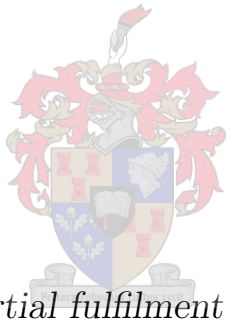
Coverage Directed Algorithms for Test Suite Construction From LR-Automata

by

C. J. Rossouw



Thesis presented in partial fulfilment of the requirements for the degree of Master of Science in Computer Science in the Faculty of Science at Stellenbosch University

Supervisor: Prof. Bernd Fischer

April 2022

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: April 2022 .....

Abstract

Coverage Directed Algorithms for Test Suite Construction From LR-Automata

Christoffel Jacobus Rossouw

Division of Computer Science,
Department of Mathematical Sciences,
University of Stellenbosch,
Private Bag X1, 7602 Matieland, South Africa.

Thesis: MSc Computer Science

April 2022

Bugs in software can have disastrous results in terms of both economic cost and human lives. Parsers can have bugs, like any other type of software, and must therefore be thoroughly tested in order to ensure that a parser recognizes its intended language accurately. However, parsers often need to recognize many different variations and combinations of grammar structures which can make it time consuming and difficult to construct test suites by hand. We therefore require automated methods of test suite construction for these systems.

Currently, the majority of test suite construction algorithms focus on the grammar describing the language to be recognized by the parser. In this thesis we take a different approach. We consider the LR-automaton that recognizes the target language and use the context information encoded in the automaton. Specifically, we define a new class of algorithm and coverage criteria over a variant of the LR-automaton that we define, called an LR-graph. We define methods of constructing positive test suites, using paths over this LR-graph, as well as mutations on valid paths to construct negative test suites.

We evaluate the performance of our new algorithms against other state-of-the-art algorithms. We do this by comparing coverage achieved over various systems, some smaller systems used in a university level compilers course and

other larger, real-world systems. We find good performance of our algorithms over these systems, when compared to algorithms that produce test suites of equivalent size.

Our evaluation has uncovered a problem in grammar-based testing algorithms that we call bias. Bias can lead to significant variation in coverage achieved over a system, which can in turn lead to a flawed comparison of two algorithms or unrealized performance when a test suite is used in practice. We therefore define bias and measure it for all grammar-based test suite construction algorithms we use in this thesis.

Uittreksel

Coverage Directed Algorithms for Test Suite Construction From LR-Automata

Christoffel Jacobus Rossouw
Verdeling van Rekenaar Wetenskap,
Department Wiskundige Wetenskappe,
Universiteit van Stellenbosch,
Privaatsak X1, 7602 Matieland, Suid Afrika

Tesis: MSc Rekernaar Wetenskap
April 2022

Foute in sagteware kan rampspoedige resultate hê in terme van beide ekonomiese koste en menselewens. Ontleders kan foute hê soos enige ander tipe sagteware en moet daarom deeglik getoets word om te verseker dat 'n ontleder sy beoogde taal akkuraat herken. Ontleders moet egter dikwels baie verskillende variasies en kombinasies van grammatikastrukture herken wat dit tydrowend en moeilik kan maak om toetsreekse met die hand te bou. Ons benodig dus outomatiese metodes van toetsreeks-konstruksie vir hierdie stelsels.

Tans fokus die meeste toetsreeks-konstruksiealgoritmes op die grammatika wat die taal beskryf wat deur die ontleder herken moet word. In hierdie tesis volg ons 'n ander benadering. Ons beskou die LR-outomaat wat die teikentaal herken en gebruik die konteksinligting wat in die outomaat geënkodeer is. Spesifiek, ons definieer 'n nuwe klas algoritme en dekkingskriteria oor 'n variant van die LR-outomaat wat ons definieer, wat 'n LR-grafiek genoem word. Ons definieer metodes om positiewe toetsreekse te konstrueer deur paaie oor hierdie LR-grafiek te gebruik, asook mutasies op geldige paaie om negatiewe toetsreekse te konstrueer.

Ons evalueer die werkverrigting van ons nuwe algoritmes teenoor ander moderne algoritmes. Ons doen dit deur dekking wat oor verskeie stelsels behaal is, te vergelyk, sommige kleiner stelsels wat in 'n samestellerskursus

op universiteitsvlak en ander groter werklike stelsels gebruik word. Ons vind goeie werkverrigting van ons algoritmes oor hierdie stelsels, in vergelyking met algoritmes wat toetsreekse van ekwivalente grootte produseer.

Ons evaluering het 'n probleem in grammatika-gebaseerde toetsalgoritmes ontdek wat ons vooroordeel noem. Vooroordeel kan lei tot aansienlike variasie in dekking wat oor 'n stelsel behaal word, wat weer kan lei tot 'n gebrekkige vergelyking van twee algoritmes of ongerealiseerde prestasie wanneer 'n toetsreeks in die praktyk gebruik word. Ons definieer dus vooroordeel en meet dit vir alle grammatika-gebaseerde toetsreeks-konstruksiealgoritmes wat ons in hierdie tesis gebruik.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Bugs in software can have catastrophic effects, both in terms of cost and potentially human lives. For example, the loss of the Mars Climate Orbiter in 1998 resulted in a loss of a spacecraft worth $125 million [2]. More recently, a bug in the flight control system for the Boeing 737-Max resulted in the deaths of 346 people [3]. Parsers are not generally classed as "mission critical" software systems since they are mostly associated with compilers. However, bugs in parsers can also have very severe consequences, since parsers are not only used in compilers, but many other systems attempting to recognize structured input.

For example, in 2017 Cloudflare, one of the world's biggest cloud network solution providers, discovered a vulnerability where private information was leaked through corrupt HTML pages that were produced by their HTTP rewrite service [20], which parsed and modified their customers' web pages on the fly. This leaked information included a private key of one of Cloudflare's internal servers. The leak was caused by a hand-written parser that did not reflect the intended HTML grammar rule correctly. Below we can see the grammar rule that shows how the HTML elements were *intended* to be parsed.

$$\textit{htmlElement} \rightarrow \texttt{<}\ \textit{htmlAttribute}*\ (\texttt{>}\ (\textit{htmlContent}\ \texttt{<}\ \texttt{/}\ \textit{tagName}\ \texttt{>})? \mid \texttt{/>});$$

However, the bug was in allowing malformed HTML elements like `<script type=` at the end of an HTML page. This led to a buffer overflow which in turn led to data outside of a buffer, used in the on-the-fly page modification, being written into the resulting HTML page.

Testing parsers to minimize the bugs they contain is thus crucial. Our goal is to construct a test suite that provides assurance that a parser will accept all valid words in the (context-free) language it is intended to recognize

and reject all invalid words that are not in the language. However, in large, real-world systems it is often unfeasible to manually construct test suites due to the scale of these grammars. Therefore, methods for automated test suite construction are very important.

Currently, the majority of approaches for generating these test suites do so directly from the input grammar, and construct derivations (and thus tests) that satisfy a variety of coverage criteria. In this thesis we propose new algorithms that work on a *recognizer* corresponding to the input grammar. Specifically, we look at a graph corresponding to this recognizer and formulate coverage as paths over this graph. This allows us to explore the extra context information encoded into the structure of the recognizer. We also describe methods of mutating the paths that produce positive test cases such that they produce guaranteed negative test cases which should be rejected by the system under test. For example, the Cloudflare bug we discussed before could have been caught by a test suite which covers the *htmlElement* rule at the end of an HTML page and uses a deletion mutation to delete the closing >. Such a test can be generated by the methods described in this thesis.

1.1 Grammar-Based Testing

The goal of software testing is to detect bugs in a software system under test (SUT) to give developers a certain level of confidence in the correctness of their software. Grammar-based testing is no different.

In general software testing, a test suite is comprised of a set of individual test cases which include test input data x and an expected output y. The SUT U is executed over x and its output $U(x)$ is then compared to y. This then gives a *pass* or *fail* verdict.

For grammar-based testing, the test input is a word x and the expected output is either *accept* or *reject*. x can either be a positive test case, where the output is *accept* if $x \in L(G)$ for the language L described by the grammar G, or a negative test case, where the output is *accept* if $x \notin L(G)$.

The goal of grammar-based test suite construction is therefore to produce a suite of tests that exercise an arbitrary SUT U which processes inputs described by a grammar G. It is common for G to be the grammar for a programming language and U to be a parser or compiler. However, G may be the grammar for any input language and U can be any system that contains an input reader for the language described by G. U does not even necessarily have to "read" the inputs: G may also describe the user interaction with the user interface of U or the calling conventions of an API for U. G also does not have to be the exact grammar for U, it may be an approximation.

The efficacy of a test suite for a software system is often measured as the coverage it achieves over the target software system, while accounting for the test suite's size. In white-box testing, test cases are constructed to exercise specific execution paths of the SUT. This method of testing can be very challenging for testing parsers, especially in an automated way. Therefore, grammar-based testing resorts to a form of black box testing, specifically partition testing [38], using the grammar as a specification. Equivalence classes can be defined in terms of the grammar. For example, for the rule coverage criterion, two test cases can be considered to form part of the same equivalence class if they exercise the same grammar rule. Most other grammar-based testing algorithms also define equivalence classes directly from the input grammar. The algorithms we propose have their coverage criterion, and hence their equivalence classes, entirely formulated in terms of paths over an LR-automaton for the input grammar. This is the main difference between the algorithms proposed in this thesis and the current state of the art algorithms [24, 29].

1.2 Problem Statement

In this thesis we solve the problem of generating test cases for a SUT from an input grammar from a different perspective. Instead of directly generating our test cases from the grammar, we construct our test cases by considering coverage over a recognizer for the grammar, specifically an LR-automaton. We intend to cover all combinations of transitions and the top-of-stack contexts which they may be exercised in. We do not define any aspect of the algorithms in terms of the input grammar but wholly focus on the structure of the automaton corresponding to the input grammar. In doing this we essentially produce tests by using a different model for a recognizer for the target language, the LR-automaton, whereas the majority of the current state-of-the-art methods model this system by using a context-free grammar. We solve this problem while keeping in mind the following constraints.

Better coverage over a system under test One of the most common ways of quantifying whether a test suite construction algorithm produces effective test suites is by considering the coverage such test suites achieve over a system under test. However, most current state-of-the-art grammar-based test suite construction algorithms [24, 29, 47] do not consider the structure of the system for which the test suite is produced, only the structure of the input grammar. They are also not feedback-directed, and so do not take the system under test into account. As such, it is difficult to make assertions

regarding which structures in the system under test will be exercised by the produced test suite.

Generally, larger test suites produce better coverage over a system under test. However, since the solution space for possible test cases for most real-world grammars is infinite and the time available to test a system is not, the challenge lies in producing effective, small test suites that achieve sufficient coverage.

Fair comparison of different algorithms and coverage criteria It is difficult to compare, in an unbiased way, algorithms that have a very different structure in how they produce test cases. There are many choices that an algorithm makes that may be considered equivalent within the context of the input grammar. However, these seemingly equivalent choices often result in certain parts of the SUT being exercised more than others. This can lead to a biased comparison of two different algorithms where an algorithm that would generally perform worse appearing as if it performs better than an algorithm that generally performs considerably better.

Guaranteeing negative test cases The primary method for producing negative test cases [35,47] is by mutating positive test cases to transform them into invalid words. However, this should ideally be done in a way such that no further checks, after generating the test suite, are necessary to guarantee that the test cases in the test suite are indeed negative. Therefore, large scale mutations are very challenging for purely grammar-based methods since they do not have all the information readily available to be able to assert that a mutation is valid in many possible mutation locations.

Concise test cases The usefulness of a test case that reveals an error in the SUT is largely determined by its ability to help determine the location of the error. This depends greatly on the size of a test case and the number of grammar structures that are simultaneously covered. If a test case helps reveal an error in the SUT but covers many grammar rules it may be difficult to locate the actual source of the error. Therefore, constructing test cases that are concise, and test minimal parts of the system at a time, is very important.

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

Figure 1.1: Example grammar G_{toy} (adapted from [35])

1.3 Bias in Grammar-Based Test Suite Construction

The different grammar-based test suite construction algorithms are based on different criteria, which obviously impact the results. For example, Purdom's algorithm [30,34] constructs a minimal set of sentences such that each production is used at least once in a derivation, while the PLL coverage algorithm [47] constructs the set of shortest sentences with $S \Rightarrow^* \alpha A \omega \Rightarrow^* \alpha a \beta \omega \Rightarrow^* w$ for each non-terminal A and each terminal $a \in \text{first}(A)$.

These algorithms have several degrees of freedom and allow some implementation variants, which can impact their results in subtle and often unexpected ways. For example, Purdom's algorithm can use any rule $A \rightarrow \alpha$ still unused when A is expanded. Likewise, the PLL coverage algorithm can use any derivation $A \Rightarrow^* a\alpha \Rightarrow^* av$ leading to a shortest yield av for A.

However, any implementation of a grammar-based test suite construction algorithm must ultimately resolve these choices in one way or another—typically deterministically, in some seemingly arbitrary but fixed manner (e.g., selecting symbols in alphabetic order or rules in textual order). Unfortunately, the resolution of these choices introduces *biases* into the generated test suites. Consider for example the grammar in Figure 1.1. This allows two different derivations of a shortest yield for *stmt* (*stmt* ⇒ `sleep` and *stmt* ⇒ `return` *expr*? ⇒ `return`, respectively), and likewise for *expr* and *type*. Figure 1.2 shows on the left the test suite generated for this grammar by the generic cover algorithm shown in Algorithm 3, with rule coverage as criterion, textual rule order as strategy to choose between rules leading to different shortest yields, and minimal height derivation trees for embeddings. We can see that the selected shortest yield for *stmt* leads to a over-representation or *bias* in favor of `sleep`-statements and an underrepresentation or bias against `return`-statements: `sleep` occurs in eight of the fifteen generated tests, while

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

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

Figure 1.2: Test suite for G_{toy} satisfying *rule*-coverage. (left) Shallowest embeddings (right) Shortest yield embeddings.

`return` occurs only in the two tests that explicitly cover the corresponding rules.

These different implementation choices matter, since they can result in substantially different test suites for the same grammar and the same criterion, which can in turn result in substantially different coverage of the SUT. For example, for a SQL grammar with 586 rules the generic cover algorithm produces different test suites satisfying rule coverage that achieve between 22.1% and 26.2% statement coverage over the SQLite 3.36.0 system, i.e., an 18% difference between worst and best case (see Table 5.2 for details). Moreover, rule coverage leads in many cases to a better coverage than more comprehensive criteria such as context-dependent rule coverage (CDRC [29]).

The introduced biases therefore constitute a threat to validity for any experiments that compare different grammar-based testing algorithms based on coverage over a SUT, and make replication of experiments harder.

1.4 Research Objectives

The goal of this work is to develop a class of grammar-based test suite construction algorithms that take a different approach from the current state-of-the-art by exploring paths in the LR-automaton corresponding to a context-free grammar. We also propose methods of producing negative test suites by mutations of these paths and the stack of the automaton. We measure the performance of these algorithms and mutations and compare them to the current state of the art, in terms of coverage achieved over the system-under-test, errors revealed and the size of the test suite.

Specifically, we answer the following research questions through our evaluation:

1. How does the coverage achieved by the automaton-based algorithms and the size of the test suites produced by them compare to the current state-of-the-art algorithms?

2. Are there any structural differences in test cases produced by the automaton-based algorithms compared to similar sized test suites by the current state-of-the-art algorithms?

3. How well do the edge and stack mutations reveal errors in a system under test compared to a positive test suite or test suite produced by current rule mutation algorithms?

4. What is the effect of bias on test suites generated by various different coverage criteria and algorithms?

1.5 Overview of Approach

Before we define our coverage criterion and algorithms we first define the concept of an LR-graph. This is a graph construct, based on an LR-automaton, that enables us to use graph theory based ideas like paths and edges. This is especially important when it comes to dealing with problems like reachability of reductions in an LR-automaton and shift/reduce and reduce/reduce conflicts. Our LR-graph enables us to transform the automaton into a graph with push and pop edges, corresponding to the stack actions associated with transitions in the automaton (see Section 3.1).

We then describe the criterion of coverage over of all edges of the newly introduced LR-graph. We describe a simple, general traversal algorithm for achieving this coverage, based on breadth-first search (see Section 3.2). This allows for the cyclic nature of valid paths over the LR-graph and avoids issues related to infinite loops in order to guarantee termination in a finite amount of time.

From this first algorithm we then propose a more efficient algorithm that scales to very large real world grammars, yet still maintains the original coverage criterion of the first algorithm (see Section 3.3). It does this by focusing on covering all reductions in all possible top-of-stack contexts, i.e., all pop edges, which in turn covers all transitions in the underlying automaton with a considerably smaller test suite size than the first algorithm.

We also formulate negative mutations based on the LR-graph and paths constructed by the positive test suite construction algorithms (see Section

3.4). These mutations can be edge-based, for example, inserting or deleting an edge which produces a guaranteed invalid path over the LR-graph, or they can be stack-based, mutating entire sub-paths, which we call reduction paths. Edge-based mutations are similar to token-based mutations found in grammar-based algorithms, although we generally have more mutation locations in the LR-graph than grammar-based algorithms (see Table 5.5 for details) have in a grammar. Stack-based mutations are more complex than edge-based mutations in that these are large scale mutations that mutate large sections of the original positive test case. This can help reveal errors in the underlying system that are caused by a combination of interconnected errors.

We formalize the concept of bias in grammar-based testing as a severe threat to validity in evaluations that consider coverage achieved by a test suite over a SUT as well as confidence in a coverage criterion when testing real world systems (see Section 5.1). In the context of grammar-based testing we refer to bias as an over-representation of specific grammar elements and derivations in a test suite due to equivalent choices made during the test suite generation process. These are choices between grammar elements that are considered equal in the test suite generation process, but often result in very different coverage over the target system.

Finally, we thoroughly evaluate the performance of the proposed algorithms, keeping in mind the concept of bias as discussed before (see Chapter 5). We compare the new algorithms' results over systems that have been used by others in their experiments [35,46], as well as over large real world systems. This allows us to evaluate our new algorithms in a multitude of settings and make stronger assertions about their performance as compared to the current state-of-the-art algorithms.

1.6 Original Contributions

Our main contribution is the development of a new coverage criterion and corresponding algorithms that take a different view of the problem of grammar-based test suite construction and use the extra context information encoded in a LR-automaton to produce effective test suites. Our test suites cover all transitions and top-of-stack contexts in which they can occur. This is a novel contribution and the main problem we solve in this thesis. However, we also make a number of other foundational contributions upon which this work is built.

Path coverage over automaton using graph constructs Reduction transitions pose a significant challenge to defining coverage over an automaton. There are often multiple possible top-of-stack contexts for a reduction. There may also be conflicts due to ambiguities in the input grammar. By introducing pop edges to explicitly solve the reachability problem of reductions and handle all possible choices for resolving conflicts, we are able to easily define paths over the automaton and ensure that crucial contexts in which a reduction could occur are not ignored.

Quantifying bias in grammar-based testing Bias is a hitherto unexplored facet of grammar-based testing. Bias poses a significant threat to validity for any evaluation that uses coverage results achieved by a test suite over a SUT. In practical uses of grammar-based testing it can also lead to unexpected bad performance from a coverage criterion. Quantifying and measuring bias allows for evaluation between our algorithms and other state-of-the-art algorithms to be fair. It enables us to make assertions about their performance in comparison to these other algorithms with a good degree of certainty.

This work has in part been published and presented at international conferences as *Test case generation from context-free grammars using generalized traversal of LR-automata* [39] and *Vision: Bias in Systematic Grammar-Based Test Suite Construction Algorithms* [40].

1.7 Outline

In Chapter 2 we present the theoretical background on grammars, testing, LR-automata and different forms of derivations. Chapter 3 defines and explains the novel construct of an LR-graph and its components. It also covers the two different positive test suite construction algorithms. We then discuss all the different types of negative mutations. Chapter 4 deals with the intricacies of our implementation and the optimization that we made in order to scale our algorithms to large, real world systems.

Chapter 5 deals with the evaluation of all our algorithms and compares it to the state-of-the-art. We also define the concept of bias and illustrate how it occurs in almost all grammar-based test suite construction algorithms. We then show the impact that bias may have on an evaluation and in practice.

In Chapter 6 we discuss and compare the work related to our algorithms. Lastly, in Chapter 7, we conclude this thesis and propose future work and research that may be performed.

Chapter 2

Theoretical Background

This chapter gives an overview of the topics that this work builds on and defines some key concepts used in later chapters. First we start off by defining context-free grammars and the notations we will be using throughout this thesis. We then discuss the concept of derivations, including shortest yield and shallowest embedding that play an important role in our definition of bias. We discuss the topic of parsing as well as the automaton used in parsing a word according to a grammar. Lastly, we also discuss testing and the current state-of-the-art approaches to grammar-based testing.

2.1 Context-Free Grammars and Notation

A context-free grammar (CFG) is a formal replacement system that specifies the replacement of designated individual symbols by arbitrary symbol strings. A CFG can be used to define the syntactic structure of most programming languages in use today. CFGs have a logical recursive structure which allow them to be easily used in parsers and compilers. All languages that can be described by CFGs can be recognized by a push-down automaton, a property we use extensively in this thesis.

Formally, a *context-free grammar* is a four-tuple $G = (N, T, P, S)$ with $N \cap T = \varnothing$, $P \subset N \times (N \cup T)^*$, and $S \in N$, where

1. T is the set of *terminal* symbols. These are the symbols that are used to make up the words in the language defined by the grammar.

2. N is the set of *non-terminal* symbols. Non-terminal symbols themselves represent sets of strings which are defined by the set of productions.

3. P is the set of *productions*. A production is a rule that consists of a non-terminal mapped to sequence of terminals and non-terminals.

4. S is the *start symbol* which is one of the non-terminal symbols.

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 $V = N \cup T$, w, x, y, z for strings of terminals or *words*, $\alpha, \beta, \gamma, \ldots$ for strings of grammar symbols or *phrases*, with $|\,\alpha\,|$ denoting the length of α and ε denoting the empty string.

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

$$
\begin{aligned}
E &\to E \texttt{ + } F \\
&\mid E \texttt{ - } F \\
&\mid F \\
F &\to \texttt{num} \\
&\mid \texttt{id} \\
&\mid \texttt{(}\ E\ \texttt{)}
\end{aligned}
$$

Figure 2.1: Example CFG

In Figure 2.1 we can see a grammar for a simple expression language. In this case the set of non-terminals is $\{E, F\}$ and the set of terminals is $\{$ `+` , `-` , `(` , `)` , `id` , `num` $\}$. The set of productions are the rules shown in Figure 2.1 and the start symbol is E in this case.

