Breaking Student Parsers with Generated Test Cases

Filip Johansson
E16, Lund University, Sweden
fi2541jo-s@student.lu.se

Abstract

In this paper we try to increase the coverage of a handwritten test suite used to test student parser grammars. To do this we construct a tool, called NeoTestGen, that generates test cases for parsers based on an input grammar. This is done by extending the parser generator NeoBeaver. NeoTestGen can generate both positive and negative test cases, that is test cases that the parsers should pass or fail respectively. The resulting tool can quickly generate a large test suite with large coverage; however, there some tests in the handwritten suite that could not be generated. Furthermore, no tests in the generated suite were found to increase the coverage of the handwritten one.

1 Introduction

In the Compiler course EDAN65 at Lund University, students write an LR parser grammar for a simple C subset called SimpliC. The students' grammars are tested on a test suite, TS_{ref}, containing small test cases written by the teachers. However, due to a lack of coverage, there may still be incorrect grammars that pass TS_{ref}. To find such misses and eventually improve TS_{ref}, the goal of this project is to build a tool, called NeoTestGen, that can automatically generate a test suite TS_{gen} with greater coverage. NeoTestGen could also be used to compare and find errors in other grammars.

Both TS_{ref} and TS_{gen}, contains two kinds of tests: positive and negative, i.e. tests that the parser should pass and fail respectively. To generate positive tests there are several algorithms. For example, Purdom [8] has designed an algorithm that generates a test suite that utilizes every production in a grammar. For the generation of negative tests, there has recently been significant developments as Raselimo, Taljaard, and Fischer [9] have come up with two new algorithms: word mutation and rule mutation. Word mutation works by mutating positive tests to ensure that the result is faulty and rule mutation works by mutating the grammar and then generating positive tests for these faulty grammars.

To produce NeoTestGen, we develop an implementation of these aforementioned algorithms as an extension to the NeoBeaver parser generator. The information needed for the algorithms, such as symbols and productions, will be gathered from the attributed syntax tree (AST) that NeoBeaver produces after having parsed the grammar file. To evaluate NeoTestGen, we compare TS_{gen} to TS_{ref}. In particular, the main question we try to answer in this article is if TS_{gen} will cover cases that TS_{ref} does not, in other word if we can find new tests which find errors in student grammars that were previosuly not found.

Course paper, EDAN70, Lund University, Sweden
January 15, 2020.

2 Background

In this section we establish nomenclature and describe the different algorithms that we have implemented in NeoTestGen.

2.1 Nomenclature

In this subsection we define concepts that will be used repeatedly over the paper. We use the same definitions of context-free grammar (CFG), language, the first- and follow-set, and items as Aho et al. [1]. Briefly, a CFG, G, is a four-tuple of the set of terminals, T, the set of non-terminals, N, the start symbol, S, and the set of productions, P. The language of G, $L(G)$, is the set of token strings that can be generated from, G.

The first-set of a symbol, *first*(X), is defined as the set of terminals that is the first symbol in some token string derived from X and the follow-set of a symbol, *follow*(X), is the set of terminals coming directly to right of X in any production. Moreover, an item is defined as being a production rule where a dot keeps tracks of the position of the current parsing, for example $p = A \rightarrow \alpha \bullet \beta$. The set of all items for a grammar will be referred to as $P^{\bullet}$.

For the definition of the last- and precede-set we use similar definitions as Raselimo et al. [9]. The last-set of a symbol, *last*(X), and the precede-set, *precede*(X), are defined, in juxtaposition to the first- and follow-set, as the set of terminals that is the last string defined from X, and respectively as the set of terminals coming directly to left of X in any production.

2.2 Purdom's Algorithm

Purdom [8] presents a method to produce a positive test set which uses every production in the grammar at least once. Purdom's algorithm is divided into three, distinct, sequentially executed parts. The first of these is the *Shortest Terminal String Algorithm*, which determines the data structure $SHORT$. For every non-terminal $n \in N$, $SHORT[n]$ contains the production used to obtain one of the shortest string derivable from n. The second step is the *Shortest Derivation Algorithm*. This algorithm calculates $PREV$. $PREV[n]$ is also defined for every $n \in N$ and contains the production that introduces n into the shortest derivation. Observe, that since the start symbol is always on the left hand side of rules, it will never be introduced by another non-terminal; thus, $PREV[S] = null$. The last part is the *Sentence Generation Algorithm*, which utilizes *SHORT* and *PREV* to generate positive tests. The non-deterministic test set generated by Purdom's algorithm will be referred to as TS_{Pur}.

