Explaining Software Bugs Leveraging Code Structures in Neural Machine Translation

Parvez Mahbub
Department of Computer Science
Dalhousie University
Nova Scotia, Canada
parvezmrobin@dal.ca

Ohiduzzaman Shuvo
Department of Computer Science
Dalhousie University
Nova Scotia, Canada
oh599627@dal.ca

Mohammad Masudur Rahman
Department of Computer Science
Dalhousie University
Nova Scotia, Canada
masud.rahman@dal.ca

***Abstract*—Software bugs claim $\approx$ 50% of development time and cost the global economy billions of dollars. Once a bug is reported, the assigned developer attempts to identify and understand the source code responsible for the bug and then corrects the code. Over the last five decades, there has been significant research on automatically finding or correcting software bugs. However, there has been little research on automatically explaining the bugs to the developers, which is essential but a highly challenging task. In this paper, we propose Bugsplainer, a transformer-based generative model, that generates natural language explanations for software bugs by learning from a large corpus of bug-fix commits. Bugsplainer can leverage structural information and buggy patterns from the source code to generate an explanation for a bug. Our evaluation using three performance metrics shows that Bugsplainer can generate *understandable* and *good* explanations according to Google's standard, and can outperform multiple baselines from the literature. We also conduct a developer study involving 20 participants where the explanations from Bugsplainer were found to be more accurate, more precise, more concise and more useful than the baselines.**

***Index Terms*—software bug, bug explanation, software engineering, software maintenance, natural language processing, deep learning, transformers**

I. Introduction

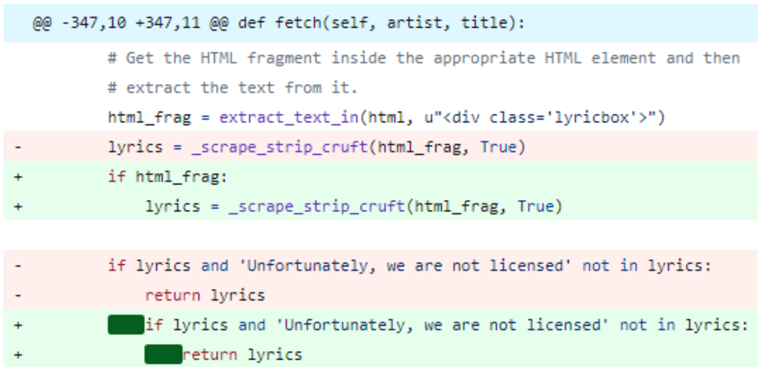
A software bug is an incorrect step, process, or data definition in a computer program that prevents the program from producing the correct result [1]. Bug resolution is one of the major tasks of software development and maintenance. According to several studies, it consumes up to 40% of the total budget [2] and costs the global economy billions of dollars each year [3], [4].

When an end-user reports a software bug, the assigned developer attempts to identify and understand the source code responsible for the bug and then corrects the code. Over the last five decades, there have been numerous approaches to automatically find the location of a bug [4], [5]. However, they often identify certain parts of the code as buggy without offering any meaningful explanation [6]. Developers are thus generally responsible for understanding a bug from the identified code before making any changes. Understanding bugs by looking at the code claims a significant chunk of debugging time. In fact, developers spend $\approx$ 50% of their time comprehending the code during software maintenance [7]. However, neither many studies attempt to explain the bugs in the source code to the developers, nor are they practical and scalable enough for industry-wide use [4], [6].

Explaining a bug in the software code is essential to fix the bug, but a highly challenging task. Many static analysis tools such as FindBugs [8], PMD, SonarLint, PyLint, and pyflakes [9] employ complex hand-crafted rules to detect the bugs and vulnerabilities in software code. Upon detection, they use pre-defined message templates to explain the bugs and vulnerabilities. Unfortunately, their utility could be limited due to their high false-positive results and the lack of actionable insights in their explanations [10]–[12]. In particular, their explanations are often too generic and unaware of the context due to their pre-defined, templated nature [13]. Thung *et al.* [14] also suggest that static analysis tools suffer from a large number of *false negative* results, which could leave the software systems vulnerable to bugs.

Unlike traditional, rule-based approaches (e.g., static analysis tools), explaining software bugs can be viewed as a translation task, where the buggy code is the source language and the corresponding explanation is the target language. In recent years, machine translation, especially neural machine translation (NMT) [15], has found numerous applications in several domains [16], [17]. NMT has also been used in different software engineering tasks including, but not limited to, code summarization [18], [19], code comment generation [20], [21], and commit message generation [22]–[25]. Traditional NMT models often consist of two items: *encoder* and *decoder*. The encoder first converts the words of the source language into an intermediate numeric representation. Then the decoder generates the target words one by one using the intermediate representation and previous words from the generated sequence [15]. However, explanation generation from the buggy source code using neural machine translation poses two major challenges as follows.

Understanding the structures of source code: Natural language is loosely structured, which exhibits phenomena like ambiguity and word movement [15]. Word movement is the appearance of words in a sentence in different orders but still being grammatically correct. On the contrary, programming languages are more structured, syntactically restricted, and less ambiguous [27]. From the two programs having the same vocabulary, one could be buggy and the other could be



```
@@ -347,10 +347,11 @@ def fetch(self, artist, title):
         # Get the HTML fragment inside the appropriate HTML element and then
         # extract the text from it.
         html_frag = extract_text_in(html, u"<div class='lyricbox'>")
-        lyrics = _scrape_strip_cruft(html_frag, True)
+        if html_frag:
+            lyrics = _scrape_strip_cruft(html_frag, True)

-        if lyrics and 'Unfortunately, we are not licensed' not in lyrics:
-            return lyrics
+            if lyrics and 'Unfortunately, we are not licensed' not in lyrics:
+                return lyrics
```

Fig. 1: An example of buggy source code

TABLE I: Generated explanations for buggy code

Technique	Generated Explanation
Ground Truth	Fix a bug where the lyricswiki fetcher would try to unescape an empty (None) response and crash
CommitGen [22]	Small bug fix for error handling
NNGen [25]	fix UnicodeDecodeError with non-ASCII text
Fine-tuned CodeT5 [26]	Don't try to get lyrics if we are licensed
pyflakes [9]	*no error found*
Bugsplainer	fix crash when lyrics not found

correct due to their structural differences (e.g., 4b, 4c). Thus, capturing and understanding the structures of code is essential to explaining the buggy code. Unfortunately, traditional NMT-based techniques often treat source code as a sequence of tokens and thus might fail to capture the structures of source code properly [28].

Understanding and detecting buggy code patterns: From a high-level perspective, NMT models translate words from the source language into words from the target language. However, to generate explanations from the buggy code, the model must be able to accurately reason about the bug from the buggy code and its structures. Such reasoning is non-trivial and warrants the model to be aware of buggy code patterns. Traditional NMT models might not be sufficient to tackle all these challenges due to their simplified assumptions about sequential inputs and outputs. According to Ray *et al.* [29] buggy code is less repetitive (a.k.a., *unnatural*) than regular code, which could exacerbate the above challenges.

In this paper, we propose *Bugsplainer*, a novel transformer-based generative model, that generates natural language explanations for software bugs by learning from a large corpus of bug-fix commits (i.e., commits that correct bugs). Our solution is able to address the above challenges, which makes our work *novel*. First, Bugsplainer can leverage code structures in explanation generation by applying structure-based traversal [19] to the buggy code. Second, we train Bugsplainer using both buggy source code and its corrected version, which helps the model to understand and detect buggy code patterns during its explanations generation for the buggy code.

We train and evaluate Bugsplainer with ≈ 150K bug-fix commits collected from GitHub using three different metrics – BLEU [30], Semantic Similarity [31] and Exact Match. We find that the explanations from Bugsplainer are *understandable* to *good*. We compare our technique with four appropriate baselines – pyflakes [9], CommitGen [22], NNGen [25], and Fine-tuned CodeT5 [26]. Bugsplainer outperforms all four baselines in all metrics by a statistically significant margin. One major strength of Bugsplainer is understanding the structure of the code and buggy code patterns, where the baselines might be falling short. To further evaluate our work, we conduct a developer study involving 20 developers from six countries, where the identities of both our tool and the baselines were kept hidden. The study result shows that explanations from Bugsplainer are more accurate, more precise, more concise, and more useful compared to that of the baselines.

