Human-in-the-Loop Oracle Learning for Semantic Bugs in String Processing Programs

Charaka Geethal Kapugama
charaka.kapugamawasangamagedon@monash.edu
Monash University, Australia

Van-Thuan Pham
thuan.pham@unimelb.edu.au
University of Melbourne, Australia

Aldeida Aleti
aldeida.aleti@monash.edu
Monash University, Australia

Marcel Böhme
marcel.boehme@acm.org
MPI-SP, Germany; Monash University, Australia

ABSTRACT

How can we automatically repair semantic bugs in string-processing programs? A *semantic bug* is an unexpected program state: The program does not crash (which can be easily detected). Instead, the program processes the input incorrectly. It produces an output which users identify as unexpected. We envision a fully automated debugging process for semantic bugs where a user reports the unexpected behavior for a given input and the machine negotiates the condition under which the program fails. During the negotiation, the machine learns to predict the user's response and in this process learns an automated oracle for semantic bugs.

In this paper, we introduce Grammar2Fix, an automated oracle learning and debugging technique for string-processing programs even when the input format is unknown. Grammar2Fix represents the oracle as a regular grammar which is iteratively improved by systematic queries to the user for other inputs that are likely failing. Grammar2Fix implements several heuristics to maximize the oracle quality under a minimal query budget. In our experiments with 3 widely-used repair benchmark sets, Grammar2Fix predicts passing inputs as passing and failing inputs as failing with more than 96% precision and recall, using a median of 42 queries to the user.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

Automated Test Oracle, Software Debugging, Grammar Inferencing

ACM Reference Format:
Charaka Geethal Kapugama, Van-Thuan Pham, Aldeida Aleti, and Marcel Böhme. 2022. Human-in-the-Loop Oracle Learning for Semantic Bugs in String Processing Programs. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3533767.3534406

ISSTA '22, July 18–22, 2022, Virtual, South Korea
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9379-9/22/07.
https://doi.org/10.1145/3533767.3534406

1 INTRODUCTION

The recent hype around Github Co-pilot shows:Developers are striving for the automation of mundane software development tasks in order to focus on the more creative aspects of programming. A recent study [25] on developer trust for automated program repair (APR) revealed that the majority of developers (72%) would review auto-generated patches. The most-widely used APR tools are test-suite-driven [16, 21, 23, 27, 28]. In their seminal work introducing GenProg [16], Le Goues et al. cast the APR problem as an optimization problem: Given a program P and a failing test suite T, find a version of P that maximizes the number of passing test cases in T. Nguyen et al. [23] cast the APR problem as a constraint satisfaction problem where the pertinent part of the program is encoded as a constraint, a culprit statement is identified, and finally fixed using constraint-based program synthesis. However, in practice there is often only a single user-provided failing test input instead of a full-fledged test suite. How can we facilitate the automated repair of semantic bugs with a single failing test input where only the users can really tell whether an input was processed correctly?

Recently, Böhme, Geethal, and Pham [6] proposed a framework within which the machine, by systematic queries to the user, would learn an SMT(Linear Real Arithmetic(LRA))-constraint under which a number-processing program fails. All inputs that satisfy this numerical constraint over inequalities are predicted as failing. However, the proposed framework is fundamentally limited to number-processing programs and builds on recent advances in learning for *numerical constraints* [13]. The problem of learning automated oracles for other types of input remains open.

In this paper, we investigate the problem of learning an automatic oracle for *string-processing programs*. Unlike numeric inputs, string inputs often follow a certain structure or format. This structure is often represented by a grammar. While we are excited about recent advances in solving string constraints [34], we are not aware of automated learners of string constraints. Instead, we explore techniques from automated *grammar learning* to negotiate the failure condition for string-processing programs—even when the original input grammar is unknown. We develop several heuristics, e.g., based on input minimization, to maximize the prediction accuracy of the grammar-based bug oracle given only a limited query budget. Our experiments demonstrate that Grammar2Fix can learn grammar-based oracles with over 90% accuracy, using about 40 queries, with these heuristics.

Discovering the failure condition through a single failing string input is a challenging task. For a single failing input, there could be multiple reasonable explanations for why it fails. Let's take a simplified example. The input nnn might fail (i) because it contains exactly three characters, (ii) because it starts with the letter n, (iii) because it contains only the letter n but no other letter, or (iv) another reason. We observe (1) that *these failure conditions are indistinguishable without further failing inputs* (e.g., nil also fails) and (2) that *the size of the string-processing* program *is insubstantial when describing (or representing) the failure condition.*

Grammar2Fix builds on techniques from automated grammar inference [19]. However, before we could apply existing grammar inference techniques to our task we had to overcome several challenges. Regular Positive and Negative Inference (RPNI) [8], the GOLD algorithm [10] and L^* [3] are some regular grammar inference techniques while Inductive CYK [22] is a popular context-free grammar inference technique. All these algorithms work under a finite alphabet that should be predetermined. In addition, as the alphabet gets larger, the number of examples required for accurate grammar inference significantly increases. Similarly, many queries have to be sent to the oracle in active grammar inference algorithms (e.g. L^*). The grammar representing the failure condition can be inferred from failing inputs; however, we cannot expect that failing inputs have a small alphabet in most cases. Therefore, under these algorithms, many training examples (failing inputs) would be required to infer a grammar for a failure condition, which is a significant limitation of these techniques.

In this paper, we present an active grammar inference approach and effective heuristics to infer a general condition by queries to the user that a string input should satisfy to expose an erroneous program execution.

The inference process begins with one failing input of the bug. First, Grammar2Fix finds the smallest failing input. The user-provided failing input can have components (character sequences) not contributing to the failure. Removing such components helps to identify the actual cause of the failure. Grammar2Fix applies a test case minimization algorithm [33] to the given failing input to remove the components not contributing to the failure and to trace the *minimal input* reproducing the failure from the given failing input. The one-minimal failing input cannot be divided further to have more failing inputs [33]. As a by-product of the minimization, the test case minimization procedure generates some additional *passing* and *failing* inputs.

Second, from the minimized input and the test inputs generated during minimization, Grammar2Fix constructs a first draft of the failure condition, called basic level grammar. The *basic level grammar* is a grammar that accepts all the failing inputs and rejects all the passing inputs obtained during minimization. To construct it, we apply *Regular Positive and Negative Inference* (RPNI) [8] with a slight modification to its merging algorithm. This results in a *Deterministic Finite Automata* (DFA) [14] that describes how the minimal failing input should be positioned in the failing inputs obtained during test case minimization. This DFA represents the basic level grammar.

Third, Grammar2Fix generalizes the basic level grammar to accept more failing inputs. The basic level grammar can be over-fitting with the training examples. Thus, Grammar2Fix applies *Mutational Fuzzing* [7] on the initial failing inputs to generate more failing inputs. For each newly generated failing input, we apply the first three steps of the algorithm, develop a new grammar, and combine it with the previous grammar. At the end of the process, Grammar2Fix returns a collection of DFAs connected via disjunction that together represent the generalized failure condition of the buggy program.

Grammar2Fix generates new test inputs in failure grammar inference which are then labelled by the user. Concretely, when requested, the user labels a generated input provided together with the corresponding actual output as passing or failing. The objective of our failure grammar inference is to learn to predict accurately the label that a user would assign to a test input. The labelled test inputs are used to create the repair test suite to be used with an automated program repair tool.

We conducted several experiments using 329 program subjects, containing bugs of varying complexity, from three (3) widely-used program repair benchmark sets. Our results demonstrate that

(1) Grammar2Fix induces high-quality automated oracles. For the majority of bugs, the learned oracle identifies failing test inputs with more than 97% precision and recall.
(2) Each heuristic step in Grammar2Fix improves the quality of the learned oracle. The proportion of correctly identified failing inputs increases from <20% for the first step to 97% for the last.
(3) For the majority of bugs, a human would need to answer at most 42 "Yes/No" questions about the correct processing of string inputs (i.e., less than 3 minutes assuming 4 seconds per question). Given that bugs in stable real-world systems are quite rare and that the query load can otherwise be distributed, we believe that this human effort is reasonable.
(4) Grammar2Fix produces high-quality patches. The median patch produced by Grammar2Fix passes all (100%) of validation tests for the corresponding subject while the median patch produced using the manually test suites that are provided with the repair benchmark passes only about 90% of validation tests.