2.2 Derivations

We use $\alpha A \beta \Rightarrow \alpha\gamma\beta$ to denote that $\alpha A \beta$ *derives* $\alpha\gamma\beta$ by application of the rule $A \to \gamma \in P$, with $\Rightarrow^k$ denoting its k-fold repetition and $\Rightarrow^*$ its reflexive-transitive closure. For example, for the grammar in Figure 2.1, `(` F `)` derives `(` `id` `)` by application of the rule $F \to$ `id`. A *sentential form* is a phrase α with $S \Rightarrow^* \alpha$, a *sentence* is a word w with $S \Rightarrow^* w$. Extending the previous example, both `(` F `)` and `(` `id` `)` are sentential forms but only `(` `id` `)` is a sentence over the grammar. We also use a *simultaneous derivation* relation $\Rrightarrow$, where $X_1 \ldots X_n \Rrightarrow \gamma_1 \ldots \gamma_n$ if $X_i \to \gamma_i \in P$ for all $X_i \in N$ and $\gamma_i = X_i$ for all $X_i \in T$. If we consider derivations AA for $A \to \gamma$ we can see that it will take two repetitions to derive $\gamma\gamma$, however it only takes one repetition of simultaneous derivation to derive the same sentence. In this case we can see that $\Rrightarrow^1$ is contained in $\Rightarrow^2$.

The *yield* of a phrase α is the set of all words that can be derived from it, i.e., $\text{yield}(\alpha) = \{w \in T^* \mid \alpha \Rightarrow^* w\}$. The *language* $L(G)$ *generated by a grammar* G is the yield of its start symbol, i.e., $L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$. We generalize the usual definition of the first set to $\text{first}(X) = \{Y \in V \mid X \Rightarrow^* Y\alpha\}$ and last set to $\text{last}(X) = \{Y \in V \mid X \Rightarrow^* \alpha Y\}$, i.e., allow non-terminal symbols.

An *embedding of* X *in* A is a phrase $\alpha X \omega$ with $A \Rightarrow^* \alpha X \omega \Rightarrow^* w$. It is called a *shortest yield embedding* if for all $A \Rightarrow^* \beta X \gamma \Rightarrow^* v$ we have $|w| \leq |v|$. It is called 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 S and a leaf X (i.e., a terminal or un-expanded non-terminal).

A derivation tree, is a method of representing derivations as an ordered tree, with nodes being labeled with non-terminal and terminal symbols. If all leaf nodes are labeled with terminal symbols it may also be called a parse tree. Otherwise it may be called a partial parse tree. We can determine the derivation a tree represents by reading off the leaf nodes in order, from left to right.

The *height* of a symbol X is the smallest height of a derivation tree rooted in X whose leafs are all terminal symbols. 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.

$$
\begin{array}{lll}
\textit{stmt} & \rightarrow & \texttt{sleep} \\
 & \mid & \texttt{return}\ \textit{expr\_optional} \\
\textit{expr\_optional} & \rightarrow & \textit{expr} \\
 & \mid & \epsilon \\
\textit{expr} & \rightarrow & \textit{expr}\ \texttt{==}\ \textit{expr} \\
 & \mid & \textit{expr}\ \texttt{+}\ \textit{expr} \\
 & \mid & \texttt{id} \\
 & \mid & \texttt{num}
\end{array}
$$

Figure 2.2: Example CFG with different options for shortest yield and shallowest embeddings, respectively

If we consider the two derivation trees in Figure 2.3 we can see that, in terms of shortest yield, we have two options of length one for *stmt*. However, this is not the case for shallowest derivations since we only have one option with minimal height of one. This shows that, although there are often significant

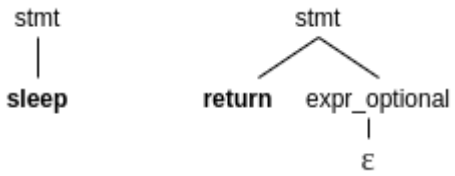


Figure 2.3: Derivation trees for grammar in Figure 2.2

overlaps between the sets of shortest yields and shallowest derivations for a symbol, these sets can be quite different, depending on the structure of the grammar.

2.3 LR-Parsing and Push-Down Automata

When parsing input according to a context-free grammar, there are two main methods that may be used, *top-down* parsing and *bottom-up* parsing. The key difference between the two methods is that top-down parsing attempts to construct a preorder parse tree from the root node, i.e., top down, for a given input while bottom-up parsing attempts to reduce an input from the leaves of the parse tree to the root node, i.e., bottom up. We focus on the latter in this thesis.

2.3.1 Push-down automata

A push-down automaton (PDA) is a finite state machine that uses a stack, with all stack operations being performed on the element at the top of the stack in the standard way, and can be used in bottom-up parsing. The language accepted by a PDA is defined as the set of all words that are accepted by the PDA. The class of languages that PDAs can accept are equivalent to the class of languages that may be described by a CFG.

We say a word is *accepted* by a PDA if, after processing the entire word, the current state is in the set of accepting states or the stack is empty. It can be shown that, for any language accepted by a PDA by using the empty stack, we can construct an equivalent automaton that accepts the same language by a final state [41]. A word is rejected if, during the parsing process, there is no valid transition for the next input symbol or, after processing the entire word, the current state is not in the set of accepting states and the stack is not empty.

Formally a push-down automaton is defined as a 7-tuple $P = (Q, \zeta, \rho, \delta, q_0, Z_0, F)$ [26], where

1. Q is a finite set of *states*.
2. ζ is a finite set of symbols composing the *input alphabet*.
3. ρ is the *stack alphabet*, the collection of symbols that are allowed on the stack.
4. δ is the *transition function*. It takes a triple (q, a, X), comprising a state q, an input symbol $a \in \zeta$ or ϵ and a stack symbol $X \in \rho$, and returns a finite set of pairs (p, γ) where $p \in Q$ is the destination state and $\gamma \in \rho$ is the string of stack symbols that replace $X \in \rho$ on top of the stack.
5. $q_0 \in Q$ is the *start state* of the automaton.
6. $Z_0 \in \rho$ is the *start symbol*. This symbol is on the stack before any transition is made.
7. $F \subseteq Q$ is the set of *accepting states*.

A PDA may be either deterministic or non-deterministic. If there exists at most one valid transition, given q, a and X, as defined above, we say the PDA is deterministic. If there exists more than one valid transition, the PDA is non-deterministic. A non-deterministic PDA can recognize the entire class of context-free languages, while a deterministic PDA can only recognize a subset of this, the class of deterministic context-free languages.

2.3.2 LR-Automata

An LR-automaton is a specific type of deterministic PDA that is constructed from a CFG. One of the main differences between the LR-automaton and general PDAs is that one transition in the LR-automaton may result in many elements being popped off the stack, as compared to popping at most one element per transition in a general PDA.

There are multiple different variants of this type of automaton (LR(k), SLR, LALR, GLR etc. [14, 23, 45]), mostly dealing with different methods of resolving conflict via a lookahead or a breadth-first search. In this thesis we focus on generating tests from the LR(0)-automaton, which does not include a lookahead.

LR-automata are commonly used to perform a type of bottom-up parsing, known as *shift-reduce* parsing, which utilizes a stack along with shift and reduce transitions. A shift transition simply pushes input symbols onto the stack. A reduction on the other hand pops a certain number of elements from

the top of the stack and pushes an appropriate non-terminal symbol. This continues until either only the start symbol is left on the stack or the input is rejected. This can be seen in detail in Algorithm 2.

A state in a LR(0)-automaton can be viewed as a set of LR(0)-items. In this context, an *item* is simply a grammar rule with a *dot* indicating how much of the rule we have seen at this point. For example, the item *stmt* → `sleep` • indicates that, by parsing some input, we have seen this entire rule and may reduce the top of the stack to *stmt*. However, the item *expr* → *expr* • `+` *expr* indicates that we should expect to possibly see the token `+` next.

In order to construct the item sets for a LR(0)-automaton for a grammar G we first need to define an augmented grammar G'. This is done by introducing a new start symbol S' and a new production $S' \rightarrow S$ for the original start symbol S. This allows for the parser to know when to accept input, since this would occur if the parser is about to perform the reduction for $S' \rightarrow S$.

Closure of item sets The closure of an item set, i is defined as the set of items that may be constructed from a state, given the following two rules [7]:

1. Add every item in i to the closure of i.
2. If $A \rightarrow \alpha \bullet B\ \beta$ is in the closure of i and there exists a production $B \rightarrow \gamma$ then add $B \rightarrow \bullet\ \gamma$ to the closure of i, if it does not already exist. Do this until no new items can be added to the closure of i.

We can divide items into two classes, kernel items and nonkernel items. Kernel items are the initial item $S' \rightarrow \bullet\ S$ and all other items who do not have a dot on the left end of a production rule. Nonkernel items are those items with a dot on the left end of a production, excluding the initial item.

A goto function may be used to define transitions from one state to another in terms of an item set i and a grammar symbol X. It is defined as the closure of the set of items of the form $A \rightarrow \alpha\ X \bullet \beta$ for all $A \rightarrow \alpha \bullet X\ \beta$ in the set i. This gives the transitions of the LR(0)-automaton, since the item sets correspond to states in the automaton, goto specifies the transition from one state to another, given an input X. We can see the algorithm for computing the collection of item sets for the LR(0)-automaton in this way in Figure 1. In Table 2.1 we can see the LR(0) item sets calculated for the grammar in Figure 2.1

The primary functions in the LR-parsing algorithm are ACTION and GOTO (This is not the same as the goto used in computing the sets of the LR(0)-automaton). They are defined as follows:

Algorithm 1: Computing an LR(0)-automaton sets (adapted from [7])

input : Items of the grammar G
output : LR(0)-automaton

```
1  C = {closure({S' → •S})}
2  while new items to be added to C do
3  |  for I ∈ C do
4  |  |  for X ∈ T ∪ N do
5  |  |  |  if goto(I, X) ≠ ∅ and goto(I, X) ∉ C then
6  |  |  |  |  C = C ∪ {goto(I, X)}
7  |  |  |  end
8  |  |  end
9  |  end
10 end
```

ACTION The ACTION function takes a state i and a terminal a as input. It can then return one of four values. It can shift a state j onto the stack to represent a. It can also reduce $A \to \beta$ which reduces a substring β to A by popping $|\beta|$ elements off the stack and pushing A. Lastly, ACTION can accept the input and finish parsing or report an error.

GOTO The GOTO function takes as input a state i and a non-terminal A and maps this pair to a corresponding state j. This is the pushing component of a reduction (see line 10, Algorithm 2) and must happen after symbols have been popped off the stack.

We can see an example of an LR(0)-automaton in Figure 2.5 with its parsing table in Figure 2.4. The solid lines represent shift transitions and the dotted lines the reduction transitions. All terminal labeled solid lines correspond to line 5 of Algorithm 2 and solid lines labeled with non-terminal correspond to line 10. The dotted lines with square boxes are reductions actions (see line 7) and result in the popping of elements of the stack (see line 8).

Algorithm 2: LR-Parsing Algorithm (adapted from [7])

input : An input string w and an LR-parsing table with functions ACTION and GOTO for a grammar G
output : If w is in $L(G)$, the reduction steps of a bottom-up parse for w; otherwise an error indication

```
let a be the first symbol of w
while true do
    let s be the state on top of the stack
    if ACTION[s, a] = shift t then
        push t onto the stack
        let a be the next input symbol
    else if ACTION[s, a] = reduce A → β then
        pop |β| symbols off the stack
        let state t now be on top of the stack
        push GOTO[t, A] onto the stack
        output the production A → β
    else if ACTION[s, a] = accept then
        break
    else
        call error-recovery routine
    end
end
```

State	Item set
0	{S' → • E $, E → • E + F, E → • E - F, E → • F, F → • num, F → • id, F → • (E)}
1	{S' → E • $, E → E • + F, E → E • - F}
2	{E → F • }
3	{F → num • }
4	{F → id • }
5	{F → (• E), E → • E + F, E → • E - F, E → • F, F → • num, F → • id, F → • (E)}
6	{E → E + • F, F → • num, F → • id, F → • (E)}
7	{E → E - • F, F → • num, F → • id, F → • (E)}
8	{F → (E •), E → E • + F, E → E • - F}
9	{E → E - F • }
10	{E → E + F • }
11	{F → (E) • }
acc	{S' → E $ • }

Table 2.1: LR(0) item sets for automaton in Figure 2.5.

2.3.3 Conflicts during Parsing

When using shift-reduce parsing there are grammars for which inputs can not be parsed uniquely. This stems from ambiguity in the grammar which

	ACTION							GOTO	
	+	-	(	)	id	num	$end	E	F
0			s5		s4	s3		1	2
1	s6	s7					acc		
2	r3	r3	r3	r3	r3	r3	r3		
3	r4	r4	r4	r4	r4	r4	r4		
4	r5	r5	r5	r5	r5	r5	r5		
5			s5		s4	s3		8	2
6			s5		s4	s3			9
7			s5		s4	s3			10
8	s6	s7	s11						
9	r1	r1	r1	r1	r1	r1			
10	r2	r2	r2	r2	r2	r2			
11	r6	r6	r6	r6	r6	r6			

Figure 2.4: LR(0) parsing table for automaton in Figure 2.5. The numbers and letters correspond to shift and reduce under the ACTION columns. So s6 means *shift 6* and r1 means reduce using *rule 1*. The GOTO columns correspond to line 10 of Algorithm 2. An empty table entry would result in an error as it means there is no valid transition from the current state given the current input symbol.

conflicts with LR-Parsers being deterministic. We can classify these conflicts into reduce/reduce and shift/reduce conflicts.

Shift/Reduce Conflicts Shift/Reduce conflicts occur when the parsing table contains multiple entries for a given input symbol, specifically a combination of shift and reduce actions. Consider the grammar fragment shown below. If the stack is of the form $\langle \ldots,$ `if`, *expr*, `then`, *stmt*, $\rangle$ (with the top of stack to the right) and the next input is `else` it is unclear whether the parser should shift the `else` next or whether the current top of stack should be reduced to *stmt*.

stmt → `if` *expr* `then` *stmt*
| `if` *expr* `then` *stmt* `else` *stmt*
| `other`

Reduce/Reduce Conflicts Reduce/Reduce conflicts happen when there are multiple possible reductions from one top-of-stack configuration. If we consider the grammar fragment below we can see that `id` can resolve to both *stmt* and *expr* if the top of stack is $\langle \ldots,$ `id`, `(`, `id`, $\rangle$. This results in a conflict which cannot be deterministically resolved without introducing further constructs like rule precedence.

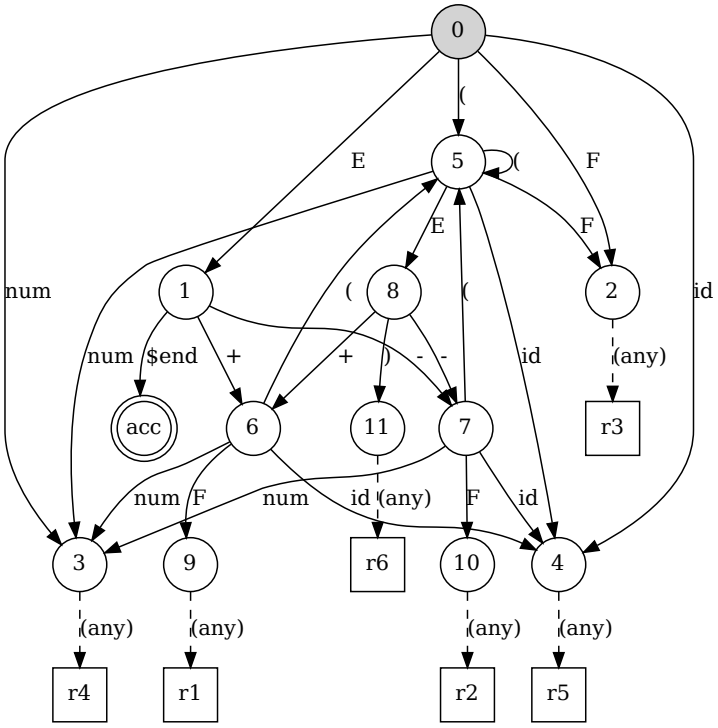


Figure 2.5: Example LR-Automaton for CFG in Figure 2.1, with the start state denoted by the grey node and the accept state denoted by a double circle. The square boxes denote a reduction rule. So r1 would correspond to rule one, $E \rightarrow E$ `+` F

.

stmt	→	`id (` *parameter_list* `)`
		| *expr* `:=` *expr*
parameter_list	→	*parameter_list* `,` *parameter*
		| *parameter*
parameter	→	`id`
expr	→	`id (` *expr_list* `)`
		| `id`
expr_list	→	*expr_list* `,` *expr*
		| *expr*

2.4 Testing

Software testing is a dynamic method used to demonstrate that a system-under-test functions correctly and to reveal any bugs that may be present in the system. There are many different approaches to software testing but most follow the general method of executing the target system over a test suite. Such a *test suite* is comprised of individual *test cases* which in turn are made up of a *test input* $\vec{x}$ and an *expected output* y. When a SUT is executed over $\vec{x}$ we can compare its output to y in order to generate a *pass* or *fail* verdict [9, 13].

There are two main approaches to testing, white-box and black-box testing, which affect how test suites for each approach is produced. White-box testing needs access to the SUT in order to construct a test suite since test cases produced using this method are generally created to exercise specific execution paths in the SUT. It is generally used to test specific aspects of the code using methods like branch and loop testing [16]. In contrast, black-box testing does not have access to the SUT when generating a test suite. Test suites used in black-box testing are constructed according to a specification for the SUT. Common methods of black box testing include partition testing and boundary value testing [11, 31]. In general, test suites used in black-box testing contain more redundant test cases but test suites for this method can be constructed much more quickly.

2.4.1 Systematic Grammar-based Testing

Grammar-based testing is a form of black-box testing, since the test cases are constructed from an input grammar, the specification, for a system that attempts to read input according to the input grammar. In grammar-based testing, the test input is a word $x \in T^*$ and the expected output is either *accept* (for *positive* tests $x \in L(G)$) or *reject* (for *negative* tests $x \notin L(G)$). Note that the verdict is also *pass* if the program rejects a negative test case, i.e., identifies an expected syntax error.

Specifically, grammar-based testing can be viewed as an instance of partition testing, where equivalence classes are defined by a coverage criterion. For example, for rule coverage (discussed below), the equivalence classes would be defined in terms of the rules used to construct a test case, i.e., if two test cases cover the same grammar rule they belong to the same equivalence class. Therefor, testing all partitions would require constructing a test suite containing test cases that cover each rule at least once.