We thus make the following contributions in this paper:

(a) A novel transformer-based technique, *Bugsplainer*, that can explain software bugs by leveraging the structural information and buggy code patterns from source code.
(b) A novel pre-training technique, namely – Discriminatory Pre-training, that is shown to be effective in generating better explanations.
(c) A benchmark dataset containing ≈ 150K instances of buggy code, corrected code, and corresponding explanations written by human developers. To the best of our knowledge, this is the first benchmark of its kind.
(d) A comprehensive evaluation and validation of the Bugsplainer technique using both popular performance metrics (e.g., BLEU score) and a developer study.
(e) A replication package that includes our working prototype, experimental dataset, and other configuration details for the replication or third-party reuse[1].

II. Motivating Example

To demonstrate the capability of our technique – *Bugsplainer*, let us consider the example in Fig. 1. The code snippet is taken from `beetbox/beets` repository at GitHub[2]. The buggy code attempts to scrape the lyrics of a song from an HTML fragment. However, if the HTML fragment is empty, then the program crashes. Table I shows both developer's explanation for the buggy code (a.k.a., reference) and the explanations generated by different techniques including Bugsplainer. We see that the explanations generated by CommitGen [22] (i.e., RNN-based technique) and NNGen [25] (i.e., Information Retrieval-based technique) are not helpful. On the other hand, the explanation from Fine-tuned CodeT5 [26] is not accurate as the bug has nothing to do with licensing. Pyflakes, a static analysis-based technique, does not provide any explanation since it was not able to detect the bug using its pre-defined rules. On the other hand,

[1]https://bit.ly/3H7R1aI
[2]https://bit.ly/3PGnkzK

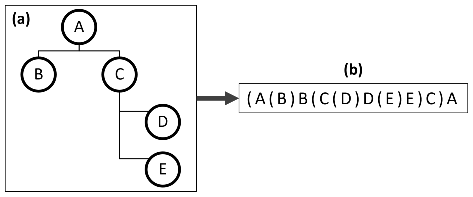


Fig. 2: Structure-based traversal (SBT) – (a) an example tree, and (b) corresponding SBT sequence

the explanation generated by our technique, Bugsplainer, is *accurate* as it expresses the same information as the ground truth and *precise* as it expresses no unnecessary information. Moreover, we see that in the fixed version of the code (Fig. 1), an **`if`** condition was used to check whether the HTML fragment exists (i.e., lyrics were found) or not, which reflects the solution implied by our explanation.

III. Background

A. Neural Machine Translation

Neural machine translation (NMT) is a deep neural network-based approach for automated translation [32]. In recent years, NMT has achieved rapid progress and has drawn the attention of both the research community and the practitioners. Generally, an NMT model is composed of two different blocks: *encoder* and *decoder*. The encoder accepts an input sequence and produces a numerical, intermediate representation of the input. Then, this intermediate representation is passed to the decoder. Based on this intermediate representation, the decoder starts to generate the target sequence, one token at a time. While generating each token, all the previously generated tokens are also passed to the decoder. This is known as *autoregressive process* where the current output is based on all previously generated outputs [33]. Bahdanau *et al.* [34] demonstrate how certain locations of the input sequence can be emphasized over others for an effective translation, which is known as the attention mechanism. The attention mechanism makes the training process faster and helps the NMT models translate long sequences [33]. In our research, we use Transformer [17], [33], the state-of-the-art NMT model along with the attention mechanism, as a part of Bugsplainer, to generate explanations for the buggy source code.

B. Structure-Based Traversal

Traditional NMT models treat their input as sequential data (e.g., English language texts). However, source code is rich in structures (e.g., syntactic or data dependencies), which are essential to convey the semantics of the code. One way to represent the syntactic structure of a source code document is an abstract syntax tree (AST). To leverage this structural information, several studies represent the tree structure into a sequence of code tokens and use it as the input to sequence-to-sequence models [19], [24]. Hu *et al.* [19] propose structure-based traversal (hereby SBT) to convert an AST node into a token sequence that can preserve the structural information. Fig. 2 shows an example tree and its corresponding SBT sequence. We use the SBT algorithm of Hu *et al.* to generate the sequence (see Algorithm 1 for details).

To generate the SBT sequence of a tree, we first use a pair of brackets to represent the tree structure and put the root node (i.e., `A`) behind the right bracket, i.e., `(A)A`. Next, we traverse the sub-trees of the root node and put all root nodes of sub-trees into the brackets, i.e., `(A(B)B(C)C)A`. Recursively, we traverse each sub-tree until all nodes of a tree are traversed. For example, we get the following SBT sequence –`(A(B)B(C(D)D(E)E)C)A` – for the example tree in Fig. 2a.

IV. Bugsplainer

Fig. 3 shows the schematic diagram of our proposed technique – *Bugsplainer* – for explaining software bugs. We discuss different steps of our technique in detail as follows.

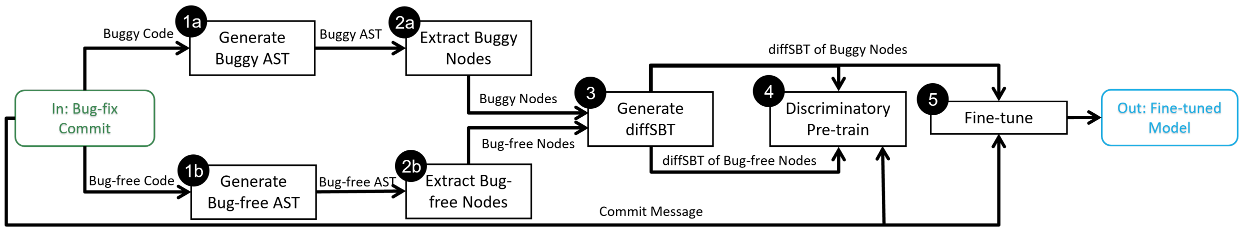
A. Extract Buggy and Bug-free AST Nodes from Commit

First, we construct abstract syntax trees (AST) of both buggy and bug-free code using the information from a bug-fix commit. A bug-fix commit contains the bug-free version of the code while being connected to its parent commit containing the buggy version. From these two versions of the source code, we construct two different ASTs – the buggy AST (Step 1a, Fig. 3a) and the bug-free AST (Step 1b, Fig. 3a). A commit also contains references to both removed lines (i.e., buggy lines) and added lines (i.e., bug-fix lines). Using these line numbers, we extract the buggy nodes from the buggy AST (Step 2a, Fig. 3a) and bug-free nodes from the bug-free AST (Step 2b, Fig. 3a). If a multi-line expression touches these line numbers, we extract the whole expression node (Line 11, Algorithm 1). Besides the affected lines, the contextual information (e.g., surrounding lines) often provides useful clues about why the code was changed. Asaduzzaman *et al.* [35] suggest that three lines of code around a target line might be sufficient to capture the contextual information. While extracting the buggy and bug-free nodes, we thus also extract the nodes representing three lines above and below the changed lines in the code.