In summary, the *main contributions* of this work are as follows:

(1) We introduce a systematic approach to discover the condition under which a string-processing program fails based on a single failing string input. This failure condition is explicitly represented as a grammar.
(2) We introduce a set of techniques to query the user systematically with alternative string inputs in the active grammar inference. Furthermore, we present some heuristics to reduce the number of queries to the user to learn the grammar under minimal examples.
(3) We present an approach to generate a repair test suite that test-driven automated program repair tools can use to produce high-quality patches for buggy string-processing programs, avoiding repair over-fitting.
(4) We conduct a set of experiments on 329 buggy programs and show that our approach is effective in learning grammar for failure conditions of different kinds of programs.

Reproducibility. To facilitate the reproduciblilty, we make our implementation of Grammar2Fix, our collection data, and scripts available at: https://github.com/charakageethal/grammar2fix.

2 MOTIVATING EXAMPLE

We demonstrate the necessity of having a grammar for failing inputs through an example Python program shown in Listing 1. The expected functionality of this program is to count the vowels (A, a, E,e, I, i, O, o, U, u) of a given string. However, there is a bug in Line 3 due to which the program does not count the 'a's in a string. Thus, the program returns incorrect outputs for all strings containing 'a's, which is a functional bug.

```
1 def find_vowel_count(input_str):
2     n_vowels=0
3     vowels=['A','E','e','I','i','O','o','U','u
          '] #Bug - no `a'
4     for c in input_str:
5         if( c in vowels ):
6             n_vowels+=1
7     return n_vowels
```

Listing 1: Buggy Python Program

Assume that a user finds that this program fails under the input "coverage". With only this single input and without program analysis, it is challenging to identify why the program fails. Similarly, a single failing input is insufficient to identify and fix this bug automatically. To locate the failure and validate the generated fixes, automated repair and debugging techniques need more passing and failing inputs.

As this is a semantic bug, only the human (the user or the developer) can answer whether a test case is passing or failing. However, finding more passing and failing inputs with human intervention is inefficient. This difficulty can be overcome by developing an automated test oracle [4] for this buggy program. Such a test oracle can be developed based on the pattern of failing inputs. The pattern of a set of strings can be formally given as a grammar. Therefore, our objective is to develop an automated approach to induce a grammar for the failing inputs based on the given failing input of the bug. A grammar describing the pattern of the failing inputs can be used as an automated test oracle to produce more passing and failing test inputs automatically. Also, such a grammar can be used to analyze the nature of the bug.

To induce a grammar for the failing inputs, we need more than one failing input and some passing inputs. Taking the given failing inputs as the base, it is possible to generate more test inputs. However, this should be done systematically to support grammar inference rather than randomly, as the human labels the generated test cases. We can observe that the program inputs of Listing 1 are not structured, i.e., the program inputs do not adhere to a particular grammar. Thus, input grammar, i.e., the structure of valid inputs, is not an applicable concept to induce a grammar for the failing inputs of the given buggy program.

Considering these challenges, we present a novel technique Grammar2Fix to infer a grammar for the failing inputs of a buggy program, using only a single failing input.

3 METHODOLOGY

Figure 1 shows the general procedure of Grammar2Fix, our automatic technique that learns to identify failure-inducing string inputs. Grammar2Fix takes as input one failing input f and a bug oracle ($O_{\mathcal{B}}$) and produces as output a grammar oracle ($O_{\mathcal{G}}$). From the failing input f, Grammar2Fix systematically generates alternative inputs which are then labelled by $O_{\mathcal{B}}$ as either passing or failing. From the generated and labeled test inputs, Grammar2Fix infers a grammar oracle ($O_{\mathcal{G}}$) whose objective is to accurately predict the label which the $O_{\mathcal{B}}$ would assign to a test input.

The *bug oracle* ($O_{\mathcal{B}}$) can be the user or developer who discovered the bug or some other mechanism. $O_{\mathcal{B}}$ compares the corresponding program output o_p of the given input with the expected, correct output o_c. If both are equal ($o_p = o_c$), the test input is labelled as *passing*, otherwise it is deemed as *failing*. As this work focuses on functional bugs, program crashes and hangs [17] cannot be used as the bug oracle, so we assume the bug oracle is a human.

One failing string input is not enough to infer the pattern of many other failing inputs of a buggy program. Firstly, the reason for why a string failed can be interpreted in multiple ways. Secondly, a failing input can contain parts that are not relevant to the failure. Removing such parts does not turn a failing input into a passing one. To address these issues, Grammar2Fix generates more test inputs based on the given failing input (f). Grammar2Fix then removes the parts of f that are not related to the failure so as to identify the actual cause for the failure.

Grammar2Fix represents the bug oracle $O_{\mathcal{B}}$ and thus the set of passing and failing inputs as a formal grammar. Grammar2Fix infers this grammar incrementally using several heuristics to explain the pattern of the failing inputs such that the failing inputs adhere to the grammar and the passing inputs do not. The first version of the grammar can be over-fitting, as a single failing input might not contain all the possible characters of the failing inputs. To address this, Grammar2Fix generalizes and extends the grammar adequately to capture the failure condition of the program.

According to this process, Grammar2Fix has the following main steps (Figure 1).

(1) Minimise the given failing input (f) to the smallest failing input, using delta debugging minimisation.
(2) Grammar inference, which infers a grammar from the test inputs generated during delta debugging minimisation.
(3) Grammar Generalization, which generalises the grammar using additional failing and passing input.
(4) Grammar Extension, which extends the grammar created after step three by finding more failing inputs through mutating the given failing input (f).

The four steps of Grammar2Fix are based "Zooming in & Zooming out" research methodology that is applied in various domains, including decision making and organizational studies [24]. In the first two steps, Grammar2Fix "zooms in" into the root cause of the failure and generates a first-level grammar. The first-level grammar could be overfitting with the root cause. Thus, in the last two steps, Grammar2Fix "zooms out" applying generalization and extension steps to avoid overfitting.

3.1 Delta Debugging Minimization

Delta Debugging Minimization (ddmin) [33] is an algorithm that can reduce a failing input to a minimal input (f_{min}) reproducing the bug. A minimal failing input (f_{min}) is a failing input that cannot be divided further to obtain more failing inputs [33]. *ddmin* uses a *divide and conquer* approach to reduce the given failing input. As

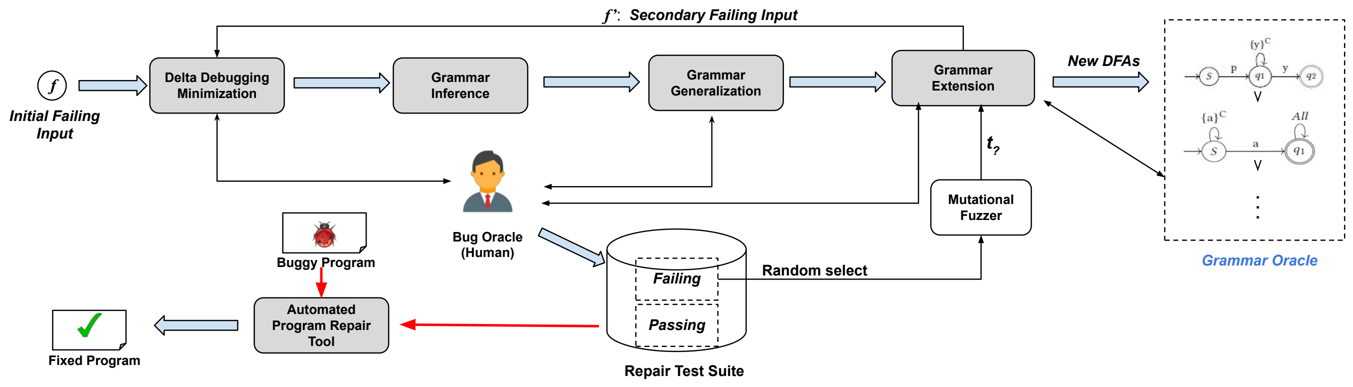


Figure 1: Overall process of Grammar2Fix

the failing input is divided, some new test cases are intermediately generated. Thus, *ddmin* can be used as a test generation method as well.

