

# Fully Automated Compiler Testing of a Reasoning Engine via Mutated Grammar Fuzzing

Yavuz Koroglu

Dependable Systems Group (DSG)  
Department of Computer Engineering  
Bogazici University  
Istanbul, Turkey  
yavuz.koroglu@boun.edu.tr

Franz Wotawa

Christian Doppler Laboratory  
for Quality Assurance Methodologies  
for Autonomous Cyber-Physical Systems (QAMCAS)  
Institute for Software Technology  
Graz University of Technology  
Graz, Austria  
wotawa@ist.tugraz.at

**Abstract**—A reasoning engine infers logical consequences from a set of fixed axioms and observations. However, before it can make an inference, it must compile the axioms and observations which are given in a predefined format. Any attempt to test the correctness of a reasoning engine assumes that it compiles inputs correctly, but that may not be the case. In this work, we implement a mutated grammar fuzzer to automatically generate tests for the compilation stage of Assumption-based Truth Maintenance System (ATMS), a reasoning engine for model-based diagnosis. We also implement a recognizer as an oracle and automatically evaluate the correctness of compiler output. We automatically generate, execute, and evaluate more than a million tests in two weeks. We show that while tests generated from the true grammar of ATMS find no faults, tests generated from mutated grammars uncover an important fault in the compiler. We also show that mutated grammars achieve higher code coverage with fewer tests and the original grammar cannot cover any code that is not covered by mutated grammars. To the best of our knowledge, ours is the first work that provides a practical implementation and evaluation of a mutated grammar fuzzer. We make the implementation available online along with small examples, tests generated for this paper, and steps to reproduce our experiments.

**Index Terms**—compiler testing, fuzz testing, mutation testing, grammar fuzzing

## I. INTRODUCTION

Reasoning engines infer logical consequences from a set of fixed axioms and observations. Reasoning engines are used in many areas including automated diagnosis [1] and debugging [2] to increase reliability. However, a reasoning engine may increase the reliability of a system only if the reasoning engine itself is reliable. Therefore, testing these engines is important to ensure overall reliability.

Reliability of a reasoning engine is dependent on two functions. First, a reliable reasoning engine must never infer illogical consequences. Second, a reliable reasoning engine must accept only valid inputs and reject all invalid inputs. Therefore, we investigate reasoning engines by analyzing them in two parts, *compiler* and *reasoner*. The compiler ensures that only valid inputs are accepted and the reasoner ensures that all consequences are logical.

In order to test the reasoner, we must first assume that the compiler is reliable. We implement *gFuzzer*, a simple grammar fuzzer, to test this assumption. *gFuzzer* takes a grammar and automatically generates, executes, and evaluates valid test

inputs using that grammar. However, it only generates valid test inputs. We must also test if the reasoning engine accepts some invalid test inputs. In order to make the reasoning engine wrongfully accept an invalid test input, we argue that the invalid test input should be similar to a valid test input. Therefore, we implement *mgFuzzer* on top of *gFuzzer*. *mgFuzzer* is a grammar fuzzer that mutates the original grammar and then fuzzes the mutated grammar, thus enabling invalid test input generation. We propose six mutation operators for *mgFuzzer*. We evaluate *gFuzzer* and *mgFuzzer* by generating tests on a reasoning engine called Assumption-based Truth Maintenance System (ATMS) and show that these mutation operators are useful for finding faults in ATMS and achieve higher code coverage than simple grammar fuzzing.

Main contributions of this paper are as follows.

- 1) We show the effectiveness of fuzzing mutated grammars instead of fuzzing the original grammar. We generate 1,490,388 tests from the original grammar and find no faults whereas we generate 1,026 tests from mutated grammars and find an important fault in the compiler. We show that mutated grammars achieve 8.5% higher coverage and the original grammar cannot cover any code that is not covered by mutated grammars.
- 2) We implement a fully automated approach that generates, executes, and evaluates tests, called *gFuzzer*. *gFuzzer* can fuzz any given grammar in Backus-Naur Form (BNF), execute generated tests on the compiler under test, and evaluate test outputs as passed or failed.
- 3) We implement *mgFuzzer* on top of *gFuzzer*. *mgFuzzer* mutates a grammar and generate tests from the mutated grammar. To the best of our knowledge, ours is the first work that provides a practical implementation and evaluation of a mutated grammar fuzzer. We make the implementation available online [3] along with small examples, tests generated for this paper, and steps to reproduce our experiments.

We organize the remaining of this paper as follows. We describe ATMS and context-free grammars in Section II. We describe our approach in Section III. We evaluate our approach and describe faults reported by our implementation in Section IV. We explain the challenges and issues in Section V.

We discuss the related work in Section VI. We conclude by summarizing our work and future avenues of research in Section VII.