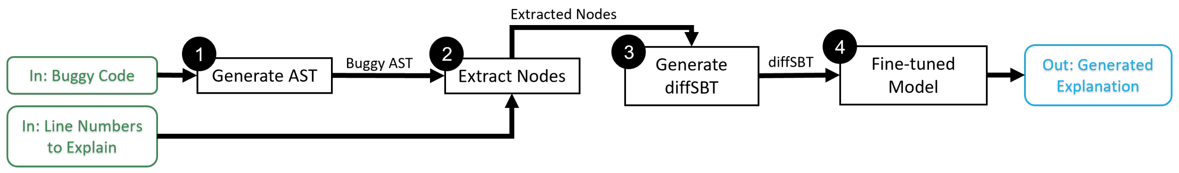
B. Generate diffSBT Sequence

In this step, we convert the buggy and bug-free AST nodes into diffSBT sequences (i.e., preserve structural information) using the diffSBT algorithm (Step 3, Fig. 3a). Algorithm 1 shows our algorithm – *diffSBT* – for structure-preserving sequence generation from commit diff, which is an adaptation of SBT algorithm by Hu *et al.* [19]. We create two versions of the diffSBT sequence. One of them contains both buggy and bug-free nodes to aid the discriminatory pre-training (see Section IV-C1). The other contains only buggy nodes to aid the fine-tuning.

To illustrate the generation of diffSBT sequences from a commit diff, let us consider Fig. 4. Fig. 4a contains a bug-fix commit. The source code before submitting this commit was buggy (Fig. 4b), and the source code after the commit is bug-free (Fig. 4c). The bug is that the `sanitize()` function



(a) Training of Bugsplainer



(b) Explanation generation for buggy code

Fig. 3: Schematic diagram of Bugsplainer

Algorithm 1 Generate diffSBT sequence from commit diff

```
 1: function DIFFSBT(c)  ▷ Generate diffSBT sequence for commit
 2:     buggyAST ← BUILDAST(c.buggyCode)
 3:     bugfreeAST ← BUILDAST(c.bugFreeCode)
 4:     buggyNodes ← INTERSECTIONS(buggyAST, c.removed)
 5:     bugfreeNodes ← INTERSECTIONS(bugfreeAST, c.added)
 6:     return SBT(buggyNodes) + $\langle /s \rangle$ + SBT(bugfreeNodes)
 7: end function

 8: function INTERSECTIONS(r, ln)
 9:     nodes ← $\phi$                ▷ Initialize nodes with an empty list
10:     for all n in r do        ▷ Get intersections for all nodes in $r$
11:         if ISINSIDE(n, ln) or ISEXPRESSION(n) then
12:             APPEND(nodes, n)
13:         else if STARTSINSIDE(r, ln) then      ▷ Keep node $r$ but
    prune the children outside $ln$
14:             r.children ← INTERSECTIONS(r.children)
15:             APPEND(nodes, n)
16:         else if ENDSINSIDE(r, ln) then     ▷ Node $r$ starts before
    the $ln$. Return only the children of $r$ that intersect with $ln$.
17:             children ← INTERSECTIONS(r.children)
18:             APPEND(nodes, children)
19:         end if
20:     end for
21:     return nodes
22: end function
```

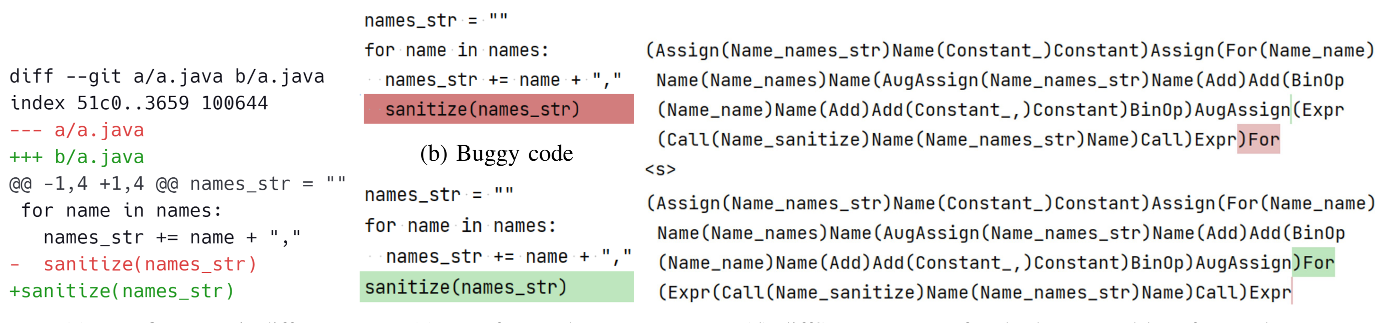
was called inside the `for` loop rather than outside the `for` loop. The buggy code resembles the fixed version of the code as both code segments also have the same vocabulary. However, they differ by white spaces, as shown in the commit diff (Fig. 4a), which is a scope-related problem according to Python programming language. Thus, if the source code is simply considered as a sequence of tokens without the structural information (as many studies [22], [25] do), the bug is really hard to understand. Fig. 4d shows the diffSBT sequence for this bug-fix change. We see that, in the buggy version, the `For` block closes at the end of the diffSBT sequence. On the contrary, the `For` block closes before the `Expr` block in the bug-free version. Such a placement ensures that the `Expr` block (i.e., `sanitize(name_str)`) is outside the `For` block. Thus, with the help of diffSBT, Bugsplainer can identify the difference in the structure of code, which could be useful to explain the bug.

C. Train Bugsplainer

In Fig. 3a, Steps 4-5 explain the training of Bugsplainer. Our training phase is divided into two steps – discriminatory pre-training and fine-tuning. In both steps, we use a RoBERTa tokenizer [36], pre-trained on GitHub CodeSearchNet dataset [37]. Due to its pre-train dataset (CodeSearchNet), this RoBERTa tokenizer has common code elements in its vocabulary, which can reduce the length of tokenized code sequence by 30%-45% [26]. We use this tokenizer to tokenize and encode the inputs (e.g., diffSBT sequence) and decode the outputs (e.g., commit message). In the following sections, we describe the training phase in detail.

1) Discriminatory Pre-training: Pre-trained language models have been found to be effective in improving many natural language understanding tasks (e.g., news title generation, question-answering) [16], [17]. During pre-training, a model acquires a general knowledge about a domain which allows it to *understand* the input (e.g., text, image) [17]. In natural language processing, pre-training is often performed in an unsupervised fashion (e.g., Word2Vec [38], missing token prediction [16]). However, many domains use supervised pre-training as well (e.g., Multi-Task Learning [17], [26]). Bugsplainer uses both unsupervised and supervised pre-training to equip the model with a comprehensive understanding of the programming language and its bugs.

We use a pre-trained model – CodeT5 [26] – to perform our discriminatory pre-training with buggy and bug-free code. CodeT5 is a transformer model based on the Text to Text Transfer Transformer (T5) architecture [17], [33]. It has two versions – 60M parameters and 220M parameters. We use the 60M parameter version for Bugsplainer, which is pre-trained



```
diff --git a/a.java b/a.java
index 51c0..3659 100644
--- a/a.java
+++ b/a.java
@@ -1,4 +1,4 @@ names_str = ""
 for name in names:
   names_str += name + ","
-  sanitize(names_str)
+sanitize(names_str)
```

(a) Bug-fix commit diff

```
names_str = ""
for name in names:
  names_str += name + ","
  sanitize(names_str)
```

(b) Buggy code

```
names_str = ""
for name in names:
  names_str += name + ","
sanitize(names_str)
```

(c) Bug-free code

```
(Assign(Name_names_str)Name(Constant_)Constant)Assign(For(Name_name)
 Name(Name_names)Name(AugAssign(Name_names_str)Name(Add)Add(BinOp
 (Name_name)Name(Add)Add(Constant_,)Constant)BinOp)AugAssign(Expr
 (Call(Name_sanitize)Name(Name_names_str)Name)Call)Expr)For
<s>
(Assign(Name_names_str)Name(Constant_)Constant)Assign(For(Name_name)
 Name(Name_names)Name(AugAssign(Name_names_str)Name(Add)Add(BinOp
 (Name_name)Name(Add)Add(Constant_,)Constant)BinOp)AugAssign)For
 (Expr(Call(Name_sanitize)Name(Name_names_str)Name)Call)Expr
```

(d) diffSBT sequence for the buggy and bug-free code

Fig. 4: An example of diffSBT sequence generation from buggy code and commit diff

on GitHub CodeSearchNet data [37] for three unsupervised tasks. CodeSearchNet contains ≈ 6M methods written in popular programming languages accompanied by natural language documentation. Thus, the CodeT5 model has a significant understanding of both programming and natural languages, which makes it an ideal choice for our pre-training task.