Criteria for the Generic Cover Algorithm

Algorithm 3: 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 ← ∅
2 for X ∈ V do
3 |   compute S ⇒*≤ αXω
4 |   for θ ∈ C(X) do
5 |   |   compute αθω ⇒*≤ w
6 |   |   TS.add(w)
7 |   end
8 end
9 return TS
```

Most current grammar-based testing algorithms follow a very similar structure, as shown in Algorithm 3. The algorithm iterates over all symbols of the grammar and computes their minimal derivations according the coverage criterion specified. These different coverage criteria result in very different test suites, however the general structure for constructing these test suites remain the same.

The various coverage criteria considered in this thesis are described below.

Rule Coverage This coverage criterion ensures that for every production $A \to \alpha$ it is included in a sentence S in the test suite such that $S \to \gamma A \beta$ and forms a test case t such that $t \to \gamma\alpha\beta$. We can formulate it as a coverage criterion for the generic cover algorithm as $\text{rule}(X) \hat{=} \{\alpha \mid X \to \alpha \in P\}$. This criterion tends to result in relatively small test suites as compared to the other grammar-based test suite construction coverage criteria we consider.

Context-Dependent Rule Coverage The original rule coverage algorithm does not take into account the context in which a rule is applied. This means that a test suite might not reveal a bug with a rule in a different context than the one considered by that test suite. Context-dependent rule [29] coverage attempts to solve this problem by ensuring a rule is applied in all possible contexts in which it directly occurs. For the generic cover algorithm this criterion can be formulated as $\text{cdrc}(X) \hat{=} \{\alpha\gamma\omega \mid X \to \alpha Y \omega \in P, Y \to \gamma \in P\}$.

This idea is expanded upon by the breadth-first coverage that applies a rule in all contexts up to a depth k and can be formulated as $\text{bfs}_k(X) \mathrel{\hat{=}} \{\alpha Y \omega \mid X \Rrightarrow^k \alpha Y \omega, Y \in V\}$.

K-Path (Step) Coverage K-path coverage [24] utilizes the derivation tree for a grammar to produce a test suite. This is done by considering paths in the derivation tree and limiting the number of nodes in the path to k. Formally this can be stated as $\text{step}_k(X) \mathrel{\hat{=}} \{\alpha Y \omega \mid X \Rightarrow^k_\leq \alpha Y \omega, Y \in V\}$ for the generic cover algorithm. For small k test suite sizes are similar to that of breadth-first coverage, however the test suite size grows much less rapidly than breadth-first coverage as k is increased.

Derivation Coverage Derivation coverage constructs a test suite by ensuring that all shortest paths between any pair of symbols are covered. This often results in a very effective yet compact test suite when compared to other coverage criteria. It is defined as $\text{deriv}(X) \mathrel{\hat{=}} \{\alpha Y \omega \mid X \Rightarrow^*_\leq \alpha Y \omega, Y \in V\}$.

PLL Zelenov et al. [47] propose a method for test suite generation that ensures that all non-terminal symbols are covered by test cases such that there exists sentential forms for all non-terminals that start with each terminal in its first set. This is done by first constructing derivation chains from the non-terminal to a start symbol. This is then used to construct a sentential form for the non-terminal of the form $\alpha A \beta$ for a non-terminal A. Variants of this sentential form are then created for each terminal symbol in the first set of A and other non-terminals in the sentential form are ground out to form test cases. Although the original algorithm for constructing a test suite according to this criterion is not the same as the generic cover algorithm, we may also formulate it as a coverage criterion for the generic cover algorithm as $\text{pll}(X) \mathrel{\hat{=}} \{a\omega \mid X \Rightarrow^*_\leq a\omega, X \in N, a \in \text{first}(X)\}$.

Other approaches

There are a few grammar-based test generation algorithms that do not conform to the same structure as the generic cover algorithm. For example, Purdom's algorithm [30, 34] proceeds in multiple phases and selects rules based on different conditions than the generic cover algorithm. Another approach is the PLR algorithm proposed by Zelenov et al. [47] which utilizes some information from the LR-automaton for a grammar in generating test suites.

2.4.2 Grammar-based fuzzing

Software fuzzing is a popular way of testing real world software systems. It uses randomization of inputs in order to detect bugs and crashes in these systems. In grammar-based fuzzing a probabilistic context-free grammar is often used, with probabilities being attached to a grammar's production rules [21,43,44]. These probabilities are then used to generate words from the grammar along with other constraints, like the maximum length of a word. This method of grammar-based testing makes up the bulk of the work in the field of grammar-based testing at this time.

Although very effective at finding bugs, fuzzing is often very resource expensive and requires large test suites to be truly effective. Due to the random nature of fuzzing, a smaller test suite may be very biased towards certain grammar structures (see Chapter 5 for the effect of small random biases on coverage achieved over a system). Feedback directed fuzzing for grammar-based testing is fairly unexplored and not much more effective than existing grammar-based fuzzing methods [8].

Chapter 3

Generating Test Suites by Covering LR-Graphs

In this chapter we discuss the foundational topic of LR-graphs. We then describe two different test suite construction algorithms, both of which cover the LR-graph. Finally, we define two different classes of negative mutations on valid paths over these graphs that allow us to construct negative test cases.

3.1 LR-Graphs

One method of parsing input according to a CFG is by using a push down automaton known as an LR-Automaton (see Section 2.3). This type of automaton contains a stack that is altered by the two possible transitions in the LR-Automaton. A *shift* transition reads one element from the input stream and pushes it onto the stack. The *reduce* transitions applies a grammar rule by popping the corresponding number of elements off the stack and pushing the appropriate non-terminal symbol onto the stack.

We extend the LR-itemset construction by defining an *LR-graph* $LR_G = (V, E, v_0, v_{acc})$ for a CFG $G = (N, T, P, S)$. LR_G is a labeled directed graph with a set of vertexes V and a set of edges $E = E_{\rightarrow} \cup E_{\dashrightarrow}$. The vertexes correspond to the states of the LR-Automaton, with a single start state represented by a vertex $v_0 \in V$ and a single accept state represented by a vertex $v_{acc} \in V$. In Figures 3.1 and 3.2 the vertex for the start state is shown in grey and the vertex for the accept state is shown by a double-circle outline around the vertex.

Edges may be either *push* edges $E_{\rightarrow} \subseteq V \times (N \cup T) \times V$ or *pop* edges $E_{\dashrightarrow} \subseteq V \times (N \times \mathbf{N}) \times V$. Push edges are written as $u \rightarrow_X v$ with label X and pop edges are written as $u \dashrightarrow_{A/|\gamma|} v$ with label $A/|\gamma|$ for a pop edge

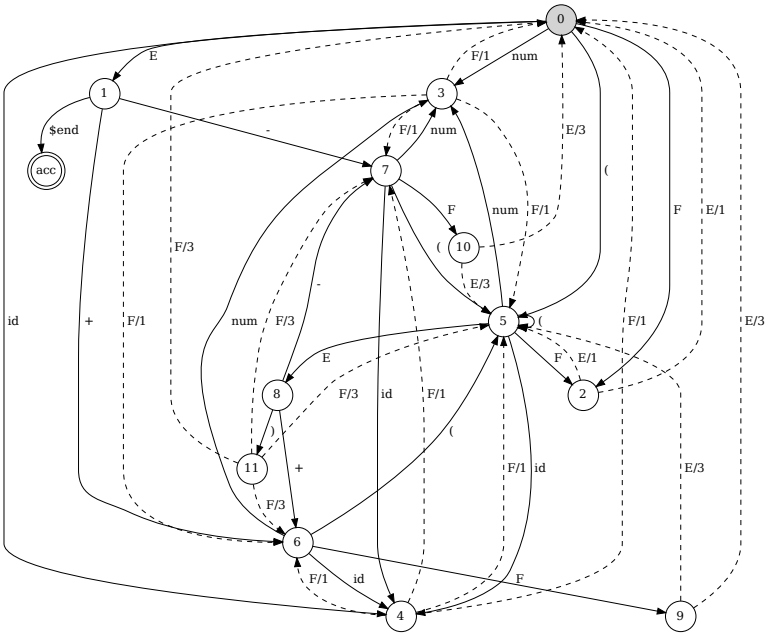
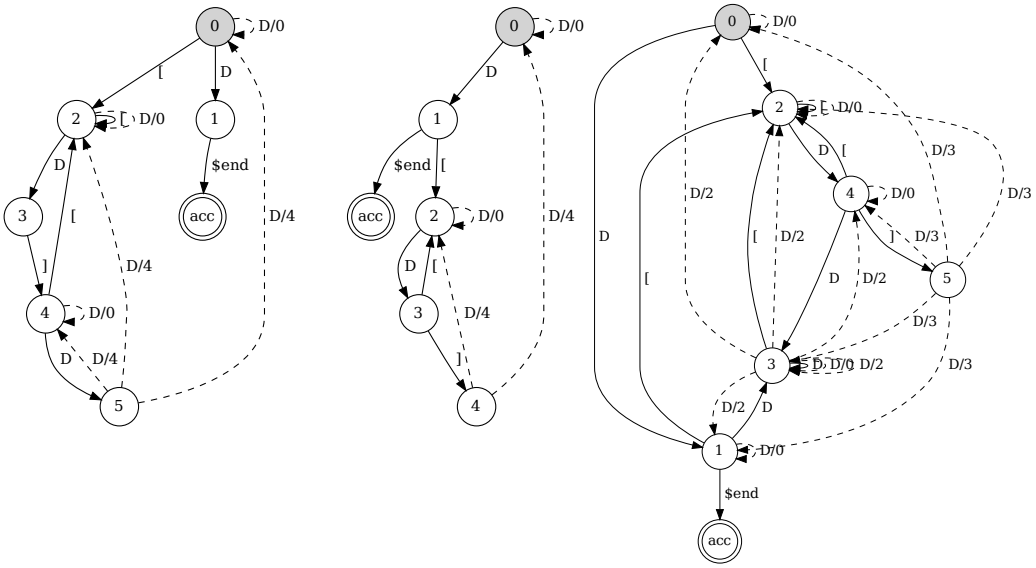


Figure 3.1: LR-graph for LR-automaton in Figure 2.5

corresponding to a rule $A \rightarrow \gamma \in P$. Push edges labeled with a terminal symbol correspond to shift actions in the LR parsing table and push edges labeled with a non-terminal symbol correspond to the GOTO transitions in the LR parsing table. Pop edges are the part of the LR-graph that deviate the most from the LR-automaton. They correspond to the reduce actions in the LR parsing table. However, they also encode specific top-of-stack contexts in which these reduce actions may occur. Therefore, a single reduce action may result in multiple pop edges. We can determine the pop edges corresponding to a reduce transition by traversing back from the current state, using only push edges, and finding all paths with a length of $|\gamma|$. For example, in Figure 2.5 we can see that, at state 9, we should reduce using the first rule (from the grammar in Figure 2.1), $E \rightarrow E + F$. There are two possible top-of-stack contexts which may result in this reduction, $\langle \dots, v_0, v_1, v_6, v_9 \rangle$ and $\langle \dots, v_5, v_8, v_6, v_9 \rangle$ which are shown by the two pop edges $v_9 \dashrightarrow_{E/3} v_0$ and $v_9 \dashrightarrow_{E/3} v_5$ in Figure 3.1. The concept of pop edges allow us to explicitly encode unique top-of-stack contexts into the graph, which allow us to ensure coverage of these contexts in our coverage criteria. When either a push edge or a pop edge may be used the edge is written as $\rightsquigarrow$ and $\leadsto$.

A path in LR_G is the sequence of edges $e_1 \dots e_n$. Paths are called valid if



(a) $D \rightarrow \epsilon \mid [\ D\ ]\ D$ (b) $D \rightarrow \epsilon \mid D\ [\ D\ ]$ (c) $D \rightarrow \epsilon \mid [\ D\ ]\ \mid D\ D$

Figure 3.2: Dyck grammar variants and corresponding LR-graphs. These grammars all describe the same language with different locations of recursion in the grammar.

the sequence of edges satisfies a set of conditions. In paths over LR_G edges are only valid in specific contexts. A push edge $e = u \rightarrow_a v$ labeled with a terminal symbol $a \in T$ can exists in a valid path iff the edge directly preceding it was a push edge or it is the first edge in the path. This means that a terminal push edge is only valid if not preceded by a pop edge, since the pop action of reduction must be followed by pushing the result of the GOTO function (which is shown by a non-terminal push edge). For example, in Figure 3.2 (a) $v_2 \rightarrow_[v_2$ is only valid when it is preceded by $v_0 \rightarrow_[v_2$ and not $v_2 \dashrightarrow_{D/0} v_2$.

A pop edge is valid if it is directly preceded by a valid path that results in the top-of-stack context for which this pop edge was constructed. For example, in Figure 3.1, the pop edge $v_9 \dashrightarrow_{E/3} v_5$ is only valid if the path directly preceding it results in a stack of the form $\langle \dots, v_5, v_8, v_6, v_9 \rangle$. This can only be achieved by the sequence of push edges $v_5 \rightarrow_E v_8 \rightarrow_+ v_6 \rightarrow_F v_9$ (see Section 3.1 below for more details on how this sequence of push edges are determined).

Push edges labeled with a non-terminal symbol correspond to the GOTO transitions in the LR parsing table. This means that a push edge labeled with

a non-terminal symbol is valid iff it is preceded by a pop edge such that $X = A$ for push edge $u \rightarrow_X v$ and pop edge $g \dashrightarrow_{A/|\gamma|} u$. In Figure 3.2 (a) $v_0 \rightarrow_D v_1$ is only valid if preceded by $v_0 \dashrightarrow_{D/0} v_0$ or by $v_5 \dashrightarrow_{D/4}$. By construction there does not exist more than one non-terminal push edge with the same label originating from the same vertex. This is because having more than one non-terminal push edge with the same label from a vertex would be equivalent to a shift/shift conflict, which is not possible in LR-parsers due to the way in which item sets (and thus states) are constructed.

We identify the (labeled) edges with relations over pairs of vertices in the usual way, and write $\rightsquigarrow \circ \rightsquigarrow$ to denote the composition of two relations corresponding to $\rightsquigarrow$ and $\rightsquigarrow$, $\rightsquigarrow^*$ to denote the corresponding transitive closure, and define $\rightsquigarrow_I = \bigcup_{i \in I} \rightsquigarrow_i$. We omit I if it is clear from the context.

We represent vertexes along a path $p = v_1 \rightsquigarrow_{l_1} v_2 \rightsquigarrow_{l_2} \rightsquigarrow \ldots \rightsquigarrow_{l_{n-1}} v_n$ as $v(p) = v_1 \ldots v_n$. A *word over the path* p is defined as the sequence $w(p) = \langle a_i \mid v_i \rightarrow_{a_i} v_{i+1} \in p,\ a_i \in T \rangle$ of terminal symbols labeling the push edges in p. We call such a path p *accepting* if $v(p) = v_0 \ldots v_{acc}$ and the sequence of states in $v(p)$ reflect a valid sequence of states in a pushdown automaton $A = (V, T, T \cup N, \delta, v_0, S, \{v_{acc}\})$ with an appropriate transition relation δ. For example, the path $v_0 \dashrightarrow_{D/0} v_0 \rightarrow_D v_1 \rightarrow_{[} v_2 \dashrightarrow_{D/0} v_2 \rightarrow_D v_3 \rightarrow_{]} v_4 \dashrightarrow_{D/4} v_0 \rightarrow_D v_1 \rightarrow_{\$end} v_{acc}$ in Figure 3.2 (b) is an accepting path corresponding to the word `[ ]`.

Construction of Pop Edges and Reduction Paths

One of the main differences between LR-graphs and LR-Automata is how reduction transitions are represented. In the automaton a reduction is a single transition resulting in items being popped off the stack and a single non-terminal symbol being pushed onto the stack. However, in the LR-graph this transition is split into a pop-edge and a push edge. We do this in order to encode the stack contexts in which a reduction can be applied

A single reduction rule can thus result in multiple pop edges. For example, in Figure 3.2 (a), state 5 contains the reduction transition for the rule $D \rightarrow$ `[` D `]` D. This results in the three pop edges $v_5 \dashrightarrow_{D/4} v_4$, $v_5 \dashrightarrow_{D/4} v_2$, $v_5 \dashrightarrow_{D/4} v_0$ corresponding to the three possible stack contexts in which the reduction may be applied. This is because all possible prefix path segments for a reduction rule are calculated and connected with a pop edge from the state at which the reduction rule is applied to the source of the prefix path segment.

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 Figure 3.2 (c) there are multiple reduce/reduce conflicts at state 3. For example, the reduction transition for the rule $D \to \epsilon$ is always valid at state 3, which conflicts with the reduction transition for the rule $D \to D\ D$ whenever it is also valid. Instead of having to use methods such as lookaheads to resolve this, we simply add pop edges for both rules, namely $v_3 \dashrightarrow_{D/0} v_3$ and $v_3 \dashrightarrow_{D/2} v_1$, since the algorithms for edge coverage will ensure both rules are considered by covering both pop edges. This then allows us to use an LR(0) automaton as base for the construction of the LR-graph since conflicts do not pose an issue to the construction or coverage of the LR-graph.

Reduction Paths We describe a rule application over the LR-graph using reduction paths. A reduction path is path of the form

$$r = \underbrace{\iota(p \in E_{\to}^{|\gamma|}, vert(p) = v \dots u)}_{part\ 1} \circ \underbrace{(u \dashrightarrow_{A/|\gamma|} v)}_{part\ 2} \circ \underbrace{\iota(v \to_A v')}_{part\ 3}$$

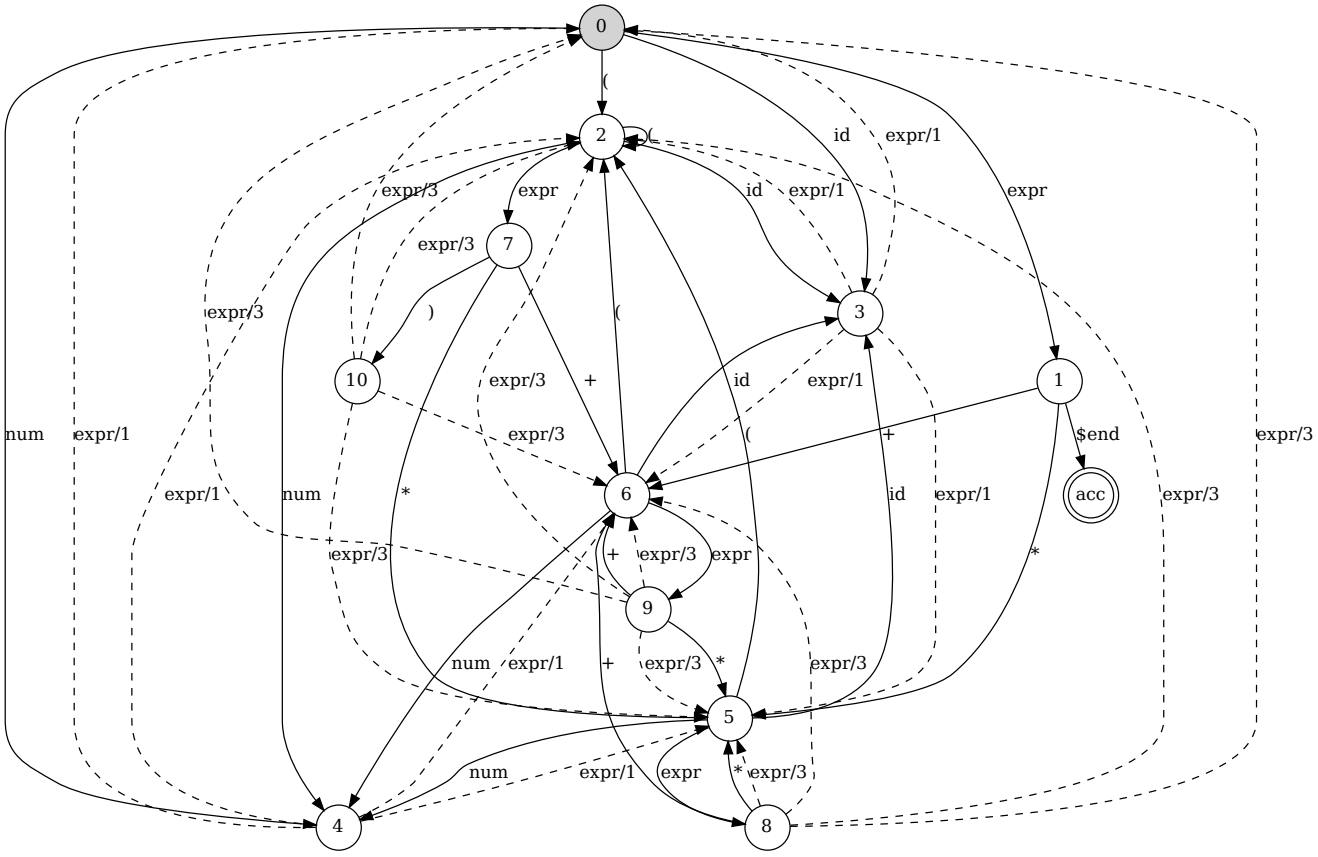
that is uniquely determined by its pop edge $u \dashrightarrow_{A/|\gamma|} v$. We say a reduction path is unique for a pop edge since its two components, other than the pop edge (part 2), namely the push path component $\iota(p \in E_{\to}^{|\gamma|}, vert(p) = v \dots u)$ (part 1) and the non-terminal push edge that follows the pop edge, $\iota(v \to_A v)$ (part 3), are unique by construction. Specifically, the push path component must be unique since it directly corresponds to the prefix path component that results in the top-of-stack context for which the pop edge was constructed and each top-of-stack context for a reduction rule generates a new pop edge.

Reduction paths capture the required stack context for a pop edge and the resulting reduction. For example, the pop edge $v_4 \dashrightarrow_{D/4} v_0$ in Figure 3.2 (b) corresponds to the reduction path $v_0 \to_D v_1 \to_{[} v_2 \to_D v_3 \to_{]} v_4 \dashrightarrow_{D/4} v_0 \to_D v_1$. This corresponds directly to the rule $D \to D\ \texttt{[}\ D\ \texttt{]}$. The reduction path indicates that this pop edge requires the stack to be of the form $\langle \dots, D, \texttt{[}, D, \texttt{]}, \rangle$, originating from v_0. It also indicates that these four items will be popped off the stack and replaced by the non-terminal D.

Since the set of all reduction paths will cover all pop edges, and all push edges must lead to pop edges in order for only the start symbol and $end to be left on the stack when the accept state is reached in a valid path, reduction path coverage will lead to coverage over all edges.

3.2 Traversing the LR-graph

The first algorithm we present in this thesis is comprised of two stages, namely the flooding and path completion phases. The flooding phase generates paths



(a) *expr* → *expr* `*` *expr* | *expr* `+` *expr* | `(` *expr* `)` | `id` | `num`

Figure 3.3: A LR-graph for an expression grammar

Algorithm 4: Main LR-graph traversal algorithm

input : An LR Graph for a CFG $G = (N, T, P, S)$
output : A test suite for the CFG