## II. BACKGROUND

### A. Assumption-based Truth Maintenance System (ATMS)

Classical logic (see e.g. [4] for an introduction) like propositional logic or first-order logic provides means for representing knowledge about the world in an accurate way preventing from ambiguous interpretations. When adding new facts or rules to a logical formula or theory, we are always able to derive the same or an increased amount of facts. This monotonicity property of classical logic, however, leads to trouble when dealing with common sense knowledge, i.e., formulae or rules that are not always applicable. Let us consider the following logical theory  $Th$  representing the behavior of two bulbs connected in parallel with a battery. For such a scenario we would expect that both bulbs are illuminated, and whenever we see one of the bulbs transmitting light, there must be a voltage available provided by the battery:

$$Th = \left\{ \begin{array}{l} voltage \rightarrow bulb1\_light, \\ voltage \rightarrow bulb2\_light, \\ (bulb1\_light \vee bulb2\_light) \rightarrow voltage \end{array} \right\}$$

Now let us assume that we observe  $bulb1$  glowing but  $bulb2$  not, e.g.,  $OBS = \{bulb1\_light, \neg bulb2\_light\}$ . When bringing together the observations and the theory, i.e.,  $Th \cup OBS$ , the resulting logical formula becomes inconsistent allowing us to derive anything. Because we are always interested in consistent theories, we have to come up with a method that allows removing inconsistencies, and the ATMS [5] provides such means. The ATMS can handle assumptions, which we write starting with a capitalized letter, directly. An assumption is a proposition where its truth value is set by the ATMS and not the theory itself. The ATMS always tries to make an assumption true unless this leads to inconsistency. Let us have a look at theory  $Th'$  capturing again the battery-bulb circuit. This time we say that the battery is delivering a voltage if it is working as expected, e.g.,  $OK\_bat$  is true. Each bulb can only be illuminated if it is ok, and there is a voltage available.

$$Th' = \left\{ \begin{array}{l} OK\_bat \rightarrow voltage, \\ (OK\_bulb1 \wedge voltage) \rightarrow bulb1\_light, \\ (OK\_bulb2 \wedge voltage) \rightarrow bulb2\_light, \\ (bulb1\_light \vee bulb2\_light) \rightarrow voltage \end{array} \right\}$$

When combining the new theory with the observations  $Th' \cup OBS$  the ATMS computes all sets of assumptions leading to an inconsistency. Such a set is also called a conflict. For  $Th' \cup OBS$  the ATMS only returns  $\{OK\_bulb2\}$  stating that  $bulb2$  has to be broken.

The ATMS and similar methods allowing to retain consistencies for logical formulae are of use for diagnosis [1], [6], self-adaptive systems [7], and also debugging [2].

### B. Context-Free Grammars in Chomsky Normal Form

Our reasoning engine, ATMS, accepts axioms and observations according to a Context-Free Grammar (CFG). Formally, a CFG  $G = (V, \Sigma, R, S)$  is a 4-tuple where

- $V$  is the set of non-terminal rules.
- $\Sigma$  is the set of terminal symbols.
- $R$  is a finite relation from  $V$  to  $(V \cup \Sigma \cup \{\epsilon\})^*$ . It is also called the rule set of the CFG. Each element of  $R$  is called a production rule or *rule* in short. Each element in the domain of  $R$  is called an *expansion* of a non-terminal.
- $S \in V$  is the root.

Our grammar is in Chomsky Normal Form (CNF) so we can develop a recognizer for it. CNF is the same as CFG, except  $R$  is now a finite relation from  $V$  to  $(V \times V) \cup \Sigma \cup \{\epsilon\}$ . This ensures that every non-terminal has only three types of rules. These are two consecutive non-terminals, a single terminal, or an empty string ( $\epsilon$ ).

## III. METHODOLOGY

Our aim in this section is to design and implement a fully automated tool for testing the ATMS compiler. In order to fully automate the testing process, all the following tasks must be automated.

- 1) Test Generation
- 2) Test Execution
- 3) Test Oracle

Fig. 1 shows the overview of our approach. In the test generation phase, we generate tests by giving the ATMS Grammar to two generator tools,  $gFuzzer$  and  $mgFuzzer$ . Dotted lines show the flow of  $gFuzzer$  while dashed lines show the flow of  $mgFuzzer$ . Solid lines are used by both approaches. In the test execution phase, we execute the generated test inputs on ATMS. In the test oracle phase, we use a Cocke-Younger-Kasami (CYK) Recognizer [8] to decide if the test input should be accepted or rejected. If ATMS agrees, then the test is passed, otherwise, the test is failed. Since  $gFuzzer$  uses the original grammar, it always generates valid inputs and therefore we only need the CYK recognizer for  $mgFuzzer$ .

### A. Test Generation

Our aim in this part is to generate tests that check two types of errors in a parser, Type I and Type II. Type I errors are cases where the parser accepts an invalid input (false positive). Type I errors can be exploited to force invalid inferences from the reasoning engine. Type II errors are cases where the parser rejects a valid input (false negative). Type II errors can cause the reasoning engine not to function as it was intended to. We implement two approaches,  $gFuzzer$  and  $mgFuzzer$ , to generate tests for these errors.

