limited by the number of pop edges. This also provides an upper bound for the size of the test suite; note that the words extracted from the different paths can be identical, so that the test suite can be smaller. In the example, the algorithm finds the four embeddings shown in Fig. 3, but these give rise to only three different test cases. In Section 8 we show that this improvement allows the algorithm to scale to production-level grammars such as SQLite that induces 400 710 pop edges in the LR(1)-graph.

Complexity. The number of test cases generated by the PEC algorithm is bounded by the number of pop edges in the LR-graph; it can be lower, because different pop edges can induce the same accepting path, and because different accepting paths can induce the same word. The number of pop edges is maximal if each state contains a reduction item for each rule and is reachable from each state by push paths corresponding to each right-hand side of each rule; in the worst case the number is thus $|Q| \times |P| \times |Q|$.

In practice, however, the number is much lower, which allows us to handle realistic grammars.[1] Since not every state is a reduction state the number of states with outgoing pop edges is smaller than $|Q|$, and since the number of reduce/reduce conflicts is small, the number of different types of outgoing pop edges is much smaller than $|P|$ as well. Similarly, the number of target states for each pop edge is limited by the length of the production, since each target state must be connected to the source

[1] For example, with the LR(1)-graph for the SQLite grammar, we have $|Q| = 27349$ and $|P| = 586$, but the number of pop edges is 366 648, or less than 0.0001% of the worst case (see Table 1).

Algorithm 3: Pop Edge Coverage

```
  input :  A CFG $G = (N, T, P, S)$
  local :  An LR-graph $LR_G^{\to} = (Q, E, q_{init}, q_{acc})$
  output:  A test suite for $G$ covering $E$
1 fun rpath($q \dashrightarrow_{A/|\alpha|} q'$) =
2     $\iota(p \in E_{\to}^{|\alpha|} \mid \upsilon(p) = q' \ldots q) \circ (q \dashrightarrow_{A/|\alpha|} q') \circ \iota(q' \to_A q'')$
3 fun embed(r) =
4     $(q_{init} \rightsquigarrow_{\leq}^{*} q) \circ r \circ (q' \rightsquigarrow_{\leq}^{*} q'' \to_{\$} q_{acc})$
5 $LR_G^{\to} \leftarrow$ compute_graph$(G)$
6 test_suite $\leftarrow \varnothing$
7 for $e \in E_{\dashrightarrow}$ do
8     test_suite.$\cup$ $\{w(embed(rpath(e)))\}$
9 return test_suite
```

state by a push path of that length, and thus the number of states with incoming pop edges is also much smaller than $|Q|$. Note, however, that each production $A \to \epsilon$ induces a pop-edge "loop" $q \dashrightarrow_{A/0} q$ at each state q with an outgoing non-terminal push edge $q \to_A q'$ labeled with A; ϵ-productions thus substantially increase the number of pop-edges.

6. Generating negative tests by path mutations

Positive test suites cannot reveal false positives, i.e., bugs where the SUT accepts inputs that it should be rejecting. Consider for example a faulty SUT that accepts function call expressions of the form

fun_expr	$\to$	`id`(*expr_list*)
expr_list	$\to$	*expr* | *expr_list* , *expr* | ϵ

i.e., it allows argument lists to start with a comma. Since the SUT also accepts all well-formed function call expressions, positive tests generated from the correct reference grammar alone are not sufficient to reveal the bug, and negative tests or counterexamples (i.e., inputs that we expect the SUT to reject) are required.

One way to generate a negative test suite is to use a simple generate-and-test approach: we randomly generate token sequences and test (using a separate parser for the reference grammar as oracle) whether these are in the target language or not. However, such test suites tend to be of low quality because the errors tend to occur towards the beginning of the sequence and do not require successful parsing of complex prefixes. We can produce negative tests with more meaningful errors if we start with a positive test suite and systematically mutate these tests, again using an oracle parser to test that the mutations indeed result in negative tests. However, this still incurs the extra cost of running the oracle parser.

Our goal is to develop an algorithm that produces tests that are guaranteed to be negative and so avoids the need for an oracle parser, similar to the mutation-based approach by Raselimo et al. (2019). However, here we mutate valid paths (rather than words or rules) such that the words extracted from the mutated paths are guaranteed to be invalid. This requires two checks, first that the path prefix leading to the mutated edge can after the mutation no longer be completed into a valid path, and second that there is no alternative valid path that induces the mutated word.

We introduce two distinct types of mutations, *edge mutations* that mutate terminal push edges in a similar way to string edit operations, and *stack mutations* that mutate multiple symbols at once and may be seen as editing the stack. We use insertion, deletion, and substitution as mutation operators. The mutations depend on the notion of *follow/precede sets*, *witnesses*, and *first/last sets* that we define below. These definitions do not rely on the traversal having reached a specific state and are therefore approximations. Our algorithm may thus miss some possible mutations, in order to guarantee negative test cases without the need for further validation.

Follow and precede sets. In a grammar, the follow set of a symbol is defined as the set of tokens that can follow that symbol in any valid derivation. We extend the idea to be the *follow set of a vertex*, i.e., the set of labels of the next terminal push edges that are reachable from the vertex. This is equivalent to the terminals that can follow a valid prefix that is recognized by the LR-automaton at this state. More formally we define the follow set of a vertex q in an LR-graph $LR_G^{\to} = (Q, E, q_{init}, q_{acc})$ as $F(q) = \{a \in T \mid \exists q' \in Q \cdot q \twoheadrightarrow^{*} \circ \to_a q'\}$.

We similarly define the precede set as the set of labels on the last terminal push edges that are covered before the vertex a is reached, i.e., $P(q) = \{a \in T \mid \exists q' \in Q \cdot q \to_a \circ \twoheadrightarrow^{*} q'\}$.

We call a vertex q a *witness* for the pair (a, b) if $a \in P(q)$ and $b \in F(q)$ and denote the set of all witnesses for (a, b) by $W(a, b)$; any valid path for a word $uabv$ must contain a witness for (a, b). For $T_1, T_2 \subseteq T$, we define $W(T_1, T_2) = \bigcup_{a \in T_1, b \in T_2} W(a, b)$. Note that $W(\varnothing, T_2) = W(T_1, \varnothing) = \varnothing$.

First and last sets. We can also extend the concepts of first and last sets for grammar symbols to reduction paths. A token is in the first or last sets of a reduction path if it is the label of the first or last terminal push edge in the reduction path, after all non-terminal push edges have been eliminated.

For a reduction path r of the form $q \to_a q' \rightsquigarrow^{*} q'''$ the first-set is $\{a\}$. If it is of the form $q \to_A q' \rightsquigarrow^{*} q'''$ then $\mathrm{first}(r) = \{a \in \mathrm{first}(r') \mid r' = (q \rightsquigarrow^{*} q'') \circ (q'' \dashrightarrow_{A/|\gamma|} q) \circ \iota(q \to_A q')\}$. In other words, if the first edge of a reduction path is a terminal push edge, the first set contains only the terminal symbol from the edge label. If the reduction path starts with a non-terminal push edge, the first-set is the union of all first sets for reduction paths ending with this non-terminal push edge.

Last sets are defined similarly by considering the last push edge in the push path component of the reduction path, instead of the first push edge.

6.1. Edge mutations

Edge mutations change the terminal labels at push edges in valid paths. They thus effectively perform string edit operations on the word over the path, e.g., deleting a terminal push edge results in deleting a terminal symbol from the resulting word. Edge mutations may be performed "on the fly" as the word is extracted from the path, since they only affect one edge at a time. We describe two approaches to ensure that the mutation results in a negative test, based on a path-sensitive analysis and its global approximation, respectively.

6.1.1. Path-sensitive analysis

If the LR-automaton does not contain any shift/reduce or reduce/reduce conflicts, we can ensure that the word over the mutated path is a negative test, simply by checking at the *mutation location* (i.e., the start vertex q_k of the designated edge $q_k \to_b q_{k+1}$) that the path prefix leading to the mutated edge can after the mutation no longer be completed into a valid path. However, if there are conflicts we need to check this for all *equivalent* vertices that are reachable with the same prefix.

In the following, we assume a valid accepting path $p = q_{init} \rightsquigarrow^{*} q_k \to_b q_{k+1} \rightsquigarrow^{*} q_n \to_{\$} q_{acc}$ where we mutate the terminal push edge $q_k \to_b q_{k+1}$. We call $p' = q_{init} \rightsquigarrow^{*} q_k$, and by abuse of notation also $w(p')$, the *left prefix* to the mutation and define the *left frontier* $LF(p') = \{q \mid w(q_{init} \rightsquigarrow^{*} q) = w(p')\}$ as the set of vertices at the end of valid paths that induce the same word as the left prefix. Similarly, we define the *right frontier* $RF(p', a) = \{q' \mid q \in LF(p'), q \twoheadrightarrow^{*} \circ \to_a q'\}$ as the set of vertices reachable from the left frontier through a single push edge labeled with a.

Table 1
Quantitative evaluation of PEC-algorithm over different LR-graphs. Numbers in parenthesis denote states with corresponding conflicts.

	AMPL		SQLite		GCCGo	
	LR(0)	LR(1)	LR(0)	LR(1)	LR(0)	LR(1)
$\lvert Q\rvert$	153	483	966	27 349	480	2963
$\lvert E_{\rightarrow}\rvert$	316	1 115	21 586	354 127	3770	24 053
$\lvert E_{\dashrightarrow}\rvert$	277	998	26 124	400 710	4728	30 006
shift/red. conflicts	63 (44)	0	12 392 (290)	225 825 (3171)	988 (101)	1432 (144)
red./red. conflicts	48 (1)	0	40 261 (132)	202 556 (2545)	2212 (24)	45 (6)
Positive	141	628	22 289	366 648	3478	23 510
Negative	184 236	1 035 777	19 736 911	–	5 457 397	43 803 722
Edge insertions	68 397	311 276	10 089 563	[a]	2 764 229	22 141 014
Edge substitutions	66 507	297 997	9 564 944	[a]	2 669 182	21 715 296
Edge deletions	1502	6 480	81 889	[a]	27 050	196 767
Stack insertions	49 114	350 939	[a]	[a]	[a]	[a]
Stack substitutions	63 007	471 737	[a]	[a]	[a]	[a]
Stack deletions	623	3 016	41 170	[a]	17 155	120 824

[a] Denotes time-outs (>168 h).

Insertion. This mutation inserts at the mutation location any terminal symbol $d \in T$ for which either (i) $d \notin \bigcup_{q \in LF(p')} F(q)$ or (ii) $b \notin \bigcup_{q \in RF(p',d)} F(q)$ hold, i.e., the inserted symbol d cannot be the label on the next push edge following any of the vertices recognizing the left prefix (and thus d cannot follow the preceding symbol of the word), or the original symbol b cannot be the label on the *next* push edge following any of the vertices reachable through a push edge labeled with d (and thus b cannot follow the inserted d). Hence, no valid accepting path exists, and the mutated word is indeed invalid.

For example, given the LR-graph shown in Fig. 1(a), consider the path $p = q_{init} \rightarrow_{[} q_2 \dashrightarrow_{D/0} q_2 \rightarrow_D q_3 \rightarrow_{]} q_4 \dashrightarrow_{D/0} q_4 \rightarrow_D q_5 \dashrightarrow_{D/4} q_{init} \rightarrow_{\$} q_{acc}$, with $w(p) =$[], which has three terminal push edges.

For $e_1 = q_{init} \rightarrow_{[} q_2$, we have the left prefix $p'_1 = \epsilon$, $w(p'_1) = \epsilon$, and left frontier $LF(p'_1) = \{q_{init}\}$. We can thus insert] because it is not in $F(q_{init}) = \{[, \$\}$ (i.e., condition (i) holds) but not [because it is in $F(q_{init})$ and because $RF(p'_1, [) = \{q_2, q_3\}$ and $[\in F(q_2) = \{[,]\}$ (i.e., both conditions (i) and (ii) fail).