The pre-training with the CodeSearchNet dataset provides the model with general knowledge about programming and language syntax. However, to reason about a bug in the source code, the model should be able to differentiate between buggy and bug-free code. To equip the model with such a reasoning capability, we use diffSBT sequences of both buggy and bug-free AST nodes (Step 4, Fig. 3a). We pre-train the Bugsplainer model to predict commit messages from the diffSBT sequences of the buggy and bug-free code. We refer to this pre-training step as *discriminatory pre-training* since Bugsplainer learns to discriminate between buggy and bug-free code. The diffSBT sequences for the buggy and bug-free code are separated by a special token (</s>). We hypothesize that the model can differentiate and attend to (i.e., selectively focus on) the changes in both sides of the separator token and generate the commit message (a.k.a., bug explanation) accordingly. Our experimental result reports the effectiveness of discriminatory pre-training in explaining software bugs (see RQ$_2$ in Section V-C).

2) Fine-tuning: Once the discriminatory pre-training is complete, we also train Bugsplainer to generate explanations from only buggy code. We take diffSBT sequences of only buggy code as the input and corresponding explanation (i.e., commit message) as the output. We pass both input and output to the RoBERTa tokenizer. Then, we fine-tune our pre-trained model from the previous phase to generate explanations from the diffSBT sequence of buggy code (Step 5, Fig. 3a). The output of the fine-tuning step is the Bugsplainer model for bug explanation generation.

D. Generate Explanation

Once the training phase is complete, we test our model using the testing instances. Fig. 3b shows how Bugsplainer generates an explanation from buggy code. During the generation phase, Bugsplainer takes two inputs – the buggy code and the line numbers within the code that need an explanation. From the buggy code, Bugsplainer constructs the AST (Step 1, Fig. 3b) and extracts the AST nodes that intersect with the given line numbers (Step 2, Fig. 3b). Subsequently, Bugsplainer converts the intersecting nodes into a diffSBT sequence (Step 3, Fig. 3b). Then, it tokenizes the diffSBT sequence using the same RoBERTa tokenizer and passes the tokens to the fine-tuned model (Step 4, Fig. 3b). Finally, the fine-tuned model generates an explanation for the buggy code.

V. Experiment

We curate a large dataset of ≈ 150K bug-fix commits and evaluate Bugsplainer using three appropriate metrics from the relevant literature – BLEU score [30], Semantic Similarity [31], and Exact Match. To place our work in the literature, we compare our solution – Bugsplainer – with four relevant baselines. We also conduct a developer study to assess the quality of our automatically generated explanations (e.g., accuracy, usefulness) for software bugs. In our experiments, we thus answer four research questions as follows.

- **RQ$_1$**: How does Bugsplainer perform in explaining software bugs in terms of automatic evaluation metrics?
- **RQ$_2$**: How do (a) structural information and (b) discriminatory pre-training influence the performance of Bugsplainer in generating explanations for software bugs?
- **RQ$_3$**: Can Bugsplainer outperform the existing baseline techniques in terms of automatic evaluation metrics?
- **RQ$_4$**: How accurate, precise, concise, and useful are the explanations of Bugsplainer compared to baselines?

A. Dataset Construction

To conduct our experiments, we curate a dataset of ≈ 150K bug-fix commits from GitHub[3] using its REST API[4]. We discuss different steps of dataset construction as follows.

1) Repository Selection: First, to ensure high-quality commits, we aim to find ≈ $10K$ Python repositories with high star counts. We choose Python since it is the second most popular programming language according to StackOverflow survey 2021[5]. As GitHub's search API does not return more than $1K$ results from a single query, we use small buckets of

[3]Accessed: April 18, 2022
[4]https://docs.github.com/en/rest
[5]https://bit.ly/3cmooLv

star counts to renew our query contents. We found the $10{,}000^{th}$ repository falls in the bucket of 300-399 stars. Thus, we collect all the repositories that have a star count of ≥ 300, which led us to a total of $10{,}154$ repositories.

2) Collection of Bug-fix Commits: We collect all the commits from the above repositories, which led to a total of $\approx$ 11.8M commits. Then, we attempt to find the bug-fix commits from them. Similar to previous studies [39], [40], we consider a commit as a *bug-fix* commit if it contains either 'fix' or 'solve' in its commit message. This filtration step led us to a total of $\approx$ 1.4M bug-fix commits.

3) Filtration of Noisy Commits: To ensure commit quality, we perform a manual analysis of 500 commits that were randomly sampled from the above commit collection. We found seven machine-generated templates in the commit messages (e.g., "Merge branch X to master") that can be easily detected using appropriate regular expressions (see replication package for details). We remove these machine-generated templates from commit messages. If a commit message contains only machine-generated texts, then the whole commit is discarded from the dataset. We also note that Python repositories contain non-Python files (e.g., configuration files) and test scripts in their commits, which are out of the scope of this work. We thus keep the commits that have at least one modified Python file (excluding test scripts) in them.

Vaswani *et al.* [33] report that the complexity of transformer models increases quadratically with the length of input and output sequences. Therefore, we set a limit to the maximum length of both commit diff and commit message. In particular, we retain such commits that have $\leq$30 tokens in their commit message and $\leq$170 tokens in their commit diff. These limits cover $>$85% of both commit messages and commit diffs from the $\approx$ 1.4M bug-fix commits. Then, we remove commits with less than five tokens in their messages to discard trivial commits. We also keep only the commits that have one diff hunk (i.e. change location) to avoid tangled commits (i.e. commits doing more than one task). After performing all these noise filtration steps, we end up with $\approx$ 180K bug-fix commits.

To determine the reliability of our constructed dataset, we perform a manual analysis using 385 commits. We randomly sample these commits from $\approx$ 180K commits above with a 95% confidence level and 5% error margin. We find that 92.1% of these commits are bug-fix and 5.2% are style-fix, which indicates a negligible amount of noise in our constructed dataset. Previous studies [39], [40] also use datasets with similar amount of noise. Furthermore, manually filtering $\approx$ 180K commits was prohibitively costly or impractical, which possibly justifies our choice of using the current version of the dataset.

4) Embedding Structural Information: We generate diffSBT sequence for each commit as described in Section IV-B. We first generate AST for both the buggy and bug-free code from the commit using the `ast` parser of Python 3[6]. After discarding syntactically incorrect programs, we find a total of $\approx$ 150K diffSBT sequences.

5) Construction of Training and Testing Data: First, we randomly select 110K entries as the training dataset for both pre-training and fine-tuning steps. Second, we randomly split the remaining 40K entries into four sets that are allocated for validation and testing in both pre-training and fine-tuning steps. Thus, the training data are shared by both pre-training and fine-tuning steps whereas the validation and testing data are not shared. Finally, we remove the part of the diffSBT sequence that corresponds to the fixed version of the source code from the three fine-tuning splits (training, validation and testing) since Bugsplainer aims to generate an explanation from the buggy code only.

B. Evaluation Metrics

To evaluate the explanations generated by Bugsplainer, we use three different metrics – BLEU score [30], Semantic Similarity [31], and Exact Match. Relevant studies [17], [26], [32], [33] frequently used these metrics, which justifies our choice. They are defined as follows.

1) BLEU: Bi-Lingual Evaluation of Understanding: BLEU score [30] is a widely used performance measure for NMT. It has been used in software engineering context as well [19], [22]–[25], [28]. It calculates the similarity between auto-generated and reference sequences in terms of their n-grams precision as follows.

$$BLEU = BP \cdot exp\left(\sum_{n=1}^{N} w_n log(p_n)\right) \quad (1)$$

Here, p_n is the ratio between overlapping n-grams (from both generated and reference sequences) and the total number of n-grams in the generated sequence, and w_n is the weight of the n-gram length. Following the existing studies [19], [23], we use $N = 4$ and $w_n = 0.25$ for all $n \in [1, N]$. That is, we compute the mean BLEU score for all n-gram lengths. The brevity penalty, BP, lowers the BLEU score if the generated sequence is too small.