1) *Grammar Fuzzer (gFuzzer)*: Generates a random test input from a given grammar. We show how  $gFuzzer$  works in Algorithm 1. The CFG  $G$  is the input of this algorithm. First, we initialize the test input  $t$  as  $S$  in Line 1. The test input  $t$  is basically an  $n$ -tuple where each tuple is an element of  $V \cup \Sigma \cup \{\epsilon\}$ . Initially, there is only one tuple, so we set  $n$  to 1 in Line 2. We denote the  $i^{\text{th}}$  tuple as  $t_i$ . For all tuples, if the tuple is a non-terminal, we pick a random rule to expand the non-terminal in Line 7. We denote the expansion of  $t_i$  as  $t'_i$ . If  $t_i$  is not a non-terminal, we set  $t'_i$  to  $t_i$  in Line 9. We update  $t$  in Line 13. We use the short  $t'_{1,n}$  notation to denote the concatenation  $t'_1 \dots t'_n$ . Note that each  $t_i$  is an element of  $V \cup \Sigma \cup \{\epsilon\}$  whereas each  $t'_i$  is an element of  $(V \cup \Sigma \cup \{\epsilon\})^m$ .

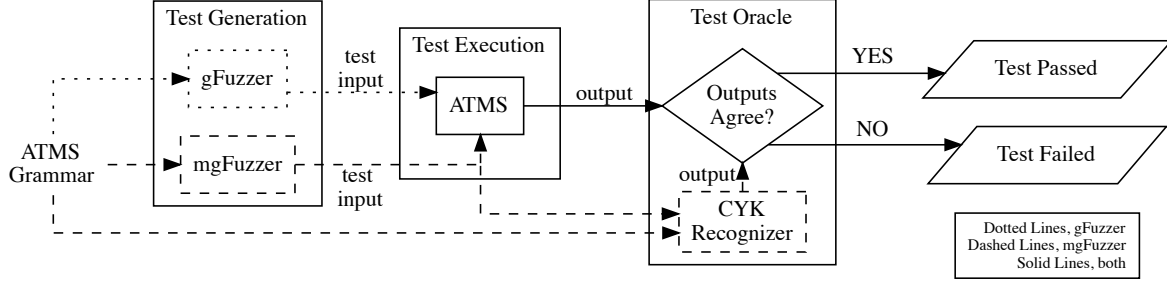


Fig. 1. Overview of Our Approach

---

**Algorithm 1** Grammar Fuzzer (gFuzzer)

---

**Require:** CFG  $G = (V, \Sigma, R, S)$

**Ensure:** Test Input  $t \in (\Sigma \cup \{\epsilon\})^*$

```

1:  $t \leftarrow S$  ▷ Start from the root
2:  $n \leftarrow 1$  ▷ Initially, there is only one tuple
3: repeat
4:    $n' \leftarrow 0$ 
5:   for  $i$  from 1 to  $n$  do ▷ For all tuples in  $t$ 
6:     if  $t_i \in V$  then
7:        $t'_i \leftarrow \text{random } t'_i \text{ s.t. } (t_i, t'_i) \in R$  ▷ Expand  $t_i$ 
8:     else
9:        $t'_i \leftarrow t_i$  ▷ Do not expand  $t_i$ 
10:    end if
11:     $n' \leftarrow n' + m$  where  $t'_i \in (V \cup \Sigma \cup \{\epsilon\})^m$ 
12:  end for
13:   $t \leftarrow t'_{1,n}$  ▷ Update  $t$ 
14:   $n \leftarrow n'$  ▷ Update  $n$ 
15: until  $t \in (\Sigma \cup \{\epsilon\})^*$ 

```

---

This means that with each expansion,  $t$  grows, so we calculate the new tuple count in Line 11 and update  $n$  accordingly in Line 14. We repeat this process until there are no non-terminals left in  $t$ .

We give the grammar of ATMS to *gFuzzer* so it generates valid inputs. *gFuzzer* always generates valid inputs, so it only checks Type II errors.

We need invalid inputs to test for Type I errors. ATMS immediately rejects completely random strings, so we argue that we should generate invalid inputs that look similar to the valid ones in order to fool ATMS. Therefore, we develop grammar mutation operators to perturb the original CFG so that *gFuzzer* generates invalid inputs that look similar to the valid ones.

Formally, a grammar mutation operator or *mutation operator* in short, is a partial function that takes a CFG and returns a modified CFG. We denote it as  $\delta(O) = G$  where  $\delta$  is the mutation operator function,  $O$  is the original CFG and  $G$  is the mutated CFG. Mutation operators are partial functions because there may be grammars for which a mutation operator is inapplicable and therefore not defined.

2) *Mutated Grammar Fuzzer (mgFuzzer)*: Generates a random test input that is probably invalid. We show how *mgFuzzer* works in Algorithm 2. There are two inputs, the

---

**Algorithm 2** Mutated Grammar Fuzzer (mgFuzzer)

---

**Require:** CFG  $O = (V, \Sigma, R, S)$  and Mutation Set  $\Delta$

**Ensure:** Test Input  $t \in (\Sigma \cup \{\epsilon\})^*$

```

1:  $\delta \leftarrow \text{random } \delta \in \Delta \text{ s.t. } \delta(O) \text{ is defined}$ 
2:  $G \leftarrow \delta(O)$ 
3: execute Algorithm 1

```

---

original CFG  $O$  and a mutation set  $\Delta$ . A *mutation set* is a set of mutation operators. First, we pick a random mutation operator  $\delta$  from  $\Delta$  such that  $\delta$  is defined on the original CFG  $O$ . Then we apply the mutation operator  $\delta$  on the original CFG  $O$  and get the mutated grammar  $G$ . Then we execute Algorithm 1 with the mutated grammar  $G$ . Note that, for each test input, we re-mutate the original grammar.