For $e_2 = q_3 \rightarrow_{]} q_4$, we have $p'_2 = q_{init} \rightarrow_{[} q_2 \dashrightarrow_{D/0} q_2 \rightarrow_D q_3$, $w(p'_2) =$[, and $LF(p'_2) = \{q_2, q_3\}$. Condition (i) therefore fails for both [and]. Since $RF(p'_2, [) = \{q_2, q_3\}$ and $[\in F(q_2)$, and similarly, $RF(p'_2,]) = \{q_4\}$ and $[\in F(q_4) = \{[,], \$\}$, condition (ii) also fails for both [and].

Finally, for $e_3 = q_5 \rightarrow_{\$} q_{acc}$, we have $p'_3 = q_{init} \rightarrow_{[} q_2 \dashrightarrow_{D/0} q_2 \rightarrow_D q_3 \rightarrow_{]} q_4 \dashrightarrow_{D/0} q_4 \rightarrow_D q_5 \dashrightarrow_{D/4} q_{init}$, $w(p'_3) =$ [], and $LF(p'_3) = \{q_1, q_3, q_4, q_5\}$. Hence, condition (i) fails, as above; condition (ii) fails for], as $RF(p'_3,]) = \{q_4\}$ and $\$ \in F(q_4)$, but succeeds for [, as $RF(p'_3, [) = \{q_2\}$ and $\$ \notin F(q_2)$, and we can insert an open bracket at the end of the word.

Overall, the insertion mutation thus yields the two negative tests][] and [][from this path. Note that in this example longer paths do not allow any further insertions at the inner terminal push edges of the path, due to the small terminal set and the densely connected structure of the LR-graph.

Substitution. This mutation replaces the label b of the mutated edge by another terminal symbol $d \in T$. However, in contrast to the insertion mutation, we need to look ahead two terminal symbols on the accepting path p, in order to formulate general conditions for the substitution. We therefore require that p is of the form $q_{init} \rightsquigarrow^* q_k \rightarrow_b q_{k+1} \twoheadrightarrow^* \circ \rightarrow_c q_n \rightsquigarrow^* q_{acc}$ with $c \in T \cup \{\$\}$ and $q_n \rightsquigarrow^* \circ \rightarrow_{\$} q_{acc}$ if $c \neq \$$. We can then substitute d for b if either (i) $d \notin \bigcup_{q \in LF(p')} F(q)$ or (ii) $c \notin \bigcup_{q \in RF(p',d)} F(q)$ hold, similar to the case of the insertion mutation.

For the example path p above, we now get the two mutation edges e_1 and e_2 only. For e_1, we can substitute] for [because $LF(p'_1) = \{q_{init}\}$ and $] \notin F(q_{init})$. For e_2, we can substitute [for] because $RF(p'_2, [) = \{q_2\}$ and $\$ \notin F(q_2)$. The substitution mutation thus yields the two negative tests]] and [[from this path; again, longer paths do not yield more tests here.

Deletion. This mutation simply ignores the label b of the mutated edge if $c \notin \bigcup_{q \in LF(p')} F(q)$, i.e., if the labels on all of the next terminal push edges cannot follow the states recognizing the left prefix.

For the same example path p as above, we again get only two mutation edges, e_1 and e_2, and we can delete the labels on both of them, for the same reasons as above. The deletion mutation thus yields the two negative tests] and [from this path; longer paths also yield at most two tests: if the induced words start with [[or end with]], the conditions for the deletion do not hold even at the start or the end of the path, and no negative tests are generated.

6.1.2. Global approximation

The computation of the left frontiers for each of the left prefixes, which is used in the path-sensitive analysis needs to explore all possible parses of these prefixes. We can avoid these computations by reformulating the checks in terms of witnesses for appropriate precede and follow sets, independent of reachability. We ensure that these tests are an overapproximation, i.e., if there are no corresponding witnesses then we cannot have a valid accepting path for the mutated word. Witnesses (more precisely, their absence) thus play the same role as *poisoned pairs* in word and rule mutation algorithms (Raselimo et al., 2019) for negative test suite construction. We can therefore derive conditions for edge mutations in the same way as for word mutations.

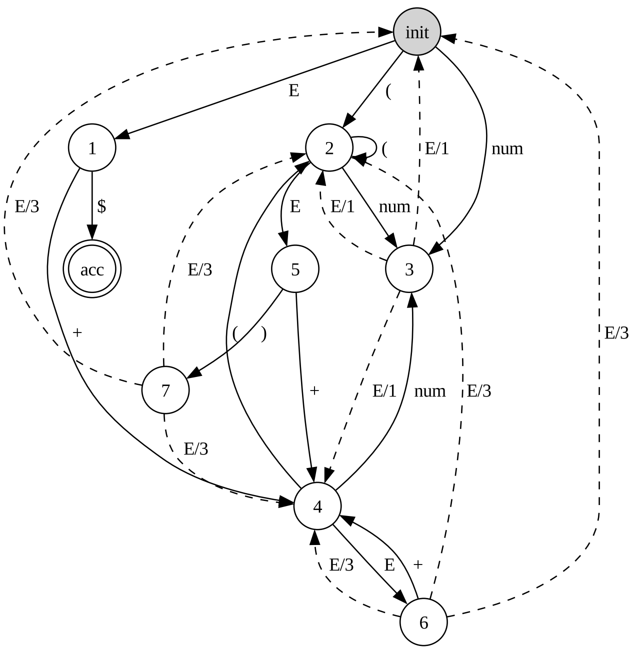
In the following we assume that $w(p) = bcv$ if $w(q_{init} \rightsquigarrow^* q_k) = \epsilon$ and $w(p) = uabcv$ otherwise, with $c \in T \cup \{\$\}$. We can then overapproximate $LF(p)$ by all vertices $w(a, b)$ that are a witness for (a, b) or by $\{q \mid b \in F(q)\}$ if $w(q_{init} \rightsquigarrow^* q_k) = \epsilon$.

Insertion. We can insert $d \in T$ at the mutation location if (i) there is no witness for (a, d) for $w(q_{init} \rightsquigarrow^* q_k) \neq \epsilon$ and $d \notin F(q_{init})$ otherwise or (ii) there is no witness for (d, b).

As with the path-sensitive analysis above, we can insert] at the beginning (i.e., at e_1) because $] \notin F(q_{init})$ (but not [), and [at the end (i.e., at e_3) because there is no witness for $([, \$)$ (but not] because q_3 and q_4 are witnesses for $([,])$ and $(], \$)$, respectively). Likewise, because q_2 and q_3 are witnesses for $([, [)$ and $([,])$, respectively, we cannot insert [at e_2 (i.e., between [and]), and similarly for]. Hence, the global approximation leads to the same result as the path-sensitive analysis.

Substitution. Substitutions create negative test cases similar to insertions, by replacing the label b of the mutation edge with $d \in T$ if (i) there is no witness for (a, d) for $w(q_{init} \rightsquigarrow^* q_k) \neq \epsilon$ and $d \notin F(q_{init})$ otherwise or (ii) there is no witness for (d, c).

We can thus substitute] for [at e_1 and [for] at e_2, for the same reasons as in the case of inserting the same terminal symbol at the beginning resp. end of the path. Note that in this example substitutions are impossible at the inner terminal push edges of longer paths because there are witnesses for all four combinations of [and].



$$E \rightarrow E + E \mid (E) \mid \texttt{num}$$

Fig. 4. Simple expression grammar and corresponding LR-graph.

Deletion. Deletions simply ignore the label b of the mutation edge when the word is extracted during the path traversal. This is guaranteed to create negative test cases if there is no witness for (a, c) for $w(q_{init} \rightsquigarrow^* q_k) \neq \epsilon$ and $c \notin F(q_{init})$ otherwise.

In this case, we can delete the symbols at both e_1 and e_2 (i.e., [and]) because $] \in F(q_{init})$ and $w([, \$) = \emptyset$, respectively. Note that for longer paths no other deletions are possible, for the same reasons as for the substitution mutations.

6.2. Stack mutations

In contrast to edge mutations, which affect individual terminal symbols, stack mutations affect an entire sequence of operations on the stack, so that multiple tokens in the resulting word may be affected at once as well. Hence, stack mutations must be implemented in the path construction phase because they work with entire reduction paths and can no longer simply be grafted on to the word extraction phase.

Below we formalize the applicability conditions for these mutations using the witness sets; however, as we are working with paths, rather than with individual edges, we need to use the first and last sets of these paths, respectively. Let $p' = q_{init} \rightsquigarrow^* q$ and $p'' = q'' \rightsquigarrow^* q_{acc}$ be paths, and $r = q \dashrightarrow_{A/|\alpha|} q' \rightarrow_A q''$ be a reduction path for the pop edge $q \dashrightarrow_{A/|\alpha|} q'$ (respectively $r = q \rightarrow_a q''$ be a single terminal push edge) such that $p' \circ r \circ p''$ has a valid expansion p.

Insertion. This mutation splices a valid reduction, represented by a reduction path $\bar{r} = \bar{q} \dashrightarrow_{B/|\beta|} \bar{q}' \rightarrow_B \bar{q}''$, into an invalid position on the path p such that $p' \circ \bar{r} \circ r \circ p''$ has no valid expansion. Specifically, this is the case if (i) $\bar{r}$ is not nullable, and either (ii) $W(\text{last}(p'), \text{first}(\bar{r})) = \emptyset$ if p' is not nullable, and $W(F(q_{init}), \text{first}(\bar{r})) = \emptyset$ otherwise (i.e., the first token pushed onto the stack to set up the reduction cannot follow the last token processed) or, vice versa, (iii) $W(\text{last}(\bar{r}), \text{first}(r)) = \emptyset$. Hence, the inserted reduction path $\bar{r}$ results in a token sequence which is not valid, given the top-of-stack when q' (or any other possible node $\hat{q}$ that can be reached through any of the last tokens on p') is reached in the traversal, or is invalid before q'' is reached in the traversal. Note that the mutation also works if the mutation location is q_n, i.e., at the end of the word.

For example, we can insert the reduction path $\bar{r} = q_4 \rightarrow_(q_2 \rightarrow_E q_5 \rightarrow_) q_7 \dashrightarrow_{E/3} q_4 \rightarrow_E q_6$ after q_5 into the path $q_{init} \rightarrow_(q_2 \rightarrow_{num} q_3 \dashrightarrow_{E/1} q_2 \rightarrow_E q_5 \rightarrow_) q_7 \dashrightarrow_{E/3} q_0 \rightarrow_E q_1 \rightarrow_\$ q_{acc}$ since neither the path nor $\bar{r}$ are nullable and the first set of $\bar{r}$ only contains ((which is not in the last set of the path segment leading up to q_5.

Substitution. Like an insertion mutation, a substitution mutation also splices a valid reduction path $\bar{r}$ into an invalid position on p, but it first removes an existing valid reduction path (resp. terminal push edge) r from p, such that $p' \circ \bar{r} \circ p''$ has no valid expansion. The conditions for substitution mutations are therefore similar to those of insertion mutations; specifically, the nullability condition is relaxed, i.e., (i) $\bar{r}$ is nullable only if r is nullable, the "left" condition is identical, i.e., (ii) $W(\text{last}(p'), \text{first}(\bar{r})) = \emptyset$ if p' is not nullable, and $W(F(q_{init}), \text{first}(\bar{r})) = \emptyset$ otherwise, while the "right" condition needs to check against the remainder p'' rather than against r, i.e., (iii) $W(\text{last}(\bar{r}), \text{first}(p'')) = \emptyset$. This mutates the top-of-stack sequence caused by the traversal from q to q'' by replacing it with a sequence that cannot occur between q and q''.