There exist several variations of the BLEU score. In our study, we use case-insensitive BLEU score with *add one smoothing* [41] which aligns the most with human judgement [23].

2) Semantic Similarity: Although the BLEU score is widely adopted for evaluating machine translation, it does not take the meaning of the text into account. Haque *et al.* [31] conduct a human study to determine which metric better represents the perception of human evaluators. They find that Sentence-BERT encoder [42] with cosine similarity has the highest correlation with the human evaluated similarity. Sentence-BERT provides a fixed-length numeric representation for any given text. As suggested by Haque *et al.* [31], we use `stsb-roberta-large`[7] pre-trained model to generate the embedding for the input text. We compute the Semantic Similarity as follows.

$$SemSim(ref, gen) = cos(sbert(ref), sbert(gen)) \quad (2)$$

[6]https://docs.python.org/3/library/ast.html

[7]https://bit.ly/3dR9mxD

TABLE II: Performance of Bugsplainer

Model	Dataset	BLEU	Semantic Similarity	Exact Match
Bugsplainer	Random split	33.15	55.76	22.37
	No CodeSearchNet Repository	34.53	56.67	19.55
	Cross-project	17.16	44.98	7.15
Bugsplainer 220M	Random split	33.87	56.35	23.50
	No CodeSearchNet Repository	35.59	57.29	20.74
	Cross-project	23.83	49.00	15.47

where $sbert(x)$ is the numerical representation from Sentence-BERT for any input text x, ref is the reference explanation, and gen is the generated explanation.

3) Exact Match: We also use the Exact Match metric to evaluate our explanation. As the name suggests, Exact Match checks whether a generated explanation exactly matches the corresponding reference explanation. It is analogous to string equality check in many programming languages, which is case-sensitive and space-sensitive.

C. Evaluating Bugsplainer

Answering RQ_1 – Performance of Bugsplainer: Table II shows the performance of Bugsplainer in terms of BLEU score, Semantic Similarity, and Exact Match.

When the dataset is split randomly into training, validation and testing sets, Bugsplainer achieves a BLEU score of 33.15, which is considered as *understandable* to *good* translation according to Google's AutoML Translation documentation[8]. Explanations from Bugsplainer also have an average of 55.76% Semantic Similarity, which indicates a major semantic overlap with the explanations from developers. Finally, 22.37% of the explanations exactly match the reference explanations. To achieve an Exact Match with the reference, an NMT model warrants a substantial knowledge of the domain. All these statistics are highly promising and demonstrate the high potential of our technique in explaining software bugs.

Allamanis [43] report that an overlap between training and testing datasets might lead to an overestimation in performance measurement. In our experiment design, we ensure that there is no overlap between our training and testing datasets (see Section V-A5). However, we also use a pre-trained CodeT5 [26] model which is pre-trained on millions of code snippets from thousands of repositories in CodeSearchNet dataset [37]. As a result, there might be an unavoidable overlap between pre-training and testing datasets. To ensure a fair evaluation, we thus discard the testing instances from CodeSearchNet repositories ($\approx$ 14% instances) to avoid any possibility of overlap and re-evaluate Bugsplainer. Table II shows that after discarding the overlapping repositories, Bugsplainer demonstrates a marginal improvement both in BLEU score and Semantic Similarity.

In the real world, when adopting Bugsplainer for a new project, data from the new project might not always be available to re-train Bugsplainer. Therefore, we investigate how the performance of Bugsplainer varies in a cross-project setting. That is, each of the training, validation, and testing datasets contain commits from mutually exclusive projects. From Table II, we see that even though the performance of Bugsplainer decreases in a cross-project setting, it is still promising, especially in terms of the Semantic Similarity metric. We see that the BLEU score decreases by 48% whereas the Semantic Similarity decreases by only 19%. That is, in the cross-project setting, the generated explanations might express similar information but with different words. To verify the case, we manually compare a sample of 385 explanations from Bugsplainer (95% confidence level and 5% error margin) with the reference explanations. We find that a substantial amount of generated explanations express information either more precisely or with different phrases, which might cause the BLEU score to be low. For instance, for a particular bug[9], Bugsplainer generates *"Improve the message in IncompleteRead.__init__"*, whereas the reference is *"fixing incorrect message for IncompleteRead."* Even though the generated explanation is accurate and more precise, it returns a BLEU score of only 11. Such a phenomenon also explains the low BLEU score and comparatively high Semantic Similarity score for the cross-project setting of Bugsplainer.

Recent studies suggest that increasing the model size can significantly improve the performance of deep learning models [17], [26], [36]. We thus were interested to see how the performance of Bugsplainer changes with an increased number of parameters. For this experiment, we train a 220M parameter variant of Bugsplainer and call it *Bugsplainer 220M*. Both variants share the same architecture (i.e., T5) but they have different hyperparameters. The detailed hyperparameter values can be found in the replication package. Table II also shows the performance of Bugsplainer 220M in the random split and cross-project settings. We see improved performance in both cases, which aligns with the existing findings [17], [26]. Interestingly, in cross-project settings, Bugsplainer 220M achieves a big bump of $\approx$ 39% in BLEU score and $\approx$ 117% in Exact Match. Such a finding suggests that Bugsplainer 220M can generalize the acquired knowledge better across multiple projects than Bugsplainer.

Finally, we investigate how the performance of Bugsplainer is affected by the input and output length. Table III shows the metric scores categorized by the number of words in the input buggy code segments. The table shows no clear correlation between the input length and the performance. With increasing input length, the performance both increases and decreases.

TABLE III: Performance of Bugsplainer by Input Length

#Words	BLEU	Semantic Similarity	Exact Match
$\# < 50$	32.05	54.90	17.84
$50 \leq \# < 100$	34.22	56.25	18.65
$100 \leq \# < 150$	34.72	56.99	21.10
$150 \leq \# < 200$	32.92	56.50	23.91

[8]https://bit.ly/3wGpCIx

[9]https://bit.ly/3RoSpsT

TABLE IV: Performance of Bugsplainer by The Length of Ground Truth

#Words	BLEU	Semantic Similarity	Exact Match
$\# < 10$	35.75	56.32	22.72
$10 \leq \# < 20$	27.70	53.16	8.93
$20 \leq \# < 30$	21.62	52.03	1.26

On the contrary, Table IV shows that the performance of Bugsplainer tends to decrease with increasing ground truth lengths. Interestingly, the drop in Semantic Similarity is not as strong as the BLEU score or Exact Match score. This suggests that even with increasing output length, Bugsplainer can provide explanations that are semantically coherent with the ground truth.

Summary of RQ$_1$: Bugsplainer can generate bug explanations that are *understandable* and *good* according to Google's standard. It shows promising results not only in random split settings but also in cross-project settings. With a higher number of parameters, Bugsplainer can better generalize the acquired knowledge across multiple projects.

Answering RQ$_2$ – Role of structural information and discriminatory pre-training in Bugsplainer: In this experiment, we analyze the impact of structural information and discriminatory pre-training on bug explanation generation. We remove one of these two components from Bugsplainer and keep the rest as is. Such an experiment helps us understand the contribution of individual components toward Bugsplainer.

To analyze the impact of structural information, we use raw commit diff as input rather than diffSBT sequences. In the pre-train dataset, we keep the commit diff as is, while in the fine-tuning dataset, we remove the added lines (i.e., bug-free lines) from the commit diff. From Table V, we see that the BLEU score of Bugsplainer reduces by 7.15% due to the absence of structural information. Interestingly, the Exact Match score also drops by 31.07%, which is significant.