We define six mutation operators for *mgFuzzer* through Algorithms 3-8. We based our mutation operators in Algorithms 3-5 on previous work [9]. To the best of our knowledge, all the remaining mutation operators are novel.

- 1) Terminal Replacement ( $\delta_{TR}$ ) : Swaps two terminals of the grammar. Formally, it finds two rules  $(A, a), (B, b) \in R$  such that  $\exists i, j$  where  $a_i, b_j \in \Sigma$  and  $a_i \neq b_j$ . Then in the mutated rule set  $R'$ , these rules are replaced with  $(A, a_{1,i-1} \cdot b_j \cdot a_{i+1,n})$  and  $(B, b_{1,j-1} \cdot a_i \cdot b_{j+1,m})$ . This mutation operator is *undefined* for CFGs that has no such a pair of rules.
- 2) Deletion ( $\delta_{DE}$ ) : Replaces all rules of a random non-terminal other than the root symbol of a CFG with empty string. Formally, it finds a non-terminal  $A \in V - \{S\}$  and ensures that  $(A, \epsilon) \in R'$  is the only rule of  $A$  in the mutated CFG. This mutation operator is *undefined* for CFGs where  $V = \{S\}$ .
- 3) Duplication ( $\delta_{DU}$ ) : Duplicates a rule. Formally, it randomly picks a non-terminal  $A \in V$ . Then it creates a new non-terminal  $A'$  such that  $V' = V \cup \{A'\}$  where  $V'$  is the mutated set of non-terminals. In the mutated rule set  $R'$ , all previous rules  $(A, p) \in R$  are replaced with  $(A', p) \in R'$  and the new rule  $(A, A') \in R'$  is added. This operator is defined for all CFGs.
- 4) Exchange ( $\delta_{EX}$ ) : Finds a rule that has exactly two non-terminals and nothing else. Then it swaps these non-terminals. Formally, it finds a non-terminal  $A \in V$  such that  $(A, B \cdot C) \in R$  where  $B, C \in V$  and  $B \neq C$ . Then in the mutated rule set  $R'$ , it replaces this rule

**Algorithm 3** Terminal Replacement

---

```

1: function  $\delta_{\text{TR}}(O = (V, \Sigma, R, S))$ 
2:    $a_i \leftarrow \text{random } a_i \text{ s.t. } \exists(A, a) \in R, \exists i, a_i \in \Sigma$ 
3:    $b_j \leftarrow \text{random } b_j \text{ s.t. } \exists(B, b) \in R, \exists j, b_j \in \Sigma - \{a_i\}$ 
4:    $R' \leftarrow R - \{(A, a), (B, b)\}$ 
5:    $R' \leftarrow R' \cup \{(A, a_{1,i-1} \cdot b_j \cdot a_{i+1,n})\}$ 
6:    $R' \leftarrow R' \cup \{(B, b_{1,j-1} \cdot a_i \cdot b_{j+1,m})\}$ 
7:   return  $G = (V, \Sigma, R', S)$ 
8: end function

```

---

**Algorithm 4** Deletion

---

```

1: function  $\delta_{\text{DE}}(O = (V, \Sigma, R, S))$ 
2:    $A \leftarrow \text{random } A \in V - \{S\}$ 
3:    $R' \leftarrow R - \{(A, p) | \forall p, (A, p) \in R\} \cup \{(A, \epsilon)\}$ 
4:   return  $G = (V, \Sigma, R', S)$ 
5: end function

```

---

**Algorithm 5** Duplication

---

```

1: function  $\delta_{\text{DU}}(O = (V, \Sigma, R, S))$ 
2:    $A \leftarrow \text{random } A \in V$ 
3:    $V' \leftarrow V \cup \{A'\}$ 
4:    $R' \leftarrow R \cup \{(A', p) | \forall p, (A, p) \in R\}$ 
5:    $R' \leftarrow R' - \{(A, p) | \forall p, (A, p) \in R\}$ 
6:    $R' \leftarrow R' \cup \{(A, A' \cdot A')\}$ 
7:   return  $G = (V', \Sigma, R', S)$ 
8: end function

```

---

with  $(A, C \cdot B)$ . This operator is *undefined* for CFGs which do not contain non-terminals with a rule that has two non-terminals.

- 5) Recursion Insertion ( $\delta_{\text{RI}}$ ) : Makes a rule recursive. Formally, it finds a rule  $(A, a) \in R$  and adds  $(A, A \cdot a)$  to the mutated rule set  $R'$ . This operator is defined for all CFGs.
- 6) Terminal Insertion ( $\delta_{\text{TI}}$ ) : Randomly picks a rule and inserts a terminal to the rule. Formally, it randomly picks a rule  $(A, a) \in R$ . In the mutated rule set, the rule is replaced with either  $(A, x \cdot a)$  or  $(A, a \cdot x)$  where  $x \in \{a, A, ., !, @, \&, \%, +, ?, *, 0, 1, -, \_, ;\}$ . This mutation operator is defined for all CFGs and it is the only mutation operator that modifies the set of terminals  $\Sigma$ .