In the LR-graphs shown in both of Figs. 1 and 4 there are no cases where substitution may be applied, due to the small sizes and highly recursive nature of the grammars. However, in larger and more structured grammars it proves to be very useful (see the experimental results in Section 8.2.4).

Deletion. We can delete the reduction path r from p if $W(\text{last}(p'), \text{first}(p'')) = \emptyset$ if p' is not nullable, and $W(F(q_{init}), \text{first}(p'')) = \emptyset$ otherwise; note that these conditions already imply that r is not nullable. This will result in gaps in the top-of-stack context required after q'', and so produce a negative test case, since the tokens following q'' cannot occur after q in any valid word in $L(G)$.

For example we can delete the reduction path $\bar{r} = q_2 \rightarrow_{num} q_3 \dashrightarrow_{E/1} q_2 \rightarrow_E q_5$ from the path $p = q_{init} \rightarrow_(q_2 \rightarrow_{num} q_3 \dashrightarrow_{E/1} q_2 \rightarrow_E q_5 \rightarrow_) q_7 \dashrightarrow_{E/3} q_{init} \rightarrow_E q_1 \rightarrow_\$ q_{acc}$ as neither of the remaining path segments are nullable and $W(\text{last}(p'), \text{first}(p'')) = \emptyset$

7. Implementation

We have implemented both algorithms described in this paper using Python 3. Both the LR-BFS and PEC implementations largely follow the pseudocode presentations here. Both implementations are available through the github repository https://github.com/TheLonelyNull/Pytomata.

Hyacc. We used HYacc (Smith, 2020), a Yacc (Johnson, 1975) variant, to produce an initial LR-automaton which we then used to construct an LR-graph. HYacc can construct LR(0)-, LALR(1)-, and LR(1)-automata. In principle, the LR(0)- and LALR(1)-automata are isomorphic, since they use the same itemset construction, but LALR(1)-parsers rely on lookahead to resolve reduce/reduce conflicts whereas LR(0)-parsers do not. In principle, the LR-automata should therefore induce the same LR-graphs which contain pop-edges for each of the possible reductions, whether the conflict is resolved or not. In practice, however, HYacc's implementation of LALR(1)-algorithm inserts additional error-transitions, without leaving sufficient log information to reconstruct all reductions, so that the number of pop edges may be lower for LALR(1)-graphs. We also noticed similar effects when using Bison as parser generator. In our experimental evaluation, we therefore do not use the LALR(1)-automata.

Extending these LR-automata into an LR-graph is fairly straightforward. All push transitions are immediately translated into push edges. Then all reductions, even those which result in conflicts, are used to calculate all pop edges which satisfy these reductions. Where duplicate pop edges are created we prune these to be left with a single pop edge. Extra information such as reduction paths are also calculated here and cached as meta data in the nodes or edges that require this information during the computation that follows this construction step.

Optimizing pop-edge coverage. The PEC algorithm is inherently more efficient than the LR-BFS algorithm since it builds the paths from larger fragments than just individual edges. However, the embedding step can still result in a large amount of redundant computation, especially as embeddings are recomputed every time a reduction paths is embedded into a larger path. We optimize this by constructing an "embedding tree", in linear time, out of the reduction paths. Together with an appropriate caching structure, this enables us to quickly read the order of embeddings required by simply reading the path from a node in this tree to the root node to construct a valid path out of a reduction path.

Our Python implementation of the optimized PEC algorithm performed well in practice, even on the larger grammars we used in the evaluation. It typically performed comparably to a Prolog-based implementation of the generic coverage algorithm when this was called with coverage criteria producing test suites of similar sizes.

Negative test case generation. We did not implement the path-sensitive analysis but only the global approximation algorithm, due to the complexity of the left frontier calculations. Moreover, our implementation does not insert any symbols right before final $q \to_{\$} q_{acc}$ edges.

8. Evaluation

In our experimental evaluation, we focus on the new PEC algorithm since the original LR-BFS algorithm did not scale to larger input grammars, even after the optimizations described in Section 4.

Our first goal is to quantitatively analyze the test suites constructed by the PEC algorithm. We specifically address the following two research questions:

RQ1. Can the PEC algorithm generate test suites for complex, production-quality grammars?

RQ2. How does the size of the test suites produced by the PEC algorithm vary with the choice of the LR-automaton?

We then compare the PEC algorithm, to current state-of-the-art grammar-based test suite generation algorithms, specifically the generic cover algorithm with different coverage criteria, using the same grammars as above, as well as corresponding SUT implementations.

RQ3. How do the PEC and generic cover algorithm (instantiated with different coverage criteria) compare in terms of (a) the test suite sizes and (b) the induced SUT code coverage, respectively?

We finally evaluate the fault detection capabilities of the different test suites for a number of student grammars for the same target language. Here, we use grammars with known faults that have been previously used in experiments to investigate fault localization and grammar repair (Raselimo and Fischer, 2021).

RQ4. How well do the different positive and negative test suites reveal errors in the language accepted by a grammar?

8.1. Experimental setup

8.1.1. Design

For RQ1 and RQ2, we use HYacc to produce LR(0)- and LR(1)-automata for three different medium to large input grammars (see Section 8.1.2 below for more details), and construct positive and negative test suites over the derived graphs, as described in Sections 5 and 6.

For RQ3(a), we compare the sizes of the test suites generated by the PEC algorithm from LR(0)- and LR(1)-graphs with those generated by the generic cover algorithm when it is instantiated with the different coverage criteria shown in Algorithm 1.

For RQ3(b), we compare the performance of the PEC and generic cover algorithms through the statement and branch coverage of the SUTs when they are run over the respective generated test suites. We consider an algorithm to be better when it achieves higher coverage. However, it is important to remember that coverage is only one possible proxy for test suite quality, and other metrics (e.g., number of bugs discovered) are possible. We chose coverage, in line with other work (Havrikov and Zeller, 2019; van Heerden et al., 2020), because it can be collected reliably and provides more fine-grained measurements (i.e., shows more differences between different algorithms) than for example the number of bugs discovered.

Since we are comparing different algorithms for RQ3, we carefully consider in our experimental design the effects of bias, as described by Rossouw and Fischer (2021). In order to mitigate against *equivalent choice* bias in the generic cover algorithm, we generated 100 equivalent grammar versions where we randomly shuffled the order of the rules in the grammar, and generated the test suites for each coverage criterion from the versions. In our implementation of the cover algorithm, this induces a corresponding random resolution of equivalent choices. For the PEC algorithm, we similarly generate 100 different test suites, randomly resolving equivalent edge choices. In order to mitigate against *embedding* bias in the generic cover algorithm, we generated 100 test suites using shallowest and shortest yield embeddings, respectively, and compared the coverage for these two variants independently with that of the PEC algorithm (which does not suffer from embedding bias).

For RQ4, we cannot measure the fault revealing capabilities of the different test suites simply as the fraction of failing tests, because a single fault can cause any numbers of tests to fail. Instead, we used spectrum-based fault localization (Raselimo and Fischer, 2019) to evaluate the effectiveness of the different test suites at finding known errors in a grammar: we consider a test suite better if the fault localization process using this test suite incurs a lower *wasted effort*. More specifically, we used the Ochiai metric (Ochiai, 1957) to assign a suspiciousness score to each rule in the grammar, based on how often these rules were exercised by the passing and failing test cases in the test suite. This allows us to rank all suspicious rules and determine how many rules must be examined before the first true, known error is found.

For this process we need SUTs with known grammar-level errors; here, we used grammars with known faults that have been submitted for an assignment in a compiler engineering course and have previously been used in experiments to investigate fault localization and grammar repair (Raselimo and Fischer, 2021). We manually inspected the grammars and identified the faulty rules; in cases where multiple rules acted together to cause a single error, we identified a unique root cause rule, based on our understanding of the grammar and possible repairs for the fault.

Runtime measurements. For the quantitative evaluation, we focus on test suite sizes and their induced SUT coverage; a runtime comparison of the PEC and generic cover algorithms is unreliable because both implementations are un-optimized research implementations and their runtimes are impacted substantially by string processing and file output times; we therefore refrain from reporting more detailed timing measurements.

Both algorithms also compute and cache some grammar metadata (e.g., nullability of non-terminals, first and last sets, shortest embeddings, or reduction paths); the PEC algorithm additionally constructs the LR-graph from the LR-automaton. The runtime of these pre-processing steps depends on the size and structure of the grammar; in our implementation, and using a standard laptop, the pre-processing times for our subject grammars were between 0.5 and 15 s for both algorithms.

The runtime of the SUT over the generated test suites cannot easily be predicted from the test suites themselves; however, both algorithms construct short tests (via shortest paths resp. minimal derivations), and in our evaluation the SUT runtimes for the individual tests were roughly constant. We therefore refrain from reporting more detailed timing measurements.

8.1.2. Subjects

For the construction and evaluation of the positive test suites we used three different grammars of varying complexity (AMPL van Heerden et al., 2020, SQLite SQLite, 2021b, and Go GCCGo, 2021), together with corresponding SUTs, as described in more detail below.

The grammars were originally written in EBNF-style for LL-parsers; we eliminated the EBNF constructs and converted them into HYacc notation to make them suitable as input for our implementation, but we did not eliminate any shift/reduce or reduce/reduce conflicts. The resulting grammars are also available at https://github.com/TheLonelyNull/Pytomata. The SUTs, however, use different parsers that are implemented manually (AMPL, GCCGo) or were generated from a third grammar (SQLite). Hence, both the original LL-grammars and the derived HYacc-versions we used in our experiments serve only as models for the target language.

All three SUTs are implemented in C. We used the `gcov` tool to measure statement and branch coverage of the SUTs over the different test suites, as described in more detail below.

AMPL. AMPL is an example language ($|N| = 43, |T| = 49, |P| = 87$ in the BNF version) designed for a computer architecture course. van Heerden et al. (2020) have previously also used AMPL in their experiments, together with a code base of 61 student submissions implementing a single-pass compiler with a recursive-descent parser translating from AMPL to Java byte code. The compilers comprise on average about 1300 non-comment non-blank lines of code (LoC). We have used the same code base here.

We ran each test case in the respective test suites over the individual compilers separately and aggregated the coverage information for all tests in a test suite into a cumulative statement and branch coverage for each test suite and compiler combination. However, we take the average coverage a test suite induces over the 61 student compilers as the primary metric, and use the average over each of the 100 test suite variants to determine the coverages for an algorithm or coverage criterion as the main metrics for the comparison (cf. Table 3). We also report the minimal and maximal coverage values we observed for a variant, as well as the standard deviation of the coverage values.

We also consider the coverage of merged test suites (i.e., the union of all 100 test suite variants for a given algorithm or coverage criterion, respectively). This gives us additional estimates of the variance and upper-bound performance of an algorithm. We finally consider the union of these merged test suites for the PEC algorithm and for the generic cover algorithm with each coverage criterion, respectively, as an indication to which extent the two algorithms are complementary.

SQLite. SQLite is an SQL database engine written in C that is widely used as an on-device datastore for mobile and IoT applications. We used version 3.8.x of the SQLite grammar ($|N| = 201, |T| = 155, |P| = 586$ in the BNF version) from the ANTLR grammar repository (ANTLR, 2021) as input grammar and compiled an executable for SQLite (v3.36.0) from the sources, which comprised 234 229 LoC in total. SQLite itself uses a parser generated by the Lemon LALR(1) parser generator from a different grammar (SQLite, 2021a).

SQLite is obviously much larger and more complex than the AMPL compilers; it is also not a single-pass system so that the test cases do not penetrate the SUT as deeply. This effect is magnified by the structure of the test suites that, due to the complexity of the grammar, only comprise individual SQL statements but no longer scripts.