```
paths ← ∅
for prefix ∈ flooding(LR_G) do
    paths.∪ {complete_path(LR_G, prefix)}
end
return {w(p) | p ∈ paths}
```

from v_0, which are prefixes of valid words in $L(G)$, in a layer-by-layer manner and ensures all edges in the graph are covered. The path completion phase

Algorithm 5: Flooding Phase

input : An LR Graph $LR_G = (V, E, v_0, v_{acc})$
output : A List of valid paths starting at v_0 covering all edges in LR_G

$Q \leftarrow \langle\langle v_0 \rangle\rangle$
$seen\_edges \leftarrow \varnothing$
while $seen\_edges \neq E$ **do**
  $p = Q.dequeue\,()$
  if $p.last = u \dashrightarrow_{A/n} v$ **then**
    $Q.enqueue\,(p \circ \iota(v \rightarrow_A v' \in E))$
    $seen\_edges.\cup \{v \rightarrow_A v'\}$
    $continue$
  end
  for e *in* $\{(v \rightsquigarrow v') \in E \mid p.last = u \rightsquigarrow v\}$ **do**
    if $e \in E_{\rightarrow} \vee (e \in E_{\dashrightarrow} \wedge valid(p \circ e))$ **then**
      $Q.enqueue\,(p \circ e)$
      $seen\_edges.\cup \{e\}$
    end
  end
end
return Q

then completes these prefixes with the shortest possible postfix that results in a valid path in the grammar. The test suite is then derived from words over these valid paths.

Both of these phases use breadth-first search (BFS). Breadth-first search is used since it explores the LR-graph in layers, which results in shorter test cases. There is also a bound on the maximum path length required to cover all edges in the graph. This means that we can guarantee termination of the algorithm after exploring a finite number of layers, without cycle detection or avoidance. This is due to cycles being required by all valid paths. Consider the form of the first two parts of a reduction path in Section 3.1, a sequence of push edges from a node u to a node v followed by a pop edge from v to u. This forms a necessary cycle. Other cycles may also be necessary, depending on the structure of the LR-graph. For example, in Figure 3.2 (a) the edges $v_2 \rightarrow_d v_3$ and $v_3 \rightarrow_] v_4$ must be visited a minimum of two times in order to explore the edge $v_4 \rightarrow_d v_2$.

Normally BFS appends children nodes to the end of a queue and continues until the queue is empty. Instead of only storing next nodes, we duplicate the current path and stack before exploring valid, reachable edges from the

Algorithm 6: Path Completion Phase

input : An LR Graph $LR_G = (V, E, v_0, v_{acc})$ and a valid path starting at v_0
output : A valid path starting at v_0 and ending in v_{acc}

$Q \leftarrow \langle path \rangle$
while *true* **do**
  $p = Q.dequeue()$
  if $p.last = u \rightsquigarrow v_{acc}$ **then**
    return p
  end
  if $p.last = u \dashrightarrow_{A/n} v$ **then**
    $Q.enqueue(p \circ \iota(v \rightarrow_A v' \in E))$
    continue
  end
  for e *in* $\{(v \rightsquigarrow v') \in E \mid p.last = u \rightsquigarrow v\}$ **do**
    if $e \in E_{\rightarrow} \vee (e \in E_{\dashrightarrow} \wedge valid(p \circ e))$ **then**
      $Q.enqueue(p \circ e)$
    end
  end
end

current node. The exploration continues until all edges are covered for the flooding phase, and until v_{acc} is reached for the path completion phase.

By duplicating the current path and stack it enables the algorithm to handle conflicts in a similar way to how GLR parsers resolve conflicts. When a conflict is encountered, we do not attempt to choose a specific traversal and instead perform both traversals in parallel.

Algorithm The main algorithm (see Algorithm 4) calls the flooding phase (see Algorithm 5) to obtain a set of prefixes. Each prefix is then individually completed (see Algorithm 6) and the word over the complete path is extracted and added to the test suite.

Both the flooding and path completion phases dequeue a path and extend this path at its head. If the last edge in the path was a push edge, the path is duplicated and all valid edges, those edges that could be appended to the current path so that the new path remains valid (see Section 3.1 for when different types of edges are valid), originating from the current node are explored in parallel. This entails copying the current path, appending the edge being explored and then enqueueing this new path.

If the previous edge was a pop edge then only the corresponding non-terminal push edge from the current state may be explored. This edge is determined uniquely by construction as indicated by the ι quantifier (see Algorithm 5 line 6 and Algorithm 6 line 8). This non-terminal push edge completes the reduction action that was started by the pop edge that came before it.

A pop edge $u \dashrightarrow_{A/|\gamma|} v$ is considered valid if the target v is in the current path and the current top-of-stack context resulting from the current path satisfies the rule corresponding to the pop edge. In Figure 3.2 (c), for a path $v_0 \dashrightarrow_{D/0} v_0 \rightarrow_D v_1 \dashrightarrow_{D/0} v_1 \rightarrow_D v_3$ we can see that only $v_3 \dashrightarrow_{D/0} v_3$ and $v_3 \dashrightarrow_{D/2} v_0$ are valid pop edges since the top-of-stack context is $... v_0 \rightarrow_D v_1, v_1 \rightarrow_D v_3$. Stack context may be viewed as a path comprised of only push edges, less the push edges which have been removed by pop actions as this directly relates to the symbols that are currently on the stack.

Example For the LR-graph shown in Figure 3.2(a), we can see the layer-by-layer exploration of the flooding phase in Table 3.1. It shows how the queue growth accelerates as more and more layers are explored. We can observe that, from layer 7, the only edge that remains to be covered is $v_4 \dashrightarrow_{D/4} v_2$, however the queue grows from 3 to 5 paths until the edge is finally covered in layer 10 of the exploration. This is due to the non-deterministic nature of the breadth-first search algorithm. Unnecessary stack duplication can be negated to some extent by turning the queue in Algorithms 5 into a priority queue where unseen edges have a higher priority.

A priority queue allows the breath-first exploration algorithm to cover slightly larger LR-graphs like the one seen in Figure 3.3. From this figure we can also see how the number of edges grows rapidly as more rules are introduced in a grammar. Even with a priority queue, the breath-first search algorithm produces 620 paths that require completion. If we extend the expression grammar from Figure 3.3 with a few extra rules and non-terminal symbols, as can be seen in Figure 3.4, the breath-first search algorithm does not terminate in a reasonable amount of time.

This is due to *stubborn edges* that arise from the structure of the LR-graph. We have found these edges to exist in LR-graphs for many other grammars as well. They are pop-edges that require a long, specific prefix to become valid. These prefixes also often consist of edges that will 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 attempted to counteract this by methods such as bi-directional search and a more intricate edge priority

$$
\begin{aligned}
e &\rightarrow e \texttt{ + t} \\
&\quad \mid e \texttt{ - } t \\
&\quad \mid t \\
t &\rightarrow t \texttt{ * f} \\
&\quad \mid t \texttt{ / } f \\
&\quad \mid f \\
f &\rightarrow \texttt{num} \\
&\quad \mid \texttt{id} \\
&\quad \mid \texttt{(} e \texttt{)}
\end{aligned}
$$

Figure 3.4: A slightly larger expression grammar

implementation. However, in each case it failed to prove effective against stubborn edges for real world grammars. It is this problem that we solve by the algorithm presented in Section 3.3.

3.3 Solving for Pop-Edges

The second algorithm we present in this thesis uses pop-edge coverage to avoid the pitfalls of the first algorithm. One of the main drawbacks of the breadth-first exploration algorithms is that it only considers the immediate edges that are adjacent to the current node when constructing paths that explore the graph. This leads to a rapid growth in redundant test cases as the size of the graph is increased, due to the high branching factor in many nodes leading to a substantial growth in queue size as the required lengths of paths are increased. By using pop-edge coverage, we can avoid this drawback by making the calculation much more efficient by avoiding this branching factor altogether.

The pop-edge coverage algorithm, that we describe in this section, exploits extra information that is contained in the LR-graph by construction. Since each pop edge corresponds to a pop action in a specific top-of-stack context, we know that there exists only one top-of-stack configuration (we focus on the top $|\gamma|$ elements of the stack for a pop edge $u \dashrightarrow_{A/|\gamma|} v$ as described in Section 3.1) per pop edge.

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. These reduction paths may then be embedded in other reduction paths in a fixed point computation to find the shallowest embeddings of each reduction path into a valid path in the graph. For example, for the

Layer:	Queue:
0	v_0
1	$v_0 \dashrightarrow_{D/0} v_0$
2	$v_0 \dashrightarrow_{D/0} v_0 \rightarrow_{[} v_1$
3	$v_0 \dashrightarrow_{D/0} v_0 \rightarrow_{[} v_1 \rightarrow_{\$end} v_{acc}$,
	$v_0 \dashrightarrow_{D/0} v_0 \rightarrow_{[} v_1 \rightarrow_{[} v_2$
4	$v_0 \dashrightarrow_{D/0} v_0 \rightarrow_{[} v_1 \rightarrow_{\$end} v_{acc}$,
	$v_0 \dashrightarrow_{D/0} v_0 \rightarrow_{[} v_1 \rightarrow_{[} v_2 \dashrightarrow_{D/0} v_2$
5	$v_0 \dashrightarrow_{D/0} v_0 \rightarrow_{[} v_1 \rightarrow_{\$end} v_{acc}$,
	$v_0 \dashrightarrow_{D/0} v_0 \rightarrow_{[} v_1 \rightarrow_{[} v_2 \dashrightarrow_{D/0} v_2 \rightarrow_{D} v_3$
6	$v_0 \dashrightarrow_{D/0} v_0 \rightarrow_{[} v_1 \rightarrow_{\$end} v_{acc}$,
	$v_0 \dashrightarrow_{D/0} v_0 \rightarrow_{[} v_1 \rightarrow_{[} v_2 \dashrightarrow_{D/0} v_2 \rightarrow_{D} v_3 \rightarrow_{[} v_2$,
	$v_0 \dashrightarrow_{D/0} v_0 \rightarrow_{[} v_1 \rightarrow_{[} v_2 \dashrightarrow_{D/0} v_2 \rightarrow_{D} v_3 \rightarrow_{]} v_4$
7	$v_0 \dashrightarrow_{D/0} v_0 \rightarrow_{[} v_1 \rightarrow_{\$end} v_{acc}$,
	$v_0 \dashrightarrow_{D/0} v_0 \rightarrow_{[} v_1 \rightarrow_{[} v_2 \dashrightarrow_{D/0} v_2 \rightarrow_{D} v_3 \rightarrow_{[} v_2 \dashrightarrow_{D/0}$,
	$v_0 \dashrightarrow_{D/0} v_0 \rightarrow_{[} v_1 \rightarrow_{[} v_2 \dashrightarrow_{D/0} v_2 \rightarrow_{D} v_3 \rightarrow_{]} v_4 \dashrightarrow_{D/4} v_0$
8	$v_0 \dashrightarrow_{D/0} v_0 \rightarrow_{[} v_1 \rightarrow_{\$end} v_{acc}$,
	$v_0 \dashrightarrow_{D/0} v_0 \rightarrow_{[} v_1 \rightarrow_{[} v_2 \dashrightarrow_{D/0} v_2 \rightarrow_{D} v_3 \rightarrow_{[} v_2 \dashrightarrow_{D/0} \rightarrow_{D} v_3$,
	$v_0 \dashrightarrow_{D/0} v_0 \rightarrow_{[} v_1 \rightarrow_{[} v_2 \dashrightarrow_{D/0} v_2 \rightarrow_{D} v_3 \rightarrow_{]} v_4 \dashrightarrow_{D/4} v_0 \rightarrow_{D} v_1$
9	$v_0 \dashrightarrow_{D/0} v_0 \rightarrow_{[} v_1 \rightarrow_{\$end} v_{acc}$,
	$v_0 \dashrightarrow_{D/0} v_0 \rightarrow_{[} v_1 \rightarrow_{[} v_2 \dashrightarrow_{D/0} v_2 \rightarrow_{D} v_3 \rightarrow_{[} v_2 \dashrightarrow_{D/0} \rightarrow_{D} v_3 \rightarrow_{[} v_2$,
	$v_0 \dashrightarrow_{D/0} v_0 \rightarrow_{[} v_1 \rightarrow_{[} v_2 \dashrightarrow_{D/0} v_2 \rightarrow_{D} v_3 \rightarrow_{[} v_2 \dashrightarrow_{D/0} \rightarrow_{D} v_3 \rightarrow_{]} v_4$,
	$v_0 \dashrightarrow_{D/0} v_0 \rightarrow_{[} v_1 \rightarrow_{[} v_2 \dashrightarrow_{D/0} v_2 \rightarrow_{D} v_3 \rightarrow_{]} v_4 \dashrightarrow_{D/4} v_0 \rightarrow_{D} v_1 \rightarrow_{\$end} v_{acc}$,
	$v_0 \dashrightarrow_{D/0} v_0 \rightarrow_{[} v_1 \rightarrow_{[} v_2 \dashrightarrow_{D/0} v_2 \rightarrow_{D} v_3 \rightarrow_{]} v_4 \dashrightarrow_{D/4} v_0 \rightarrow_{D} v_1 \rightarrow_{[} v_2$
10	$v_0 \dashrightarrow_{D/0} v_0 \rightarrow_{[} v_1 \rightarrow_{\$end} v_{acc}$,
	$v_0 \dashrightarrow_{D/0} v_0 \rightarrow_{[} v_1 \rightarrow_{[} v_2 \dashrightarrow_{D/0} v_2 \rightarrow_{D} v_3 \rightarrow_{[} v_2 \dashrightarrow_{D/0} \rightarrow_{D} v_3 \rightarrow_{[} v_2 \dashrightarrow_{D/0} v_2$,
	$v_0 \dashrightarrow_{D/0} v_0 \rightarrow_{[} v_1 \rightarrow_{[} v_2 \dashrightarrow_{D/0} v_2 \rightarrow_{D} v_3 \rightarrow_{[} v_2 \dashrightarrow_{D/0} \rightarrow_{D} v_3 \rightarrow_{]} v_4 \dashrightarrow_{D/4} v_2$,
	$v_0 \dashrightarrow_{D/0} v_0 \rightarrow_{[} v_1 \rightarrow_{[} v_2 \dashrightarrow_{D/0} v_2 \rightarrow_{D} v_3 \rightarrow_{]} v_4 \dashrightarrow_{D/4} v_0 \rightarrow_{D} v_1 \rightarrow_{\$end} v_{acc}$,
	$v_0 \dashrightarrow_{D/0} v_0 \rightarrow_{[} v_1 \rightarrow_{[} v_2 \dashrightarrow_{D/0} v_2 \rightarrow_{D} v_3 \rightarrow_{]} v_4 \dashrightarrow_{D/4} v_0 \rightarrow_{D} v_1 \rightarrow_{[} v_2 \dashrightarrow_{D/0} v_2$

Table 3.1: Queue Growth for flooding phase over LR-graph in Figure 3.2 (b)

edge $v_2 \dashrightarrow_{D/0} v_2$ in Figure 3.2 (b), we do not require any push edges so the reduction path is $v_2 \dashrightarrow_{D/0} v_2 \rightarrow_{D} v_3$. This reduction path can then be embedded into another reduction path to give us a complete, valid path $v_0 \dashrightarrow_{D/0} v_0 \rightarrow_{D} v_1 \rightarrow_{[} v_2 \dashrightarrow_{D/0} v_2 \rightarrow_{D} v_3 \rightarrow_{]} v_4 \dashrightarrow_{D/0} v_0 \rightarrow_{D} v_1 \rightarrow_{\$end} v_{acc}$.

Algorithm The algorithm contains two main functions that is used within the main loop, namely the reduction path function and the embed function

(see lines 1 and 2 of Algorithm 7).

For each reduction path we can see that the push path component, $\iota(p \in E_{\rightarrow}^{|\gamma|}, vert(p) = v \dots u)$, and the non-terminal push edge component, $\iota(v \rightarrow_A v')$, are by construction uniquely determined by the pop edge in the reduction path. Multiple push path components would imply multiple top-of-stack contexts which would in turn result in multiple pop edges. For the non-terminal push edge component, this is unique by construction as there can be no two non-terminal push edges originating from the same state with the same label. This means that there can be only one correctly labeled non-terminal push edge originating from the destination node of the pop edge (see Section 3.1 for more details regarding this property of the LR-graph).

The embed function takes a reduction path p and embeds it into other reduction paths where there are non-terminal push edges in the push path component that are equal to the final non-terminal push edge in the current reduction path. This continues until the resultant path is of the form $(v_0 \rightsquigarrow \dots \rightsquigarrow v) \circ p \circ (v' \rightsquigarrow \dots \rightsquigarrow v'' \rightarrow_{\$end} v_{acc})$. This effectively results in a shallowest embedding. Any non-terminal push edge in these embeddings, that is not yet preceded by a pop edge, is substituted out by the shortest reduction path that contains this non-terminal push edge as its final edge. For example, in Figure 3.2 (b), the pop edge $v_4 \dashrightarrow_{D/4} v_2$ results in the reduction path $v_2 \rightarrow_D v_3 \rightarrow_[v_2 \rightarrow_D v_3 \rightarrow_] v_4 \dashrightarrow_{D/4} v_2 \rightarrow_D v_3$. It may be embedded into the reduction path $v_0 \rightarrow_D v_1 \rightarrow_[v_2 \rightarrow_D v_3 \rightarrow_] v_4 \dashrightarrow_{D/0} v_0 \rightarrow_D v_1$ since the non-terminal push edge $v_2 \rightarrow_D v_3$ is contained in the push path component of this reduction path. This gives the path $v_0 \rightarrow_D v_1 \rightarrow_[v_2 \rightarrow_D v_3 \rightarrow_[v_2 \rightarrow_D v_3 \rightarrow_] v_4 \dashrightarrow_{D/4} v_2 \rightarrow_D v_3 \rightarrow_] v_4 \dashrightarrow_{D/0} v_0 \rightarrow_D v_1$. The remaining non-terminal push edges that are not preceded by a pop edge, for example $v_0 \rightarrow_D v_1$, can then be ground out to give the valid path $v_0 \dashrightarrow_{D/0} v_0 \rightarrow_D v_1 \rightarrow_[v_2 \dashrightarrow_{D/0} v_2 \rightarrow_D v_3 \rightarrow_[v_2 \dashrightarrow_{D/0} v_2 \rightarrow_D v_3 \rightarrow_] v_4 \dashrightarrow_{D/4} v_2 \rightarrow_D v_3 \rightarrow_] v_4 \dashrightarrow_{D/0} v_0 \rightarrow_D v_1 \rightarrow_{\$end} v_{acc}$

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

Example We can observe the reduction paths and embeddings found by the pop edge coverage algorithm for Figure 3.2 (b), an LR-graph for a dyck grammar, in Table 3.2. From this we can clearly see that the number of paths considered is far less than the breadth-first search algorithm in Table 3.1. It is clear that the number of valid paths is limited by the number of pop edges and provides an upper bound for the size of the test suite. Even though we have found four embeddings in Table 3.2, the size of the test suite is three

since the extracted words over the valid paths resulted in some duplicate words. This is about half the number of test cases produced by the traversal algorithm, which produces a test suite containing five test cases, while still covering all the edges of the LR-graph. In Chapter 5 we will show that this performance improvement not only applies to small grammars and graphs but scales to production-level grammars such as SQLite that induces 26419 pop edges.

Algorithm 7: Pop Edge Coverage

input : An LR Graph $LR_G = (V, E, v_0, v_{acc})$
output : A test suite covering all edges in LR_G

1 $reduction\_path(u \dashrightarrow_{A/|\gamma|} v) = \iota(p \in E_{\rightarrow}^{|\gamma|}, vert(p) = v \dots u) \circ (u \dashrightarrow_{A/|\gamma|} v) \circ \iota(v \rightarrow_A v')$
2 $embed(rPath) = (v_0 \rightsquigarrow \dots \rightsquigarrow v) \circ rPath \circ (v' \rightsquigarrow \dots \rightsquigarrow v'' \rightarrow_{\$end} v_{acc})$
3 $test\_suite \leftarrow \varnothing$
4 **for** $e \in E_{\dashrightarrow}$ **do**
5 $\quad test\_suite. \cup \{w(embed(reduction\_path(e)))\}$
6 **end**
7 **return** $test\_suite$

Pop Edge:	Reduction Path:	Embedding:
$v_0 \dashrightarrow_{D/0} v_0$	$v_0 \dashrightarrow_{D/0} v_0 \rightarrow_D v_1$	$v_0 \dashrightarrow_{D/0} v_0 \rightarrow_D v_1 \rightarrow_{\$end} v_{acc}$
$v_2 \dashrightarrow_{D/0} v_2$	$v_2 \dashrightarrow_{D/0} v_2 \rightarrow_D v_3$	$v_0 \dashrightarrow_{D/0} v_0 \rightarrow_D v_1 \rightarrow_[v_2 \dashrightarrow_{D/0} v_2 \rightarrow_D v_3 \rightarrow_] v_4 \dashrightarrow_{D/0} v_0 \rightarrow_D v_1 \rightarrow_{\$end} v_{acc}$
$v_4 \dashrightarrow_{D/4} v_2$	$v_2 \rightarrow_D v_3 \rightarrow_[v_2 \rightarrow_D v_3 \rightarrow_] v_4 \dashrightarrow_{D/4} v_2 \rightarrow_D v_3$	$v_0 \dashrightarrow_{D/0} v_0 \rightarrow_D v_1 \rightarrow_[v_2 \dashrightarrow_{D/0} v_2 \rightarrow_D v_3 \rightarrow_[v_2 \dashrightarrow_{D/0} v_2 \rightarrow_D v_3 \rightarrow_] v_4 \dashrightarrow_{D/4} v_2 \rightarrow_D v_3 \rightarrow_] v_4 \dashrightarrow_{D/0} v_0 \rightarrow_D v_1 \rightarrow_{\$end} v_{acc}$
$v_4 \dashrightarrow_{D/0} v_0$	$v_0 \rightarrow_D v_1 \rightarrow_[v_2 \rightarrow_D v_3 \rightarrow_] v_4 \dashrightarrow_{D/0} v_0 \rightarrow_D v_1$	$v_0 \dashrightarrow_{D/0} v_0 \rightarrow_D v_1 \rightarrow_[v_2 \dashrightarrow_{D/0} v_2 \rightarrow_D v_3 \rightarrow_] v_4 \dashrightarrow_{D/0} v_0 \rightarrow_D v_1 \rightarrow_{\$end} v_{acc}$

Table 3.2: Paths constructed by pop-edge algorithm over LR-graph in Figure 3.2 (b)

3.4 Negative Mutations

A negative test case is a test case that we expect to be rejected by the SUT. The goal of using a negative test suite on a SUT is to reveal any bugs which allow invalid input through the parser. For example, the Cloudflare bug described in Chapter 1 could have been revealed be a negative test case that tests invalid input at the end of the HTML page. Negative test suites therefore help reveal instances where a parser is not strict enough in rejecting invalid input.

One way to generate a negative test suite is to start with a positive test suite and mutate the positive test cases into negative test cases. We do not simply want to randomly mutate the positive test cases and check if the mutation resulted in negative test case. Instead, we want to perform mutations that are guaranteed to result in negative test cases, without any further checks being required.

The negative mutations in this thesis mutate positive paths to generate paths that are guaranteed to be invalid. The negative mutations can be divided into two distinct types, *edge mutations* that locally 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. Mutations include insertion, deletion, substitution and prefix cutting. These mutations depend on the notion of follow sets and almost accepting states. These notions do not rely on the traversal that has reached a specific state and are therefore under-approximations. This results in our algorithm discarding some valid mutations in order to guarantee negative test cases without the need for further validation.

Follow Set In a grammar, the follow set of a symbol is defined as the set of tokens that can follow the 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 can define the follow set of a vertex v as $F(v) = \{a \in T \mid \exists u \in V \cdot (v, u) \in \dashrightarrow^* \circ \rightarrow_a\}$, given an LR-graph $LR_G = (V, E, v_0, v_{acc})$.

Precede Set We reverse the idea of the follow set to the precede set, the set of labels on the last terminal push edges that are covered before a vertex is reached. This is equivalent to the terminals that can precede a valid suffix that is recognized by the LR-automaton at this state. More formally, given

an LR-graph $LR_G = (V, E, v_0, v_{acc})$, we can define the precede set of a vertex as $P(v) = \{a \in T \mid \exists u \in V \cdot (v, u) \in \rightarrow_a \circ \dashrightarrow^*\}$.

Almost Accepting State We say a vertex v is *almost accepting* if the accept state v_{acc} can be reached through a path without any further terminal push edges, i.e., without consuming any further input. Formally we can define whether a vertex is an almost accepting state as $AA(v) \Leftrightarrow (v, v_{acc}) \in (\rightarrow_N \cup \dashrightarrow^*)$.

Hence, if $p = v_0 \rightsquigarrow \ldots \rightsquigarrow v$ is a valid path and the word over p is in the language, then v must be almost accepting, i.e., $w(p) \in L(G) \Rightarrow AA(v)$, or, by contraposition, if v is not almost accepting, then the word over p cannot be in the language, i.e., $\neg AA(v) \Rightarrow w(p) \notin L(G)$,

First and Last sets We can extend the concept 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 ground out.

For a reduction path r, of the form $v \rightarrow_a v' \rightsquigarrow \ldots$, its first set is $\{a\}$. If it is of the form $v \rightarrow_A v' \rightsquigarrow \ldots$ then $\mathit{first}(r) = \{first(r') \mid r' = (u \rightsquigarrow \ldots \rightsquigarrow u) \circ (u \dashrightarrow_{A/|\gamma|} v) \circ \iota(v \rightarrow_A v')\}$. Last sets may be computed similarly by considering the last push edge in the push path component of the reduction path, instead of the first push edge.

3.4.1 Edge Mutations

Edge mutations focus on terminal push edges in valid paths. By mutating these edges we are effectively performing string edit operations on the word over the path, i.e., 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.

Insertion On any edge $u \rightsquigarrow v$ traversed during word extraction, the mutation inserts any terminal symbol $a \notin F(v)$. This is guaranteed to be invalid as the next input at the current point in the derivation. Hence, no alternative valid accepting path exists, and the mutated word is indeed invalid.

For example, in Figure 3.2(a) we may insert `[` when have just traversed the push edge $v_2 \rightarrow_d v_3$, since it is not in $F(v_3)$. However, we cannot insert anything after v_4 since $T - F(v_4) = \varnothing$.

Substitution This mutation creates negative test cases similar to insertion mutations, by replacing the label a on traversing a terminal push edge $u \rightarrow_a v$ with a terminal $b \notin F(u)$ that is guaranteed to be invalid as the next input at u. Again, no alternative valid accepting path exists, and the mutated word is indeed invalid.

For example, in Figure 3.2(a) we may replace `]` by `[` when we traverse the edge $v_3 \rightarrow_{\texttt{]}} v_4$, since $F(v_3) = \{\texttt{]}\}$. However, the terminal from the edge $v_2 \rightarrow_{\texttt{[}} v_2$ cannot be replaced since $\texttt{]} \in F(v_2)$

Deletion Deletion creates negative test cases by removing the terminal a at a push edge $u \rightarrow_a v$ from the word when it is extracted from the path. The result is guaranteed to be a negative test if $F(u) \cap F(v) = \varnothing$ and $\neg AA(u)$. This is because any terminal symbol that should follow state u can not follow state v in any valid path.

In Figure 3.2, none of the LR-graphs allows any deletion of a terminal push edge, since all terminal push edges either have almost-accepting source vertices or have a non-empty overlap in the follow set of the source and target vertexes. However, in Figure 3.3 we may delete the token `id` on the edge $v_0 \rightarrow_{id} v_3$ since $F(v_0) = \{\texttt{id}, \texttt{num}, \texttt{(}\}$ and $F(v_3) = \{\texttt{+}, \texttt{*}, \texttt{)}\}$ so $F(v_0) \cap F(v_3) = \varnothing$

3.4.2 Stack Mutations

Stack mutations affect an entire sequence of events that would occur on the stack by performing mutations that may use entire reduction paths. Therefore, multiple tokens in the resulting word may be affected at once, meaning these mutation may require backtracking if performed while extracting a word from a path. These mutations are somewhat similar to rule mutations on non-terminal symbols.

Insertion We can insert a reduction path r for a pop edge $u \dashrightarrow_{A/|\gamma|} v$ into a path $v_0 \rightsquigarrow \ldots \rightsquigarrow a \rightsquigarrow b \rightsquigarrow \ldots \rightsquigarrow v_{acc}$ after vertex a if $F(a) \cap \mathit{first}(r) = \varnothing \vee P(b) \cap \mathit{last}(r) = \varnothing$ and r is not nullable. Hence, the inserted reduction path results in a token sequence which is not valid, given the current top-of-stack, sequence when node a is reached in the traversal and before b is reached in the traversal.

We can insert the reduction path $r = v_2 \rightarrow_{\texttt{(}} v_2 \rightarrow_{expr} v_7 \rightarrow_{\texttt{)}} v_{10} \dashrightarrow_{expr/3} v_2 \rightarrow_{expr} v_7$ into the path $v_0 \rightarrow_{\texttt{(}} v_2 \rightarrow_{num} v_4 \dashrightarrow_{expr/1} v_2 \rightarrow_{expr} v_7 \rightarrow_{\texttt{)}} v_{10} \dashrightarrow_{expr/3} v_0 \rightarrow_{expr} v_1 \rightarrow_{\$end} v_{acc}$ over the LR-graph in Figure 3.3 after v_7, since $F(v_7) = \{\texttt{+}, \texttt{*}, \texttt{)}\}$ and $\mathit{first}(r) = \{\texttt{(}\}$ so $F(v_7) \cap first(r) = \varnothing$.

Substitution Substitution performs a combination of deletion and insertion. Here we replace a reduction path p of the form $a \rightsquigarrow ... \rightsquigarrow b$ with a reduction path r if $F(a) \cap \mathit{first}(r) = \varnothing \vee P(b) \cap \mathit{last}(r) = \varnothing$ and r may only be nullable if p is not nullable. This mutates the top-of-stack sequence caused by the traversal from a to b by replacing it with a sequence that can not occur between a and b.

In both Figure 3.2 and Figure 3.3 there do not exist any cases where substitution may be applied. In Figure 3.2 this is due to the small alphabet and Figure 3.3 there are no nullable reduction paths and most states have a follow set that contains all possible first tokens and precede sets that contain all possible last tokens. We have however verified that substitution is possible on larger grammars such as SQLite.

Deletion We may delete a reduction path r, of the form $a \rightsquigarrow ... \rightsquigarrow b$, in a traversal if r is not nullable and $F(a) \cap F(b) = \varnothing$. This will result in gaps in the top-of-stack context required after b to produce a negative test case. This is caused by the tokens following b not being able to occur after a in any valid word in $L(G)$.

For example, we may delete the reduction path $r = v_2 \rightarrow_{num} v_4 \dashrightarrow_{expr/1} v_2 \rightarrow_{expr} v_7$ over the LR-graph in Figure 3.3 from any path which contains it since r is not nullable and $F(v_2) \cap F(v_7) = \varnothing$;

Prefix Cutting If the target vertex v of an edge is not an almost-accepting state (i.e., $\neg AA(v)$ we can guarantee that the prefix which has reached v it is not a valid word for the grammar. Cutting therefore terminates the word whenever it reaches a vertex that is not a almost-accepting state. This equivalent to popping all elements off the stack and accepting the invalid prefix path as a valid path by the empty stack.