2.3 Word Mutations

Raselimo et al. [9] introduce new algorithms for negative parser test generation. The first one, word mutation, mutates words $w \in L(G)$ into $w_{mut} \notin L(G)$. Central to this method are poisoned pairs. A poisoned pair is a pair of symbols (A, B), that cannot occur next to each other. Furthermore, they denote $PP(G)$ as the set of all poisoned pairs in the grammar G. The pair (A, B) is in $PP(G)$ if and only if $A \notin precede(B)$ or equivalently, if and only if $B \notin follow(A)$. Since any word with a poisoned pair cannot be formed, such word is not in the language.

To perform the mutations, Raselimo et al. employ the string editing operations that are used to calculate the Damereau-Levenshtein string edit distance, as mutation operations [3, 6]. The mutations are defined as follows, where $a, b, c, d \in T$, and x, y are possibly empty strings of terminals:

1. Token deletion: if $xabcy \in L(G)$ and $(a, c) \in PP(G)$, then $xacy \notin L(G)$.
2. Token insertion: if $xacy \in L(G)$, and either $(a, b) \in PP(G)$ or $(b, c) \in PP(G)$, then $xabcy \notin L(G)$.
3. Token substitution: if $xabcy \in L(G)$, and either $(a, d) \in PP(G)$ or $(d, c) \in PP(G)$, then $xadcy \notin L(G)$.
4. Token transposition: if $xabcdy \in L(G)$, and either $(a, c) \in L(G)$ or $(c, d) \in PP(G)$ or $(b, d) \in PP(G)$, then $xacbdy \notin L(G)$.

2.4 Rule Mutations

Raselimo et al. [9] outline another mutation based approach for generating negative test cases called rule mutations . First, rules in the grammar G are mutated forming the mutated grammar G_{mut}. The types of mutations possible are similar to token deletion, token insertion, and token substitution for word mutations; the difference being, they are performed on right-hand symbols in rules instead of on tokens in words. A word $w \in L(G_{mut})$ is not in $L(G)$ if the mutated rule is in the derivation of w. The word w can thus be used as a negative test when the aforementioned condition is fulfilled.

To be able to determine when to mutate the rules, there are two auxiliary functions called left and right. These functions operate on items and are defined as follows, where α, β are possibly empty strings of symbols and $A \in N$:

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

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

An item of a mutated rule $A \longrightarrow \alpha \bullet \beta$ will lead to a negative test when (1) or (2) is true.

$$follow(left(A \longrightarrow \alpha \bullet \beta)) \cap right(A \longrightarrow \alpha \bullet \beta) = \emptyset \qquad (1)$$

$$left(A \longrightarrow \alpha \bullet \beta) \cap precede(right(A \longrightarrow \alpha \bullet \beta)) = \emptyset \qquad (2)$$

2.5 Pretty Printing

The tests in TS_{ref} are small examples of program text in the SimpliC language. However, the aforementioned algorithms output token strings. Thus, to be able to compare the generated tests to TS_{ref}, the token strings need to be pretty printed to program text. This is done in a simplistic manner, where each token is translated to a corresponding character or string. For example, the NUMERAL token translates to 1 and the ID token translates to *a* in all cases.

3 Implementation

We extended the NeoBeaver parser generator by first parsing the input grammar. The grammar that is used to parse the input file is called GGrammar. After parsing, NeoBeaver traverses the generated AST to collect the information about all symbols and productions needed for generating a new parser.

The implementation of Purdom's [8] two first parts were straight forward according to the descriptions in the paper. However, Purdom's description of the *Sentence Generation Algorithm* is convoluted and therefore hard to translate into code. This is an issue recognized by Malloy and Power [7]. They have thus attempted to describe the algorithm in pseudo-code. Once we had implemented Malloy's and Power's pseudo-code, we discovered that it was faulty. In particular, Malloy's and Power's subroutine `load_ONCE()`. This method omits to check whether a non-terminal is ready to be assigned a rule it is supposed to use, and thus overwrites some rules that have not yet been used, resulting in the algorithm finishing prematurely. By comparing to Purdom's description we eventually discovered and fixed this error. An example of a sentence in the language of SimpliC generated by Purdom's algorithm is:

```
int a(int a) {
  int a;
}
```

To implement negative test generation by word mutation, the follow set needed to be implemented to be able to determine the poisoned pairs in SimpliC, $PP(SimpliC)$. This was done in a similar way to Appel's and Palsberg's [2] fixed point iteration (FPI) to calculate the nullable-, first- and follow-set.