We apply *ddmin* to the given failing input (f). The new test inputs generated intermediately should be labelled (i.e., check whether they are passing or failing). Grammar2Fix uses the bug oracle ($O_{\mathcal{B}}$) for this purpose. At the end of the step, *ddmin* returns the minimal failing input exposing the bug (f_{min}) traced through f.

Let F be the set of failing test cases, and P be the set of passing test cases generated in *ddmin*. The sets F and P are disjoint, i.e., $F \cap P = \emptyset$. Also, $f, f_{\text{min}} \in F$. Let $O_{\mathcal{B}}$ be the bug oracle.

As an example, consider the buggy program in Listing 1 (Section 2). *ddmin* produces the following outcomes for the given failing input "coverage".

- F = { "coverage", "rage", "ra", "a"}
- P = { "cove", "r"}
- f_{min} = 'a'

The given failing input (f) and the test cases generated in addition to f_{min} can be considered as the neighborhood of f_{min}. Therefore, test inputs generated by *ddmin* illustrate the characteristics of the failure condition more concretely than randomly generated test inputs. A basic intuition about the failure condition can be built based on the minimal failing input (f_{min}) and the intermediate passing (P) and failing ($F \setminus f_{\text{min}}$) inputs. Hence, we use these test inputs in the next steps.

3.2 Grammar Inference (GI)

Given the test inputs generated in *ddmin*, Grammar2Fix induces a grammar given as a Deterministic Finite Automata (DFA). To induce a DFA from F and P, we apply *Regular Positive and Negative Inference* (RPNI) [8] with a modification to its merging technique.

Definition 1. *A Deterministic Finite Automaton (DFA) can be defined by a 5-tuple $(Q, \Sigma, \delta, q_0, \mathcal{F})$ where Q is a finite set of states, Σ is a finite set of symbols called the alphabet, δ is the transition function: $\delta : Q \times \Sigma \longrightarrow Q$, q_0 is the initial state ($q_0 \in Q$), and $\mathcal{F}$ is a set of final/accept states ($\mathcal{F} \subseteq Q$). A DFA can have self-transitions, i.e., transitions from one state to itself, and inter-state transitions, i.e., transitions between two states.*

A DFA can be modeled to accept any given set of strings while rejecting the others even without knowing the complete alphabet (e.g. a prefix tree acceptor [8]). Also, by adding more characters to the transitions, a DFA can be extended to accept more strings. Moreover, even though a single DFA may have a limited expressive power, a collection of DFAs combined with disjunctions can model more complex patterns. Grammar2Fix uses RPNI [8] to infer DFAs, as it is more accurate compared to other DFA inference algorithms such as GOLD [10]. Also, it has a flexible merging technique that generalizes the DFA through merging the states.

Given a set of positive and negative examples, RPNI creates a DFA that accepts all the positive examples and rejects all the negative examples. As the grammar is modeled for failing inputs, we consider F as the positive examples and P as the negative examples. Following RPNI algorithm, first, we develop a *Prefix Tree Acceptor* (PTA) based on F, which results in a DFA accepting only the strings in F. Next, following RPNI merge [8], the possible states of the DFA are merged iteratively such that no string in P is accepted.

Grammar2Fix uses an additional constraint compared to RPNI merging algorithm as follows. Given a pair of states, if there are inter-state transitions that take any character of f_{min}, those states are not merged.

This constraint enforces the characters of f_{min} to appear only in the inter-state transitions of the resulting DFA, which makes those characters mandatory in the grammar for failing inputs. In a deterministic finite automaton, the characters in a self transition can have zero or more occurrences in a string accepted by the DFA [14]. Thus, the self-transitions indicate the characters that are not relevant to the failure. The DFA obtained in this step shows where f_{min} can appear in failing inputs.

As an example, consider the F obtained for the failing input "coverage" in Section 3.1. The PTA for all the failing inputs in F is given in Figure 2. The states q_8, q_{10}, q_{12} and q_{13} are the final states that correspond to "coverage", "ra", "rage" and "a", respectively. The inter-state transitions between q_0 and q_{13}, q_9 and q_{10}, and q_5 and q_6 take 'a'. As 'a' is a character of f_{min}, these states are not merged according to our constraint. The remaining states can be safely merged such that no string in P is accepted. The resulting DFA is given in Figure 3 and it accepts all the strings in F. This DFA

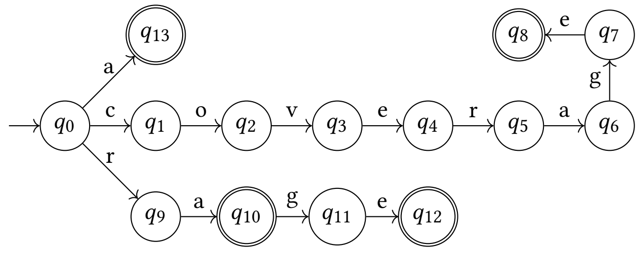


Figure 2: Prefix Tree Acceptor (PTA) for F derived from the failing input "coverage" of the buggy program in Listing 1

indicates that 'a' is mandatory in the failing inputs, and the other characters ('c','o','v','e','r','g') can appear zero or more times.

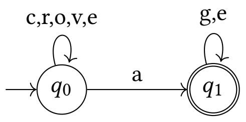


Figure 3: Basic level grammar with the modification to RPNI's merging technique. f_{min} **='a'**

The resulting grammar is named *Basic Level Grammar.*

3.3 Grammar Generalization

We can obtain only a limited amount of information about the failure condition from the given failing input (f) and the test cases generated by *ddmin*. Therefore, the basic level grammar can overfit with the failing inputs generated through *ddmin* (F). To avoid this, we apply generalization steps to the basic level grammar.

3.3.1 Basic Generalisation (BG). The Basic Generalisation (BG) step focuses on generalising the characters of the self-transitions in the DFA.

The self-transition of a state has few characters, as we used only the failing input set F (Section 3.1) to develop the DFA. Nevertheless, it can take any character except the characters used in the inter-state transitions, as the characters mandatory to form failing inputs are already in the inter-state transitions. Therefore, we generalize these transitions as follows. Given a state q_i, with one or more interstate-transitions

i) BG identifies the set of characters (C_i) in the outgoing inter-state transitions of s_i.
ii) BG updates the characters of the self transition of s_i to include any character except the characters in C_i (${C_i}^{\text{C}}$ - complement of C_i).

After the conversion, we name such self-transitions as *Complementary Self-Transitions.*

Example: Consider the DFA given in Figure 3. The state q_0 has an outgoing inter-state transition taking 'a' to reach q_1. Also, it has a self transition taking 'c', 'r', 'o', and ,'v'. In this generalization, the self transition is changed as it can happen under any character *except* 'a' ($\{\text{a}\}^{\text{C}}$), which is the *complementary self-transition* for q_0. q_1 also has a self-transition that takes 'g' and 'e'. However, this state has no outgoing inter-state transitions. Thus, this generalization changes the self-transition in q_1 as it can happen under any character (Because, $\emptyset^{\text{C}} = U; U$ set of all elements). The DFA after this generalization is in Figure 4.

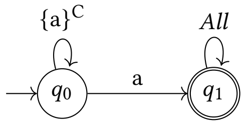


Figure 4: After adding complementary transition to S **and** q_1**. *All* is the set of all characters**

3.3.2 Handling Special Cases (HSC). Positioning the characters of f_{min} at the beginning or end of failing test inputs could lead to the initial state without complementary self-transitions and final states without complementary self-transitions and outgoing transitions. This could be either an attribute of failing inputs or overfitting to the failing inputs. To check this case, we extend f, as *ddmin* cannot explore beyond the given failing input. *ddmin* avoids possible overfittings in the other states, as it generates new test cases through fragmenting f.

The DFA at this point may have the following properties, which are considered special cases.

i) The initial state (q_0) has no complementary self-transition.
ii) Final states have no complementary self-transitions and outgoing inter-state transitions.

If the DFA has i), it could mean that the failing inputs must start with the characters in the outgoing-transitions of q_0. Otherwise, the current grammar overfits to the given failing input (f) and it should be generalized. To check if overfitting occurs, we create a random string without any character of f_{min} and add it to the front of f. If the resulting test input is failing, it signals the overfitting, and we add a complementary self-transition to the q_0.