For example, in Figure 3.2(a) we can cut after reaching vertex v_2 since every path leaving from this vertex passes through $v_3 \rightarrow_{\texttt{]}} v_4$, so $\neg AA(v_2)$. However, we cannot cut after reaching the vertex v_4 since the path $v_4 \dashrightarrow_{d/0} v_4 \rightarrow_d v_5 \dashrightarrow_{d/4} v_0 \rightarrow_d v_1 \rightarrow_{\texttt{\$end}} v_{acc}$ can reach the end of the input and the accept state without any terminal push edge.

3.4.3 Comparison with rule mutations

Rule mutations [37] have many similarities to our proposed edge and stack mutations. Edge mutations are very similar to rule mutations on terminal symbols and stack mutations are similar to rule mutations on non-terminal symbols.

However, there is a key difference in where rule mutations may be applied in comparison to our edge and stack mutations. Since rule mutations mutate the rules of a context-free grammar, it does not have knowledge of the context in which a mutation is being applied. This means that rule mutations must be conservatively applied in order to guarantee negative test cases.

By using the LR-graph, constructed from the LR-automaton, we gain extra context information encoded directly into the automaton. We also have the advantage of explicitly encoding 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, when compared to rule mutations. This is advantageous as the more contexts we can test, the more granular our testing of the SUT.

For example, the expression grammar in Figure 3.4 can not have a `-` symbol inserted before `id` in the rule $f \rightarrow$ `id`, since the `-` symbol may precede the `id` symbol in some applications of the rule. However, in an LR-graph different rule applications can result in different paths. This allows us to insert the – symbol in those paths where it should not be possible, such as the path resulting in the word `id`, turning it into the invalid word `- id`.

We can see this in our results (see page 69) when comparing the fault detecting capabilities of different positive coverage criteria and negative mutations and the number of test cases being produced.

3.5 Conclusion

We have thus described how to construct an LR-graph and which conditions apply to edge validity in a path. We have also described two methods of generating positive test suites by covering the edges of the LR-graph as well as mutations to generate negative test suites from these positive test suites.

Chapter 4

Implementation

We have implemented the algorithms developed in this thesis using Python 3. Both the breadth-first traversal and pop-edge coverage algorithms' implementation closely follow their pseudocode presentations. However, in some cases we made minor optimizations in order to improve the time it takes to generate a test suite. We used HYacc [42], a Yacc variant, to produce an initial LR(0)-automaton. We then used the automaton to construct an LR-graph. Figure 4.1 provides an overview of the data flow through our system. Detailed command-line arguments can also be found in Appendix A. In the following we describe the components in more detail.

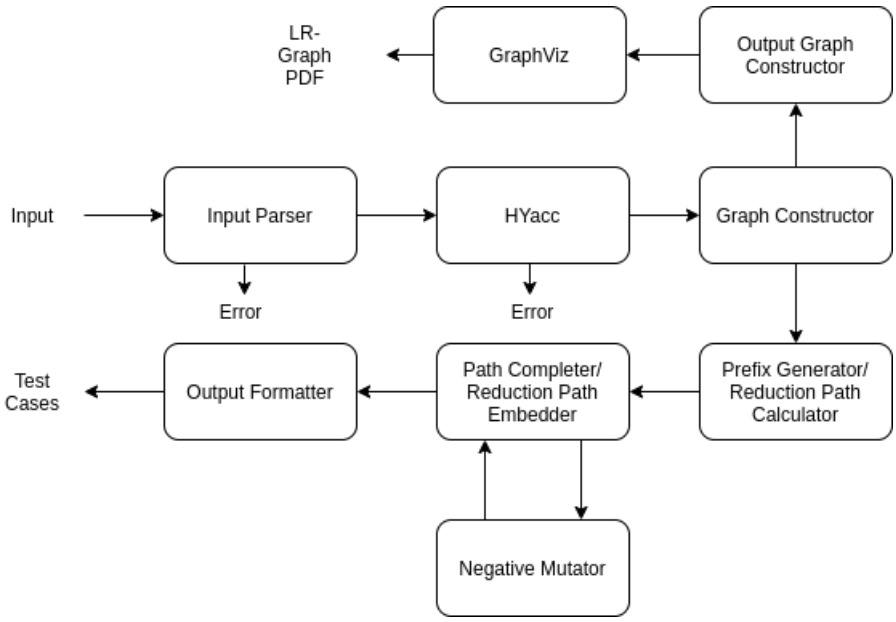


Figure 4.1: Program Data Flow

Input Parser Before passing the input grammar to HYacc we first confirm that the file format is approximately correct, however we do not check the grammar yet. We also confirm that different combinations of arguments that may be passed to the program are compatible.

HYacc HYacc parses the grammar to confirm the grammar contains all necessary information. It then produces an LR-automaton as text file. We ignore any warning about conflicts since these will be dealt with in the next phase. However, we display information regarding conflicts so that the graph structure that is produced can be understood better.

Graph Constructor We take the LR-automaton produced by HYacc and extend it into an LR-graph. This step 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.

We cache the following information in the nodes and edges of the graph to speed up computation:

1. Reduction paths for pop-edges (see Section 3.1)
2. Shallowest embedding of pop edge into a valid path (see Section 2.2)
3. Inbound edges for a node

Converting LR-graph to PDF In some instances we may want to look at the LR-graph. For this we first construct an output file from the LR-graph in the GraphViz [1] graph format. We then use GraphViz to compile a PDF of the LR-graph.

Generating Test Suites For the breadth-first traversal algorithm the implementation follows the multi-phase approach of the algorithm, first generating prefixes in the flooding phase and then completing these paths. The pop-edge coverage algorithm is implemented slightly differently than the pseudocode in Algorithm 7. Instead of iterating over all pop edges, calculating reduction paths and embeddings in one step, we first calculate all reduction paths and then embed them in a separate phase.

If the flag for generating negative test suites is set, complete paths are passed to the mutation engine which then generates mutated paths, depending on the mutation criteria selected.

Formatting Output Test cases are extracted from the set of paths that are generated by our algorithm. In order to accommodate multiple SUTs we also allow writing all test cases to a single file or writing them to individual files.

Optimizations

We optimized both our proposed algorithms using the methods we describe below.

Optimizing Breadth-First Traversal One of the largest performance problems for the breadth-first traversal algorithm stems from the high branching factor in the traversal when no loop detection can be used (see page 32). This leads to rapid growth in the length of the queue which in turn leads to rapid growth in execution time. Specifically, if a node in the loop has n valid outgoing edges that loop back to it, then it will contribute n^2 paths to the queue. When this happens for many nodes in a loop it can drastically increase the amount of time to flood the graph and to complete paths.

Even though traversing certain edges multiple times to construct one valid path is unavoidable, many edges do not require multiple traversals to be covered by a valid path. Therefore, we can improve performance of the algorithm by avoiding redundant edges (those edges we have traversed before) whenever possible. In order to do this we used a priority queue.

For the flooding phase, this means prioritizing edges that have been covered the least amount of times; in the path completion phase, this means prioritizing edges that are "closer" to the accept state. This approach leads to a less redundant exploration of the graph, in some cases being capable of covering a LR-graph in linear time (in terms of the number of edges), even if it contained loops, whereas the unmodified algorithm could only cover a LR-graph in linear times if the underlying automaton resembled a directed acyclic graph. However, this optimization could still run into problems where the structure of the graph results in many edges with the same priority, essentially reducing the priority queue to the simple queue of the unmodified algorithm.

Optimizing Pop-Edge Coverage This algorithm is inherently more efficient than the breadth-first traversal algorithm since many of the required computational steps, such as determining the reduction path, can be done in constant time since the information required can be encoded in the LR-graph. This does require storing more information for each pop-edge, specifically

which paths of length $|\gamma|$, for a pop edge $u \dashrightarrow_{A/|\gamma|} v$, lead up to this pop-edge. This enables determining all reduction paths in linear time in terms of the number of edges and nodes in the graph.

The embedding step, however, results in a large amount of redundant computation if no caching data-structure is used and in turn results in polynomial running time of this step. This is due to the inefficiency of calculating embeddings of a reduction path into other reduction paths which are then further embedded until a valid path is formed, only to have to calculate the embeddings for these reduction paths again individually. We can optimize this by constructing an "embedding tree", in linear time, out of the reduction paths. 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 for each reduction path.

The optimized pop-edge coverage algorithm compares favourably in terms of time taken to generate a test suite when compared with other *state of the art* algorithms used during our experimental evaluations. In terms of wall-clock time, this algorithm always takes less time to produce a test suite than other algorithms [24, 29, 47], implemented in Prolog, that produces similar sized test suites. It also scales much better than the breadth-first traversal algorithm, computing the entire test suite for SQLite in a few seconds, whereas the breath-first traversal algorithm could not finish computing test suites for larger expression grammars (see Figure 3.4) in a reasonable amount of time.

Chapter 5

Evaluation

With our evaluation we specifically answers the following research questions.

R1. How does the coverage achieved by the automaton-based algorithms and the size of the test suites produced by them compare to the current state-of-the-art algorithms?

R2. Are there any structural differences in test cases produced by the automaton-based algorithms compared to similar sized test suites by the current state-of-the-art algorithms?

R3. How well do the edge and stack mutations reveal errors in a system under test compared to a positive test suite or test suite produced by current rule mutation algorithms?

The main goal of our experiments is to quantitatively compare the proposed automaton-based methods for test case generation with the current *state of the art* grammar-based test suite generation algorithms. In our experiments we specifically use the pop-edge coverage method since the breadth-first traversal algorithm did not scale to the input grammars considered here. In order to do this for positive test suites, we look at coverage achieved over systems under test for three different medium to large input grammars (AMPL [46], SQLite [6], Go [5]), relative to the size of the test suite. We also investigate how our proposed algorithm compares to other algorithms in terms of relative test suite size over a number of different grammars.

For negative test suite generation we investigate the ability of the different mutations to reveal faults in a SUT, relative to those revealed by positive test suites as well grammar-based rule mutations [35]. We do this over multiple systems that have known faults and have been previously used in experiments by Raselimo et al. [36] to investigate fault localization and grammar repair.

During our initial evaluation we encountered difficulties with making meaningful comparisons between our algorithms and grammar-based algorithms and being able to accurately answer the first research question. This was due to slightly different equivalent choices in the implementation of our algorithms than the implementation of the grammar-based algorithms which led to vastly different coverage results, which we call bias. We therefore also evaluate the effect of bias on all the test suites we consider and this leads us to our final research question.

R4. What is the effect of bias on test suites generated by various different coverage criteria and algorithms?

5.1 Bias in Grammar-Based Testing

In statistics, a *sample statistic* (i.e., any quantity that is computed by aggregating values in a sample of an underlying population, such as the sample mean or variance) is *biased* if it is systematically different from the corresponding quantity of the entire population [12]. Statistical bias has many forms, including *modeling bias* (e.g., using a normal distribution to model a population with long tails), *observer bias* (e.g., the Hawthorne-effect [10]), or *sampling bias* (e.g., selecting participants with a landline phone number), although the latter is the most prevalent. It occurs (e.g., as *self-selection*, *exclusion*, or *survivorship bias*) whenever the sample is collected in a way such that specific groups of the underlying population are over-represented in the sample.

We focus on sampling bias here. More specifically, we consider as population the set of all test suites satisfying a chosen coverage criterion and as sample a test suite constructed for a specific algorithm with specific fixed choices. The biased sample statistic that we are observing is the code coverage the generated test suite achieves over a SUT. We focus on the two types of fixed choices described below. Note that we do not consider any bias caused by different grammar variation and preprocessing steps (e.g., EBNF operator elimination) because these are under the full control of the user.

Embedding Bias The generic cover algorithm and its variants use minimal derivations in the embedding (line 3) and completion (line 5) steps, and in some of the criteria. We compare the effects of shortest yield and shallowest derivations here. Our algorithms, especially the pop-edge coverage algorithm, implicitly resemble shallowest derivations due to shorter paths leading to a

natural decrease in embedding depth of reduction paths. Shortest yield derivations $\Rightarrow^*_{\leq}$ induce a very strong bias towards ϵ-productions, leaving optional elements of the grammar exercised much less often. Shallowest derivations $\Rightarrow^*_{\sqsubseteq}$ induce a bias against deeply nested tests, leading to structurally simpler tests. This often corresponds to the shortest yield derivations, but not necessarily so; for example, in G_{toy} (see Figure 1.1) the shortest yield embedding of `num` is through a `return`-statement, while the shallowest embeddings are through an `if-`, `while-`, or assignment statement. Note that the user can in principle deliberately choose the minimal embedding, taking these biases into account, although the choice is often hard-coded in the algorithm's implementation.

Regardless of the choice of embedding, it will lead to sampling bias since there will be an increased preference of sentences with the characteristics of the embedding in the test suite.

Equivalent Choice Bias Regardless of the chosen minimal derivation method $\Rightarrow^*_{\leq}$, the algorithms, whether grammar-based or based on the LR-graph, need at some points to choose between a set of equivalent (wrt. $\Rightarrow^*_{\leq}$) options. If these choices are made systematically (e.g., based on textual rule order or edge order in the LR-graph) rather than randomly, this introduces a classic sampling bias, and the generated test suite will be biased accordingly. If the choices are functionally determined by the grammar or LR-graph structure, the user could in principle control the bias by reformulating the grammar, but this becomes infeasible in practice.

5.2 Experimental Setup

5.2.1 Design

We are comparing the performance of the automaton-based algorithm to the current state-of-the-art coverage-based algorithms. We therefore compare algorithms using the metrics of test suite size, line and branch coverage by executing the test suites produced by these algorithms over the same systems under test. We also use line and branch coverage to measure the bias effects of different grammar-based test suite construction algorithms.

Since we are comparing different algorithms, we carefully consider bias as described in Section 5.1. In order to avoid *equivalent choice* bias in the grammar-based algorithm, we generated a hundred different test suites for each coverage criteria, resolving equivalent choices in the grammar-based algorithms with a random seed and shuffling of rule order. For the LR-graph based algorithms we also generate a hundred different test suites and resolve

equivalent edge choices using a random seed. We also consider embedding bias in other algorithms that we are comparing our proposed algorithms to, by generating 100 variants using shallowest embeddings and another 100 variants with shortest yield derivations. We then compare coverage for both these sets of test suites with the coverage achieved by the automaton-based methods independently. We do not generate two different test suites in this fashion for the automaton-based method since it is based on paths over the LR-graph so no such choice exists in this context (although in practice covering these paths over the LR-graph results in test cases that somewhat resembles those generated by shallowest embeddings).

5.2.2 AMPL

AMPL is a small programming language ($|N| = 45, |T| = 48, |P| = 90$) used for teaching a computer architecture course. It has previously been used by van Heerden et al. [46] in their experiments. We have also obtained and used the 61 student compilers that were used in these experiments. These compilers are written in C and are about 1300 LoC.

We ran each test case in their respective test suites over the individual compilers separately. We then used gcov to measure cumulative line and branch coverage for each test suite and compiler combination. The average coverage for a test suite is the average coverage it achieves over the 61 student compilers. The average coverage over each of the 100 test suite variants is then calculated to determine the average coverage for a coverage criterion. This is then used as the primary comparative metric between different coverage criteria and algorithms. We also consider the coverage achieved by merging all 100 test suite variants (this is the union of all 100 test suites for a coverage criterion). We also compute the union of this merged test suite with the merged test suite for the automaton-based method for each coverage criterion.

5.2.3 SQLite

SQLite is an SQL database engine written in C and is widely used as an *on-device* datastore for mobile and IOT applications. We used version 3.8.x of the SQLite grammar ($|N| = 201, |T| = 155, |P| = 586$) from the ANTLR grammar repo [4] as input grammar and compiled the C executable for SQLite (v3.36.0) from sources, which comprised 234229 LoC in total.

We executed each SQL statement individually for each test suite. Then we used gcov to measure the cumulative branch and line coverage of all test cases in a test suite, to determine the coverage of the test suite as a whole. We then look at the average coverage over all 100 variants as well as the

coverage achieved when merging all 100 variants. Unlike the experiments with AMPL, we could not measure the coverage achieved by computing the union of these merged test suites with the merged test suite for the automaton-based method, since it was not possible to compute these values in a reasonable time due to the extremely large sizes of the test suites obtained from the union. However, the size of these unioned test suites still allow us to show important structural attributes of test cases generated by the automaton-based method and how it differs from other algorithms it is evaluated against here.

5.2.4 Go

Go is a popular programming language used to build memory safe applications. In our experiments we use the version of the Go grammar ($|N| = 159, |T| = 84, |P| = 324$) found in the ANTLR grammar repo [4]. Programs written in Go can be compiled with GCC by using a frontend for GCC known as GCCGo [5]. We use GCCGo as our system-under-test. Specifically, we focus on the parser module of GCCGo in order to confirm that the conclusion that we draw from our results are not only due to semantic actions executed later in the SUT. We compile the GCC executable with the GCCGo frontend from sources, with the relevant parsing module being 5972 LoC in length.

We ran GCC with the GCCGo front-end over each test case in a test suite individually. We then use gcov to measure the cumulative branch and line coverage for the test suite over the parser module. As before, we also measure the coverage achieved by the union of all test suites for each coverage criteria as well as the union of these test suites with the union of the automaton-based algorithm test suites.

5.2.5 SUTs for Negative Test Suites

To test the ability of our proposed negative coverage criteria we used two grammars and three student parsers. Both grammars used are small programming languages used in a compiler course, similar to AMPL. The first is known as SIMPL ($|N| = 47, |T| = 47, |P| = 93$) and the second as Niklaas ($|N| = 56, |T| = 44, |P| = 111$). We used one student parser for SIMPL and two for Niklaas. The student parsers were constructed using jflex (version 1.8.2) and Java CUP (version 11b). We use these systems as opposed to the ones used for comparing positive test suites since we require systems with known errors in order to compare the error revealing characteristics of negative test suites.

We produced test cases from a "golden grammar" that we know is correct. We then ran test cases over the respective parsers individually. For positive

test suites a test cases is said to be failing if it is rejected by the system under tests, a false negative. Conversely, for a negative test suite, a test case is said to be failing if it is accepted by the system under test, a false positive. We divide the known errors between positive and negative test suites, since positive and negative test cases identify different types of errors.

We use spectrum-based fault localization techniques [35] in order to evaluate the effectiveness of different test suites at revealing errors in a SUT. Specifically, we use a scoring system by Ochiai [32] to assign a suspiciousness score to each rule in a SUT, based on whether these rules were exercised by passing or failing test cases in a test suite. This allows us to rank all suspicious rules and determine how many rules must be examined before the true, known errors are revealed. From this we calculate the amount of wasted effort to find known errors, as the percentage of rules that must be investigated before the known errors are found.

5.3 Results

5.3.1 Coverage Achieved

Tables 5.1 - 5.3 show coverage data achieved by different algorithms and coverage criteria for the three SUTs. From this data we can see that the coverage achieved by the LR algorithm favourably compares to the coverage achieved by other coverage criteria which results in much larger test suites. As a baseline, our proposed algorithm always achieved better coverage than algorithms which produce smaller test suites, like *pll* and *rule*. If we consider Table 5.1, we can see that the automaton-based algorithm, with an average line coverage of 64.4%, even out performs larger test suites like bfs_2 for both shortest yield and shallowest embedding, which achieves an average line coverage of 61.6% and 63.8% respectively, and step3 for shallowest embeddings which achieves line coverage of 62.6%.

A Mann-Whitney U test ($\rho < 0.05$) confirms that the coverage distribution of the LR algorithm is significantly different from all the other algorithms considered in Table 5.1. It is also significantly different from all but shallowest embedding bfs_2 and shortest yield *deriv* in Table 5.2. This is reflected by the fact the average line coverage for both these collections of test suites is approximately equal at 26.1% and 26.0% respectively. Table 5.3 also shows similar results with the coverage distribution being significantly different for the LR algorithm for all but *deriv* and $step_4$. This means that in most instances, other than those we have highlighted above, if the LR algorithm achieves higher coverage we can conclusively say that it performed better

over the SUT than another coverage criterion. In the instances that we have highlighted, a higher or lower coverage percentage does not conclusively allow us to state whether the LR algorithm performed better or worse in terms of coverage, since the difference in coverage distribution is not significant enough.

For coverage over GCCGo we can see that the automaton-based algorithm achieves equivalent coverage to both $step_4$ and *deriv* at between 70.2% and 70.8% on average for these algorithms. However, both of these algorithms produced significantly larger test suites at 4320.8 and 5773.4 for shallowest embedding and 4424.3 and 5942.4 for shortest yield respectively, as compared to the *lr* algorithm at 3486.3 tests. This confirms that the relative performance observed over whole systems for both AMPL and SQLite holds true, even when only the front-end of the system is considered as we did with GCCGo.

5.3.2 Test Suite Sizes

Table 5.4 shows the minimum, maximum and average number of test cases per test suite for a multitude of algorithms and coverage criteria and many different grammars. Here we do show some data for the general traversal algorithm in the lr* column. From this table it becomes immediately apparent that the pop-edge coverage based algorithm performs considerably better than the general traversal algorithm in terms of being able deal with larger grammars where the amount of possible valid paths in the LR-graph rapidly increase. From the Dyck variants we can see the new algorithm produces test suites about half the size as the previous algorithm and for the expression grammar for Figure 3.3 we can see that it is multiple orders of magnitude smaller. It is also able to produce test suites for much larger grammars where the general traversal algorithm failed to compute in a reasonable time.

We can also observe that the pop-edge coverage algorithm always results in larger test suites than both the *pll* and *rule* algorithms. For the majority of grammars considered the *lr* algorithm produced a test suite that was larger than bfs_1 and bfs_2, except in the case of the expression grammar in Figure 3.3 where the test suite produced was even smaller than that produced by bfs_1. However, for larger real-world grammars like Go and CSS3 the size of the lr test suite was larger than both bfs_1 and bfs_2. Both $step_3$ and $step_4$ generally produced larger test suites than the *lr* algorithm. However, for Go and CSS3 it produced a test suite which size fell in between that of $step_3$ and $step_4$. For some grammars like SQLite the *lr* algorithm produced a test suite considerably larger than that produced by *deriv*. However, in the vast majority of cases it produced test suites that were multiple times smaller than that produced by *deriv*.

From this data, as well as the data discussed in Section 5.3.1 we can answer the first research question.

R1. The general traversal algorithm simply did not scale to larger grammars in a way that the coverage it achieves could be meaningfully evaluated. From Table 5.4 we have shown how the size of test suites produced by this algorithm grows much more rapidly than the pop-edge coverage algorithm, which has the same coverage criterion of coverage over all edges of the LR-graph. We therefore consider the coverage and the size data of the pop-edge coverage algorithm when comparing coverage of the LR-graph's edge with other, grammar-based, coverage criteria.
The coverage achieved by the LR algorithm was always better than other coverage criteria that produce smaller test suites, and often produced better or equivalent coverage than that achieved by coverage criteria that produced larger test suites. An exception to this is *deriv* that achieved better coverage to the LR algorithm in almost all cases, with a very small test suite over SQLite. However, *deriv* often produces much larger test suites than the LR algorithm, as shown in Table 5.4.
We can thus conclude that the LR algorithm in general compares favorably with state-of-the-art algorithms and coverage criteria that produce similar sized test suites, often producing slightly smaller test suites for equivalent or better coverage over the SUT.

expr → *simple relopsimpleoptional*
relopsimpleoptional → *relop simple* | ϵ
relop → `=` | `>=` | `>` | `<=` | `/=`
optionalminus → `-` | ϵ
addoptermstar → *addop term addoptermstar* | ϵ
simple → *optionalminus term addoptermstar*
addop → `-` | `or` | `+`
mulopfactorstar → *mulop factor mulopfactorstar* | ϵ
term → *factor mulopfactorstar*
mulop → `and` | `/` | `*` | `rem`
simpleexprstaroptional → `[` *simple* `]`
| `(` *expr exprstar* `)` | ϵ
exprstar → `,` *expr exprstar* | ϵ
factor → `id` *simpleexprstaroptional* | `num` | `(` *expr* `)`
| `not` *factor* | `true` | `false`

Figure 5.1: BNF for AMPL expression sub-grammar