To analyze the impact of discriminatory pre-training, we use only fine-tuning dataset and avoid the pre-training step. In this experiment, we use the diffSBT sequences as input during the fine-tuning step. From Table V, we see that the BLEU score of Bugsplainer reduces by 8.54% due to the absence of discriminatory pre-training. Interestingly, the Exact Match score also drops by 25.70%, which is significant.

The significant performance drops due to the absence of structural information and discriminatory pre-training indicate their important roles in Bugsplainer. Our technique also performs the best when both items are incorporated.

Summary of RQ$_2$: Both structural information and discriminatory pre-training have a major contribution to the performance of Bugsplainer. Furthermore, they are the most effective when they are used together.

Answering RQ$_3$ – Comparison with existing baseline techniques: In this research question, we compare Bugsplainer with existing techniques from the literature and investigate whether Bugsplainer can outperform them in terms of various

TABLE V: Role of structural information and discriminatory pre-training

Model	BLEU	Semantic Similarity	Exact Match
Bugsplainer	**33.15**	**55.76**	**22.37**
Bugsplainer without structural information	30.78	53.74	15.42
Bugsplainer without discriminatory pre-training	30.32	53.51	16.62

evaluation metrics. To the best of our knowledge, there exists no work that explains software bugs in natural language texts. Since commit message generation is quite similar to explanation generation, we use state-of-the-art commit message generation techniques as our baseline. The main difference between commit message generation and explanation generation is the former takes both buggy and bug-free lines as input whereas the latter takes only buggy lines as input. In particular, we compare Bugsplainer with three commit message generation techniques namely – *CommitGen* [22], *NNGen* [25], and *Fine-tuned CodeT5* [26] and a static analysis tool *pyflakes* [9]. None of these existing approaches for commit message generation learns to differentiate between buggy and bug-free code. Thus, our approach has a better chance of generating meaningful explanations for the buggy code.

To generate error messages from *pyflakes*, a static analysis tool, we run pyflakes on the whole buggy source code. Once we have the error messages, we keep only the messages generated for the buggy lines. If we get multiple errors for the same data point, we keep them all and report the one with the highest automatic metric score (e.g., BLEU score).

CommitGen uses an NMT framework namely nematus [44], which we use for our replication of the technique. The authors also provide the values of all the important parameters in their paper, which were carefully adopted in our replication.

According to a recent study [23], NNGen [25] is the state-of-the-art tool for generating commit messages. Being an Information Retrieval-based technique, NNGen does not require any training phase. It solely depends on the K-Nearest Neighbours algorithm. NNGen first finds k most similar commit diffs from the training set using *bag-of-words* model (i.e., term frequency) and cosine similarity measure. Since the authors do not provide any details of their bag-of-words implementation, we use the `CountVectorizer` API of the scikit-learn library in our replication. As suggested in the paper, we set the value of k to 5. From the top-k commits, NNGen selects the message from the most similar commit (using the BLEU score) as the final translation.

The fine-tuned CodeT5 model has the same architecture and the same hyperparameters as those of Bugsplainer. However, unlike Bugsplainer, it neither uses the structural information from the source code nor performs any discriminatory pre-training.

Table VI shows the comparison between Bugsplainer and four baselines in terms of BLEU score, Semantic Similarity, and Exact Match. The results shown in the table are the mean of five runs with different random initialization of the

TABLE VI: Comparison of Bugsplainer with existing baseline techniques (Using five random runs)

Technique	BLEU	Semantic Similarity	Exact Match
pyflakes	0.49	5.68	0.00
CommitGen	9.94	35.39	1.04
NNGen	24.16	47.33	14.17
Fine-tuned CodeT5	26.19	54.52	8.85
Bugsplainer*	**32.90**	**55.22**	**18.14**

*The scores differ from the earlier tables due to five random runs

parameters. We see that Bugsplainer outperforms all the baselines in terms of all three metrics. Only Fine-tuned CodeT5 is comparable with Bugsplainer. Therefore, we perform the Mann-Whitney U rank test [45] to see whether their performances over the five runs are significantly different. We found that Bugsplainer performs significantly higher than Fine-tuned CodeT5 i.e., *p-value* = 0.008 $<$ 0.05, Cliff's d = 1.0 (*large*) for all three metrics.

CommitGen relies on certain patterns in commit messages that might be generated by machines [25]. However, we removed auto-generated messages to ensure high-quality dataset (Section V-A3). CommitGen is also based on the LSTM architecture that does not perform well with long inputs [33]. Thus, the low scores of CommitGen are explainable.

According to Google's AutoML Translation documentation, a BLEU score between 20 and 29 indicates that the gist of a generated message is clear, but has significant errors. NNGen reuses existing commit messages from the training set and thus cannot analyze the dynamic behaviour of software programs. Thus, such errors in the translation are also explainable.

Thung *et al.* [14] report that static analysis tools produce a lot of false negatives. This means they do not produce any output for potentially buggy code in many cases. Our experiment with *pyflakes*, shows a similar result, generating error messages for only 7.70% cases. Therefore, its poor metric scores are also understandable.

Summary of RQ$_3$: Bugsplainer outperforms all four baselines in terms of three performance metrics. According to our statistical tests, our technique outperforms the closest competitor – Fine-tuned CodeT5 – by a statistically significant margin.

Answering RQ$_4$ – Evaluation of Bugsplainer using a developer study: The metric-based evaluation demonstrates the benefit of our technique in bug explanation generation. We also conduct a developer study to further demonstrate the benefit of Bugsplainer in a practical setting. Given the reference explanations of a software bug (e.g., the message of a bug-fix commit), we ask the developers to assess how accurate, precise, concise, and useful the explanations are. During the study, *we anonymize the model names* to avoid any bias.

Study participants: The target population of our study is English-speaking software engineers with experience in Python programming language. We invite our participants in two ways. First, we contact software companies with a history of participation in academic studies to contribute to

TABLE VII: Quality aspects of generated explanations

Quality	Overview
Accurate	It provides the same factual information as the reference.
Precise	It can pinpoint the issue in the code.
Concise	It is short and still conveys the whole message.
Useful	The provided information has the potential to fix the bug.

this research. Second, we advertise the study on the authors' social networks to reach potential participants and increase the diversity of samples. As of August 31, 2022, we receive a total of 20 responses to our developer study. The participants have professional software development experience of 1 to 10 years and bug-fixing experience of 1 to 7 years. All of them are familiar with Python programming language as well. Such experience makes them suitable candidates for our study.

Study setup: In the developer study, each participant worked with 15 bug-fix commits and spent 30 minutes on average. To select these examples for our developer study, we apply random sampling without replacement to the testing set. To avoid information overload, we take the examples that (1) do not have more than five changed lines or more than 15 word-tokens in a single line within the commit diff, and (2) do not require any project-specific knowledge to understand the bug. We take the first 15 randomly sampled examples matching these two criteria.

We ask the participants to assess the accuracy, precision, conciseness, and usefulness of the explanations from Bugsplainer and baselines with respect to the reference explanations. Table VII provides our definitions for these aspects. The participants assess these four aspects using a five-point Likert scale, where 1 indicates strongly disagree and 5 indicates strongly agree. Please note that *we anonymize the model names and do not show the participants which explanation comes from which model to avoid any potential bias*. We collect a total of 300 data points (15 questions $\times$ 5 explanations $\times$ 4 aspects) from each participant.

Study result and discussion: Table VIII summarizes our findings from the developer study. We note that the participants find the explanations from Bugsplainer to be the most *accurate*, most *precise*, most *concise* and most *useful*. Based on the median and mode values, we see that the participants agree the most with explanations from Bugsplainer. Similar to our findings in RQ$_3$, according to the developers, the closest competitor of Bugspaliner is Fine-tuned CodeT5. According to the mode values, the developers agree with Fine-tuned CodeT5 in many cases. However, looking at the 2^{nd} mode values, we see that the developers strongly disagree with Fine-tuned CodeT5 in a noticeable manner, making the mean agreement poor. We also perform the Wilcoxon Signed Rank test to check whether the developers' agreement with Bugsplainer is significantly higher than that of Fine-tuned CodeT5. For accuracy, conciseness, precision and usefulness, the p-values are $5.16e{-23}$, $2.74e{-17}$, $7.45e{-18}$ and $2.63e{-23}$ respectively, all are below the threshold of 0.05, which makes the difference significant.