To see where the mutations should be applied, we iterate over a list of tokens that form a positive test (i.e. tests in TS_{Pur}) and record were a mutation will result in a poisoned pair. Then the positive test is mutated and printed. A negative test generated by this method is:

```
int a(int  % a) {
  int a;
}
```

Here the token REST was inserted in-between the INT and ID tokens, because the pair $(INT, REST) \in PP(SimpliC)$. At first, there were some issues with positive cases being generated by the algorithm. One such example was:

```
int awhile () {
}
```

A WHILE token was inserted since $(ID, WHILE) \in PP(SimpliC)$. However, due to the simplicity of the pretty printer, it could not figure out spacing for faulty tests, and thus the ID token and the WHILE token were printed without any spaces between them. For a scanner 'awhile' is just an ID. We fixed this issue by making sure there is sufficient with spaces next to ID tokens.

For the implementation of rule mutations the *last* set and *precede* set needed to be implemented. These were implemented similarly to the way the *first* and *follow* set were implemented. After this, we implemented the *left* and *right* set. The left attribute implemented in JastAdd looks as follows:

```
syn Set<String> GRule.left(int pos) {
  GGrammar g;
  GComponent leftComp;
  String leftSym;
  g = getGrammar();
  if (pos <= 0) {
    return g.precede(leftSide());
  }
  leftComp = getGComponent(pos-1);
  leftSym = leftComp.getName();
  if (g.nullable(leftSym)) {
    Set<String> l = g.last(leftSym);
    l.addAll(left(pos-2));
    return l;
  } else {
    return g.last(leftSym);
  }
}
```

The code above returns the left set for an item with its dot at position pos. It returns the last-set of the symbol on the left of the dot. If the symbol left of the dot is nullable it will return the left-set of the position one to the left. If there is no more symbol on the left of the dot the precede set of the left hand side of the rule will be returned as per definition. We implemented the right set in a similar way, using the first and follow attribute instead of last and precede. The mutations are thus generated by iterating over every rule and checking when a mutation could be done. A mutation in this sense is that one copies the GGrammar AST and modifies a GRule. Following the mutation, a test needs to generated for this mutated grammar. Raselimo et al. do not provide a method for this in their paper [9]. Furthermore, one cannot use Purdom's algorithm [8] straight off, since it is possible that test cases that doesn't use the mutated rule are generated (and the resulting test is not be negative). However, it possible to use the two first parts in Purdom's algorithm to generate short tests that make sure to utilize the mutated rule. The pseudo-code of out implementation is provided below, where `short(sym)` is a method that returns the shortest word derivable from the symbol `sym`, `lhs` means the left-hand-side of a rule and `rhs` means the right-hand-side of a rule:

```
input: GGrammar g  // mutated grammar
input: Symbol s    // start symbol
input: GRule mRule // mutated rule in g
output: test
method genRuleMutationTest:
  set test to NULL
  set SHORT using Purdom first part
  set PREV using Purdom second part
  set m to mRule
  if SHORT[s] equals NULL:
    end method // test cannot be formed
  if PREV[m.lhs] equals NULL:
    end method // test cannot be formed

  set shortM to empty list
  for each symbol i in m.rhs:
    add i to shortM
  if m.lhs equals s:
    set test to shortM
    end method

  set shortW to empty list
  do:
    set p to PREV[m]
    set cWord to empty list
    for each symbol i in p.rhs:
      if i equals mRule.lhs:
        add shortM to cWord
      else if i equals shortW.lhs:
        add shortW to cWord
        else
          add short(i) to cWord
        set shortW to cWord
        set m to p

    set test to shortW
    end method
```

For some mutated grammars it is impossible to generate a test based on how the grammar mutated. This is why we check if `SHORT[s]` or `PREV[m.lhs]` equals `NULL`. Such a scenario might happen due to generated recursion in the CFG. One example of a test that was generated by rule mutation looks as follows:

```
int a() {
  a(1 int a() {
  }
  , 1);
}
```

Here the *function_declaration* non-terminal was added into the *argument_list* rule.

4 Evaluation

To evaluate the implementation, we compare the generated test suite, TS_{gen}, to the teachers' test suite, TS_{ref}. For SimpliC, the implementation generates 7 positive tests and 9316 negative tests. However, since running the student parsers through a large test suite takes a lot of time, only 900 (about 10%) negative tests are included in TS_{gen}. [1] On the other side, TS_{ref}, contains 128 tests, both positive and negative.