```
false * num * num
false * num / num
false * num and num
false * num rem num
false * true * true
false * true / true
false * true and true
false * true rem true
false + num + num
false + num - num
false + num or num
false + true + true
false + true - true
false + true or true
false - num + num
false - num - num
false - num or num
false - true + true
false - true - true
false - true or true
false / num * num
false / num / num
false / num and num
false / num rem num
false / true * true
false / true / true
false / true and true
false / true rem true
false and num * num
false and num / num
false and num and num
false and num rem num
false or num + num
false or num - num
false or num or num
false or true + true
false or true - true
false or true or true
false rem num * num
false rem num / num
false rem num and num
false rem num rem num
false rem true * true
false rem true / true
false rem true and true
false rem true rem true
num * false * false
num * false / false
num * false and false
num * false rem false
num * true * true
num * true / true
num * true and true
num * true rem true
num + false + false
num + false - false
num + false or false
num + true + true
num + true - true
num + true or true
num - false + false
num - false - false
num - false or false
num - true + true
num - true - true
num - true or true
num / false * false
num / false / false
num / false and false
num / false rem false
num / true * true
num / true / true
num / true and true
num / true rem true
num and false * false
num and false / false
num and false and false
num and false rem false
num and true * true
num and true / true
num and true and true
num and true rem true
num or false + false
num or false - false
num or false or false
num or true + true
num or true - true
num or true or true
num rem false * false
num rem false / false
num rem false and false
num rem false rem false
num rem true * true
num rem true / true
num rem true and true
num rem true rem true
true * false * false
true * false / false
true * false and false
true * false rem false
true * num * num
true * num / num
true * num and num
true * num rem num
true + false + false
true + false - false
true + false or false
true - false + false
true - false - false
true - false or false
true - num + num
true - num - num
true - num or num
true / false * false
true / false / false
true / false and false
true / false rem false
true / num * num
true / num / num
true / num and num
true / num  rem  num
true and false * false
true and false / false
true and false and false
true and false rem false
true and num * num
true and num / num
true and num and num
true and num rem num
true or false + false
true or false - false
true or false or false
true or num + num
true or num - num
true or num or num
true rem false * false
true rem false / false
true rem false and false
true rem false rem false
true rem num * num
true rem num / num
true rem num and num
true rem num rem num
```

Figure 5.2: Test cases exclusive to LR algorithm for AMPL expression grammar in Figure 5.1

5.3.3 Structure of Test Cases

If we consider Tables 5.1, 5.2 and 5.3 we can see from the size of the test suites comprised of the union of merged automaton-based test suites and other merged suites that a significant amount of the test cases present in the automaton-based algorithm's test suites are not found in the other coverage criteria's test suites. For example, the largest overlap in Table 5.1 is with shallowest embedding $step_4$ test suites of 55%. and in Table 5.2 shallowest yield bfs_2 has an overlap of 13%.

From Tables 5.1 and 5.2 we can see that there is generally a larger overlap between the automaton-based algorithm's test suite and shallowest embeddings. This is likely due to the manner in which the structure of the automaton is covered roughly resembling a shallowest embedding. However, it is important to note that, even with a larger overlap, a significant amount of test cases produced by the LR algorithm is not produced by any other algorithm or coverage criterion that we considered.

To illustrate the structure of the unique test cases that are only produced by the automaton-based algorithm, we used the expression sub-grammar from AMPL (see Figure 5.1) and again generated 100 different variants for each algorithm. From this we then extracted all the test cases that were unique to the automaton-based algorithm. These test cases can be seen in Figure 5.2.

This set of test cases show that majority of the unique test cases are ones that cover deeply nested structures in the grammar. This is shown by at least three embeddings of the *simple* token chained together, which requires multiple layers of embeddings to achieve, given the recursive nature of the expression grammar. Some algorithms like *bfs* will eventually overlap completely with the test suite generated with the automaton-based method. We generated bfs_3 which resulted in a superset of the *lr* suite. However, this came at a rather significant cost of test suites which on average contained 3605.8 test cases as compared to 53.0 for the *lr* test suites. This leads to our answer for the second research question.

R2. From the data in Tables 5.1 - 5.3 we can clearly see that the LR algorithm produces many test cases not found in test suites of similar size produced by state-of-the-art algorithms. Figure 5.2 shows that this is due to the coverage of more deeply nested grammar structures by the LR algorithm that are only covered by state-of-the-art coverage criteria that produce considerably larger test suites. We can therefore conclude that the LR algorithm produces test suites that are more effective at covering deeply nested grammar structures than algorithms that produce test suites of a similar size.

5.3.4 Fault Detection Capabilities

All test suites, except for those constructed by prefix cutting, revealed all of the known errors in the SUTs, with the set of known errors differing for positive and negative test suites. Therefore, in order to evaluate the efficacy of the edge and stack mutations, we compare the number of suspicious rules, the wasted effort in identifying the known errors from suspiciousness scores, the number of false positives or negatives and the size of each test suite with unmutated, positive test suites as well as grammar-based rule mutation test suites.

Positive Test Suite From Table 5.5 we can see that both the automaton-based algorithm and the context-dependent rule coverage algorithm don't reveal many suspicious rules and have very few failing tests, as compared to the negative test suites, especially those produced by edge and stack mutations. They also have a much lower amount of wasted effort, since a failing positive test case terminates in a specific location, as opposed to negative test cases which fail if the whole test case is accepted by the SUT. It is important to note that the known errors uncovered by these positive test suites are inherently different from the negative test suites since it occurs when the parser is too strict, as opposed to the faults uncovered by negative test suites which occur when the parser does not apply rules strictly enough. The latter is often more common in practice. The positive test suites are also considerably smaller in practice.

Rule Mutations Grammar-based rule mutations led to more suspicious rules than the positive automaton-based test suite, but generally still less than the majority of the edge and stack mutations, other than edge deletion and substitution for Niklaas 2 and prefix cutting. Grammar-based rule mutations generated test suites that were considerably smaller than all stack and edge mutations other than stack and edge deletion and prefix cutting.

Edge Mutations In terms of test suite size, the insertion and substitution mutations produced the largest test suites for both Niklaas and SIMPL. However, these two mutations also led to the most suspicious rules, especially the insertion mutation. We can observe that, for both Niklaas parsers, edge substitution led to less failing tests than edge insertion, but the opposite is true for the SIMPL parser. This is a result of the substitution mutation effectively being a delete mutation followed by an insertion mutation at the location of the deletion. Depending on the structure of the LR-graph, this may result in situations where the substitution acts more like a deletion,

which generally has less possible mutation locations which in turn results in smaller test suites, due to lack of possible tokens to insert at the location of the substitution, which happens with the Niklaas test suites. However, in the optimal situation it acts as a unique combination of deletion and insertion to allow for more mutations and more failing tests, as is the case for the SIMPL.

We can see that the deletion mutation reveals less failing tests and suspicious rules than the insertion and substitution mutation. It does however come at a much reduced cost in terms of test suite size. This is due to grammars with a lot of optional token sequences resulting in less locations where a deletion would cause an error. There are also not multiple mutation options at a single mutation location, as is the case for substitution or insertion.

Stack Mutations Stack mutations produce very similar sized or smaller test suites than their corresponding edge mutations. This is because the increased possible mutation locations are being counteracted by there generally being less reduction paths in the LR-graph than push edges. We can see that, although the insertion and substitution stack mutations produce very similar sized test suites to their edge mutation counterparts, the stack deletion mutation produces test suites many orders of magnitude smaller than the edge deletion mutation. This is due to there being less locations in which reduction paths can be deleted as compared to push edges.

Stack mutations over the three systems generally led to a similar number of suspicious rules as edge mutations. However, the key difference is the number of false positives produced by stack mutations. For example, the stack insertion mutation for Niklaas produced about three times more false positives than the edge insertion mutation and about five times more for SIMPL. This increase in false positives produced is observed for the substitution and deletion mutations as well.

For all three parsers considered in this evaluation the prefix cutting mutation did not detect any false positives, with test suite sizes comparable to deletion. This does not mean that it would never detect faults, rather that the type of fault it attempts to find, invalid termination of input, was not present in the systems that we considered. For example, prefix cutting would reveal the Cloudflare bug discussed in Chapter 1.

From this data we may answer our next research question.

R3. All types of edge and stack mutations, with the exception of prefix cutting, found the known errors in the SUTs considered, the same as rule mutations, and found different types of errors to those identified by the positive test suites, as expected. Edge and stack mutations also revealed the known errors in a greater number of contexts than rule mutations, as shown by the number of rules marked as suspicious. However, this did come at the cost of more wasted effort in identifying the true cause of the known errors using the suspiciousness scores assigned to the rules. This leads us to conclude that, although having more examples of contexts in which an error occurs can be good, in larger systems it may be difficult to identify the true cause of an error. However, the SUTs considered all contained errors that could be revealed by fairly simple mutations and do not fully showcase the ability of the edge and stack mutation methods to use a greater amount of context information and thereby test for errors in more locations than is possible using only rule mutations.

5.3.5 Equivalent Choice Bias

Figures 5.3 and 5.4 and Tables 5.1 - 5.3 show a high variance in the code coverage under the different re-ordered grammar variants, especially for shallowest embeddings, and even though this generally decreases for larger test suites such as *deriv*, $step_3$, or bfs_2, it remains significant. We also see variance in coverage for the different LR variants. For example, if we consider, for shallowest embeddings, the *best* line coverage achieved using *rule* coverage over SQLite (26.2%), we see that this outperforms the *average* line coverage of all other criteria but $step_3$ (26.5%)—which is almost two orders of magnitude bigger in terms of average test suite size.

Hence, any comparative study of such criteria and algorithms can be significantly skewed if the effect of bias is not taken properly into consideration; this is particularly relevant when different implementations are compared, where not all bias-inducing choices are necessarily known.

Table 5.3 shows similar variance in coverage when compared to Tables 5.1 and 5.2, even though only coverage over the parser was considered for GCCGo, unlike AMPL and SQLite where coverage was measured over the whole system. This shows us that the observed variance is not simply due to semantic actions being exercised later on in a SUT but also affects coverage observed over purely the recognizer component of a SUT.

Figure 5.3 and 5.4 also shows that the coverage achieved is not generally normally distributed, with clusters often occurring near the minimum and

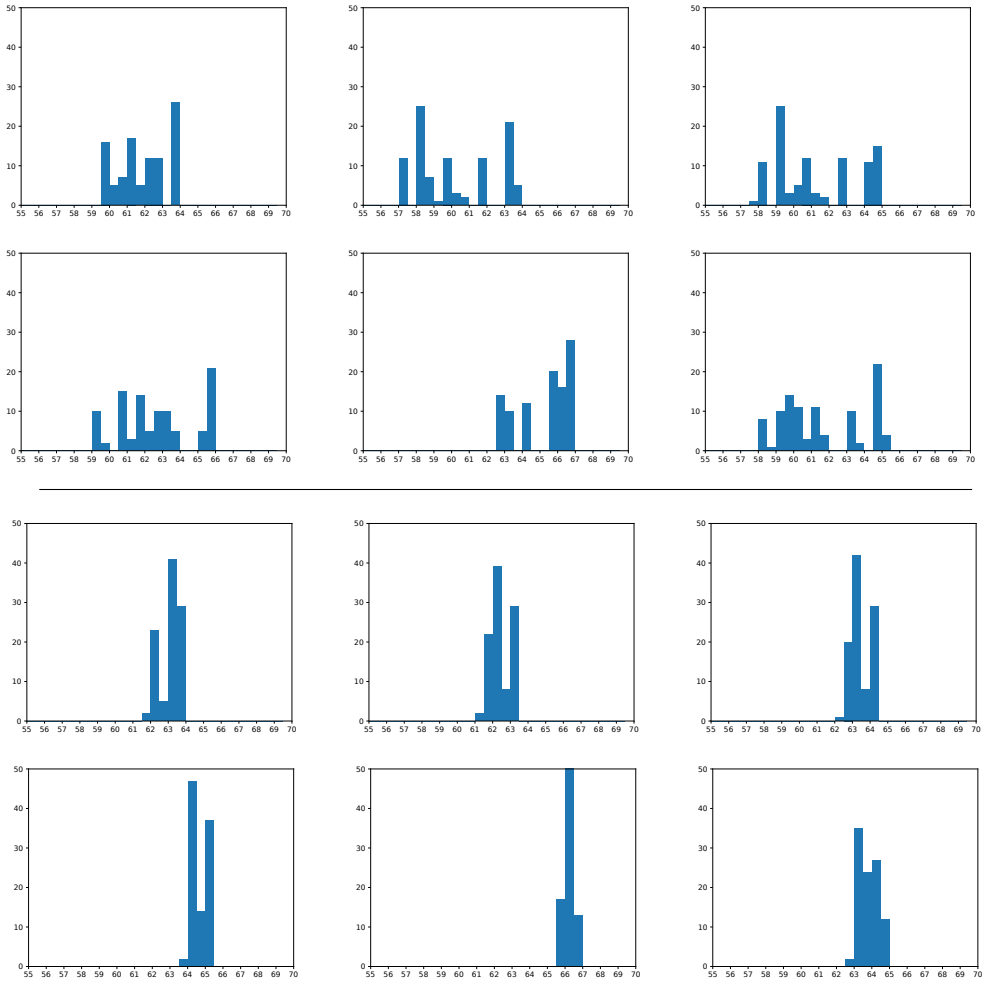


Figure 5.3: Statement coverage distribution for AMPL. Shallowest embedding results are shown above the line and shortest yield embedding results are shown below the line. From left to right, top to bottom, for each section, we have *pll*, *rule*, *cdrc*, $step_3$, *deriv*, and bfs_2 coverage. Each chart shows the observed statement coverage on the x-axis and the corresponding number of test suites on the y-axis.

maximum measured coverage. Test suites are thus more likely to achieve coverage near the extremes than near the average, reinforcing the bias effect.

5.3.6 Embedding Bias

Unlike for equivalent choice bias, the results for the different embeddings show noticeable coverage differences between AMPL, SQLite and GCCGo.

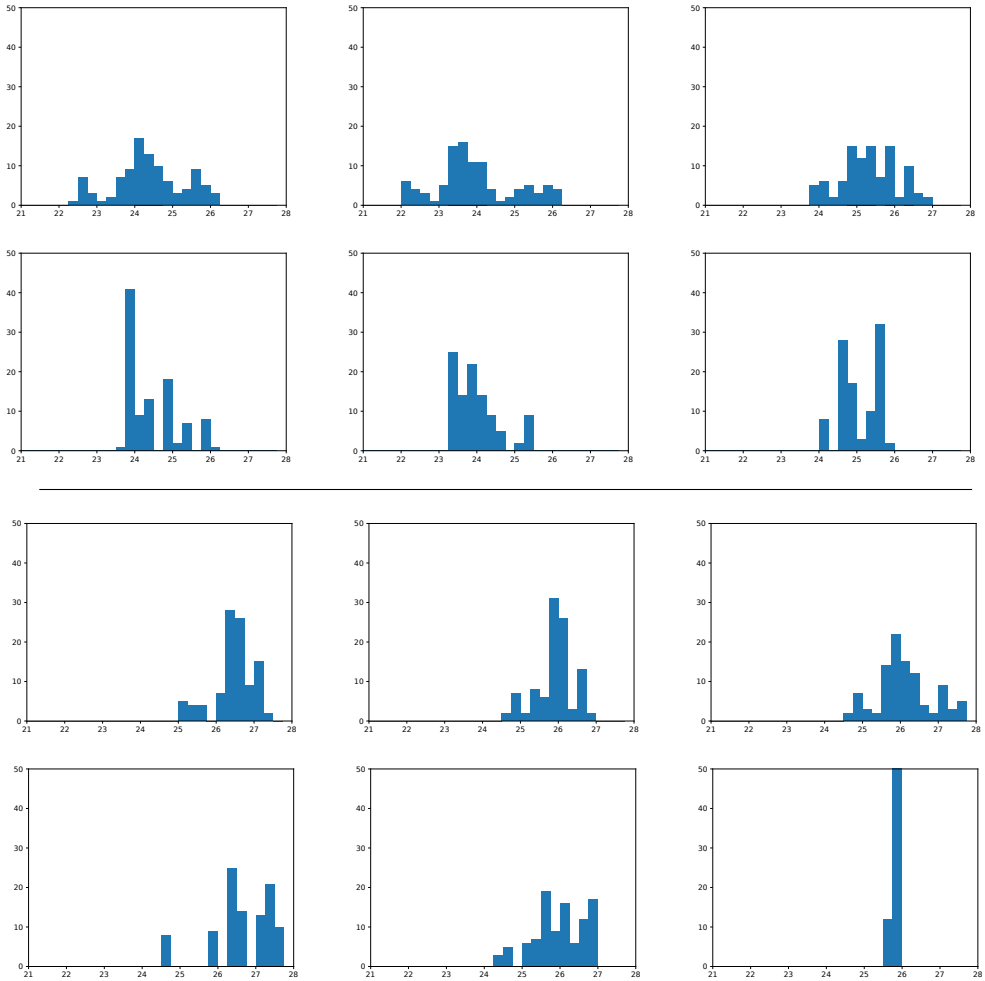


Figure 5.4: Statement coverage distribution for SQLite. Shallowest embedding results are shown above the line and shortest yield embedding results are shown below the line. From left to right, top to bottom, for each section, we have *pll*, *rule*, *cdrc*, $step_3$, *deriv*, and bfs_2 coverage. Each chart shows the observed statement coverage on the x-axis and the corresponding number of test suites on the y-axis.

For AMPL, we can see in Table 5.1 a large difference in the code coverage achieved between shallowest embeddings and shortest yield embeddings. The latter leads to a substantially higher average code coverage for the different criteria (0.9–2.3%-points for line coverage, with 2.0%-points on average, and 1.6–4.0%-points for branch coverage, with 3.4%-points on average) than the former, with a substantially smaller variance at the same time. However, the best variants under both embeddings achieve roughly the same coverage for

all criteria.

For SQLite, the average code coverage achieved is approximately similar across all criteria (see Table 5.2), but Figure 5.4 shows that the coverage distributions for shallowest and shortest yield embeddings are very different for most criteria, and a two-tailed Mann-Whitney U test ($\rho < 0.05$) confirms that the difference in distributions is significant for each coverage criterion, with the exception of *rule*-coverage. We observe similar results for GCCGo. Since shallowest embeddings have a large variance, several variants should be run to achieve reliable results, while shortest yield embeddings are more stable and should be preferred if the budget only allows a single run.

In our experiments none of the two embedding algorithm appears to universally outperform the other. Rather, the relative coverage achieved by either algorithm depends on the structure of the input grammar as well as the structure of the SUT. However, generally there are more possible shallowest embeddings than shortest yields, which is reflected in the greater observed coverage variance in test suites constructed using shallowest embeddings. These test suites thus often achieve better maximal coverage.

5.3.7 Randomization

The large variance in coverage is not necessarily only negative, and we can indeed exploit it to boost overall coverage by merging the various variants into one overall test suite, i.e., creating the union of the 100 different test suites. This obviously results in a larger test suite that achieves better coverage than the individual test suites. However, in certain cases it is even possible to create a combined test suite that has a smaller size and still achieves better coverage than some larger test suites for more complex criteria. For example, for SQLite the combined *rule* test suite for shallowest embedding, with a line coverage percentage of 26.5%, achieves equivalent coverage to both bfs_2 and $step_3$, which achieve an average line coverage of 26.1% and 26.5% respectively, while being only about one fifth and one third of their sizes, respectively.

The data discussed in Sections 5.3.5 to 5.3.7 allow us to answer our final research question.

R4. We observed the effects of both embedding bias and equivalent choice bias in all the systems we considered while evaluating positive test suites (AMPL, SQLite, GCCGo). Especially equivalent choice bias was found to be prevalent in both grammar-based test suite construction algorithms and our LR-graph based algorithms. Embedding bias is less obvious in our LR-graph based algorithms but still present as shallowest embeddings for paths are encoded by construction. We have shown that the effects of bias can pose a significant problem when comparing different implementations of algorithms, especially different classes of algorithms, if one is not aware of theses biases. However, we have also shown that it is possible to use these biases to construct concise test suites that achieve similar coverage to much larger test suites.

			Shallowest Embedding													
algorithm	lr (144.91)		pll (72.3)		rule (48.0)		cdrc (81.0)		$step_3$ (182.0)		$step_4$ (429.0)		deriv (426.1)		bfs_2 (150.0)	
Coverage Type	line	branch	line	branch	line	branch	line	branch	line	branch	line	branch	line	branch	line	branch
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
avg	64.4	50.2	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
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
collapsed	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
collapsed size	1817		615		552		1294		4002		10241		7967		3627	
union lr	-	-	66.5	55.7	66.21	54.27	66.2	54.3	66.8	55	67.5	55.8	67.7	57.7	67	55.2
union size	-		2293		2232		2869		5432		11633		9455		5141	
lr exclusive	-		83%		84%		69%		50%		45%		59%		60%	
			Shortest yield													
algorithm	lr (144.91)		pll (62.0)		rule (48.0)		cdrc (81.7)		$step_3$ (183.6)		$step_4$ (435.2)		deriv (439.8)		bfs_2 (153.0)	
Coverage Type	line	branch	line	branch	line	branch	line	branch	line	branch	line	branch	line	branch	line	branch
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
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
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
collapsed	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
collapsed size	1817		211		198		520		1923		4953		4941		1553	