Fig. 5 shows the distribution of participants' agreement

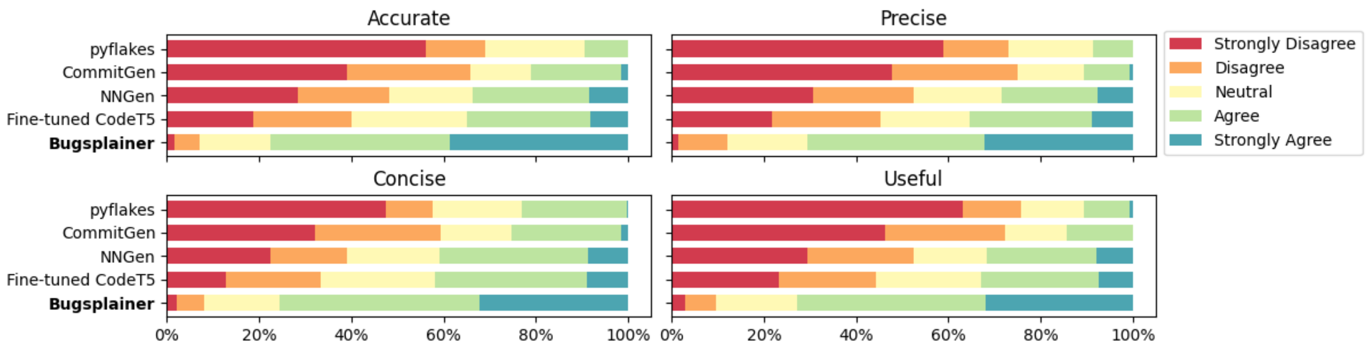


Fig. 5: Comparison of Bugsplainer with the baselines using Likert scores

levels in different aspects. We see that the participants disagree with Bugsplainer very few times (highest 14% times in precision) with a substantial amount of agreement (highest 76% in accuracy). Nearly half of the time the developers strongly disagree with pyflakes and CommitGen. Such disagreement with pyflakes is explicable since it does not generate any error message for 13 out of 15 cases. However, CommitGen, even after explaining all cases, receives a high disagreement due to its generic and less informative explanations.

Summary of RQ$_4$: Professional developers with bug-fixing experience find the bug explanations from Bugsplainer to be accurate, precise, concise, and useful. Their preference levels for Bugsplainer over other baseline techniques are also significantly higher.

VI. Related Work

A. *Explanation of Software Bugs*

Software bugs are errors, flaws, or defects in a program that prevent the program from working correctly [1]. They claim 50% of development time and cost the global economy billions of dollars every year [46]. While there have been numerous approaches to find or repair software bugs, neither many approaches attempt to explain the bugs in the source code to the developers, nor are they practical and scalable enough for industry-wide use [4], [6]. Several tools (e.g. FindBugs [8], PMD, SonarLint, PyLint, and pyflakes [9]) attempt to explain bugs using static analysis. Unfortunately, their utility could be limited due to their high false-positive results and lack of meaningful, actionable explanations [10]–[12]. Furthermore, their explanations can be too generic and limited by their templated natures [13]. Recent studies suggest complementing these messages with rule graphs [47], assertive error explanation [10], and interactive feedback from developers [48].

Besides the static analysis, there have been several attempts to explain a program's behaviours, failed tests, bug-fixing patches, undocumented code, and intelligent behaviours. Ko and Myers [48] design an interrogative debugging system for the Alice programming environment [49] where a novice learner can inquire why a program behaves unexpectedly or why it does not show an expected behaviour. Lim *et al.* [50] later suggest that these why and why not questions are essential to improve a user's understanding or perception of an intelligent system. Zhang *et al.* [51] explain a failed test case by automatically performing failure-correcting edits (e.g., replacement of identifiers' values) and synthesizing a code comment from them. Befrouei *et al.* [52] use program execution traces to explain concurrency bugs. Later Bragaglio *et al.* [53] remove irrelevant information from the execution traces to understand the cause of the unexpected behaviour. Liang *et al.* [54] investigate what should be included in a patch explanation, such as expected program behaviours or a high-level summary of code changes. Recently, Pornprasit *et al.* [55] also explain why the changed code can be defect-prone by visualizing how specific local rules are satisfied by a change. Although all these studies and approaches are relevant and are a source of our inspiration, they might be restricted to only specific problem contexts (e.g., Alice [49], failed tests [51]) or certain types of bugs (e.g., concurrency bugs [52]).

TABLE VIII: Comparison of Bugsplainer with baselines using developer study

Quality	Model	Mean	Median	Mode	2^{nd} Mode
Accurate	pyflakes	1.841	1	1	3
	CommitGen	2.176	2	1	2
	NNGen	2.653	3	1	4
	Fine-tuned CodeT5	2.842	3	4	3
	Bugsplainer	**4.074**	**4**	**5**	**4**
Precise	pyflakes	1.768	1	1	3
	CommitGen	1.884	2	1	2
	NNGen	2.529	2	1	2
	Fine-tuned CodeT5	2.772	3	4	2
	Bugsplainer	**3.891**	**4**	**4**	**5**
Concise	pyflakes	2.182	2	1	4
	CommitGen	2.350	2	1	2
	NNGen	2.881	3	4	1
	Fine-tuned CodeT5	3.044	3	4	3
	Bugsplainer	**3.974**	**4**	**4**	**5**
Useful	pyflakes	1.724	1	1	3
	CommitGen	1.960	2	1	2
	NNGen	2.576	2	1	4
	Fine-tuned CodeT5	2.728	3	4	1
	Bugsplainer	**3.923**	**4**	**4**	**5**

Unlike these traditional approaches, Bugsplainer is not restricted to any specific context or bugs. Besides, it can generate explanations that resemble that of humans, and are accurate, precise, concise, and useful (check RQ_4 for details).

B. Translation of Source Code into Texts

Our work is also related to code translation into natural language texts. Existing approaches translate code into the natural language to generate code comments, review comments, and commit messages.

Earlier works on code comment generation utilize hand-crafted templates [56], [57] and information retrieval [58], [59], while recent works more depend on learning-based approaches [19], [28], [60]. Wei *et al.* [61] combine both IR and NMT in comment generation. Recently, Mastropaolo *et al.* [62] use the Text-To-Text Transfer Transformer (T5) to perform several tasks including code comment generation. Their comments explain what is happening in the code rather than what makes the code buggy. On the contrary, Bugsplainer not only explains the buggy code but also suggests useful information to correct the bug in the code (e.g., Table I).

To generate code review comments, Tufano *et al.* [20] make use of the CodeT5 [26] model with Stack Overflow data and fine-tune their model with pairs of function and review comment pairs from GitHub. Later, Hong *et al.* [21] use Gestalt Pattern Matching (GPM) to mine candidate review comments of similar methods from a large corpus of source code repositories. However, both approaches treat source code as regular texts (e.g., a sequence of tokens) overlooking the structures. On the other hand, Bugsplainer leverages the structures of code through diffSBT sequences and learns to differentiate between buggy and bug-free code as a part of explanation generation.

To generate commit messages, several studies adopt an attention mechanism with RNN [18], [22], [24], [63] and leverage structural information [24], [63]. Xu *et al.* [63] jointly model the semantic representation and structural representation of code changes where they substitute identifier names with placeholders in the code. Liu *et al.* [24] capture both ASTs of code changes where they convert each AST into path sequence. However, the conversion of each change from the AST into its own path might lead to long, redundant sequences, which could hurt the translation performance. Liu *et al.* [25] show that a simple information retrieval-based approach, NNGen, has promising capability in commit message generation due to its repetitive nature. Both techniques above are closely related to ours due to their nature of translation. We thus compare Bugsplainer with them using experiments and the details can be found in Section V-C.