Both TS_{gen} and TS_{ref} were run on 724 student parsers, out of which 341 passes TS_{ref}. No generated tests in TS_{gen} were discovered that increases coverage. Furthermore, six submissions were failed by the teachers' test suite that passed the test suite. Figure 1 shows the number of parsers failed by tests for TS_{gen}.

As seen by the distribution, there were about 35 negative tests that failed at least one parser submission. This means that there were 865 negative tests that provided no coverage. The positive test cases failed a lot of parser submissions per test, one even failed almost 250 submissions.

Figure 2 shows the number of submissions failed for tests in TS_{ref}. As seen, there are more than 60 tests that fails parser submissions. This is a larger amount than for TS_{gen}. Furthermore, it is also clear that TS_{ref} contain a higher proportion of positive tests that have an impact than TS_{gen}. One reason for these differences could be that the hand written tests were specifically written to test one language feature, while the generated tests inadvertently could test many features at a time and thus reducing the number of tests needed. Having a test suite with fewer tests is usually an advantage as it saves disk space and also testing time. However, the advantage of being able to track the failed test to a specific issue in the parser grammar probably is the most important aspect in this case. Another reason for there being a higher proportion of positive tests in TS_{ref}, could be the testing order. For TS_{gen} all the negative tests are run first and after this the positive tests are run. This is in contrast to TS_{ref}, where there is no clear execution order. Some tests might not have been needed if another test, covers the same feature. There is an indication of that this is impacting the results, as test 35 in TS_{gen} is identical to test 1 in TS_{ref}; however, they fail a different amount of parser submissions in their respective test suite. This means some submissions that were covered by test 35 in TS_{gen} had already been failed by previous negative tests. These negative tests are perhaps superfluous.

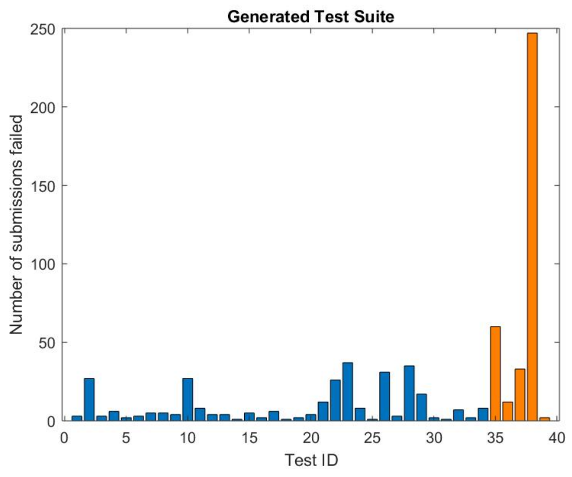

Figure 1. The number of failed parser submissions per test. The blue tests are negative and the orange are positive.

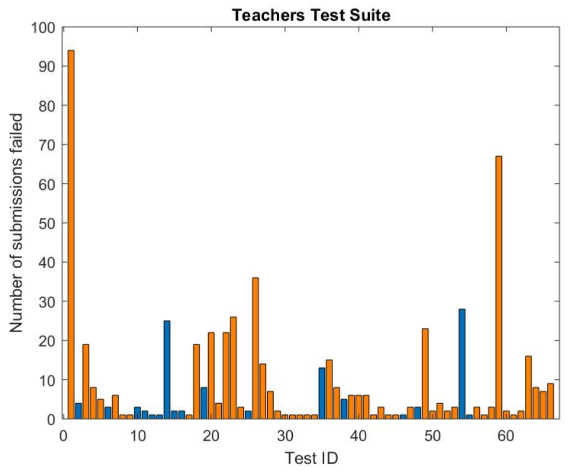

Figure 2. The number of failed parser submissions per test. The blue tests are negative and the orange are positive.

[1]It took almost 3 days to run all the parsers on the tests.

By analyzing the submissions that were failed by TS_{ref}, but passed TS_{gen} we found both positive and negative tests that could not be generated by our implementation. An example of such a positive test case is:

```
int f() {
  return -(3 + 2);
}
```

In this specific case, the rule, factor → MINUS factor, has to be in the derivation of the sentence. The issue is that Purdom's algorithm stipulates that it is only necessary to use a rule once, and the rule, factor → MINUS factor, had already been used to generate another shorter test. Thus, the aforementioned positive test can not be generated by Purdom's algorithm. The other positive tests in TS_{ref} that could not be generated are also caused by the same issue. An example of a negative test case in TS_{ref} that could not be generated is:

```
int f(int x) {
  x = x = 0;
}
```

This negative test case could not be generated since the sentence lacks any poisoned pairs. Thus neither word mutation or rule mutation could result in tests like this. The same is true for all negative test cases that could not be generated.