Test inputs generated by *mgFuzzer* are not always invalid. Test inputs generated by *mgFuzzer* might not kill the mutation, in other words, the test input may not be generated by a mutated rule. Then the test input must be valid. Even if the test input kills the mutation, it still might be accidentally valid. Therefore, *mgFuzzer* generates tests for both Type I and Type II errors.

**B. Test Execution**

ATMS originally takes inputs from a Graphical User Interface (GUI). We modified the source code of ATMS so it now accepts text input, parses the input and gives either a compile failed message or all the inferred consequences as output. We

**Algorithm 6** Exchange

---

```

1: function  $\delta_{\text{EX}}(O = (V, \Sigma, R, S))$ 
2:    $(A, B, C) \leftarrow$ 
       random  $A, B, C \in V$ 
       s.t.  $B \neq C, (A, B \cdot C) \in R$ 
3:    $R' \leftarrow R - \{(A, B \cdot C)\}$ 
4:    $R' \leftarrow R' \cup \{(A, C \cdot B)\}$ 
5:   return  $G = (V, \Sigma, R', S)$ 
6: end function

```

---

**Algorithm 7** Recursion Insertion

---

```

1: function  $\delta_{\text{RI}}(O = (V, \Sigma, R, S))$ 
2:    $(A, a) \leftarrow \text{random } (A, a) \in R$ 
3:    $R' \leftarrow R \cup \{(A, A \cdot a)\}$ 
4:   return  $G = (V, \Sigma, R', S)$ 
5: end function

```

---

**Algorithm 8** Terminal Insertion

---

```

1: function  $\delta_{\text{TI}}(O = (V, \Sigma, R, S))$ 
2:    $\Sigma_{\text{TI}} \leftarrow \{a, A, ., !, @, \&, \%, +, ?, *, 0, 1, -, \_, ;\}$ 
3:    $A \leftarrow \text{random } A \in V$ 
4:    $x \leftarrow \text{random } x \in \Sigma_{\text{TI}}$ 
5:    $R' \leftarrow \text{random } R' \in \{R \cup \{(A, A \cdot x)\}, R \cup \{(A, x \cdot A)\}\}$ 
6:    $\Sigma' \leftarrow \Sigma \cup \Sigma_{\text{TI}}$ 
7:   return  $G = (V, \Sigma', R', S)$ 
8: end function

```

---

TABLE I  
CONFUSION MATRIX OF CYK RECOGNIZER AND ATMS OUTPUTS

Recognizer	ATMS	Compile Failure	Consequence List
	Reject	Test Passed	Type I Error
	Accept	Type II Error	Test Passed

feed the fuzzer output to the modified ATMS and feed the modified ATMS output to the test oracle.

**C. Test Oracle**

Our aim in this part is to design an automated test oracle that decides if the test is failed or passed. In our case, the test is passed only if the input is parsed correctly. Test inputs generated by *gFuzzer* are always valid, so the test is passed if and only if ATMS does not produce compile failure message. For inputs generated by *mgFuzzer*, we require a recognizer that ultimately decides whether the test input should be accepted by a given grammar or not.

We implement a Cocke-Younger-Kasami (CYK) Recognizer [8], which is a generic bottom-up parser for context-free grammars in Chomsky Normal Form. The CYK Recognizer outputs either *accept* or *reject* whereas ATMS outputs either a compile failure message or a list of consequences. Table I shows how our automated oracle decides if the test is passed or not by looking at the outputs of CYK Recognizer and ATMS. The test is passed if and only if the two outputs agree, otherwise, failed with either a Type I or Type II error.

TABLE II  
TEST GENERATION RESULTS FOR *gFuzzer* AND *mgFuzzer* IN ONE WEEK

	Failed	Passed	Total	Rule Cov. (%)	Code Cov. (%)
<i>gFuzzer</i>	0	1,490,388	1,490,388	100	67.6
<i>mgFuzzer</i>	2	1,024	1,026	100	75.9
Both	2	1,491,412	1,491,414	100	75.9

TABLE III  
FAILED TEST INPUTS

Input #1	x1, falsex2()x3->x2. Assumption1.
Input #2	Assumption1, x2(false, x3) false. Assumption3. Assumption2. x1.

#### IV. EVALUATION

We executed *gFuzzer* and *mgFuzzer* on ATMS, each for one week. We show the test results in Table II.

Even though we executed both tools for the same duration, *mgFuzzer* generated much fewer test inputs than *gFuzzer*. This slowdown is caused by the CYK Recognizer. The largest test input created by *mgFuzzer* has 26544 statements, which corresponds to more than 100K tokens. CYK Recognizer's time complexity is  $O(n^3)$  where  $n$  denotes the number of tokens. We also note that there can be multiple tokenizations of the test input and in that case, CYK Recognizer has to check all of them. Processing this input alone took slightly less than a day with our implementation.