We followed a similar measurement approach as in the AMPL experiments. We executed each test (i.e., SQL statement) individually and aggregated the coverage information for all tests in a test suite into a cumulative statement and branch coverage for each test suite. We use the average over the 100 test suite variants to determine the coverages for an algorithm or coverage criterion as the main metrics for the comparison (cf. Table 4), as above. We also compute the coverage induced by the merged test suites, as above. However, due to the size of these, we did not run the SUT over the union of the merged PEC test suite with the respective merged test suites for the different coverage criteria, but simply report the static information.

Go. Go is a popular programming language that is often used to build memory-safe applications. In our experiments we used the version of the Go grammar ($|N| = 159, |T| = 84, |P| = 324$ in the BNF version) found in the ANTLR grammar repository (ANTLR, 2021). Programs written in Go can be compiled with GCC by using a frontend for GCC known as GCCGo (GCCGo, 2021). It uses a recursive-descent parser that closely follows the grammar provided in the Go specification. We use only its parser module as SUT, in order to assess whether the observations we made for AMPL and SQLite are skewed by the semantic actions associated with the parsing phase or by the SUTs' later phases (e.g., query optimization). We used GCC to compile a GCCGo frontend executable from sources. The parsing module comprises 5972 LoC.

We followed the same measurement approach as in the SQLite experiments; due to the smaller test suite sizes, we were also able to collect coverage information for the different unions of the merged test suites.

SUTs for fault detection capabilities. We evaluated the impact of the test suites produced by the different positive and negative coverage criteria on our ability to identify (and localize) errors in a grammar over nine AMPL grammars that contain errors, out of 25 submitted in total in response to an assignment in a compiler engineering course. Here, the students were given the same EBNF as in the computer architecture course; their main task was to transform this into a form and format suitable as input for jflex (version 1.8.2) and JavaCUP (version 11b).

8.2. Results

8.2.1. Test suite sizes

PEC results. Table 1 summarizes the quantitative evaluation of the PEC algorithm. The first block shows the sizes of the two different LR-graphs for our three subject grammars, specifically the number of nodes $|Q|$ (which is the number of states in the underlying LR-automaton), push edges $|E_{\rightarrow}|$ and pop edges $|E_{\dashrightarrow}|$, as well as the number of *shift/reduce* resp. *reduce/reduce* conflicts and the number of states with either one or both of those conflicts. The second block shows the sizes of the corresponding test suites generated by the PEC algorithm; for negative tests, we show the number of tests constructed using each of the individual mutation operations as well as the total number of different tests, which is not necessarily the sum of the individual test suite sizes, as different mutations can create the same tests.

We can see that the PEC algorithm can indeed construct (positive) test suites for complex grammars, independent of the type of the underlying LR-graph. For all three grammars, the algorithm produces roughly proportionally larger test suites for the larger LR(1)-graphs than for the LR(0)-graphs, i.e., the number of pop edges and the number of tests grow roughly at the same rate. However, the growth rates (see Table 1) increase with the sizes of the different grammars, from approx. 4 (AMPL) to approx. 14 (SQLite).

The runtimes are negligible in most cases, and our implementation constructs between 1000 and 5000 tests per second, which is comparable to the performance of the generic cover algorithm. We did not observe large performance degradations for large test suites, and even using the LR(1)-graph derived from the SQLite grammar, it takes less than two minutes to generate all tests.

For negative test suites, the results indicate some scalability issues, as the number of tests grows explosively, to (at least) three orders of magnitude more than in the corresponding positive test suites.

Edge insertions and substitutions both check the same conditions for each terminal symbol they try to insert (resp. replace by) at each mutation location; however, insertion applies at one additional location on each path, so that it the number of edge insertions is slightly higher. In both cases, the number grows linearly with the product of the number of terminal symbols and the number of positive tests. The number of edge deletions is obviously independent of the number of the terminal symbols, and since each terminal shift edge on the path of each

Table 2
Sizes of positive test suites generated by PEC and generic cover algorithms using different coverage criteria, averaged over 100 runs. Ratio is computed between maximum- and minimum-sized test suites. $\Rightarrow^*_\sqsubseteq$ denotes shallowest embeddings, $\Rightarrow^*_\leq$ denotes shortest yield embeddings.

		PEC		pll		rule		cdrc		step$_3$		step$_4$		deriv		bfs$_2$	
		LR(0)	LR(1)	$\Rightarrow^*_\sqsubseteq$	$\Rightarrow^*_\leq$	$\Rightarrow^*_\sqsubseteq$	$\Rightarrow^*_\leq$	$\Rightarrow^*_\sqsubseteq$	$\Rightarrow^*_\leq$	$\Rightarrow^*_\sqsubseteq$	$\Rightarrow^*_\leq$	$\Rightarrow^*_\sqsubseteq$	$\Rightarrow^*_\leq$	$\Rightarrow^*_\sqsubseteq$	$\Rightarrow^*_\leq$	$\Rightarrow^*_\sqsubseteq$	$\Rightarrow^*_\leq$
AMPL	Avg	144.9	631.9	69.4	59.0	45.0	45.0	78.0	78.7	179.0	180.6	426.0	432.2	423.1	436.8	147.0	150.0
	Min	141	625	64	59	45	45	78	78	179	179	426	426	400	420	147	147
	Max	149	639	76	59	45	45	78	80	179	185	426	447	461	467	147	156
	Ratio	1.06	1.02	1.19	1.00	1.00	1.00	1.00	1.03	1.00	1.03	1.00	1.05	1.15	1.11	1.00	1.06
SQLite	Avg	22 364.7	318 972.6	4834.9	4184.1	367.7	370.0	1697.3	1680.6	31 865.5	30 895.8	634 483.6	622 907.7	2935.5	2991.4	57 776.2	56 876.0
	Min	22 126	23 457	4539	4148	362	363	1615	1607	28 923	28 142	519 912	519 981	2680	2841	53 225	48 315
	Max	22 526	367 344	5047	4289	373	383	1742	1743	33 027	32 167	656 079	656 974	3080	3157	59 021	58 869
	Ratio	1.02	15.66	1.11	1.03	1.03	1.06	1.08	1.08	1.14	1.1.4	1.26	1.26	1.15	1.11	1.11	1.22
Go	Avg	3486.3	23 525.9	617.8	611.2	163.6	163.8	318.6	322.1	1247.1	1265.5	4320.8	4424.3	5773.4	5942.4	462.7	466.7
	Min	3455	23 444	602	588	163	163	318	318	1245	1246	4319	4314	4631	5813	462	462
	Max	3521	23 621	638	638	164	164	319	326	1248	1281	4322	4511	6044	6123	463	471
	Ratio	1.02	1.01	1.06	1.09	1.01	1.01	1.00	1.03	1.00	1.03	1.00	1.05	1.31	1.05	1.00	1.02

positive test gives rise to at most one negative test, the number of edge deletion mutants is much smaller. Overall, the behavior of the edge mutation is similar to that of the word mutation algorithm (Raselimo et al., 2019) which produces similarly sized negative test suites if it is applied to positive bases of similar sizes than those produced by PEC.

The algorithm computes the edge mutations very fast (up to 100000 negative tests per second) as part of the word extraction, and thus completes for all three grammars when using LR(0)-graphs, but fails to complete even within one week when using the LR(1)-graph for SQLite, as the number of paths is too large.

The implementation of the stack mutations is substantially more complex, as it is integrated into the path construction. It is thus even slower than the positive test suite construction. Consequently, it completes (with the exception of stack deletions) only for the smallest of the three grammars, AMPL. Here, the number of the different stack mutations is roughly similar to the number of the corresponding edge mutations, for both types of LR-graphs. We conjecture that this follows from the fact that the grammar is LL(1), with roughly the same number of terminal and non-terminal symbols ($|N| = 43, |T| = 49$). However, the number of stack mutations is 10× (for LR(0)-graphs) resp. 80× (for LR(1)-graphs) larger than the number of rule mutations by Raselimo et al. (2019), even though the latter use both terminal and non-terminal mutations (i.e., insertions, substitutions, and deletions). The main reason for this difference is that the (edge and) stack mutations are applied to all paths that the PEC algorithm constructs and thus explore the same mutation in different contexts, while the rule mutation algorithm embeds each mutated rule only once.

Summary. We can answer our first two research questions based on the results shown in Table 1 and their discussion above. For RQ1, we can conclude that the new PEC algorithm can indeed generate (positive) test suites for complex, production-quality grammars, but faces scalability issues in the generation of the corresponding negative test suites. For RQ2 we can conclude that the choice of the substantially larger LR(1)-graphs does not impact the algorithm's ability to generate positive test suites, but makes it impossible for the algorithm to generate negative test suites for the largest grammar (SQLite). The algorithm produces roughly proportionally larger test suites for LR(1)-graphs than for LR(0)-graphs, i.e., the number of pop edges and the number of tests grow roughly at the same rate. However, the growth rates increase with the size of the grammars from approx. 4 (AMPL) to approx. 14 (SQLite).

Comparison to generic cover algorithm. Table 2 shows the average as well as minimum and maximum test suite size generated by the different methods for our evaluation grammars. We can see that the size of the test suites depends in a complex fashion on the size and structure of the grammar. PEC produces test suites that are larger than those induced by the widely used *pll*, *rule*, and *cdrc* coverage criteria, but with increasing lengths of the covered derivations, the generic cover algorithm begins to produce larger test suites than the PEC algorithm. For LR(0)-graphs this happens at derivation lengths of 3 or 4, but for the larger LR(1)-graphs this happens only with derivation lengths of 4 to 6, depending on the grammar.

Table 2 also shows the effects of the different biases, reflected in the minimum and maximum test suite sizes. For PEC, the effects are generally small, with a difference between minimum and maximum sizes of about 2% to 6% for LR(0)-graphs, and about 1% to 2% for LR(1)-graphs. The only outlier is SQLite, where the smallest LR(1) test suite is only about 6% of the size of the largest; moreover, we observed a clear size dichotomy—about 20% of the test suites are within 1% of the minimum, while the remaining 80% are within 1% of the maximum. We conjecture that this is a consequence of the large number of nullable non-terminal symbols in the grammar, which lead to many shift/reduce and reduce/reduce conflicts, and thus in turn to many test suites with many identical tests extracted from different paths computed by the *embed*-function. For the generic cover algorithm we can see for both embedding styles that the variance increases with increasing lengths of the covered derivations, typically becoming substantially larger than PEC's variances when the *pll* and *deriv* criteria are used, which cover the longest derivations.

Summary. As answer to RQ3(a), we find that PEC produces test suites that are larger than those produced by the generic cover algorithm with the widely used coverage criteria, but eventually become smaller as the length of the covered derivations is increased.

8.2.2. *Code coverage*

Tables 3–5 show the coverage data for the different algorithms and coverage criteria over the three SUTs AMPL, SQLite, and GCCGo. For PEC, we only use the test suites generated from the LR(0)-graphs, as those generated from the LR(1)-graphs are too large to run the repetitions. Mann–Whitney U-tests confirm that the coverage distribution of the PEC algorithm is statistically significant ($\rho < 0.05$) different from all cover methods shown; higher average coverages thus indeed indicate better performance.

Note that our experimental design follows our earlier work (Rossouw and Fischer, 2021), where we also used AMPL and SQLite as SUTs. However, we no longer had access to the test suites and had to re-generated them, and, due to the randomized construction of the variant suites, sizes and achieved SUT coverages reported here differ from the earlier results.