If the DFA has ii), it could mean that the failing input must end at these state. Otherwise, the current grammar overfits to the given failing input and it should be generalized. To check if overfitting occurs, for each such final state, first, we select a failing input that ends at the state from F (Section 3.1). We add a random string to the end of the selected failing input. If the resulting input is failing, it signals the overfitting, and we add a complementary self-transition to the final state.

As an example, 'a' is also a failing input under the bug in the motivating example (Section 2). However, if this is the initial failing input (f), the DFA at this point has no complementary self-transition in q_0 and no complementary self-transition and outgoing inter-state transition in the final state. This DFA implies that 'a' is the only failing input, which is incorrect. Nevertheless, following the process described above, we find that adding characters to the front or the end gives failing inputs. Thus, we conclude that q_0 and the final state require complementary self-transitions.

3.3.3 Finding the character class of the minimal failing input f_{min} (CCF). We discovered that there are more than one minimal failing inputs with the same length under some bugs. In such situations, more minimal failing inputs can be explored by substituting the characters of one minimal failing input. Thus, CCF step focuses on finding the character substitutions, i.e., the character class, producing minimal failing inputs.

In this step, we assume that the pattern of a group of minimal failing inputs of the same length can be abstracted in terms of the unique characters of one minimal failing input of the group. When an input grammar is unavailable, this assumption helps to reduce the search space. Based on this assumption, we substitute the unique characters with a set of random characters distinct to each other. In other words, If the set of unique characters of f_{min} is $U = \{C_1, C_2, \cdots, C_n\}$ ($C_1 \neq C_2 \neq C_3 \cdots \neq C_n$), we substitute $C_1 \longleftarrow A_1$, $C_2 \longleftarrow A_2$, $C_3 \longleftarrow A_4, \cdots, C_n \longleftarrow A_n$, where $\{A_1, A_2, A_3, \cdots A_n\}$ is the set of random characters and $A_1 \neq A_2 \neq A_3 \cdots \neq A_n$ (Line 5 Algorithm 1). The resulting test input is presented to the bug oracle (O_B) for labelling. Through our experiments, we have identified that this technique can explore minimal failing inputs with the same length effectively.

Example: If f_{min}="abab" , there are two unique characters 'a' and 'b', and we assume that the pattern of the minimal failing inputs are of the form $C_1C_2C_1C_2$ (where $C_1 \neq C_2$). Then we substitute 'a' $\longleftarrow$ 'c' and 'b' $\longleftarrow$ 'd'. This substitution produces "cdcd", and it is presented to O_B. (Here, substitutions such as 'a' $\longleftarrow$ 'c' and 'b' $\longleftarrow$ 'c' are considered invalid, as 'a' and 'c' are substituted with the same character which breaks the pattern).

We repeat this process for n iterations, and there are two main cases depending on the *passing* inputs generated in the process (Algorithm 1).

Case 1 All the generated inputs are *passing* . We conclude that f_{min} is the only minimal failing input, and the character class ($C_{f_{\text{min}}}$) is only the set of unique characters (U) (Line 13 Algorithm 1).

Case 2 Not all the generated inputs are *passing*. We conclude that except the substitutions leading to produce passing inputs (P_A), the unique characters of f_{min} can be replaced by any other set of characters distinct to each other. Thus, $C_{f_{\text{min}}} = All \setminus P_A \cup \{U\}$ (Line 16 Algorithm 1)

In Algorithm 1, P_A and $C_{f_{\text{min}}}$ are sets of sets.

Under **Case 1**, no change is done to DFA. The motivating example (Section 2) falls under **Case 1**, as the 'a' is the only minimal failing input.

Under **Case 2**, if $s \in C_{f_{\text{min}}}$, for each character in U in the inter-state transitions, there is a corresponding character in s. By replacing each character of the inter-state transitions with its corresponding character in s and changing the complementary self-transitions accordingly, a new DFA is created. This is done for all the sets in $C_{f_{\text{min}}}$, which results in a collection of DFAs that are connected with disjuctions/"OR ($\bigvee$)" operator.

Example: Consider f_{min}="abab" and the DFA presented in Figure 5. Assume **Case 2** and Algorithm 1 returns the set of character substitutions $C_{f_{\text{min}}} = \{\{C_{11}, C_{21}\}, \{C_{12}, C_{22}\} \cdots \{C_{1n}, C_{2n}\}\}$, the DFA in Figure 5 is converted to a collection of DFAs connected with "OR" operators as in Figure 6, where C_{ij} is the assignment to the i th unique character of f_{min} from j th substitution set.

3.4 Grammar Extension (GE)

One failing input can be insufficient to capture all the properties of the failure condition for certain bugs, even with our generalisation

Algorithm 1 Character Class Finding

```
Input: $f_{\text{min}}$ : Minimal Failing Input
Input: $O_{\mathcal{B}}$ : Bug Oracle
Input: $n$ : Substitution Iterations
Output: $C_{f_{\text{min}}}$: Character Class of $f_{\text{min}}$
 1: $U \longleftarrow unique\_characters(f_{\text{min}})$
 2: $P_A \longleftarrow \emptyset$
 3: $n\_success \longleftarrow 0$
 4: for $i \longleftarrow 1$ to $n$ do
 5:     $\mathcal{N} \longleftarrow get\_new\_random\_assignment()$
 6:     $t_{new} \longleftarrow replace\_unique\_characters(f_{\text{min}}, \mathcal{N})$
 7:     if $O_{\mathcal{B}}(t_{new}) = Pass$ then
 8:         $P_A \longleftarrow P_A \cup \{\mathcal{N}\}$
 9:         $n\_pass \longleftarrow n\_pass + 1$
10:     end if
11: end for
12: if $\frac{n\_pass}{n} = 1$ then
13:     $C_{f_{\text{min}}} \longleftarrow U$ //Case 1
14: else
15:     //Let $All$ be the set of all possible character substitutions
16:     $C_{f_{\text{min}}} \longleftarrow All \setminus P_A \cup \{U\}$ //Case 2
17: end if
18: return $C_{f_{\text{min}}}$
```

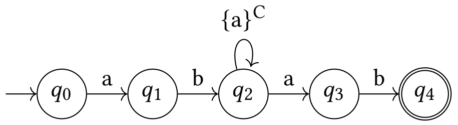


Figure 5: DFA before finding the character class of f_{min}

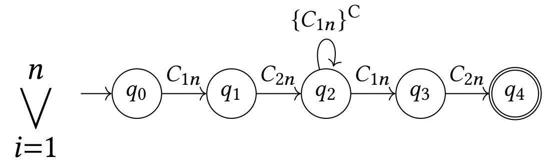


Figure 6: Abstract Representation of the collection of DFAs under $C_{f_{\text{min}}} = \{\{C_{11}, C_{21}\}, \{C_{12}, C_{22}\} \cdots \{C_{1n}, C_{2n}\}\}$

steps described in Section 3.2). As an example, there are bugs with more than one minimal failing inputs with different lengths, which cannot all be explored by the CCF step. In addition, due to the limited number of substitution iterations, CCF might not find all minimal failing inputs. To address these issues, we introduce the Grammar Extension (GE) step as described in Algorithm 2.

GE extends the grammar oracle ($O_{\mathcal{G}}$) by applying steps 1 - 3 described in Sections 3.1-3.3 to failing test inputs generated via mutational fuzzing [7]. We run N fuzzing iterations, and use the grammar oracle ($O_{\mathcal{G}}$) to predict if the inputs are *failing* or *passing*. If $O_{\mathcal{G}}$ predicts the test input (t_s) as *passing*, we present it to the bug oracle ($O_{\mathcal{B}}$) for labelling. If $O_{\mathcal{B}}$ labels t_s is *failing*, it implies that the current grammar oracle ($O_{\mathcal{G}}$) cannot correctly identify this failing input. Therefore, we apply step 1-3 of Grammar2Fix to t_s

and derive a new grammar ($\mathcal{G}_{\text{new}}$). The grammar oracle ($O_{\mathcal{G}}$) is updated with $\mathcal{G}_{\text{new}}$ with an “OR" operator, and the seed corpus (C) is updated with the new failing input t_s.

At the end of the four steps (Sections 3.1 - 3.4), we obtain the grammar describing the failure condition as a DFA or a collection of DFAs connected with “OR" operators. The test inputs labelled by the bug oracle ($O_{\mathcal{B}}$) in the grammar inference process are part of the repair test suite, as shown in Figure 1 which is used as an input to an automated program repair tool to generate a patch for the buggy program.