Although *gFuzzer* generated almost 1.5 million tests, it did not find a single failed test input whereas *mgFuzzer* generated two tests that reveal the same important fault in the ATMS. We show these failed tests in Table III. Input #1 is generated by Recursion Insertion while Input #2 is generated by Deletion. We figured out that ATMS had a Type I error that allowed a predicate immediately followed by another if the first predicate ends with a paranthesis. For these test inputs, ATMS reported unexpected consequences. Unexpected consequences may trigger fail-safe behavior in no-fail conditions. This is an important potential vulnerability for safety-critical systems. For example, an autonomous vehicle may be forced to pull over and stop, or worse, crash for no reason.

We collected the cumulative *rule coverages* of *gFuzzer* and *mgFuzzer*. Both tools achieved 100% rule coverage. Test suites generated by our tools are at least equivalent in coverage to a test suite generated by Purdom's algorithm [10]. This is a known baseline in compiler testing [11].

We also collected *code coverages* of the ATMS parser using EclEmma [12]. Table II shows that *gFuzzer* achieved 67.6% whereas *mgFuzzer* achieved 75.9% coverage. Furthermore, when both tests suites are combined, we notice that the coverage is still 75.9%. This shows us that *gFuzzer* could not cover any code that is not covered by *mgFuzzer*. Overall, our evaluation shows that *mgFuzzer* achieved 8.5% higher coverage with fewer tests. Further investigation reveals that the code that is not covered by our tools consists unused constructors and methods unrelated to the compiler.

#### V. DISCUSSION

##### A. Challenges

Some aspects of practically generating tests with our approach proved to be challenging. Fig. 2 shows the grammars we used during our study. These grammars are represented in Backus-Naur Form (BNF) and furthermore we support *regular expressions* as terminal symbols. This allowed us to state the variable naming conventions of ATMS in a convenient manner. During our initial attempts, CYK Recognizer did not recognize some of the valid inputs because we were using the grammar in Fig. 2a both for Test Generation and Test Oracle phases. This is a *generative grammar* designed to have the same variable in the same test input multiple times which is expected from a valid test input. However, mutation creates unrecognizable variable names because the grammar did not include them. We created the *recognizer grammar* in Fig. 2b in order to deal with this issue. In the end, we had to prepare slightly different grammars for the Test Generation and Test Oracle phases which increased the manual effort for test generation a little, but at least we did not interfere during the remainder of the testing process.

It turned out that the original grammar was incomplete. ATMS allowed Prolog style factual declarations such as "`:- rainy, wet.`" where there were no preconditions for being rainy and wet, these are just facts. However, the original grammar did not have such a rule and we had to add one after our initial attempts. We also noticed that the percent sign (%) denotes a comment, so our Terminal Insertion operator sometimes just commented out lines and therefore generated valid inputs which were not accepted by CYK Recognizer. We did not fix the issue but marked those tests as passed. There were eight such test cases in total.

ATMS normally accepts a fully empty string ( $\epsilon$ ), however, we did not add this to the grammar. This is because if we had added, for example, `<start> ::=  $\epsilon$` , half of the generated test inputs would be fully empty since *gFuzzer* selects rules completely randomly. We excluded this rule to generate more meaningful tests.

##### B. Issues

Now, we explain some of the issues regarding our approach. First, theoretically, there is a chance that Algorithm 1 never terminates, because it may always expand a non-terminal to more non-terminals. However, the probability is practically zero and our tools did not halt during evaluation.

Second, we made a known optimization to CYK Recognizer that assumes two alphanumeric words never appear next to each other without a separator. This assumption allows us to consume variable names as single tokens, instead of a number of character tokens. Although this optimization made CYK Recognizer significantly faster, we must keep in mind to use grammars that obey this assumption in future testing endeavors.

Third, we assumed that ATMS would not crash during evaluation. In fact, ATMS did not crash during evaluation. Still, we believe it is trivial to make the test oracle label the test as failed whenever the tool crashes. We believe our approach

<start>	::= <first_rule> <start>
<first_rule>	::= <rule> <rule_end>
<rule>	::= <atom> <entailments>
<rule>	::= <implications> <atom>
<rule>	::= <implies> <atom>
<rule>	::= 'x[1 2 3]'
<rule>	::= 'Assumption[1 2 3]'
<rule>	::= 'false'
<entailments>	::= <entails> <atom_list>
<implications>	::= <atom_list> <implies>
<atom_list>	::= <atom> <atom_list_rest>
<atom_list>	::= <id> <opt_args>
<atom_list>	::= 'x[1 2 3]'
<atom_list>	::= 'Assumption[1 2 3]'
<atom_list>	::= 'false'
<atom_list_rest>	::= <separator> <atom_list>
<atom>	::= <id> <opt_args>
<atom>	::= 'x[1 2 3]'
<atom>	::= 'Assumption[1 2 3]'
<atom>	::= 'false'
<opt_args>	::= <open_par> <close_par>
<opt_args>	::= <open_par> <args>
<args>	::= <atom_list> <close_par>
<open_par>	::= '('
<close_par>	::= ')'
<id>	::= 'x[1 2 3]'
<id>	::= 'Assumption[1 2 3]'
<id>	::= 'false'
<separator>	::= ','
<entails>	::= ':-'
<implies>	::= '->'
<rule_end>	::= '.\n'