AMPL. For AMPL (see Table 3), PEC constructs test suites that achieve statement and branch coverage of 64.4% and 50.1%, respectively, on average over all 100 variants; the generic cover algorithm performs better here, achieving its best performance (66.2% and 55.1% statement and branch coverage, respectively) using *deriv* coverage and shortest yield embeddings. However, the results are strongly correlated with the test suite sizes. The PEC algorithm thus outperforms the generic cover algorithm for the *pll*, *rule*, and *cdrc* criteria, which induce smaller

Table 3
AMPL code coverage for different embeddings and coverage criteria; line is *line* (or *statement*) coverage, cond is *condition* (or *branch*) coverage. Coverage is averaged over all 61 student compilers. The minimum and maximum coverage percentages in each row are shown in italics and bold, respectively. The average sizes of the test suites for the different criteria are shown in parentheses next to the respective names.

			Shortest yield embedding ($\Rightarrow^*_\leq$)													
Algorithm	PEC (144.9)		pll (59.0)		rule (45.0)		cdrc (78.7)		$step_3$ (180.6)		$step_4$ (432.2)		deriv (436.8)		bfs_2 (150.0)	
Coverage	line	cond	line	cond	line	cond	line	cond	line	cond	line	cond	line	cond	line	cond
Avg	64.4	50.1	63.0	51.2	*62.5*	*48.5*	63.5	50.2	64.7	51.4	66.0	53.3	**66.2**	**55.1**	63.8	50.5
Max	65.4	51.5	63.6	52.3	*63.2*	*49.9*	64.3	51.4	65.4	52.5	**66.6**	54.2	**66.6**	**55.6**	64.6	51.8
Min	62.8	48.0	61.9	49.7	*61.4*	*46.8*	62.3	48.2	63.9	49.6	65.3	52.0	**65.6**	**54.2**	62.6	48.5
Stdev	0.6	0.8	0.5	0.7	0.5	0.8	0.5	0.8	0.5	0.8	0.3	5.3	0.2	0.3	0.5	0.8
Merged	66.1	52.1	*64.9*	54.3	*64.9*	*53.8*	65.5	54.7	66.3	55.5	67.3	57.0	**67.4**	**57.6**	65.7	54.9
Merged size	1817		211		198		520		1923		4953		4941		1553	
Composite	–		*66.5*	56.1	*66.5*	*55.8*	*66.5*	55.9	66.7	56.1	67.5	57.3	**67.6**	**58.1**	66.6	56.0
Composite size	–		1978		1970		2277		3581		6548		6512		3310	
PEC exclusive	–		96%		97%		95%		87%		80%		78%		95%	

			Shallowest embedding ($\Rightarrow^*_\sqsubseteq$)													
Algorithm	PEC (144.9)		pll (69.4)		rule (45.0)		cdrc (78.0)		$step_3$ (179.0)		$step_4$ (426.0)		deriv (423.1)		bfs_2 (147.0)	
Coverage	line	cond	line	cond	line	cond	line	cond	line	cond	line	cond	line	cond	line	cond
Avg	64.4	50.1	61.8	49.6	*60.2*	*44.5*	61.2	46.2	62.6	47.5	65.1	50.8	**65.2**	**51.8**	61.6	46.6
Max	65.4	51.5	63.8	51.5	*63.5*	*48.0*	64.6	50.2	65.6	51.3	66.7	53.0	**66.8**	**55.6**	65.2	50.8
Min	62.8	48.0	59.7	47.5	*57.0*	*41.2*	57.9	42.5	59.1	43.8	**62.9**	**48.5**	62.8	47.3	58.3	43.1
Stdev	0.6	0.8	1.4	1.3	2.3	2.2	2.3	2.4	2.1	2.2	1.1	1.2	1.5	3.2	2.3	2.4
Merged	66.1	52.1	65.2	53.1	*65.1*	*51.6*	66.0	54.1	66.8	55.0	67.4	55.8	**67.6**	**57.4**	66.8	54.9
Merged size	1817		615		552		1294		4002		10 241		7967		3627	
Composite	–		66.5	55.7	*66.2*	*54.3*	*66.2*	*54.3*	66.8	55.0	67.5	55.8	**67.7**	**57.7**	67.0	55.2
Composite size	–		2293		2232		2869		5432		11 633		9455		5141	
PEC exclusive	–		83%		84%		69%		50%		45%		59%		60%	

test suites; it performs slightly better than or on par with the generic cover algorithm when the induced test suites have a comparable size (i.e., for $step_3$ and bfs_2), and is outperformed by the substantially (roughly 3×) larger test suites $step_3$ and *deriv*. This is independent of the embedding used for the generic cover algorithm, although the only slightly smaller shallowest embeddings test suites lead to substantial losses of approximately 2 resp. 4 percentage points for statement resp. branch coverage.

Note that for shallowest embeddings the performance of the generic cover algorithm varies significantly more over the different grammar variants than that of the PEC algorithm. The best case performance of the PEC algorithm is thus even further below that of generic cover algorithm for the larger test suites (e.g., *deriv* with 66.8% statement coverage) than the average performance; on the flip side, however, PEC's worst-case performance is better.

Merging all 100 variants into a single large test suite leads to 5–10× increase in the number of test cases (and even 10–20× for shallowest embeddings, due to the large variance). These merged test suites achieve coverages that are typically three to five standard deviations above the average, and typically still one standard deviation above the best individual variant. As for the individual variants, PEC outperforms the generic cover algorithm when its merged test suites are larger, performs on par when they are roughly the same size, and is outperformed when they are smaller.

SQLite. For SQLite (see Table 4), the generated test suites are one to two orders of magnitude larger than those for AMPL, due to the more complex grammar, yet their induced code coverage is much lower. This is caused by the restriction of the test suites to single SQL statements (rather than complex scripts) and the stricter phase distinction in the SQLite implementation, which causes many syntactically correct tests to be rejected early by the semantic analysis, before they exercise any code from later phases. Moreover, statement and branch coverage are much closer to each other than for AMPL. However, the difference between the worst (*rule*) and the best ($step_3$) performing method is relatively larger than for AMPL, both on average and for the outliers.

Here, PEC's test suites on average achieve a statement and branch coverage of 26.1% and 25.7%, respectively, while the best performing generic cover algorithm variant is $step_3$ with shortest yield embeddings, achieving 26.7% and 26.5% statement and branch coverage, respectively, with slightly worse results for shallowest embeddings. However, it produces approximately 40% larger test suites than PEC. Generally, methods that only explore short derivations (i.e., *rule*, *cdrc*, and even bfs_2) perform badly here, with an average statement coverage of 24.0–25.3%, while methods that explore longer derivations (i.e., PEC, $step_3$, and *deriv*) perform better. Note that *deriv* performs comparatively well, given the relatively small size of the test suites, while bfs_2 performs comparatively badly, given its large size; this is a strong indication that the covered derivation length plays a more important role than the test suite size.

As in the case of AMPL, the different test suite variants produced by PEC are more uniform than those produced by the generic cover algorithm, in particular for shallowest embeddings. The standard deviation of their induced coverages is thus much smaller than for the generic cover algorithm; likewise, merging the different variants into a single test suite produces a relatively smaller increase in size (approximately 20× compared to 50–60× for $step_3$ and bfs_2 under shallowest embeddings) and a relatively smaller improvement in coverage.

GCCGo. For GCCGo (see Table 5), PEC's test suites on average achieve a statement and branch coverage of 70.6% and 76.7%, respectively, while the best performing generic cover algorithm variant is *deriv* with shallowest embeddings, achieving a slightly higher (70.8%) statement coverage but slightly lower (76.6%) branch coverage. Generally, the methods that only explore short derivations (i.e., *rule*, *cdrc*, and bfs_2) again perform relatively badly here, with an average statement coverage of 57.7–61.8%, while the longer derivations explored by PEC, $step_3$, $step_4$, and *deriv* lead to substantially higher average statement coverages of 67.5–70.8% (with even higher branch coverages). This is hardly surprising, since the SUT is comprised only of the parsing and syntax tree building code, so that better grammar coverage translates directly into better SUT coverage. More specifically, PEC outperforms the generic cover algorithm for all coverage criteria if the more widely used shortest yield embeddings are used as well, even though it produces substantially (i.e., 20%–40%) smaller test suites than the best

Table 4
SQLite code coverage for different embeddings and coverage criteria. Coverage is obtained over the SQLite executable (v3.36.0). See Table 3 for further explanations.

Shortest yield embedding ($\Rightarrow^*_{\leq}$)

Algorithm	PEC (22 364.7)		pll (4184.1)		rule (370.0)		cdrc (1680.6)		step_3 (30 895.8)		deriv (2991.4)		bfs_2 (56 876.0)	
Coverage	line	cond	line	cond	line	cond	line	cond	line	cond	line	cond	line	cond
Avg	26.1	25.7	24.4	23.9	*24.0*	*23.5*	25.1	24.6	**26.7**	**26.4**	26.0	25.2	25.8	25.2
Max	26.7	26.3	26.0	25.4	*25.3*	*24.7*	25.8	25.3	**27.6**	**27.2**	26.9	25.4	25.9	25.4
Min	24.8	24.6	23.7	23.3	*23.3*	*22.8*	24.0	23.7	24.6	24.5	24.4	**24.9**	**25.5**	**24.9**
Stdev	0.4	0.4	0.7	0.6	0.6	0.5	0.5	0.4	0.8	0.7	0.7	0.1	0.1	0.1
Merged	27.0	26.6	26.3	25.7	*25.5*	*24.9*	26.0	25.6	**27.8**	**27.5**	27.4	26.9	27.1	26.7
Merged size	483 289		13 373		2381		30 900		894 470		56 691		1 746 918	
Composite size	–		489 656		481 516		507 003		1 356 470		530 734		2 215 709	
PEC exclusive	–		99%		99%		99%		96%		98%		97%	

Shallowest embedding ($\Rightarrow^*_{\sqsubseteq}$)

Algorithm	PEC (22 364.7)		pll (4834.9)		rule (367.7)		cdrc (1697.3)		step_3 (31 865.5)		deriv (2935.5)		bfs_2 (57 776.2)	
Coverage	line	cond	line	cond	line	cond	line	cond	line	cond	line	cond	line	cond
Avg	26.1	25.7	24.4	23.7	*24.0*	*23.4*	25.3	24.8	**26.5**	**26.0**	25.9	25.3	26.1	25.4
Max	26.7	26.3	*26.2*	*25.5*	*26.2*	*25.5*	26.9	26.4	27.4	27.0	26.9	26.4	**27.7**	**27.1**
Min	24.8	24.6	22.5	22.1	*22.1*	*21.7*	23.8	23.4	**25.1**	**24.7**	24.5	24.1	24.6	24.0
Stdev	0.4	0.4	0.9	0.9	1.0	1.0	0.7	0.6	0.5	0.5	0.5	1.1	0.7	0.7
Merged	27.0	26.6	26.7	26.2	*26.5*	*25.9*	27.3	26.8	27.7	27.5	27.3	26.8	**28.1**	**27.6**
Merged size	483 289		140 592		12 329		72 091		1 683 000		124 401		3 374 922	
Composite size	–		613 545		491 162		549 587		2 133 066		602 343		3 796 527	
PEC exclusive	–		98%		99%		99%		93%		99%		87%	

Table 5
GCCGo code coverage for different embeddings and coverage criteria. Coverage is obtained over the GCCGo parser (included with GCC version 11). See Table 3 for further explanations.