Algorithm 2 Grammar Extension

```
Input: $f$ : Initial failing input
Input: $O_{\mathcal{G}}$ : Grammar oracle
Input: $O_{\mathcal{B}}$ : Bug oracle
Input: $N$ : Fuzzing iterations
 1: Let $C$ be the seed corpus of failing inputs.
 2: $C \leftarrow \{f\}$
 3: for $i \leftarrow 1$ to $N$ do
 4:     $f' \leftarrow pick\_random(C)$
 5:     $t_s \leftarrow mutate\_fuzz(f')$
 6:     if $O_{\mathcal{G}}(t_s) = Pass$ then
 7:         if $O_{\mathcal{B}}(t_s) = Fail$ then
 8:             $\mathcal{G}_{\text{new}} \leftarrow Derive\_Grammar(t_s)$
 9:             $O_{\mathcal{G}} \leftarrow O_{\mathcal{G}} \vee \mathcal{G}_{\text{new}}$
10:             $C \leftarrow C \cup \{t_s\}$
11:         end if
12:     end if
13: end for
```

4 EXPERIMENTAL SETUP

4.1 Research Questions

RQ.1. (Grammar oracle quality) How accurately does the grammar-based oracle classify the test cases in a given test suite?

RQ.2. (Ablation study) What are the contributions of the heuristics to the accuracy of the grammar-based oracles?

RQ.3. (Labelling effort) Is the number of requests that are sent to the human oracle for labeling reasonable?

RQ.4. (Patch quality) How does the quality of patches produced through the grammar inference algorithm's auto-generated test suites compare to the quality of patches produced through the manually constructed test suites?

4.2 Experimental Subjects

We selected three *benchmarks* according to the criteria given below to evaluate GRAMMAR2FIX and answer the research questions.

(1) There should be programs that take string inputs.
(2) There should be a diverse set of real defects that lead to *functional bugs*, i.e., programs produce incorrect outputs for specific inputs. There should be one functional bug for each subject.
(3) For each subject, there should be a *golden version*, i.e., a program that produces the expected, correct output for an input. For a given input, we simulate the bug oracle's($O_{\mathcal{B}}$) task by comparing the subject's (buggy program's) output with its golden version's output. If both are different, the test case is labelled as failing.
(4) For each subject, there should be a manually constructed and labelled *training test suite*.
(5) For each subject, there should be at least one *failing* test case in the training test suite, i.e., a test input for which the buggy and the golden version produce different outputs.

We found that the benchmarks **IntroClass** [15], **Quixbugs** [18], and **Codeflaws** [29] satisfy the above criteria. **IntroClass** [15] consists of C programs that were submitted under six (6) assignments by a group of students. We selected the programs under the assignments *Syllables* and *Checksum*, as those take string inputs. Under each of the two assignments, there is a golden version and a labelled test suite. Based on the second and fifth criteria, we excluded the programs showing flaky behaviour and having no failing inputs. After that, there were 52 subjects under *checksum* and 121 subjects under *syllables* for the experiments.

We selected 4 Python programs from **Quixbugs** [18] and 152 C programs from **Codeflaws** [29] based on the first criterion. In this selection, we excluded the programs taking mixed inputs (e.g. strings and numbers together) in the benchmarks. For each of these selected subjects, there is a separate labelled test suite and a golden version.

4.3 Setup and Evaluation

For each subject, we apply our grammar inference algorithm selecting a random *failing* input from the training test suite (manually labelled test suite). After the grammar is generated, we apply it on the training test suite. The test inputs adhering to the grammar are predicted as *failing*. The others are predicted as *passing*. In this manner, we used the inferred grammar as a test oracle (i.e., *grammar oracle* ($O_{\mathcal{G}}$)). We use the test cases generated in the grammar inference process (i.e., *auto-generated test suite*) to generate repairs for the buggy programs automatically.

In the experiments, we use 20 substitution iterations ($n = 20$) in finding the character class of a minimal failing input (CCF - Section 3.3.3). Also, 5 mutational fuzzing iterations are ($N = 5$) in grammar extension (GE - Section 3.4).

Related to **RQ.1.** and **RQ.2.**, we calculate *Accuracy* (Equation 1) and *Conditional Accuracy* for failing (Equation 2) and passing inputs (Equation 3). (*Conditional Accuracy (Failing)* : recall for failing inputs, *Conditional Accuracy (Passing)* : recall for passing inputs). Here, the actual labels of the test cases are compared with the labels predicted by $O_{\mathcal{G}}$. If both the labels for a test case are similar, it is considered as the label has been correctly predicted; otherwise not. In addition to these, we count the number of queries sent to the bug oracle ($O_{\mathcal{B}}$), i.e., labelling effort, to answer **RQ.3.**.

$$Accuracy = \frac{Number\ of\ correctly\ predicted\ test\ inputs}{Number\ of\ test\ inputs\ in\ the\ test\ suite} \quad (1)$$

$$\begin{array}{c} Conditional \\ Accuracy\ (Failing) \end{array} = \frac{\begin{array}{c} Number\ of\ correctly\ predicted \\ failing\ inputs \end{array}}{\begin{array}{c} Number\ of\ failing\ inputs \\ in\ the\ test\ suite \end{array}} \quad (2)$$

$$\text{Conditional Accuracy (Passing)} = \frac{\text{Number of correctly predicted passing inputs}}{\text{Number of passing inputs in the test suite}} \quad (3)$$

$$\text{Precision-(Failing)} = \frac{\text{Number of correctly predicted failing inputs}}{\text{Total number of inputs predicted as failing}} \quad (4)$$

$$\text{Precision-(Passing)} = \frac{\text{Number of correctly predicted passing inputs}}{\text{Total number of inputs predicted as passing}} \quad (5)$$

$$\text{Recall-(Passing)} = \text{Conditional Accuracy (Passing)} \quad (6)$$

$$\text{Recall-(Failing)} = \text{Conditional Accuracy (Failing)} \quad (7)$$

The benchmarks do not provide a balanced test suite, a test suite with equal passing and failing inputs, for most selected subjects. Thus, there is an impact from the *Class Imbalance Problem* [20] in the evaluation, and *Accuracy* is not a good metric to evaluate the quality of a grammar-based oracle. Therefore, we measured *Conditional Accuracy* for both failing (Equation 2) and passing (Equation 3) inputs in addition to *Accuracy*. These three metrics together show the quality of a grammar oracle (O_G) under the class imbalance problem; therefore, we use those to answer **RQ.1.** and **RQ.2.**.

Regarding **RQ.4.**, we selected **GenProg** [16] as the automated program repair tool in the experiments due to its capability to repair large programs cost-effectively. For each subject, we run **GenProg** on the manually created test suits, given by the benchmark, and the test suite generated in the grammar inference process. Then, we measure how many subjects can be repaired and how many *validation test cases* that the repaired program can pass under the manual and auto-generated test suites separately. We use these two measures to answer **RQ.4.**.

To minimize the impact of randomness and to gain statistical power for the experimental results, we repeat each experiment 30 times per each subject. We set the maximum time to generate the grammar oracle (O_G) to 10 minutes for each subject. Also, for each test suite (manual and auto-generated), we allocate 10 minutes to generate a repair through GenProg.

5 EXPERIMENTAL RESULTS

5.1 RQ.1. Labelling Accuracy

We are interested in the quality of the automated oracle that is learned by Grammar2Fix for the bugs in our three benchmark sets. Figure 7 shows the distribution of the prediction accuracy over the different subjects in three benchmark sets as violin plots. For each subject, we computed the average values over 32 runs.

For the majority of the projects, the oracle that is learned by Grammar2Fix classifies correctly as passing or failing 92% of test inputs (median overall accuracy). For the majority of subjects, the oracle identifies passing test inputs with 94% precision and 96% recall (Conditional Accuracy - Passing) and failing inputs with 100% precision and 97% recall (Conditional Accuracy - Failing) - Figure 7a. The median overall accuracy in QuixBugs is 88.36%; in IntroClass, 99.20% and in Codeflaws, 77.70%. Also, the median Conditional Accuracy-Failing in QuixBugs is 99.31%; in IntroClass, 98.73% and in Codeflaws 86.88% - Figure 7b.