(a) Generative Grammar

<start>	::= <first_rule> <start>
<first_rule>	::= <rule> <rule_end>
<rule>	::= <atom> <entailments>
<rule>	::= <implications> <atom>
<rule>	::= <implies> <atom>
<rule>	::= '^ [a-zA-Z]+ [a-zA-Z0-9_]*\$'
<entailments>	::= <entails> <atom_list>
<implications>	::= <atom_list> <implies>
<atom_list>	::= <atom> <atom_list_rest>
<atom_list>	::= <id> <opt_args>
<atom_list>	::= '^ [a-zA-Z]+ [a-zA-Z0-9_]*\$'
<atom_list_rest>	::= <separator> <atom_list>
<atom>	::= <id> <opt_args>
<atom>	::= '^ [a-zA-Z]+ [a-zA-Z0-9_]*\$'
<opt_args>	::= <open_par> <close_par>
<opt_args>	::= <open_par> <args>
<args>	::= <atom_list> <close_par>
<open_par>	::= '('
<close_par>	::= ')'
<id>	::= '^ [a-zA-Z]+ [a-zA-Z0-9_]*\$'
<separator>	::= ','
<entails>	::= ':-'
<implies>	::= '->'
<rule_end>	::= '.\n'

(b) Recognizer Grammar

Fig. 2. ATMS Grammars for Test Generation and Test Oracle Phases

is applicable to other testing problems if the crash detection support is added to the test oracle.

Fourth, for the sake of this study, we assume that *gFuzzer*, *mgFuzzer*, and CYK Recognizer are all correctly implemented. All these tools are available online [3].

Finally, rule coverage should never be thought as the de facto measure for functionality coverage. We rather use it as a baseline to show that the most basic functionalities are tested.

## VI. RELATED WORK

Compilers are almost used anywhere and it is, therefore, no surprise that validation and verification of compilers have been in the focus of research. Early publications in this area include Purdom [10] who introduced an algorithm for generating input sentences for a parser. Kossatchev and Posypkin [11] summarized previous work on compiler testing. There the authors also discuss the different aspects to be considered when testing compilers including parser but also optimizers and other compiler-related parts. Chen et al. [13] compared different compiler testing techniques with respect to their ability to detect faults.

*Athena* [14] and *Hermes* [15] are tools that are able to find *deep bugs* in C compilers. There are two key differences between *Athena/Hermes* and *mgFuzzer*. First, *Athena* and *Hermes* directly mutate the test input whereas *mgFuzzer* mutates the grammar. Second, *Athena* and *Hermes* always generate valid test inputs and check only the semantics. *mgFuzzer*

generates both valid and invalid inputs and checks only the syntax. From this perspective, *mgFuzzer* and *Athena/Hermes* are complementary.

Palka et al. [16] show that complex compiler optimization may introduce bugs. They propose a random generation tool dedicated to finding bugs caused by compiler optimization. Their tool is similar to *gFuzzer* without grammar mutation.

We mention that we generate large test inputs during evaluation. If these test inputs are going to be reused later, they have to be stored now and then reevaluated at the time of re-testing. Large test inputs require a lot of storage space and take a lot of time to evaluate due to CYK Recognizer. Chen et al. [17] suggest that test case reduction may be used to trim test inputs. This way, we can improve on both space and time requirements of large test inputs. Another work by Chen et al. [18] suggests test case prioritization. One simple idea for our case would be to store CYK Recognizer outputs along with test inputs. This improves on time but not on space. During our evaluation, we generated our test inputs for the first time so we did not implement these ideas.

Fuzz testing was found in 1990 by Miller, Frederiksen, and So [19]. They applied simple fuzzing to UNIX utilities and found crashes in 33% of them. Grammar fuzzing was introduced in 2008 by Godefroid, Kiezun, and Levin [20]. Since then, many automated grammar-fuzzing tools were implemented [21]–[24]. These tools were successful at finding bugs in web browsers and JavaScript interpreters.

Mutation testing traditionally refers to inserting mutations into program code. In 2006, Offut, Amman, and Liu [9] proposed mutating grammars. We take three of our mutation operators, Terminal Replacement ( $\delta_{TR}$ ), Deletion ( $\delta_{DE}$ ), and Duplication ( $\delta_{DU}$ ) from their paper. To the best of our knowledge, the other three mutation operators are novel.

Based on the proposition of Offut, Amman, and Liu, a recent work of Arcaini, Gargantini, and Riccobene [25] proposes a tool called *MutRex* that mutates a regular expression to generate test inputs that check if the regular expression represents the strings that it was intended to. To the best of our knowledge, this work is the closest to our work. The main difference is that they work on regular expressions whereas our approach works on context-free grammars.

Note that our work should not be confused with grammar-based mutation analysis [9]. Grammar-based mutation analysis involves generating test inputs with a simple grammar fuzzer and then using these inputs as the seed for mutation-based analysis. The key difference is that the original test inputs are mutated instead of the grammar.