Shortest yield embedding ($\Rightarrow^*_{\leq}$)

Algorithm	PEC (3486.3)		pll (611.2)		rule (163.8)		cdrc (322.1)		step_3 (1265.5)		step_4 (4424.3)		deriv (5942.4)		bfs_2 (466.7)	
Coverage	line	cond	line	cond	line	cond	line	cond	line	cond	line	cond	line	cond	line	cond
Avg	**70.6**	**76.7**	59.9	64.9	*57.7*	*63.0*	61.2	66.2	67.5	73.9	70.3	75.8	70.2	76.3	61.6	66.4
Max	70.8	76.9	60.5	65.6	*57.9*	*63.5*	61.5	66.8	67.9	74.4	**70.9**	76.5	70.8	**77.0**	61.9	67.2
Min	**70.3**	76.5	59.5	64.4	*57.4*	*62.6*	60.9	65.5	67.4	73.7	70.0	75.6	69.7	**75.7**	61.2	65.7
Stdev	0.1	0.1	0.2	0.3	0.2	0.3	0.2	0.4	0.2	0.2	0.2	0.3	0.3	0.4	0.2	0.4
Merged	70.9	**77.1**	60.8	65.8	*58.4*	*63.8*	62.0	66.9	68.5	74.7	**71.1**	76.7	70.9	77.0	62.4	67.4
Merged size	86 735		1386		238		1097		4502		28 827		12 224		4117	
Composite	–		*71.1*	*76.9*	*71.1*	*76.9*	71.4	77.0	72.7	78.2	74.3	**79.9**	**74.4**	79.5	71.6	77.3
Composite size	–		87 914		86 850		87 672		90 993		115 172		98 477		89 951	
PEC exclusive	–		100%		100%		100%		100%		100%		99%		99%	

Shallowest embedding ($\Rightarrow^*_{\sqsubseteq}$)

Algorithm	PEC (3486.3)		pll (617.8)		rule (163.6)		cdrc (318.6)		step_3 (1247.1)		step_4 (4320.8)		deriv (5773.4)		bfs_2 (462.7)	
Coverage	line	cond	line	cond	line	cond	line	cond	line	cond	line	cond	line	cond	line	cond
Avg	70.6	**76.7**	60.3	64.8	*57.7*	*62.9*	61.5	66.6	67.7	74.0	70.2	76.1	**70.8**	76.6	61.8	66.8
Max	70.8	76.9	61.1	66.1	*58.3*	*63.7*	61.8	67.3	68.4	74.9	71.2	77.6	**72.1**	**78.3**	62.3	67.4
Min	70.3	76.5	59.2	63.6	*57.0*	*61.7*	60.8	65.5	66.9	73.0	**69.3**	**74.6**	68.7	74.2	61.0	65.6
Stdev	0.1	0.1	0.4	0.6	0.3	0.5	0.3	0.5	0.3	0.4	0.4	0.7	0.6	0.8	0.4	0.5
Merged	70.9	77.1	62.1	66.5	*59.2*	*64.2*	62.4	67.5	69.5	75.4	72.1	78.1	**74.2**	**80.0**	62.5	67.6
Merged size	86 735		3352		400		1705		6656		40 412		31 327		4117	
Composite	–		71.7	77.5	*71.4*	*77.2*	71.7	77.5	73.6	79.0	75.2	80.8	**76.5**	**82.0**	71.7	77.5
Composite size	–		89 699		86 925		88 104		92 867		126 448		117 275		90 508	
PEC exclusive	–		100%		100%		100%		99%		99%		99%		100%	

performing criteria *step*$_4$ and *deriv*, while it is outperformed only by *deriv* (which is about 65% larger) if shallowest embeddings are used.

As for AMPL and SQLite, merging all variants leads to a very large test suite (about 25× the size of the individual variants) for PEC. However, the SUT coverage remains largely consistent over the variants, despite the variance of the test suites, and the merged test suite produces equivalent or lower coverage than the merged versions for *step*$_4$ and *deriv* (especially for shallowest embeddings), despite being substantially (2–7×) larger.

Summary. As answer to RQ3(b), we find that in terms of code coverage over the three SUTs we have considered here PEC consistently outperforms the generic cover algorithm when it is instantiated with coverage criteria that explore short derivations only (i.e., *rule*, *cdrc*, and *bfs*$_2$), independent of the size of the generated test suites. It is outperformed by generic cover algorithm in some (but not all) cases when the latter is instantiated with coverage criteria *step*$_3$, *step*$_4$, or *deriv* that explore longer derivations but only when these induce larger test suites. We also observe that the generic coverage algorithm requires different combinations of coverage criterion and embedding for the different grammar to perform better than the PEC algorithm.

For the LR(0)-graphs considered here PEC is also more robust against random choices made to resolve ambiguity in the construction of the tests (e.g., the choice of a specific yield to ground out a nonterminal symbol in a derived sentential form), and the statement and branch coverage of test suite variants generated with different choices varies much less than that of the generic cover algorithm.

Table 6
Fault localization results for different AMPL grammars and test suites. - means no failing tests.

Criterion	$\lvert TS \rvert$	$\mathcal{A}_1$ ($\lvert P \rvert = 92, N_{fault} = 3$)			$\mathcal{A}_2$ ($\lvert P \rvert = 98, N_{fault} = 8$)			$\mathcal{A}_3$ ($\lvert P \rvert = 82, N_{fault} = 1$)			$\mathcal{A}_4$ ($\lvert P \rvert = 97, N_{fault} = 3$)			$\mathcal{A}_5$ ($\lvert P \rvert = 90, N_{fault} = 2$)			$\mathcal{A}_6$ ($\lvert P \rvert = 81, N_{fault} = 1$)		
		N_{fail}	N_{susp}	Ranks	N_{fail}	N_{susp}	Ranks	N_{fail}	N_{susp}	Ranks	N_{fail}	N_{susp}	Ranks	N_{fail}	N_{susp}	Ranks	N_{fail}	N_{susp}	Ranks
cdrc	79	2	12	3, 6	75	4	4	1	18	1	3	19	2, 3	5	23	2, 3	2	16	1
PEC	141	8	16	9.5, 9.5	133	6	4	1	17	1	19	19	2, 3	11	28	2, 3	5	21	1
Rule mutation	9 414	–	–	–	–	–	–	–	–	–	–	–	–	13	31	3	–	–	–
Edge insertion	68 387	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
Edge substitution	66 507	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	5	26	2
Edge deletion	1 502	–	–	–	–	–	–	–	–	–	–	–	–	4	20	3	–	–	–
Stack insertion	49 114	–	–	–	–	–	–	–	–	–	–	–	–	17	43	2	–	–	–
Stack substitution	63 007	–	–	–	–	–	–	–	–	–	–	–	–	215	72	2	–	–	–
Stack deletion	623	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
Graph mutation	184 236	–	–	–	–	–	–	–	–	–	–	–	–	217	72	2	5	26	1

		$\mathcal{A}_7$ ($\lvert P \rvert = 78, N_{fault} = 2$)			$\mathcal{A}_8$ ($\lvert P \rvert = 85, N_{fault} = 4$)			$\mathcal{A}_9$ ($\lvert P \rvert = 73, N_{fault} = 7$)		
cdrc	79	–	–	–	3	4	–	2	9	1, 4
PEC	141	–	–	–	4	4	–	5	12	2, 3, 12
Rule mutation	9 414	14	40	1, 6	206	57	4, 12, 14	14	21	3, 6, 8, 14
Edge insertion	68 387	–	–	–	591	66	2, 25, 37, 63	62	49	2, 7, 14, 26, 31, 32, 44
Edge substitution	66 507	–	–	–	72	34	2, 6, 27	4	23	3, 8, 11, 20
Edge deletion	1 502	–	–	–	14	16	6, 11, 33	1	13	4, 6, 10
Stack insertion	49 114	294	77	4, 8	1467	66	3, 37, 41, 63	60	49	2, 6, 8, 19, 27, 30, 35
Stack substitution	63 007	294	77	1, 4	1454	64	2, 27, 44	23	23	3, 8, 9, 12
Stack deletion	623	–	–	–	8	28	3.5, 27	1	13	4, 7, 8
Graph mutation	184 236	295	77	1, 5	1586	67	2, 17, 41, 47	79	49	2, 8, 10, 12, 19, 30, 35

8.2.3. *Structure of test suites*

Tables 3–5 also give us some indication of the degree of complementarity of PEC and the different versions of the generic coverage algorithm. More specifically, they show the code coverage achieved by and size of a *composite* test suite that contains *all* tests from any PEC test suite variant and any test suite variant generated by the generic cover algorithm instantiated with the corresponding coverage criterion. *PEC exclusive* gives the fraction of the PEC test cases that are produced exclusively by the PEC algorithm.

Obviously, the overlap increases as the generic coverage algorithm produces larger test suites; for example, for AMPL (see Table 3), the fraction of PEC-exclusive tests decreases from 97% to 78% as the generic cover algorithm with shortest yield embeddings goes from *rule* to *deriv*. However, for the more complex grammars the overlaps remain generally small and between 87% and almost 100% of the tests generated by PEC do not occur in the other coverage criteria's test suites.

We can also see that there is generally a larger overlap when the generic cover algorithm is used with shallowest embeddings. For example, for AMPL, the largest overlap is with the $step_4$ test suites. This is likely due to the manner in which the structure of the automaton is covered roughly resembling a shallowest embedding.

Moreover, adding the merged PEC test suite to the merged test suites from the generic cover algorithm further increases their statement coverage. For AMPL, the coverage of the composite test suite increases by 1.0–1.5%-points over that of the smaller test suites (i.e., *pll*, *rule*, and *cdrc*) although the effect is much smaller in conjunction with the larger merged test suites, in particular $step_3$ and *deriv*. For GCCGo, the increases are much greater, and amount to approximately 2.3–12.7%-points. Note that the "reverse" gains over the PEC test suites are much smaller, at less than 0.5%-points for AMPL and less than 3.5%-points for GCCGo (with the biggest gains for the larger test suites). These results show that PEC indeed constructs test suites that are complementary to those of the generic cover algorithm, and that these tests explore SUT behaviors that require "deeper" coverage criteria.

Summary. We can therefore refine our answer to RQ3(b), and conclude that PEC produces test suites that more effectively cover deeply nested grammar structures than the generic cover algorithm with coverage criteria that produce test suites of a similar size.

8.2.4. *Fault detection capabilities*

Table 6 shows for each of the nine faulty AMPL grammars $\mathcal{A}_1$ to $\mathcal{A}_9$ as SUTs how well the test suites constructed by the different algorithms and positive and negative coverage criteria allow us to identify the respective faults. Specifically, it shows the size $|TS|$ of the test suit, and for each of the grammars the number $|P|$ of productions and the number N_{fault} of faulty productions that we manually identified. It then shows for each test suite and grammar the number N_{fail} of failing tests, the number N_{susp} of suspicious (i.e., having a non-zero Ochiai score) productions in the respective grammar, as well as the score-induced ranks of all identified faulty productions. Non-integral ranks indicate ties between multiple productions with the same score. Dashes indicate that the grammar did not fail on any of the tests and thus localization is impossible. The line *graph mutation* denotes the union of all edge and stack mutations.

Positive test suites. For some grammars, both positive test suites are already sufficient to localize all ($\mathcal{A}_3$) or most ($\mathcal{A}_1$, $\mathcal{A}_4$) of the faults with the negative test suites not yielding any test failures, indicating the grammars are likely underapproximations (i.e., too strict). However, in fact only $\mathcal{A}_1$ and $\mathcal{A}_2$ are. $\mathcal{A}_1$ contains errors in the expression structure and does not allow associative applications of the binary operators, e.g., in its faulty rule *term* → *factor mulop factor*,while $\mathcal{A}_2$ contains several errors in different list structures, allowing for example only statement lists with at most two elements. Both $\mathcal{A}_3$ and $\mathcal{A}_4$ are incompatible with the reference grammar, due to simple token mix-ups (e.g., $\mathcal{A}_3$ uses a semicolon in the rule *expr_list* → ; *expr expr_list*instead of the required comma). Note that the rule and edge substitution mutations could in principle expose such errors (as the edge substitution does in a similarly structured case in $\mathcal{A}_6$), but in these cases the overapproximations in the applicability conditions reject the mutations.