These results indicate that the automated oracle that is learned by Grammar2Fix classifies test inputs with a significantly high accuracy in the majority of subjects. It implies that our grammar inference approach can accurately induce a grammar to explain the pattern of the failing inputs of a given buggy program. The high conditional accuracy for passing and failing test inputs indicates that the induced grammar is not only effective in recognising failing inputs, but also can accurately reject passing inputs.

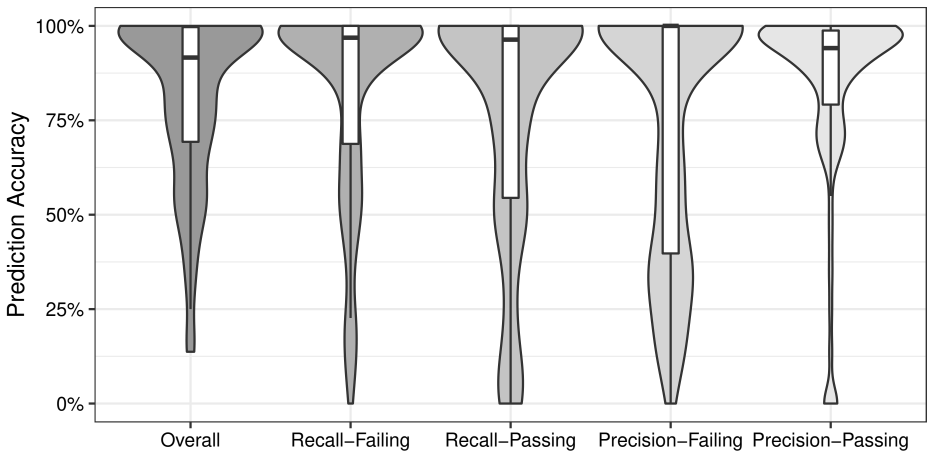
For the few subjects where the oracles were not very accurate, we identified two main reasons. Firstly, Grammar2Fix does not model dependencies among complementary-self transitions, as finding such dependencies needs more labelled inputs. However, to describe the failing input patterns of some bugs, such dependencies could be necessary. Secondly, the mutational fuzzing iterations in GE could be insufficient to identify all the constituents of the pattern of the failing inputs. Especially when the program under test accepts structured string inputs, structure-aware fuzzing might be more effective [26, 32].

Result. *Grammar2Fix induces high-quality grammars that explain failure conditions from string inputs with high overall accuracy. The precision and recall for identifying failing test cases are both above 97% for the majority of bugs among our subjects. The high conditional accuracy for failing inputs indicates that the Grammar2Fix is effective in predicting failing test inputs.*

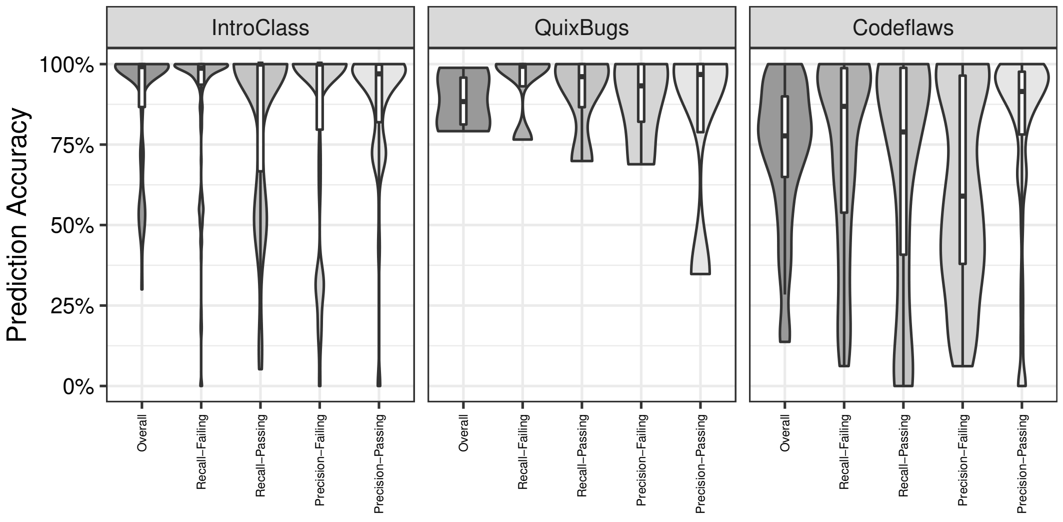
5.2 RQ.2. Ablation Study

We are interested in the contributions to accuracy from the different heuristics namely Grammar Inference (GI), Basic Generalization (BG), Handling Special Cases (HSC), Character Class Finding (CCF), and Grammar Extension (GE) (Section 3.3 and Section 3.4). Figure 8 shows the prediction accuracy over the different subjects in the three benchmark sets as violin plots. Again, for each subjects, we computed the average values over 32 runs for the different heuristic steps in the oracle inference.

According the violins of Figure 8, the overall accuracy and conditional accuracy increase with the heuristics. The median of overall accuracy increases from 60% for the first step (GI) to 92% for the last (GE; via BG 61%, HSC 73%, CCF 89%). The median of conditional accuracy failing increases from 17% the first step (GI) to 97% for the last (GE; via BG 20%, HSC 30%, CCF 88%), demonstrating that our heuristics are effective. The character class finding (CCF) significantly improves the ability of identifying failing inputs in terms of conditional accuracy because the DFA is expanded to a collection of DFAs by exploring more minimal failing input. This process significantly avoids overfitting of the grammar to training examples. Also, we observe a slight decrease from CCF to HSC in identifying passing inputs (conditional accuracy-passing), probably because some bugs have passing inputs with the same length of $f_{\min}$. The limited substitution iterations in CCF might be insufficient to identify those. If this happens, Case 2 in CCF (Algorithm 1)



(a) Overall distribution



(b) Benchmark-wise distributions

Figure 7: Violin plots of the distribution of overall accuracy, precision and recall. Figure 7a shows the overall distribution over 3 benchmarks, and Figure 7b shows the distribution under each benchmark

concludes that the unique characters of $f_{\min}$ can be assigned with any set of unique characters.

Result. *Our results demonstrate that each of the heuristic steps in GRAMMAR2FIX improves the quality of the learned oracle. The proportion of correctly identified failing inputs (conditional accuracy) increases from less than 20% for the first step to 97% for the last.*

5.3 RQ.3. Labelling Effort

Figure 9 shows the labelling effort in terms of the number of queries to the user. For each bug, the user would need to answer less than 42 queries for the majority of projects (i.e., on the median).[1] We find that this is a reasonable number for several reasons. Firstly, the user would answer simple Yes/No questions: "Does the program process this string input correctly?" Observing the actual program output, the human would respond "Yes" if the input is processed correctly; otherwise, they would say "No". Even if the user would take 4 seconds to respond to each question, the automated oracle that is used for auto-patching this bug would be automatically derived in under three (3) minutes. GRAMMAR2FIX being a blackbox appraoch, this time is independent of the size of the program. We also note that such queries do *not* need to be answered by *the same user*. If different users report the same bug, GRAMMAR2FIX will actively amplify the information about the bug during the automated oracle inference.

[1]Recall, for reasons of practicality in our experiments we used the patched versions instead of the user as bug oracles to send those queries.