To the best of our knowledge, ours is the first work that provides a practical implementation and evaluation of a mutated grammar fuzzer.

## VII. CONCLUSION

We implemented a fully automated approach which generates, executes and evaluates tests for ATMS. We argued that fuzzing mutated grammars is more effective than fuzzing the original grammar since *gFuzzer* found no failed tests after generating 1,490,388 tests whereas *mgFuzzer* found an important fault in the compiler after generating only 1,026 tests. *mgFuzzer* achieved 8.5% higher coverage with fewer tests and *gFuzzer* could not cover any code that is not covered by *mgFuzzer*. We explained our approach in detail and discussed challenges in implementing a practical mutated grammar fuzzer. We made the implementation available online [3] along with small examples, tests generated for this paper, and steps to reproduce our experiments.

A complete tester for a reasoning engine should also check if the reasoning engine is *correct*, in other words, check whether the list of consequences is sound or not. In the future, we aim to implement fully automated testing for the reasoner by utilizing SAT solvers. We also aim to compare our mutated grammar fuzzer with mutation-based analysis of grammars.

## ACKNOWLEDGMENT

The research was supported by ECSEL JU under the project H2020 737469 AutoDrive - Advancing fail-aware, fail-safe, and fail-operational electronic components, systems, and architectures for fully automated driving to make future mobility safer, affordable, and end-user acceptable. AutoDrive is funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) under the program "ICT of the Future" between May 2017 and April 2020. More information <https://iktderzukunft.at/en/> . The financial support by the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development is gratefully acknowledged.

## REFERENCES

- [1] S. P. Christopher S. Gray, Roxane Koitz and F. Wotawa, "An abductive diagnosis and modeling concept for wind power plants," in *9th IFAC symposium on fault detection, supervision and safety of technical processes*, 2015.
- [2] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, 2016.
- [3] TUGraz, "Grammar fuzzer (gfuzzer)," 2019. [Online]. Available: <https://github.com/yavuzkoroglu/gfuzzer-release>
- [4] C.-L. Chang and R. C.-T. Lee, *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.
- [5] J. de Kleer, "An assumption-based TMS," *Artificial Intelligence*, vol. 28, pp. 127–162, 1986.
- [6] R. Reiter, "A theory of diagnosis from first principles," *Artificial Intelligence*, vol. 32, no. 1, pp. 57–95, 1987.
- [7] F. Wotawa, "Reasoning from first principles for self-adaptive and autonomous systems," in *Predictive Maintenance in Dynamic Systems — Advanced Methods, Decision Support Tools and Real-World Applications*, E. Lughofer and M. Sayed-Mouchaweh, Eds. Springer, 2019.
- [8] T. Kasami, "An efficient recognition and syntax-analysis algorithm for context-free languages," *Coordinated Science Laboratory Report no. R-257*, 1966.
- [9] J. Offut, P. Ammann, and L. Liu, "Mutation testing implements grammar-based testing," in *Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006)*, 2006.
- [10] P. Purdom, "A sentence generator for testing parsers," *BIT*, vol. 12, pp. 366–375, 01 1972.
- [11] A. S. Kossatchev and M. A. Posypkin, "Survey of compiler testing methods," *Programming and Computer Software*, vol. 31, no. 1, pp. 10–19, Jan 2005.
- [12] M. R. Hoffman, "Eclemma: Java code coverage for eclipse," 2006. [Online]. Available: <https://www.eclemma.org>
- [13] J. Chen, W. Hu, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "An empirical comparison of compiler testing techniques," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, May 2016, pp. 180–190.
- [14] V. Le, C. Sun, and Z. Su, "Finding deep compiler bugs via guided stochastic program mutation," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015. New York, NY, USA: ACM, 2015, pp. 386–399.
- [15] C. Sun, V. Le, and Z. Su, "Finding compiler bugs via live code mutation," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2016. New York, NY, USA: ACM, 2016, pp. 849–863.
- [16] M. H. Palka, K. Claessen, A. Russo, and J. Hughes, "Testing an optimising compiler by generating random lambda terms," in *Proceedings of the 6th International Workshop on Automation of Software Test*, ser. AST '11. New York, NY, USA: ACM, 2011, pp. 91–97.
- [17] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: ACM, 2013, pp. 197–208.
- [18] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "Test case prioritization for compilers: A text-vector based approach," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, April 2016, pp. 266–277.
- [19] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990.
- [20] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08, 2008, pp. 206–215.
- [21] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11, 2011.
- [22] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proceedings of the 21st USENIX Conference on Security Symposium*, ser. Security '12, 2012.
- [23] T. Guo, P. Zhang, X. Wang, and Q. Wei, "Gramfuzz: Fuzzing testing of web browsers based on grammar analysis and structural mutation," in *2013 Second International Conference on Informatics Applications (ICIA)*, 2013.
- [24] R. Hodovan, A. Kiss, and T. Gyimothy, "Grammarinator: a grammar-based open source fuzzer," 11 2018, pp. 45–48.
- [25] P. Arcaini, A. Gargantini, and E. Riccobene, "Mutrex: A mutation-based generator of fault detecting strings for regular expressions," in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2017.