Both positive test suite construction algorithms perform similarly and reasonably well. The test suites are compact; as expected, the number of failing tests varies with the number and complexity of the faulty rules. The number of rules identified as suspicious remains low (generally below 30%, and often well below that), with only minor differences between the two algorithms. They both expose 9 of the 17 faults in grammars $\mathcal{A}_1$ to $\mathcal{A}_6$, and the faulty rules are typically ranked highly, often within the top three, although interactions between multiple faults can lead to lower ranks. The unexposed faults in $\mathcal{A}_2$ require more complex tests, due to the convoluted structure of the grammar. Both test suites coincidentally also expose some overapproximation faults in $\mathcal{A}_9$. However, a closer inspection of the spectra shows that this is an artefact of the spectrum collection underlying the fault localization that surfaces for this specific grammar.

Overall, there is little difference between *cdrc* and PEC in the number of exposed faults in underapproximations, the number of suspicious rules, or the fault ranks, and the faults that are exposed are typically ranked highly.

Negative test suites. For the remaining three grammars, positive test suites are insufficient to expose any ($\mathcal{A}_7$, $\mathcal{A}_8$) or all ($\mathcal{A}_9$) of the faults, as the grammars overapproximate the target language (i.e., are too loose). The grammar fragment

$$\begin{aligned}
&source \rightarrow \texttt{program}\ \mathrm{id}\ :\ decls\ \texttt{main}\ :\ body\\
&body\ \ \rightarrow decls\ stmts\\
&decls\ \ \rightarrow funcdecl\ decls \mid vardecl\ decls \mid \epsilon
\end{aligned}$$

from $\mathcal{A}_7$ illustrates this well. The root faults in $\mathcal{A}_7$ are the uses of the conflated *decls* symbol in the *source* and *body* rules, where the reference grammar only allows function and variable declarations, respectively. $\mathcal{A}_7$ will therefore correctly accept *any* positive test generated from the reference grammar, but also incorrectly accept negative tests constructed by for example swapping function and variable declarations. Most errors in $\mathcal{A}_8$ and $\mathcal{A}_9$ are similarly caused by the use of overgeneral non-terminal symbols, although each grammar also contains one faulty rule that leads to incompatibilities with the reference grammar. $\mathcal{A}_9$ in particular collapses the entire expression structure into a single non-terminal $expr$, and so allows for example nested applications of the unary minus operator (e.g., id+−id) or relational expressions in positions where only arithmetic expressions are expected.

Rule vs. Graph Mutations. For $\mathcal{A}_8$, both rule and graph mutations (specifically, stack insertions and substitutions resp. edge and stack insertions) expose all faults. However, for $\mathcal{A}_7$ and $\mathcal{A}_9$, rule mutations fail to generate the sufficiently structurally complex test cases that are required to expose the overgeneralizations in the recursive definitions of the expression structure or in the their faulty uses in statements. More specifically, rule mutations create the right wrong expressions to expose the nested application of the unary minus operator (e.g., id+−id) but do not embed this into all expression occurrences in the grammar, while the graph-based mutations do and so expose all faults. Note, however, that the exposure of the overgeneralizations at the statement level (e.g., the use of relational expressions in positions where only *simple* (arithmetic) expressions are expected) remains co-incidental, since the application conditions still prevent the identification of relational expressions as not matching the required structure of *simple* expressions.

Edge vs. Stack Mutations. For both $\mathcal{A}_8$ and $\mathcal{A}_9$, edge and stack mutations expose the same faults, at very similar ranks. This is hardly surprising as the exposure relies on the insertion of a unary minus, and the reference grammar contains the “unit rule” $addop \rightarrow -$, which allows the stack insertion of *addop* to simulate the edge insertion of − (and vice versa). However, for $\mathcal{A}_7$, a single token operation is not sufficient, and edge mutations fail to expose the two faults, while stack mutations succeed.

For AMPL, insertions expose more errors than substitutions and deletions, both at the edge and stack levels, due to the specific structure of the faults and the reference grammar. However, there are grammar structures where only substitutions will be able to expose an overgeneralization fault. Recall for example the grammar fragment from $\mathcal{A}_7$ above and consider versions in which the reference grammar only allows a single (function resp. variable) declaration, rather than a list and the SUT allows either. Here, any insertion would lead to the expected failure and not expose the fault.

Merging all graph-based mutations leads to a substantially larger test suite, but does not identify any further faults and leads to only marginally better rankings.

Summary. Our experiments with nine faulty student grammars have shown that positive and negative test suites are required to expose under- and overapproximations, respectively. For underapproximations, there is little difference between *cdrc* and PEC in the number of exposed faults, the number of suspicious rules, or the fault ranks, and the faults that are exposed are typically ranked highly. However, both approaches fail to expose all known faults. For overapproximations, LR-mutations outperform rule mutations, and stack mutations outperform edge mutations. In particular, stack insertions expose all overapproximation faults found in the grammars.

8.3. Threats to validity

Our experimental evaluation is subject to several threats to validity. We discuss these and their mitigation in the following.

Internal validity. The validity of our results is primarily threatened by errors in the implementation of our algorithms and by the data collection and analysis process. We mitigated against this threat by using a conceptually clean implementation in a high-level language (i.e., Python), by reusing an open-source implementation of the LR-automaton construction, and by automating the data collection and analysis process. We also used ANTLR parsers generated from the same grammar used by our algorithms to cross-check the validity of the generated test cases.

Internal validity is further threatened by the effects of algorithmic bias, as detailed by Rossouw and Fischer (2021), specifically the choice of equivalent minimal derivations. We followed the approach described there to mitigate against this threat, and randomly resolved these choices over 100 repetitions, and report average results.

External validity. Our experimental results may not generalize to other situations, due to limitations in the experimental set-up.

The primary threat to external validity is, as usual in such experiments, the limited number of grammars and SUTs that we used in the evaluation. We mitigated against this threat by choosing for our coverage evaluation grammars of varying complexity (ranging approximately from 50 to 200 non-terminal symbols and 100 to 600 rules) and SUTs with different characteristics (i.e., frontend code only, one-pass compiler, and multi-pass compiler with a proper phase distinction).

A second threat is the limited comparison to other grammar-based test suite generation algorithms. We were unable to compare experimentally to the WPLR and WNLR algorithms (Zelenov and Zelenova, 2005) because no implementation is publicly available and the publications do not contain enough details to implement the algorithms. Similarly, we are unable to get the implementation of Héam and M’Hemdi’s algorithm (Héam and M’Hemdi, 2015) to work for our grammars.

For the comparison with derivation-based algorithms, we used our own mature implementation of the generic cover framework, which we have also used in several other studies (Raselimo and Fischer, 2019; Raselimo et al., 2019; van Heerden et al., 2020; Raselimo and Fischer, 2021). Other generic implementations may produce different test suites, as may specialized implementations of specific coverage-based algorithms such as the PLL algorithm (Zelenov and Zelenova, 2005). We believe that the differences, if any, are minor because most algorithms (with the exception of Purdom’s algorithm (Purdom, 1972)) minimize test size rather than test suite size, and because the underlying coverage criteria limit the amount of variability. Hence, our experimental design based on randomly resolving choices over 100 repetitions also mitigates against this threat.

9. Related work

9.1. Derivation-based test suite generation algorithms

Most state-of-the-art derivation-based test suite generation algorithms (Lämmel, 2001; Fischer et al., 2011; Havrikov and Zeller, 2019) follow a very similar structure and differ only in the applied coverage criterion. Algorithm 1, which is adapted from the formulation by van Heerden et al. (2020), shows the basic structure of this generic cover algorithm. It first iterates over all symbols $X \in V$ and computes minimal embedding derivations $S \Rightarrow^*_{\leq} \alpha X \omega$ for the given minimal derivation relation. It then computes a set of minimal derivations $C(X)$ for X according to the given coverage criterion C. These minimal derivations are embedded into the embedding of X and then grounded out to form a valid sentence.

Most published algorithms were not originally phrased in this way but we can re-formulate them as a coverage criterion for the generic cover algorithm. For example, Zelenov's PLL algorithm (Zelenov and Zelenova, 2005) is originally phrased in terms of derivation chains but may also be re-formulated as a criterion for the generic cover algorithm (see Algorithm 1, line 16). Note that the different coverage criteria cause the cover algorithm to produce test suites with vastly different characteristics, as can be seen in Section 8 by the different test suite sizes and achieved coverages. However, some derivation-based algorithms do not match the structure of the generic cover algorithm, in particular Purdom's sentence generation algorithm (Purdom, 1972). This algorithm also guarantees rule coverage, but it is fundamentally different in that it minimizes the test suite size rather than the test case sizes.

These approaches differ from the algorithms we propose, in that they construct test suites through derivations over the input grammar, while we instead first construct a LR-graph from the LR-automaton, which in turn is constructed from the input grammar. This means that the key difference between derivation-based algorithms and our algorithm is that the former place emphasis on the input grammar whereas we place emphasis on the recognizer for the input grammar, which acts as an abstract model of the system-under-test.

Grammar-based fuzzing is a popular practical method of testing parsers and other grammarware. It is often based on a stochastic CFG, with probabilities attached to the production rules, and a corresponding semi-random sentence generation process (Lämmel and Schulte, 2006; Guo and Qiu, 2015; Soremekun et al., 2021; Srivastava and Payer, 2021). Fuzzing is often very effective at finding errors. However, this comes at a significant cost since the test suites can become prohibitively large which leads to longer testing times. Fuzzing also does not generally guarantee coverage over a SUT, with current feedback-directed fuzzing methods only showing marginal improvements over traditional grammar-based fuzzing (Atlidakis et al., 2020). Fuzzing is a mostly black-box testing method when used in grammar-based testing, although some work has been done to apply it in a white-box setting with better performance than existing white-box methods (Godefroid et al., 2008).

The methods we propose produce considerably smaller test suites while still achieving good coverage over the systems we tested. If there are no constraints with regards to computational budget or time then fuzzing will outperform our algorithms, and most other systematic grammar-based test suite construction algorithms. However, in most real-world applications, like CI/CD pipelines, fuzzing may not be feasible due to the long testing times.

9.2. Automaton-based test suite generation algorithms

One method for producing a test suite from an automaton is by ensuring that all combinations (p, q), where q is a state reachable from p, are covered, as proposed by Héam and M'Hemdi (2015). This coverage criterion is essentially a superset of our proposed coverage criterion, which translates to covering all combinations (p, q), such that q is reachable and adjacent to p. It will thus result in (likely considerably) larger test suites than those produced by PEC. However, the larger test suites do not exercise any parts of the automaton that is not already exercised by our algorithms, since we cover all relevant stack configurations for all reductions and also cover all shift transitions. Our approach avoids unnecessarily covering transitions multiple times.

Another method, the weak positive LR coverage algorithm (WPLR), proposed by Zelenov and Zelenova (2005), guarantees that all *states* in the automaton are explored. It constructs a test suite such that all combinations of state symbols s_i and transitions from these states labeled with a terminal symbol x are covered, i.e., that the test suite contains a sentence $S \Rightarrow \alpha x \beta$ where α is a prefix that leads to the state s_i when it is processed. However, Zelenov et al. do not provide a clear description how to construct such a test suite. While this coverage criterion guarantees some coverage of the automaton, it is fundamentally different from the one we propose in that it considers only shift transitions and states, whereas our criterion guarantees coverage of all states and transitions. In addition, we cover all reductions in combination with their relevant stack configurations while only covering states and shift transitions can result in many stack configurations for a reduction remaining untested.