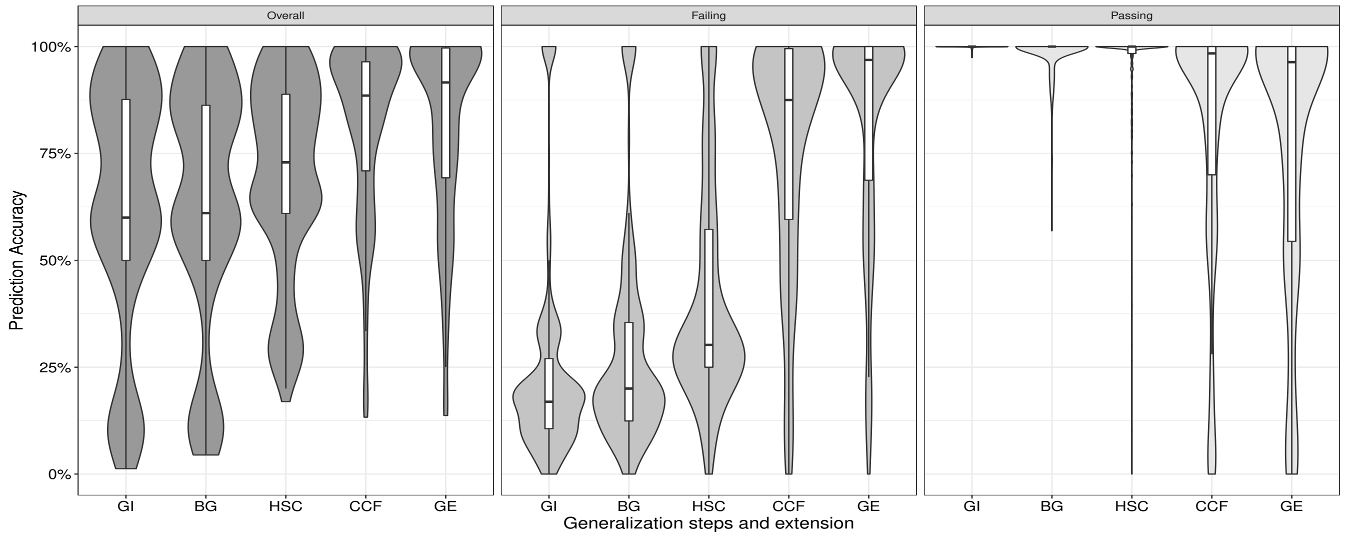


Figure 8: Violin plots of the prediction accuracy after each step of Grammar2Fix as a distribution across all subjects. Concretely, the steps are grammar inference (GI), basic level generalization (BG), handling special cases (HSC), character class finding (CCF), and grammar extension (GE).

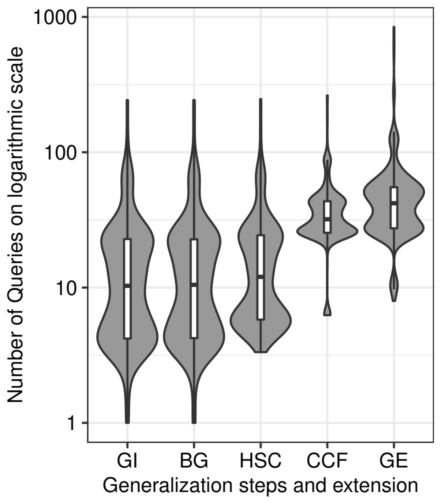


Figure 9: Violin plots (log-scale) of the cumulative number of queries to the bug oracle after each step of Grammar2Fix as a distribution across all subjects.

For the few subjects that required a greater number of queries, we found that the initially selected failing input is often significantly longer than for the other subjects. Our input minimization algorithm follows a deterministic number of steps, and after each step Grammar2Fix interacts with the bug oracle (i.e., user) to label the newly generated input. If the first failing string input is longer, the number of minimisation steps will be larger, leading to an increase in the number of queries. Future advances in input minimization algorithms will directly lead to the improvement of Grammar2Fix in terms of the minimal required number of queries.

We also investigated the labelling effort of different steps. The accumulated median number of queries after completing GI, BG, HSC, CCF, and GE steps are 10.3, 10.5, 12, 32, and 42, respectively. CCF and GE steps together generate 75% of the queries which is expected. These are the main steps at which Grammar2Fix generates substantial new failing inputs, e.g., by mutational fuzzing, to prevent the generated grammar from being overfit to the initially selected/given failing inputs. As demonstrated in Figure 7 and answered in RQ2, this additional effort leads to significant gains in labelling accuracy

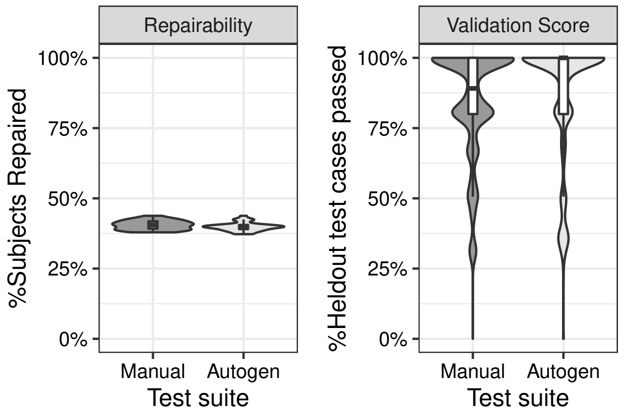
Result. *For the majority of bugs in the three benchmark sets, a human would need to answer at most 42 "Yes/No" questions about the correct processing of string inputs (i.e., less than 3 minutes assuming 4 seconds per question). Given that bugs in stable real-world systems are quite rare and that the query load can otherwise be distributed, we believe that this human effort is reasonable.*

5.4 RQ.4. Patch Quality

We are interested in the quality of the auto-generated patches after Grammar2Fix negotiated with the user $O_{\mathcal{B}}$ the failure condition $O_{\mathcal{G}}$ for buggy string-processing programs. To evaluate repair quality, we used the popular repair benchmark **Codeflaws** [29], as it provides a manually created repair test suite, a separate validation test suite (heldout test suite) for each subject, and scripts to work with *GenProg* [16]. We did not use QuixBugs because Python is not supported by *GenProg* nor Introclass because it has no separate validation test suites. We ran *GenProg* 30 times on each subject, providing the manual and auto-generated test suites.

Figure 10a shows (a) the proportion of bugs that could be repaired and (b) the proportion of validation tests that passed after the successful repair across all subjects for the manually provided test suites and those generated during the oracle inference by Grammar2Fix. As we can see, in both cases approximately 40% of the subjects can be repaired. The probability to generate a valid repair using the manually created test suites is slightly higher than for our auto-generated test suites (on the median 40.52 for the manual and 39.87 for the auto-generated test suites). However, the proportion of passing validation tests for the median subject is significantly higher for our auto-generated test suites than that for manual test suites (i.e., the median for manual test suites is 89.11% and for auto-generated 100%). This means that the auto-generated test suites by Grammar2Fix can produce high-quality patches for the buggy subjects with *GenProg*.

Result. *Grammar2Fix produces high-quality patches. The median patch produced by Grammar2Fix passes all validation tests for the corresponding subject while the median patch produced using the manually test suites that are provided with the repair benchmark passes only about 90% of validation tests.*



(a) Repairability **(b) Patch Quality**

Figure 10: Repairability and Patch Quality under GenProg

6 THREATS TO VALIDITY

Similar to other empirical studies, there are various threats to the validity of our results and conclusions. The first concern is *external validity*, i.e., to what extent our findings can be generalized. We have tested our approach in the three benchmarks, IntroClass [15], Quixbugs [18], and Codeflaws [29]. These benchmarks are widely-used APR benchmarks and contain a large number of real-world defects.

The second concern is *internal validity*, i.e., to what extent our study minimizes the systematic error. For each subject, we ran the experiment 30 times and calculated the average of accuracy (Equation 1) and conditional accuracy - passing & failing (Equation 3 and Equation 2). This technique helps to mitigate the spurious observations due to the random selection approach in Section 4.3. Also, it helps to gain statistical power for the results.

Our third concern is *construct validity*, i.e., to what extent a test measures what it claims to be measuring. To reduce this threat, we discuss at least two measures for each of the two independent variables: accuracy and labelling effort.

7 RELATED WORK

This paper was motivated by the work LEARN2FIX [6] on deriving automated test oracles for *semantic bugs*. LEARN2FIX [6] uses a single failing input of the bug and returns an SMT(LRA) (Satisfiability Modulo Linear Real Arithmetic Theory) [5] formula satisfied only by the failing inputs. The underlying learning technique, INCAL [13], learns an SMT(LRA) formula from the given set of positive and negative examples. The SMT(LRA) formula acts as the automated oracle. However, as An and Yoo [2] point out, this approach cannot be applied to string-processing programs. Recently, there have been tremendous advances in the *solving* of string constraints represented in a satisfiability modulo theory for strings [1, 12, 30, 31, 34]. However, we are not aware of any work on *learning* string constraints. Instead, our work explores an altogether different approach that is based on grammar learning. Our key observation is that grammars are a natural definition of the structure of an input and can thus represent structural aspects of bugs in string-processing programs.