union lr	-	-	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
union size	-		1978		1970		2277		3581		6548		6512		3310	
lr exclusive	-		96%		97%		95%		87%		80%		78%		95%	

Table 5.1: AMPL code coverage for different embeddings and coverage criteria. 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.

			Shallowest Embedding											
algorithm	lr (22383.24)		pll (4835.9)		rule (366.2)		cdrc (1681.2)		step_3 (31330.6)		deriv (2961.2)		bfs_2 (57173.6)	
coverage type	line	branch	line	branch	line	branch	line	branch	line	branch	line	branch	line	branch
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
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
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
collapsed	27	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**
collapsed size	483289		140592		12329		72091		1683000		124401		3374922	
union size	-		613545		491162		549587		2133066		602343		3796527	
lr exclusive	-		98%		99%		99%		93%		99%		87%	
			Shortest yield											
algorithm	lr (22383.24)		pll (4185.0)		rule (371.0)		cdrc (1681.0)		step_3 (30896.8)		deriv (2992.4)		bfs_2 (56877.0)	
coverage type	line	branch	line	branch	line	branch	line	branch	line	branch	line	branch	line	branch
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**
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
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
collapsed	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
collapsed size	483289		13373		2381		30900		894470		56691		1746918	
union size	-		489656		481516		507003		1356470		530734		2215709	
lr exclusive	-		99%		99%		99%		96%		98%		97%	

Table 5.2: SQLite code coverage for different embeddings and coverage criteria. Coverage is obtained over the SQLite executable (v3.36.0). See Table 5.1 for further explanations.

			Shallowest Embedding													
algorithm	lr (3486.3)		pll (617.8)		rule (163.8)		cdrc (318.6)		$step_3$ (1247.1)		$step_4$ (4320.8)		deriv (5773.4)		bfs_2 (462.7)	
Coverage Type	line	branch	line	branch	line	branch	line	branch	line	branch	line	branch	line	branch	line	branch
max	70.8	76.9	*58.3*	*63.7*	61.1	66.1	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	*57.0*	*61.7*	59.2	63.6	60.8	65.5	66.9	73.0	**69.3**	**74.6**	68.7	74.2	61.0	65.6
avg	70.6	**76.7**	*57.7*	*62.9*	60.3	64.8	61.5	66.6	67.7	74.0	70.2	76.1	**70.8**	76.6	61.8	66.8
stdev	0.1	0.1	0.3	0.5	0.4	0.6	0.3	0.5	0.3	0.4	0.4	0.7	0.6	0.8	0.4	0.5
collapsed	74.1	79.1	*59.2*	*64.2*	62.1	66.5	62.4	67.5	69.5	75.4	72.1	78.1	**74.2**	**80.0**	62.5	67.6
collapsed size	86735		400		3352		1705		6656		40412		31327		4117	
union lr	-	-	71.4	77.2	71.7	77.5	71.7	77.5	73.6	79.0	75.2	80.8	76.5	82.0	71.7	77.5
union size	-		86925		89699		88104		92867		126448		117275		90508	
lr exclusive	-		100%		100%		100%		99%		99%		99%		100%	
			Shortest yield													
algorithm	lr (3486.3)		pll (611.2)		rule (163.8)		cdrc (81.7)		$step_3$ (1265.5)		$step_4$ (4424.3)		deriv (4942.4)		bfs_2 (466.7)	
Coverage Type	line	branch	line	branch	line	branch	line	branch	line	branch	line	branch	line	branch	line	branch
max	70.8	76.9	*57.9*	*63.5*	60.5	65.6	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	*57.4*	*62.6*	59.5	64.4	60.9	65.5	67.4	73.7	70.0	75.6	69.7	**75.7**	61.2	65.7
avg	**70.6**	**76.7**	*57.7*	*63.0*	59.9	64.9	61.2	66.2	67.5	73.9	70.3	75.8	70.2	76.3	61.6	66.4
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
collapsed	**74.1**	**79.1**	*58.4*	*63.8*	60.8	65.8	62.0	66.9	68.5	74.7	71.1	76.7	70.9	77.0	62.4	67.4
collapsed size	86735		238		1386		1097		4502		28827		12224		4117	
union lr	-	-	71.08	76.89	71.08	76.89	71.35	76.98	72.71	78.19	74.33	79.87	74.4	79.5	71.62	77.26
union size	-		86850		87914		87672		90993		115172		98477		89951	
lr exclusive	-		100%		100%		100%		100%		100%		99%		99%	

Table 5.3: GCCGo code coverage for different embeddings and coverage criteria. Coverage is obtained over the GCCGo parser (included with GCC version 11). See Table 5.1 for further explanations.

		lr	lr*	pll		rule		bfs_1		bfs_2		$step_3$		$step_4$		deriv	
		-	-	$\Rightarrow^*_\sqsubseteq$	$\Rightarrow^*_\leq$	$\Rightarrow^*_\sqsubseteq$	$\Rightarrow^*_\leq$	$\Rightarrow^*_\sqsubseteq$	$\Rightarrow^*_\leq$	$\Rightarrow^*_\sqsubseteq$	$\Rightarrow^*_\leq$	$\Rightarrow^*_\sqsubseteq$	$\Rightarrow^*_\leq$	$\Rightarrow^*_\sqsubseteq$	$\Rightarrow^*_\leq$	$\Rightarrow^*_\sqsubseteq$	$\Rightarrow^*_\leq$
Dyck (a)	max	3.0	5.0	1.0	1.0	1.0	1.0	3.0	2.0	4.0	3.0	7.0	6.0	15.0	14.0	1.0	1.0
	min	3.0	5.0	1.0	1.0	1.0	1.0	3.0	2.0	4.0	3.0	7.0	6.0	15.0	14.0	1.0	1.0
	avg	3.0	5.0	1.0	1.0	1.0	1.0	3.0	2.0	4.0	3.0	7.0	6.0	15.0	14.0	1.0	1.0
Dyck(b)	max	4.0	8.0	1.0	1.0	1.0	1.0	3.0	2.0	4.0	3.0	7.0	6.0	15.0	14.0	1.0	1.0
	min	4.0	8.0	1.0	1.0	1.0	1.0	3.0	2.0	4.0	3.0	7.0	6.0	15.0	14.0	1.0	1.0
	avg	4.0	8.0	1.0	1.0	1.0	1.0	3.0	2.0	4.0	3.0	7.0	6.0	15.0	14.0	1.0	1.0
Dyck (c)	max	3.0	-	1.0	1.0	1.0	1.0	2.0	1.0	3.0	2.0	3.0	2.0	4.0	3.0	1.0	1.0
	min	3.0	-	1.0	1.0	1.0	1.0	2.0	1.0	3.0	2.0	3.0	2.0	4.0	3.0	1.0	1.0
	avg	3.0	-	1.0	1.0	1.0	1.0	2.0	1.0	3.0	2.0	3.0	2.0	4.0	3.0	1.0	1.0
Expr (Figure 3.3)	max	17.0	504.0	5.0	4.0	5.0	4.0	21.0	20.0	53.0	52.0	85.0	84.0	341.0	340.0	5.0	4.0
	min	17.0	504.0	5.0	4.0	5.0	4.0	21.0	20.0	53.0	52.0	85.0	84.0	341.0	340.0	5.0	4.0
	avg	17.0	504.0	5.0	4.0	5.0	4.0	21.0	20.0	53.0	52.0	85.0	84.0	341.0	340.0	5.0	4.0
Expr (Figure 3.4)	max	27.0	-	11.0	6.0	7.0	6.0	25.0	24.0	41.0	40.0	99.0	98.0	373.0	372.0	11.0	18.0
	min	25.0	-	7.0	6.0	7.0	6.0	25.0	24.0	41.0	40.0	99.0	98.0	373.0	372.0	10.0	10.0
	avg	26.2	-	9.7	6.0	7.0	6.0	25.0	24.0	41.0	40.0	99.0	98.0	373.0	372.0	10.4	15.5
Niklaas	max	303.0	-	65.0	63.0	55.0	59.0	178.0	184.0	1092.0	1100.0	628.0	652.0	2435.0	2453.0	542.0	590.0
	min	295.0	-	62.0	62.0	55.0	55.0	178.0	178.0	1092.0	1092.0	628.0	628.0	2435.0	2435.0	492.0	514.0
	avg	299.4	-	63.6	62.1	55.0	56.5	178.0	180.1	1092.0	1095.3	628.0	635.8	2435.0	2441.0	515.0	543.4
SIMPL	max	136.0	-	77.0	66.0	46.0	46.0	77.0	79.0	144.0	153.0	171.0	177.0	383.0	407.0	469.0	476.0
	min	127.0	-	71.0	66.0	46.0	46.0	77.0	77.0	144.0	144.0	171.0	171.0	383.0	383.0	418.0	438.0
	avg	132.4	-	74.3	66.0	46.0	46.0	77.0	77.7	144.0	147.0	171.0	173.1	383.0	392.2	436.56	451.6
AMPL	max	149.0	-	76.0	59.0	45.0	45.0	78.0	80.0	147.0	80.0	179.0	185.0	426.0	447.0	461.0	467.0
	min	141.0	-	64.0	59.0	45.0	45.0	78.0	78.0	147.0	78.0	179.0	179.0	426.0	426.0	400.0	420.0
	avg	144.9	-	69.4	59.0	45.0	45.0	78.0	78.7	147.0	78.7	179.0	180.6	426.0	432.2	423.1	436.8
SQLite	max	22526.0	-	5047.0	4289.0	373.0	383.0	1742.0	1743.0	59021.0	58869.0	33027.0	32167.0	-	-	3080.0	3157.0
	min	22126.0	-	4539.0	4148.0	362.0	363.0	1615.0	1607.0	53225.0	48315.0	28923.0	28142.0	-	-	2680.0	2841.0
	avg	22364.7	-	4834.9	4184.1	367.7	370.0	1697.3	1680.6	57776.2	56876.0	31865.5	30895.8	-	-	2935.5	2991.4
Go	max	3521.0	-	638.0	638.0	164.0	164.0	319.0	326.0	462.0	471.0	1248.0	1281.0	4322.0	4511.0	6044.0	6123.0
	min	3455.0	-	602.0	588.0	163.0	163.0	318.0	318.0	463.0	462.0	1245.0	1246.0	4319.0	4314.0	4631.0	5813.0
	avg	3486.3	-	617.8	611.2	163.6	163.8	318.6	322.1	462.7	466.7	1247.1	1265.5	4320.8	4424.3	5773.4	5942.4
dot	max	45.0	-	30.0	28.0	21.0	20.0	26.0	25.0	41.0	40.0	35.0	34.0	55.0	54.0	91.0	90.0
	min	45.0	-	26.0	28.0	21.0	20.0	26.0	25.0	41.0	40.0	35.0	34.0	55.0	54.0	78.0	90.0
	avg	45.0	-	27.8	28.0	21.0	20.0	26.0	25.0	41.0	40.0	35.0	34.0	55.0	54.0	84.0	90.0
CSS3	max	1796.0	-	463.0	454.0	165.0	163.0	476.0	465.0	732.0	726.0	1128.0	1129.0	2596.0	2584.0	1622.0	1770.0
	min	1738.0	-	410.0	403.0	161.0	159.0	458.0	447.0	712.0	698.0	1101.0	1042.0	2541.0	2401.0	1420.0	1514.0
	avg	1759.3	-	436.2	428.1	163.6	161.6	466.6	459.2	721.8	715.0	1114.4	1106.2	2568.2	2544.7	1519.7	1612.0

Table 5.4: Size data for different positive coverage criteria over multiple grammars. Where the grammar is not previously mentioned it was taken from the ANTLR grammar repository [4].

	Niklaas 1			
	suspicious rules	wasted effort	failing tests	test suite size
positive	18	0.0	2	302
cdrc	18	6.5	2	79
rule mutation	25	0.0 - 5.6	4	10805
edge insertion	99	6.5 - 9.3	338	158003
edge deletion	32	29.9	14	1581
edge substitution	39	0.0 - 36.4	7	153280
stack insertion	97	10.2 - 15.0	1451	144552
stack deletion	89	15.0 - 19.6	217	1679
stack substitution	98	10.3 - 14.0	1911	153280
prefix cutting	0	-	0	1781
	Niklaas 2			
	suspicious rules	wasted effort	failing tests	test suite size
positive	13	0.0	4	302
cdrc	8	0.0	1	79
rule mutation	48	26.4 - 37.3	22	10805
edge insertion	98	6.9 - 84.3	532	158003
edge deletion	32	31.3	16	1581
edge substitution	39	12.7 - 38.2	8	153280
stack insertion	99	34.3 - 44.1	1230	144552
stack deletion	93	67.6 - 87.3	221	1679
stack substitution	99	34.3 - 49.0	1687	153280
prefix cutting	0	-	0	1781
	SIMPL			
	suspicious rules	wasted effort	failing tests	test suite size
positive	15	0.0	3	130
cdrc	15	8.0	2	182
rule mutation	28	4.5	2	8549
edge insertion	85	44.3 - 75.0	270	58746
edge deletion	40	17.0 - 45.5	10	1072
edge substitution	85	51.1 - 93.1	500	57122
stack insertion	85	48.9 - 83.0	1047	50077
stack deletion	80	39.8 - 91.0	122	626
stack substitution	85	47.7 - 95.5	1626	59769
prefix cutting	0	-	0	556

Table 5.5: Suspicious rules revealed in student parsers by a selection of positive and negative test suites. We consider positive LR and the context-dependent rule coverage (cdrc) algorithms for positive test suites. For negative test suites we consider grammar-based rule mutations followed by edge and stack mutations. Wasted effort is the percentage of suspicious rules, ranked by suspiciousness, that have to be examined before finding the known errors

5.4 Conclusion

We can therefore conclude that, by using the LR-graph and indirectly the LR-automaton, our coverage directed method of test suite construction in general achieves favorable coverage relative to test suite size over the systems we have considered, when compared to the current state-of-the-art methods. We have also shown that negative mutations on paths over the LR-graph can reveal faults in a SUT in more contexts than rule mutation based methods, which can assist in intuitive fault location but may lead to more wasted effort when using automated fault localization techniques. Lastly, we have also shown how bias in grammar-based testing can have a very severe effect on the coverage achieved by an algorithm and have performed all comparisons in this thesis while carefully considering this bias.

Chapter 6

Related Work

There exists some work which is related to what we have proposed in this thesis, either in terms of related algorithms or the more foundational concepts upon which our algorithms are built. We will discuss and compare these below.

6.1 Grammar-Based Algorithms for Test Suite Generation

Most current state-of-the-art grammar-based algorithms for test suite generation follow a very similar structure [24, 29]. The generic cover algorithm (see Algorithm 3) [18] shows the basic structure of these algorithms, which differ only in coverage criterion. First, it iterates over all symbols $X \in V$ and computes minimal derivations $S \Rightarrow^*_{\leq} \alpha X \omega$. It then completes a set of minimal derivations $C(X)$ for X according to a coverage criterion C. These minimal derivations are embedded into the embedding of X and then grounded out to form a valid sentence, as described by the input grammar. Most of the state-of-the-art algorithms were not originally phrased in this way but we may formulate them as a coverage criterion for the generic cover algorithm. For example, Zelenov's PLL algorithm [47] is originally phrased in terms of derivation chains but may also be phrased as a criterion for the generic cover algorithm (see Section 2.4.1). Even though these coverage criteria make use of the same algorithm, they can still result in test suites with vastly different characteristics (as can be seen in Chapter 5 by the different test suite sizes and coverage achieved by these different criteria). However, there are some grammar-based algorithms that do not match the structure of the generic cover algorithm. For example, in Purdom's seminal paper [34] he proposes a sentence generator that constructs test suites of minimal size to exercise the

rules of an input grammar. This algorithm is fundamentally different in that it prioritizes minimizing test suite size over test case size.

These approaches differ greatly from the automaton-based algorithms we propose, in that they construct test suites directly from the input grammar. We first construct a LR-graph from the LR-automaton, which was produced from the input grammar, to more closely model the system-under-test during the test suite generation process. This means that the key difference between our algorithm and grammar-based algorithms is that grammar-based algorithms place emphasis on the input grammar whereas we place emphasis on the recognizer for the input grammar, which acts as a abstract model of the system-under-test.

In practice, grammar-based fuzzing is another popular method of testing parsers. This often involves a stochastic context-free grammar, where probabilities are attached to the production rules, and a semi-random process of sentence generation [21, 43, 44]. 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 [8]. Fuzzing is a mostly black-box testing method when used in grammar-based testing, although some work has been done to apply these black-box fuzzing techniques in a white-box setting with better performance than existing white-box methods [19].

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 suited construction algorithms. However, in most real-world applications, like CI/CD pipelines, fuzzing may not be feasible due to the long testing times.

6.1.1 Negative Test Suite Construction

Mutating valid words to produce negative test cases is a popular method of negative test suite construction and was first proposed by Harm et al. [22]. Other methods have also been proposed that use an oracle to determine whether a mutation is indeed valid [28]. In our methods we instead want to generate guaranteed negative tests by mutations, without the need for an oracle.

Zelenov et al. [47] propose two methods for producing negative test cases. The first, NLL, is produced by considering all pairs of top-of-stack symbols A in a LL-automaton and next input symbols t. Negative test cases are then

produced by using mutations of all these pairs by replacing t by t', the terminal symbols that can not be seen next in a valid word, given a top-of-stack symbol A. A similar approach, NLR, is proposed over the LR-automaton. However, in this case all pairs of states s_i and invalid next input symbols t' are used. These methods are different from our methods in that it does not include mutations of non-terminal symbols, as we do with stack mutations, and do not consider paths over the automaton in these mutations. They also do not allow for delete mutations. The algorithms to produce test suites according to the NLL and NLR test suites are not specified by Zelenov et al. and as such a deeper, quantitative evaluation was not possible.

The most similar mutation method to the methods in thesis are proposed by Raselimo et al. [37] and mutate the grammar rules. These mutations include insertion, substitution and deletion, similar to edge and stack mutations. Mutating terminal symbols in a rule is similar to edge mutations which affect terminal push edges, and mutating non-terminal symbols is similar to stack mutations, since they alter an entire sequence of tokens corresponding to the non-terminal symbols. The key difference is the context in which these mutations can be applied. Rule mutations must be valid for all contexts in which they may be used in a derivation. This can limit possible mutation locations. The benefit of our mutations over the LR-graph is that different contexts often result in different paths over the graph. This means that we may apply a mutation in one context where it is valid and omit it in another, where rule mutations may need to omit a mutation entirely since it would not result in a guaranteed negative word in all contexts.

6.2 Automaton-Based Algorithms for Test Case Generation

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 Heam et al. [25]. This coverage criterion is essentially a superset of our proposed coverage criterion, covering all edges of the LR-graph associated with the automaton, since our criterion translates to covering all combinations (p, q), such that q is reachable and adjacent to p. This will likely result in considerably larger test suites than those produced by our algorithms. This larger test suite also does 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 is therefore more fine-grained and avoids

unnecessarily covering transitions multiple times.

Another method, the Weak Positive LR coverage algorithm (WPLR), is proposed by Zelenov et al. [47] and guarantees that all states in the automaton will be explored. This is done be constructing a test suite such that all combinations of state symbols s_i and transitions from this state labeled with a terminal symbol x are covered. A pair is considered covered if the test suite contains a sentence $S \Rightarrow \alpha x \beta$ where α is a prefix that leads to the state s_i when processed. Zelenov et al. provides no clear steps as to how to construct such a test suite. This coverage criterion is fundamentally different from the one we propose in that, although it guarantees some coverage over the automaton, it only considered shift transitions and states. Our coverage criterion guarantees coverage of all states and transitions, not just shift transitions and states. Zelenov also acknowledges that their implementation does not always end up covering all shift transitions, even for a very small automaton. 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 going untested.

Esterhuizen [17] also proposes a method of both positive and negative test suite construction. It explores the automaton with a breadth-first search algorithm to cover all states and then completes the generated prefixes by using kernel items. This appears similar to our breadth-first traversal algorithm. However, it is fundamentally different in that only coverage of states is considered as a goal (this goal is not guaranteed), resulting in transitions not necessarily being covered. Negative test suite generation is done by inserting tokens which are not valid, given a prefix which leads to a specific state in the automaton when parsed. These methods are greatly different from our algorithms as it does not consider the stack context in which a transition may occur or guarantee coverage of all transitions in the automaton. Our algorithms consider both of these conditions. Esterhuizen's experimental results also show that the test suites struggled to reveal errors in the systems that it was evaluated over, while being fairly sizable. Our algorithms produce test suites many times smaller than those produced by Esterhuizen, while revealing errors in many contexts, as shown by our evaluation.

6.3 Reachability in State Machines

Since automaton-based algorithms for test case generation deal with combinations of paths between different states, calculating reachability between these states becomes a significant problem. Pottier [33] proposes 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. This is done by computing a *star* rooted at every state s from paths whose source is s and then running a modified version of Dijkstra's algorithms to determine which states are reachable by a reduction. Our algorithm contains a similar construct. The main difference in structure between the LR-automaton and the LR-graph 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 to the reduction reachability problem as pop edges. Since we do this at construction time, we simply do 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 our construction time of the 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.