9.3. Reachability in state machines

Since automata-based algorithms for test case generation deal with combinations of paths between different states, calculating reachability between these states becomes a significant problem. Pottier (2016) proposed a method for solving this problem for LR(1) parsers. One of the problems solved by Pottier is how to account for reachability when it comes to reduction actions. Pottier's solution computes a *star* rooted at every state s from paths whose source is s and then runs a modified version of Dijkstra's algorithm to determine which states are reachable by a reduction. Our algorithm contains a similar construction. The main structural difference is that reductions are split into one or more pop edges in the LR-graph. This essentially solves the reachability problem of reductions at construction time and encodes the solution as pop edges. At runtime, we simply need a layer-by-layer backwards exploration of push edges to solve for the reduction paths corresponding to a pop edge whenever a reduction is encountered. This allows us to use caching so that the construction time of our LR-graph stays linear in the number of shift transitions in the automaton. It also means we do not need to keep solving the reachability problem at runtime for different contexts.

9.4. Negative test suites

In comparison to positive test suite generation, there is only relatively little work on generating negative tests. Harm and Lämmel (2000) first proposed to use mutation to create negative test cases, but gave no algorithm. Offutt et al. (2006) mutated the grammar rules, but gave no conditions to show that the generated test cases are indeed syntactically invalid, while Köroglu and Wotawa (2019) used a CYK-parser derived from a second grammar as oracle. Zelenov and Zelenova (2005) gave negative coverage criteria for LL and LR parsers. In particular, they also gave an algorithm that derives negative test cases by covering paths in the LR automaton. However, we were unable to re-implement this algorithms from their description, and can thus not compare in detail.

The edge and stack mutations we proposed here have many similarities to rule mutations (Raselimo et al., 2019); specifically, edge resp. stack mutations are very similar to rule mutations on terminal resp. non-terminal symbols. However, the key difference is under which circumstances the mutations are applied. Rule mutations directly mutate the rules of a CFG, but cannot control the context in which the mutated rules are applied. This means that they must be conservative in order to guarantee the resulting test cases are indeed negative. Edge and stack mutations have extra context information that is encoded directly in the LR-automaton and thus LR-graph. They can also take advantage of the explicit encoding of different top-of-stack contexts for reductions via pop edges. This results in a greater number of possible mutation locations in which edge and stack mutations can be applied. This allows a more fine-grained testing of the SUT.

We can see this in our results (see Section 8.2.4) when comparing the fault detecting capabilities of different mutations and the number of test cases that are produced.

10. Conclusions and future work

Grammar-based testing uses context-free grammars as abstract structural models for the inputs to a system under test. Most approaches are derivation-based: they use the grammar rules to construct a set of derivations according to some coverage criterion, generating the actual tests as byproducts. In this paper, we presented a new graph-based approach where test suites are constructed from a set of valid paths that cover all edges in a labeled directed graph corresponding to an LR-automaton that accepts the language of the grammar. The vertices in this LR-graph correspond to states in the LR-automaton; two vertices are connected by an edge iff the top of the LR-automaton's stack can change from one state to the other, either by shifting a terminal or non-terminal symbol (i.e., push edges), or by reducing with a grammar rule (i.e., pop edges). The test suites cover all possible state transitions in the automaton, i.e., all possible pairs of states that can follow each other as the top elements of the stack. This allows us in particular to exploit the different contexts of rule applications, which are encoded in the automaton.

We described two different algorithms that follow this approach. The first algorithm uses two consecutive breadth-first traversals to cover all edges with valid word prefixes and to expand these into complete words in the language; it therefore scales badly, despite our optimization attempts. The second algorithm, which we introduced here, constructs paths bottom-up by combining larger path segments called reduction paths. Both algorithms can successfully generate a test suite, even if the LR-automaton construction leads to shift/reduce or reduce/reduce conflicts but generate only positive tests because they only explore valid paths over LR-graphs. We therefore described how the paths can be mutated so that they represent words that are guaranteed to be negative, i.e., outside the language.

We implemented and evaluated our new algorithm over LR(0)- and LR(1)-graphs constructed from several medium to large grammars and showed that it scales to grammars for real-world systems like SQLite, and that the generated test suites are of comparable quality but smaller than and complementary to those generated by best-performing derivation-based algorithms. We therefore believe that our approach is useful in practice. Our implementation is available at https://github.com/TheLonelyNull/Pytomata. However, our evaluation also indicated scalability problems for the negative test suites, in particular for the larger grammars.

Future work. We see several possible avenues for future work. First, we plan to investigate more complex coverage criteria over the LR-graph, e.g., covering all increasingly longer paths rather than only individual edges. This is similar to the deeper derivation coverage criteria investigated by Havrikov and Zeller (2019) and van Heerden et al. (2020). In the limit, this should subsume the approach by Héam and M'Hemdi (2015) that covers all reachable pairs.

Second, we plan to analyze the relationship between PEC and other automata- and derivation-based test suite generation algorithms. In particular, we are interested in a characterization of those tests that are constructed exclusively by one of the algorithm types.

Third, we plan to develop algorithms that produce better negative test cases. Consider for example the grammar $S \rightarrow abc \mid dac$. We know that, given the word dac, inserting a b between a and c produces a negative test, but the global approximation (and the word mutation algorithm by Raselimo et al., 2019) rule this out, since the first rule produces witnesses for (a, b) and (b, c). However, this witness is not reachable from the mutation location because the paths from the common dominator (here q_{init}) have disjoint first sets. We conjecture that this idea can be generalized to design more fine-grained approximations than the global approximation described in Section 6. Finally, we plan to address the explosive growth of the negative test suites, e.g., using selection strategies that limit the number of mutations applied at each edge without compromising the coverage induced over the SUTs.

CRediT authorship contribution statement

Christoff Rossouw: Conceptualization, Investigation, Methodology, Software, Validation, Writing – original draft, Writing – review & editing. **Bernd Fischer:** Conceptualization, Funding acquisition, Investigation, Methodology, Supervision, Writing – original draft, Writing – review & editing.

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

Dylan Callaghan and Moeketsi Raselimo provided feedback on an earlier draft of this paper. Moeketsi also helped with the execution of the fault localization experiments described in Section 8.2.4.

This work was partially supported by the National Research Foundation of South Africa (NRF), grant numbers 113364 and SRUG2204234463, and a PSP postgraduate bursary from Stellenbosch University.

References

Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D., 2007. Compilers: Principles, Techniques, & Tools, second ed. Addison-Wesley.

ANTLR, 2021. ANTLR grammar repo. https://github.com/antlr/grammars-v4. [Online; accessed 20-February-2023].

Atlidakis, V., Geambasu, R., Godefroid, P., Polishchuk, M., Ray, B., 2020. Pythia: Grammar-based fuzzing of REST APIs with coverage-guided feedback and learning-based mutations. CoRR abs/2005.11498. arXiv:2005.11498. URL: https://arxiv.org/abs/2005.11498.

DeRemer, F., 1971. Simple LR(k) grammars. Commun. ACM 14 (7), 453–460. http://dx.doi.org/10.1145/362619.362625.

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.

GCCGo, 2021. GCCGo. https://go.dev/. [Online; accessed 20-February-2023].

Godefroid, P., Kiezun, A., Levin, M.Y., 2008. Grammar-based whitebox fuzzing. In: Gupta, R., Amarasinghe, S.P. (Eds.), Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008. ACM, pp. 206–215. http://dx.doi.org/10.1145/1375581.1375607.

Graham-Cumming, J., 2017. Incident report on memory leak caused by cloudflare parser bug. https://blog.cloudflare.com/incident-report-on-memory-leak-caused-by-cloudflare-parser-bug/. [Online; accessed 20-February-2023].

Guo, H., Qiu, Z., 2015. A dynamic stochastic model for automatic grammar-based test generation. Softw. Pract. Exp. 45 (11), 1519–1547. http://dx.doi.org/10.1002/spe.2278.

Harm, J., Lämmel, R., 2000. Two-dimensional approximation coverage. Informatica (Slovenia) 24 (3).

Harris, L.A., 1987. SLR(1) and LALR(1) parsing for unrestricted grammars. Acta Inform. 24 (2), 191–209. http://dx.doi.org/10.1007/BF00264364.

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.

Héam, P., M'Hemdi, H., 2015. Covering both stack and states while testing push-down systems. In: Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015. IEEE Computer Society, pp. 1–7. http://dx.doi.org/10.1109/ICSTW.2015.7107406.

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.

Johnson, S., 1975. Yacc. http://dinosaur.compilertools.net/yacc/. [Online; accessed 20-February-2023].

Klint, P., Lämmel, R., Verhoef, C., 2005. Toward an engineering discipline for grammarware. ACM Trans. Softw. Eng. Methodol. 14 (3), 331–380. http://dx.doi.org/10.1145/1072997.1073000.

Köroglu, Y., Wotawa, F., 2019. Fully automated compiler testing of a reasoning engine via mutated grammar fuzzing. In: Choi, B., Escalona, M.J., Herzig, K. (Eds.), Proceedings of the 14th International Workshop on Automation of Software Test, AST@ICSE 2019, May 27, 2019, Montreal, QC, Canada. IEEE / ACM, pp. 28–34. http://dx.doi.org/10.1109/AST.2019.00010.

Lämmel, R., 2001. Grammar testing. In: Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings. 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.

Livinskii, V., Babokin, D., Regehr, J., 2020. Random testing for C and C++ compilers with YARPGen. Proc. ACM Program. Lang. 4 (OOPSLA), 196:1–196:25. http://dx.doi.org/10.1145/3428264.

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

Offutt, J., Ammann, P., Liu, L., 2006. Mutation testing implements grammar-based testing. In: 2nd Workshop on Mutation Analysis (ISSRE Workshops), 2006. IEEE Computer Society, p. 12.

Pottier, F., 2016. Reachability and error diagnosis in LR(1) parsers. In: Zaks, A., Hermenegildo, M.V. (Eds.), Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016. ACM, pp. 88–98. http://dx.doi.org/10.1145/2892208.2892224.

Purdom, P., 1972. A sentence generator for testing parsers. BIT Numer. Math. 12, 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.

Rossouw, C., Fischer, B., 2020. Test case generation from context-free grammars using generalized traversal of LR-automata. 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. 133–139. http://dx.doi.org/10.1145/3426425.3426938.

Rossouw, C., Fischer, B., 2021. Vision: bias in systematic grammar-based test suite construction algorithms. 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. 143–149. http://dx.doi.org/10.1145/3486608.3486902.

Smith, P., 2020. Hyacc. http://hyacc.sourceforge.net/. [Online; accessed 20-February-2023].

Soremekun, E.O., Pavese, E., Havrikov, N., Grunske, L., Zeller, A., 2021. Probabilistic grammar-based test generation. In: Koziolek, A., Schaefer, I., Seidl, C. (Eds.), Software Engineering 2021, Fachtagung des GI-Fachbereichs Softwaretechnik, 22.-26. Februar 2021, Braunschweig/Virtuell. In: LNI, vol. P-310, Gesellschaft für Informatik e.V., pp. 97–98. http://dx.doi.org/10.18420/SE2021_36.

SQLite, 2021a. Lemon. https://www.sqlite.org/lemon. [Online; accessed 10-October-2023].

SQLite, 2021b. Sqlite. https://www.sqlite.org/. [Online; accessed 20-February-2023].

Srivastava, P., Payer, M., 2021. Gramatron: Effective grammar-aware fuzzing. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. In: ISSTA 2021, Association for Computing Machinery, New York, NY, USA, pp. 244–256. http://dx.doi.org/10.1145/3460319.3464814.

Tomita, M., 1985. An efficient context-free parsing algorithm for natural languages. In: Joshi, A.K. (Ed.), Proceedings of the 9th International Joint Conference on Artificial Intelligence. Los Angeles, CA, USA, August 1985. Morgan Kaufmann, pp. 756–764, URL: http://ijcai.org/Proceedings/85-2/Papers/014.pdf.

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.

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

Christoff Rossouw received his M.Sc. in Computer Science from Stellenbosch University in 2022. His thesis research focused on grammar-based test suite construction using coverage-directed algorithms over LR-automata. He remains affiliated to Stellenbosch University but currently works in industry.

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.