DDSET [11] produces an outcome similar to Grammar2Fix. It starts with a user-provided grammar that specifies the general program input structure, a failing input, and a predicate (in our case the user). DDSET is targeted at programs taking highly structured inputs such as compilers while Grammar2Fix also works for programs processing unstructured string inputs. DDSET requires an input grammar which is then pruned while Grammar2Fix does not require any grammar to start with but learns a grammar that represents the pattern of bug-revealing string inputs.

ExtractFix [9] uses symbolic execution to derive a constraint under which a program produces a crash. This constraint is then used to repair the program in a constrained-based manner. We tackle the same underlying problem, i.e., that of *overfitting* in test-driven program repair. However, focussing on crashing bugs, ExtractFix assumes the availability of an automated oracle. In contrast, we are focussing on semantic bugs, i.e., bugs that can only be identified by a user. We propose a technique to learn the automated oracle from the user and develop heuristics to minimize the number of queries.

8 CONCLUSION

We introduced Grammar2Fix, a human-in-the-loop approach to inferring failure conditions of string-processing programs. Given a single bug-triggering string input, its failure condition, in the form of a regular grammar, can be identified and generalized through several carefully designed heuristics and optimizations with a manageable human effort. Our large-scale experiments on 329 subject programs from three popular program repair benchmarks show that Grammar2Fix can correctly predict passing and failing test cases with more than 96% recall while the human effort, as measured by number of queries they need to answer, is reasonably low. We could terminate the oracle learning, when the proportion of *in*correctly predicted labels drop below a certain threshold. This is a significant step toward realizing our vision of human-in-the-loop debugging and repair for semantic bugs.

ACKNOWLEDGMENTS

This research was supported by use of the Nectar Research Cloud.

REFERENCES

[1] Roberto Amadini. 2021. A Survey on String Constraint Solving. *ACM Comput. Surv.* 55, 1, Article 16 (nov 2021), 38 pages. https://doi.org/10.1145/3484198

[2] Gabin An and Shin Yoo. 2021. Human-in-the-Loop Fault Localisation Using Efficient Test Prioritisation of Generated Tests. *arXiv preprint arXiv:2104.06641* (2021). https://doi.org/10.48550/ARXIV.2104.06641

[3] Dana Angluin. 1987. Learning regular sets from queries and counterexamples. *Information and computation* 75, 2 (1987), 87–106. https://doi.org/10.1016/0890-5401(87)90052-6

[4] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *IEEE transactions on software engineering* 41, 5 (2014), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[5] Clark Barrett and Cesare Tinelli. 2018. Satisfiability modulo theories. In *Handbook of Model Checking*. Springer, 305–343. https://doi.org/10.1007/978-3-319-10575-8_11

[6] Marcel Böhme, Charaka Geethal, and Van-Thuan Pham. 2020. Human-In-The-Loop Automatic Program Repair. In *Proceedings of the 2020 IEEE International Conference on Software Testing, Verification and Validation* (Porto, Portugal) *(ICST 2020)*. 1–12. https://doi.org/10.1109/ICST46399.2020.00036

[7] Chen Chen, Baojiang Cui, Jinxin Ma, Runpu Wu, Jianchao Guo, and Wenqian Liu. 2018. A systematic review of fuzzing techniques. *Computers & Security* 75 (2018), 118–137. https://doi.org/10.1016/j.cose.2018.02.002

[8] Colin De la Higuera. 2010. *Grammatical inference: learning automata and grammars*. Cambridge University Press. https://doi.org/10.1017/CBO9781139194655

[9] Xiang Gao, Bo Wang, Gregory J. Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2021. Beyond Tests: Program Vulnerability Repair via Crash Constraint Extraction. *ACM Trans. Softw. Eng. Methodol.* 30, 2, Article 14 (feb 2021), 27 pages. https://doi.org/10.1145/3418461

[10] E Mark Gold. 1978. Complexity of automaton identification from given data. *Information and control* 37, 3 (1978), 302–320. https://doi.org/10.1016/S0019-9958(78)90562-4

[11] Rahul Gopinath, Alexander Kampmann, Nikolas Havrikov, Ezekiel O Soremekun, and Andreas Zeller. 2020. Abstracting failure-inducing inputs. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 237–248. https://doi.org/10.1145/3395363.3397349

[12] Adam Kiezun, Vijay Ganesh, Philip J Guo, Pieter Hooimeijer, and Michael D Ernst. 2009. HAMPI: a solver for string constraints. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. 105–116. https://doi.org/10.1145/1572272.1572286

[13] Samuel Kolb, Stefano Teso, Andrea Passerini, and Luc De Raedt. 2018. Learning SMT (LRA) Constraints using SMT Solvers.. In *IJCAI*, Vol. 18. 2333–2340. https://doi.org/10.24963/ijcai.2018/323

[14] Mark V Lawson. 2003. *Finite automata*. CRC Press. https://doi.org/10.1201/9781482285840

[15] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering* 41, 12 (2015), 1236–1256. https://doi.org/10.1109/TSE.2015.2454513

[16] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering* 38, 1 (2011), 54–72. https://doi.org/10.1109/TSE.2011.104

[17] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. 2006. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*. ACM, 25–33. https://doi.org/10.1145/1181309.1181314

[18] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. 55–56. https://doi.org/10.1145/3135932.3135941

[19] Peter Linz. 2006. *An Introduction to Formal Language and Automata*. Jones and Bartlett Publishers, Inc., USA.

[20] Rushi Longadge and Snehalata Dongre. 2013. Class imbalance problem in data mining review. *arXiv preprint arXiv:1305.1707* (2013). https://doi.org/10.48550/arXiv.1305.1707

[21] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*. 691–701. https://doi.org/10.1145/2884781.2884807

[22] Katsuhiko Nakamura and Takashi Ishiwata. 2000. Synthesizing context free grammars from sample strings based on inductive CYK algorithm. In *International Colloquium on Grammatical Inference*. Springer, 186–195. https://doi.org/10.1007/978-3-540-45257-7_15

[23] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 772–781. https://doi.org/10.1109/ICSE.2013.6606623

[24] Davide Nicolini. 2009. Zooming in and out: Studying practices by switching theoretical lenses and trailing connections. *Organization studies* 30, 12 (2009), 1391–1418. https://doi.org/10.1177/0170840609349875

[25] Yannic Noller, Ridwan Shariffdeen, Xiang Gao, and Abhik Roychoudhury. 2021. Trust Enhancement Issues in Program Repair. In *International Conference on Software Engineering*. 11. https://doi.org/10.48550/arXiv.2108.13064

[26] Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru R. Căciulescu, and Abhik Roychoudhury. 2021. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering* 47, 9 (2021), 1980–1997. https://doi.org/10.1109/TSE.2019.2941681

[27] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2013. Does genetic programming work well on automated program repair?. In *2013 International Conference on Computational and Information Sciences*. IEEE, 1875–1878. https://doi.org/10.1109/ICCIS.2013.490

[28] Ridwan Shariffdeen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2021. Concolic Program Repair. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 390–405. https://doi.org/10.1145/3453483.3454051

[29] Shin Hwei Tan, Jooyong Yi, Yulis, Sergey Mechtaev, and Abhik Roychoudhury. 2017. Codeflaws: A Programming Competition Benchmark for Evaluating Automated Program Repair Tools. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 180–182. https://doi.org/10.1109/ICSE-C.2017.76

[30] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2014. S3: A Symbolic String Solver for Vulnerability Detection in Web Applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. Association for Computing Machinery, New York, NY, USA, 1232–1243. https://doi.org/10.1145/2660267.2660372

[31] Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2020. Inter-Theory Dependency Analysis for SMT String Solvers. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 192 (nov 2020), 27 pages. https://doi.org/10.1145/3428260

[32] Wei You, Xuwei Liu, Shiqing Ma, David Perry, Xiangyu Zhang, and Bin Liang. 2019. SLF: Fuzzing without Valid Seed Inputs *(ICSE '19)*. IEEE Press, 712–723. https://doi.org/10.1109/ICSE.2019.00080

[33] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200. https://doi.org/10.1109/32.988498

[34] Yunhui Zheng, Xiangyu Zhang, and Vijay Ganesh. 2013. Z3-str: A z3-based string solver for web application analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 114–124. https://doi.org/10.1145/2491411.2491456