5 Related work

There are other algorithms for generating test cases for parses than the ones we implemented for NeoTestGen. For example, Lämmel [4] describes a method for positive test case generation called Context-Dependent Rule Coverage (CDRC) that makes sure that every rule is utilized in every potential location. CDRC could potentially generate the test cases that Purdom's algorithm could not. Another method by Lämmel and Schulte [5] is a smart brute force method, where certain factors are limited to avoid an explosive amount of tests.

Toolkits for generating test cases for parsers have been implemented before. For example, Xu, Zheng, and Chen [10] implemented a toolkit which employ variants of Purdom's algorithm and Lämmel's CDRC. However, it has no methods of generating negative test cases.

6 Conclusion

To see if it would be possible to increase the coverage of an existing parser grammar test suite, TS_{ref}, we implemented a tool called NeoTestGen which generates a parser test suite, TS_{gen}, based on a input grammar. The algorithm we implemented for positive test generation is Purdom's algorithm and the algorithms we implemented for negative test generation are word mutation and rule mutation. When we compared TS_{gen} to TS_{ref}, we found that TS_{gen} had less coverage. There were some positive test cases NeoTestGen could not generate due to a limitation of Purdom's algorithm. Also, negative test cases that did not contain poisoned pairs could not be generated.

This tool could be used to generate a test suite; however, one should be wary due its limitations. The major advantage of generating tests using this tool compared to writing them yourself is that it is a lot faster.

For future work, we could use CDRC instead of Purdom's algorithm. This would allow the same level of positive coverage as TS_{ref}. To increase the negative coverage one could perhaps implement a solutions that checks for poisoned strings of length n. With a sufficiently large n all negative test cases in TS_{ref} could be generated. Furthermore, one could also implement a method where you generate parsers based on the grammars from the rule mutation method. These parsers could then be run on TS_{ref}. If a parser passes all tests, then TS_{ref} is missing in coverage.

Acknowledgments

Thank you to Jesper Öqvist and Görel Hedin for invaluable supervision, and thank you to Moeketsi Raselimo and Berndt Fischer for explaining the finer parts of their paper.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2007. *Compilers: Principles, Techniques, and Tools (2Nd Edition).* Pearson Education, Inc., Boston, MA, USA.

[2] Andrew W. Appel and Jens Palsberg. 2003. *Modern Compiler Implementation in Java* (2nd ed.). Cambridge University Press, New York, NY, USA.

[3] Fred J. Damerau. 1964. A Technique for Computer Detection and Correction of Spelling Errors. *Commun. ACM* 7, 3 (March 1964), 171–176. https://doi.org/10.1145/363958.363994

[4] Ralf Lämmel. 2001. Grammar Testing. In *Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering (FASE '01).* Springer-Verlag, Berlin, Heidelberg, 201–216.

[5] Ralf Lämmel and Wolfram Schulte. 2006. Controllable Combinatorial Coverage in Grammar-Based Testing. In *Proceedings of the 18th IFIP TC6/WG6.1 International Conference on Testing of Communicating Systems (TestCom'06).* Springer-Verlag, Berlin, Heidelberg, 19–38. https://doi.org/10.1007/11754008_2

[6] Vladimir I Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. 707–710.

[7] Brian A. Malloy and James F. Power. 2001. An Interpretation of Purdom's Algorithm forAutomatic Generation of Test Cases.

[8] Paul Purdom. 1972. A sentence generator for testing parsers. *BIT Numerical Mathematics* 12, 3 (01 Sep 1972), 366–375. https://doi.org/10.1007/BF01932308

[9] Moeketsi Raselimo, Jan Taljaard, and Bernd Fischer. 2019. Breaking Parsers: Mutation-based Generation of Programs with Guaranteed Syntax Errors. In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2019).* ACM, New York, NY, USA, 83–87. https://doi.org/10.1145/3357766.3359542

[10] Zhiwu Xu, Lixiao Zheng, and Haiming Chen. 2011. A Toolkit for Generating Sentences from Context-Free Grammars. *Int. J. Software*

and Informatics 5 (01 2011), 659–676. https://doi.org/10.1109/SEFM.2010.21