6.4 LR-Parsing methods and grammar ambiguity

There are many different variations of LR parsing. These tend to trade off time and space complexity against which subset of languages in LR can be recognized by the parsing method. SLR [14] and LALR [15] are examples of parsing algorithms that can recognize input for a wider range of grammars than LR(0) by using lookahead sets. The main difference between these two algorithms lies in how these lookahead sets are calculated, with LALR being more precise and resolving a greater amount of ambiguities that may be found in a grammar. This means that an LALR parser can act as a recognizer for a greater number of grammars than an SLR parser. However, even an LR(k) parsers, which recognize more grammars than both SLR and LALR parsers, can not resolve all possible ambiguities in a grammar. Methods like GLR [45] have been proposed, which forks the stack whenever a conflict is encountered and parses using all the duplicate stacks until either the accept state is reached or an error state is reached from all forked stacks. This means that all conflicts can be resolved using a GLR parser.

Since we are generating test cases instead of parsing, we do not face the same problem with regards to ambiguity that deterministic parsers face. We can simply generate test cases using all conflicting rules and do not need to choose one at generation time. This allows us to use a simple LR(0) parser, without lookaheads. We do build on the idea of forking the stack at each point

where multiple choices can be made during the generation of paths. This can be seen explicitly in the first breath-first traversal algorithm (see Section 3.2) and is still used implicitly in the pop edge coverage algorithm (see Section 3.3), since a reduce/reduce conflict results in multiple pop edges and our algorithm ensures all pop edges are covered. Preliminary investigation also revealed that using an LR(1)-automaton instead of an LR(0)-automaton leads to unnecessary deeper nesting in recursive grammar structures and larger test suites. Therefore, we also use an LR(0)-automaton in our experiments to achieve our goal of providing concise test suites that provide good coverage of a SUT.

Chapter 7

Conclusions and Future Work

In this thesis we have introduced a new perspective and methods for generating test suites from a context-free grammar by using paths over an LR-graph to define a coverage criterion. We have shown that both positive and negative test suites generated by our new automaton-based algorithms perform well in terms of coverage achieved over a system-under-test when compared to the current state-of-the-art, grammar-based test suite construction algorithms. We have also defined and quantified bias in grammar-based test suite construction algorithms as a potentially severe problem that must be addressed when using these algorithms.

In this final chapter we summarize our main contributions, propose possible future work and conclude our research on grammar-based test suite construction using coverage defined over LR-automata.

7.1 Main Contributions

The main contribution of this thesis is the different perspective our proposed algorithms for generating test suites from an input grammar takes, as compared to the other state-of-the-art algorithms. Our algorithms do not just use the LR-automaton as part of the coverage criterion as is the case in other related work [25, 47], but instead focus completely on the structure and the context information contained in the LR-automaton to generate a test suite. This is our most novel contribution and the problem we have solved in this thesis. We have made a number of smaller, yet still novel, contributions in laying the foundation for our algorithms as well as in the comparative evaluation of our algorithms.

First, we defined the concept of an LR-graph. This is an alternative representation of the LR-automaton that enables the use of graph-theoretic

constructs in our algorithms. It also helps to solve the reachability problem of reductions in the LR-automaton at construction time, providing our algorithms with an efficient way of extracting reachability information at runtime.

We also identified and formalized the concept of bias in grammar-based test suite construction algorithms. Specifically, we identify two different types of bias. We show that the effect of these biases, either on comparisons of different algorithms or when used in the real world, can be very severe but that it can be harnessed to provide efficient test suites.

We also provide methods for negative test suite generation by mutating positive paths. We show that this allows us a greater number of mutation locations than is possible with grammar-based rule mutations.

Lastly, we provide a thorough evaluation of the algorithms proposed in this thesis and compare them to the current state-of-the-art systematic grammar-based test suite construction algorithms. We also show that our methods scale to very large, real world systems and grammars in our evaluations over SQLite and GCCGo.

7.2 Future Work

There are number of possible future extensions to the work proposed in this thesis.

Formalize relationship to other grammar-based testing algorithms In this thesis we have proposed algorithms based on the LR-automaton for a context-free grammar, and not just on the context-free grammar itself, as is the case for the coverage criteria that use the generic cover algorithm. In this thesis we compared these methods for test generation quantitatively. This gives us a good understanding of the performance of these algorithms and criteria. However, if a formal relationship between these automaton-based algorithms and the grammar-based algorithms can be established, by deriving a new coverage criterion over the LR-graph that corresponds to a coverage criterion over the CFG, it would allow for a detailed comparison of the structure of the different test suites that they generate. Conversely, we could investigate phrasing the automaton-based algorithms as a coverage criterion over a CFG.

Test suite generation over other automaton types In our initial investigation we found that using a LR(1)-automaton instead of an LR(0)-automaton led to larger test suites that did not in turn lead to covering

reductions in more useful contexts. However, it may be possible to extract more context information from other LR(k)-automatons which could lead to more fine grained negative mutations.

New coverage criteria over LR-graph We have introduced the foundational concepts for producing grammar-based test suites from an LR-automaton. The algorithms we propose are to cover all edges in the LR-graph, thus aiming at increasing coverage over a SUT. However, there are many other possible coverage criteria that are possible. For example, we could investigate the generation of test suites that specifically attempt to exercise reduce/reduce conflicts in a SUT, by covering a subset of pop-edges related to these conflicts, to determine if it resolved these conflicts correctly, as has been attempted by grammar-based methods [27].

Oracle for testing systems Test suites are used during the development of software to ensure that the software acts as expected. Therefore, it is important that the error messages, when bugs are found, are clear and accurate. All the negative mutations we propose, should lead to precisely one failure. Thus it may be possible to construct and oracle to explain the bug in the underlying system, given a specific failing test case.

More biases in grammar-based test suite construction We have identified two major biases that are present during the construction of grammar-based test suites for the algorithms discussed in this thesis. However, there are likely other biases, not only in grammar-based test suite construction algorithms but also in different classes of test suite construction algorithms. Given the threat to validity and usability that these biases pose, it is important that other biases are also quantified in the future.

Exploiting bias to increase coverage In this thesis we have already shown a simple method for increasing coverage over a SUT by using the bias in small, simple test suites and merging these test suites to produce, still compact, but more effective test suites. It should be possible to use bias in other ways to increase coverage over a SUT. For example, we could incrementally use randomization to find the asymptotic maximum coverage achievable by a coverage criterion.

7.3 Concluding Remarks

Bugs in parsers can have very severe consequences, since parsers act as gatekeepers, so allowing invalid input to pass can lead to undefined behavior in the rest of the system. Generally, grammar-based test suite construction algorithms have focused only on the input grammar when generating test suites, and not the system-under-test it is intended for. By focusing on the recognizer for the input grammar, we have an abstract model of the actions a correct system-under-test would perform and by exercising these actions gain greater confidence that a SUT is performing as expected.

We have also shown that bias in grammar-based test suite constructions algorithms can have detrimental effects on the coverage achieved over a SUT. We have investigated the effects of two types of sampling bias present in grammar-based testing and shown that they remain prevalent, even in very large test suites and very large systems. We believe that these results are significant because context-free grammars are in practice used to test full systems with context-sensitive constraints, not just parsers.

We developed new algorithms and coverage criteria, for both negative and positive test suites, that take a very different approach to the current state-of-the-art algorithms. These test suites achieved good coverage for their size when compared to the test suites produced by other algorithms. We have also demonstrated that our algorithms scale to very large systems and can be used to test real world systems.

Bibliography

[1] Graphviz. https://graphviz.org/. [Online; accessed 22-November-2021].

[2] Mars climate orbiter mishap investigation board phase 1 report. https://web.archive.org/web/20010920052120/http://sunnyday.mit.edu/accidents/MCO_report.pdf, 2001. [Online; accessed 26-November-2021].

[3] Summary of the faa's review of the boeing 737 max. https://www.faa.gov/foia/electronic_reading_room/boeing_reading_room/media/737_RTS_Summary.pdf, 2020. [Online; accessed 26-November-2021].

[4] Antlr grammar repo. https://github.com/antlr/grammars-v4, 2021. [Online; accessed 26-November-2021].

[5] Gccgo. https://go.dev/, 2021. [Online; accessed 26-November-2021].

[6] Sqlite. https://www.sqlite.org/, 2021. [Online; accessed 26-November-2021].

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

[8] Vaggelis Atlidakis, Roxana Geambasu, Patrice Godefroid, Marina Polishchuk, and Baishakhi Ray. Pythia: Grammar-based fuzzing of REST apis with coverage-guided feedback and learning-based mutations. *CoRR*, abs/2005.11498, 2020.

[9] Boris Beizer. *Software testing techniques (2. ed.)*. Van Nostrand Reinhold, 1990.

[10] Lutz Bornmann. The hawthorne effect in journal peer review. *Scientometrics*, 91(3):857–862, 2012.

[11] Tsong Yueh Chen and Yuen-Tak Yu. A more general sufficient condition for partition testing to be better than random testing. *Inf. Process. Lett.*, 57(3):145–149, 1996.

[12] Corinna Cortes, Mehryar Mohri, Michael Riley, and Afshin Rostamizadeh. Sample selection bias correction theory. In Yoav Freund, László Györfi, György Turán, and Thomas Zeugmann, editors, *Algorithmic Learning Theory, 19th International Conference, ALT 2008, Budapest, Hungary, October 13-16, 2008. Proceedings*, volume 5254 of *Lecture Notes in Computer Science*, pages 38–53. Springer, 2008.

[13] Richard A. DeMillo, W. Michael McCracken, Rhonda J. Martin, and John F. Passafiume. *Software testing and evaluation*. Benjamin/Cummings, 1987.

[14] Frank DeRemer. Simple lr(k) grammars. *Commun. ACM*, 14(7):453–460, 1971.

[15] Frank DeRemer and Thomas J. Pennello. Efficient computation of LALR(1) look-ahead sets. *ACM Trans. Program. Lang. Syst.*, 4(4):615–649, 1982.

[16] Mohd Ehmer and Farmeena Khan. A comparative study of white box, black box and grey box testing techniques. *International Journal of Advanced Computer Science and Applications*, 3, 06 2012.

[17] M. Esterhuizen. Test case generation for context free grammars. 2018.

[18] Bernd Fischer, Ralf Lämmel, and Vadim Zaytsev. Comparison of context-free grammars based on parsing generated test data. In Anthony M. Sloane and Uwe Aßmann, editors, *Software Language Engineering - 4th International Conference, SLE 2011, Braga, Portugal, July 3-4, 2011, Revised Selected Papers*, volume 6940 of *Lecture Notes in Computer Science*, pages 324–343. Springer, 2011.

[19] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 206–215. ACM, 2008.

[20] John Graham-Cumming. Incident report on memory leak caused by cloudflare parser bug. `https://blog.cloudflare.com/`

`incident-report-on-memory-leak-caused-by-cloudflare-parser-bug/`, 2017. [Online; accessed 22-November-2021].

[21] Hai-Feng Guo and Zongyan Qiu. A dynamic stochastic model for automatic grammar-based test generation. *Softw. Pract. Exp.*, 45(11):1519–1547, 2015.

[22] Jörg Harm and Ralf Lämmel. Two-dimensional approximation coverage. *Informatica (Slovenia)*, 24(3), 2000.

[23] Lawrence A. Harris. SLR(1) and LALR(1) parsing for unrestricted grammars. *Acta Informatica*, 24(2):191–209, 1987.

[24] Nikolas Havrikov and Andreas Zeller. Systematically covering input structure. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 189–199. IEEE, 2019.

[25] Pierre-Cyrille Héam and Hana M'Hemdi. 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*, pages 1–7. IEEE Computer Society, 2015.

[26] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley, 2007.

[27] Chinawat Isradisaikul and Andrew C. Myers. Finding counterexamples from parsing conflicts. In David Grove and Stephen M. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 555–564. ACM, 2015.

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

[29] Ralf Lämmel. Grammar testing. In *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software,*

ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings, pages 201–216, 2001.

[30] Brian A. Malloy and James F. Power. An interpretation of purdom's algorithm for automatic generation of test cases. In *1st ACIS Annual International Conference on Computer and Information Science*, 2001.

[31] Glenford J. Myers. *The art of software testing (2. ed.)*. Wiley, 2004.

[32] Akira OCHIAI. Zoogeographical studies on the soleoid fishes found in japan and its neighhouring regions-ii. *NIPPON SUISAN GAKKAISHI*, 22(9):526–530, 1957.

[33] François Pottier. Reachability and error diagnosis in LR(1) parsers. In Ayal Zaks and Manuel V. Hermenegildo, editors, *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 88–98. ACM, 2016.

[34] Paul Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12:366–375, September 1972.

[35] Moeketsi Raselimo and Bernd Fischer. Spectrum-based fault localization for context-free grammars. In Oscar Nierstrasz, Jeff Gray, and Bruno C. d. S. Oliveira, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2019, Athens, Greece, October 20-22, 2019*, pages 15–28. ACM, 2019.

[36] Moeketsi Raselimo and Bernd Fischer. Automatic grammar repair. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2021, Virtual Event, USA*, 2021.

[37] Moeketsi Raselimo, Jan Taljaard, and Bernd Fischer. Breaking parsers: mutation-based generation of programs with guaranteed syntax errors. In Oscar Nierstrasz, Jeff Gray, and Bruno C. d. S. Oliveira, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2019, Athens, Greece, October 20-22, 2019*, pages 83–87. ACM, 2019.

[38] Stuart Reid. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In *4th IEEE International Software Metrics Symposium (METRICS 1997), November 5-7, 1997, Albuquerque, NM, USA*, pages 64–73. IEEE Computer Society, 1997.

[39] Christoff Rossouw and Bernd Fischer. Test case generation from context-free grammars using generalized traversal of lr-automata. In Ralf Lämmel, Laurence Tratt, and Juan de Lara, editors, *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16-17, 2020*, pages 133–139. ACM, 2020.

[40] Christoff Rossouw and Bernd Fischer. Vision: Bias in systematic grammar-based test suite construction algorithms. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2021, Virtual Event, USA*, 2021.

[41] Michael Sipser. *Introduction to the theory of computation*. PWS Publishing Company, 1997.

[42] Perry Smith. Hyacc. `http://hyacc.sourceforge.net/`, 2020. [Online; accessed 26-November-2021].

[43] Ezekiel O. Soremekun, Esteban Pavese, Nikolas Havrikov, Lars Grunske, and Andreas Zeller. Probabilistic grammar-based test generation. In Anne Koziolek, Ina Schaefer, and Christoph Seidl, editors, *Software Engineering 2021, Fachtagung des GI-Fachbereichs Softwaretechnik, 22.-26. Februar 2021, Braunschweig/Virtuell*, volume P-310 of *LNI*, pages 97–98. Gesellschaft für Informatik e.V., 2021.

[44] Prashast Srivastava and Mathias Payer. Gramatron: Effective grammar-aware fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 244–256, New York, NY, USA, 2021. Association for Computing Machinery.

[45] Masaru Tomita. An efficient context-free parsing algorithm for natural languages. In Aravind K. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence. Los Angeles, CA, USA, August 1985*, pages 756–764. Morgan Kaufmann, 1985.

[46] Phillip van Heerden, Moeketsi Raselimo, Konstantinos Sagonas, and Bernd Fischer. Grammar-based testing for little languages: an experience report with student compilers. In Ralf Lämmel, Laurence Tratt, and Juan de Lara, editors, *Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2020, Virtual Event, USA, November 16-17, 2020*, pages 253–269. ACM, 2020.

[47] Sergey V. Zelenov and Sophia A. Zelenova. Automated generation of positive and negative tests for parsers. In Wolfgang Grieskamp and

Carsten Weise, editors, *Formal Approaches to Software Testing, 5th International Workshop, FATES 2005, Edinburgh, UK, July 11, 2005, Revised Selected Papers*, volume 3997 of *Lecture Notes in Computer Science*, pages 187–202. Springer, 2005.

Appendix A

Command-Line Arguments

Our implementation takes the following command-line arguments:

1. **-f** The file path to the grammar file.
2. **-g** A boolean flag that determines whether a PDF of the LR-graph should be produced.
3. **-c** The coverage type. It can be positive, neg-sub, neg-cut, neg-del and neg-add.
4. **--stack** A boolean flag for whether or not to use stack mutations. It is only applicable when neg-sub, neg-del or neg-add are set as coverage type.
5. **--classic** A boolean flag that determines which algorithm to use. By default the pop-edge coverage algorithm is used, but by setting this flag to true the breadth-first traversal algorithm is used.
6. **--seed** A number that is used to seed the random generator that is used to resolve equivalent choices.
7. **-o** The path to the output file. If it is not set the test suite is printed to `stdout`.