VII. Threats To Validity

Threats of *internal validity* relate to experimental errors and biases. Re-implementation of the existing baseline techniques could pose a threat. However, our implementation of NNGen [25] is based on the well-known k-nearest neighbour algorithm. CommitGen [22] uses a framework and reports all important hyperparameters. We use the same framework and the reported parameters for our implementation. For pyflakes, we use the officially provided package and follow the official documentation. Fine-tuned CodeT5 adopts the same model architecture as that of Bugsplainer. Thus, threats related to replication might be mitigated. We also repeat our experiments five times and compare the performance with that of baselines to mitigate any bias due to random trials.

Threats to *external validity* relate to the generalizability of our work. Even though Bugsplainer is evaluated using only Python code, the underlying algorithm is language-agnostic and can be easily adapted to any traditional programming language. The use of metric-based evaluation might also pose threat to the real-world usability of our approach [23], [31]. To mitigate this threat, we also conduct a developer study involving 20 participants from six different countries. As the developer study suggests, our bug explanations were also found to be accurate and useful in real-world scenarios.

Finally, the performance of Bugsplainer might depend on the precision of bug localization tools. To minimize this dependency, Bugsplainer accepts a range of lines containing both buggy lines and their surrounding lines as input during explanation generation. However, we do not indicate which lines among them contain a bug. Thus, precise localization of the bug either by developers or by existing tools might not be necessary to generate explanations using our tool.

VIII. Conclusion and Future Works

Software bugs not only claim precious development time but also cost billions every year. Although there have been many approaches for finding or repairing software bugs, there exists little research on automatically explaining the bugs. In this paper, we propose *Bugsplainer*, a novel technique that generates explanations for buggy code segments. Our technique can leverage both structural information and buggy code patterns from source code and employs neural machine translation with an attention mechanism to generate bug explanations. We evaluate Bugsplainer using three metrics (i.e., BLEU score, Semantic Similarity and Exact Match) where our technique outperforms the baselines. We also conduct a developer study involving 20 participants and our explanations were found to be more accurate and more useful compared to the baselines.

In future, we will investigate how to encode the structural information from source code in a more compact and efficient format and how to better leverage the structural differences between buggy and bug-free code. This might help us better understand the underlying semantics of software bugs.

Acknowledgement

This work was supported by Dalhousie University and Mitacs Accelerate International Program. We would like to thank Avinash Gopal, Ben Reaves, and Massimiliano Genta from our industry partner – *Metabob Inc*. We would also like to thank all the anonymous participants in our developer study.

References

[1] "IEEE Standard Glossary of Software Engineering Terminology," *IEEE Std 610.12-1990*, pp. 1–84, 1990.

[2] R. Glass, "Frequently forgotten fundamental facts about software engineering," *IEEE Software*, vol. 18, no. 3, pp. 112–111, 2001.

[3] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, *Reversible Debugging Software: Quantify the time and cost saved using reversible debuggers*, 2013.

[4] W. Zou, D. Lo, Z. Chen, X. Xia, Y. Feng, and B. Xu, "How practitioners perceive automated bug report management techniques," *TSE*, vol. 46, no. 8, pp. 836–862, 2018.

[5] M. M. Rahman, F. Khomh, S. Yeasmin, and C. K. Roy, "The forgotten role of search queries in ir-based bug localization: An empirical study," *EMSE*, vol. 26, no. 6, pp. 1–56, 2021.

[6] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proc. 25th, ISSTA*, 2016, pp. 165–176.

[7] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" In *2012 34th International Conference on Software Engineering (ICSE)*, IEEE, 2012, pp. 255–265.

[8] B. Pugh and A. Loskutov, *FindBugs™ - Find Bugs in Java Programs*, 2021. [Online]. Available: http://findbugs.sourceforge.net/ (visited on 01/18/2022).

[9] A. Sottile, *pyflakes: A simple program which checks Python source files for errors*, 2022. [Online]. Available: https://github.com/PyCQA/pyflakes (visited on 01/18/2022).

[10] T. Barik, "How should static analysis tools explain anomalies to developers?" In *Proc. 24th ACM SIGSOFT*, 2016, pp. 1118–1120.

[11] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" In *Proc. 35th ICSE*, IEEE, 2013, pp. 672–681.

[12] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proc.7th ACM SIGPLAN*, 2007, pp. 1–8.

[13] M. Nachtigall, M. Schlichtig, and E. Bodden, "A large-scale study of usability criteria addressed by static analysis tools," in *Proc.31st ISSTA*, 2022, pp. 532–543.

[14] F. Thung, D. Lo, L. Jiang, F. Rahman, and P. T. Devanbu, "To what extent could we detect field defects? an extended empirical study of false negatives in static bug-finding tools," *ASE*, vol. 22, no. 4, pp. 561–602, 2015.

[15] D. Jurafsky and J. H. Martin, *Speech and language processing*, 3rd ed. Hoboken, New Jersey: Prentice-Hall, Inc., 2014.

[16] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[17] C. Raffel *et al.*, "Exploring the limits of transfer learning with a unified text-to-text transformer," *JMLR*, vol. 21, pp. 1–67, 2020.

[18] P. Loyola, E. Marrese-Taylor, and Y. Matsuo, "A neural architecture for generating natural language descriptions from source code changes," *arXiv preprint arXiv:1704.04856*, 2017.

[19] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proc.26th ICPC*, IEEE, 2018, pp. 200–20 010.

[20] R. Tufano, L. Pascarella, M. Tufano, D. Poshyvanyk, and G. Bavota, "Towards automating code review activities," in *Proc.43rd ICSE*, IEEE, 2021, pp. 163–174.

[21] Y. Hong, C. Tantithamthavorn, P. Thongtanunam, and A. Alenti, "Commentfinder: A simpler, faster, more accurate code review comments recommendation," in *2022 ESEC/FSE*, ACM, 2022.

[22] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *2017 32nd ASE*, IEEE, 2017, pp. 135–146.

[23] W. Tao *et al.*, "On the evaluation of commit message generation models: An experimental study," in *2021 ICSME*, IEEE, 2021, pp. 126–136.

[24] S. Liu, C. Gao, S. Chen, N. L. Yiu, and Y. Liu, "Atom: Commit message generation based on abstract syntax tree and hybrid ranking," *TSE*, 2020.

[25] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang, "Neural-machine-translation-based commit message generation: How far are we?" In *Proc.33rd ASE*, 2018, pp. 373–384.

[26] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proc.2021 EMNLP*, 2021, pp. 8696–8708.

[27] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[28] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proc.54th ACL*, 2016, pp. 2073–2083.

[29] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the" naturalness" of buggy code," in *2016 IEEE/ACM 38th ICSE*, IEEE, 2016, pp. 428–439.

[30] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: A method for automatic evaluation of machine translation," in *Proc.40th ACL*, 2002, pp. 311–318.

[31] S. Haque, Z. Eberhart, A. Bansal, and C. McMillan, "Semantic similarity metrics for evaluating source code summarization," *2022 IEEE/ACM 26th ICPC*, 2022.

[32] Y. Wu *et al.*, “Google’s neural machine translation system: Bridging the gap between human and machine translation,” *arXiv preprint arXiv:1609.08144*, 2016.

[33] A. Vaswani *et al.*, “Attention is all you need,” *NeurIPS*, vol. 30, 2017.

[34] D. Bahdanau, K. Cho, and Y. Bengio, “Neural machine translation by jointly learning to align and translate,” *arXiv preprint arXiv:1409.0473*, 2014.

[35] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou, “Cscc: Simple, efficient, context sensitive code completion,” in *2014 ICSME*, IEEE, 2014, pp. 71–80.

[36] Y. Liu *et al.*, “Roberta: A robustly optimized bert pretraining approach,” *arXiv preprint arXiv:1907.11692*, 2019.

[37] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, “Codesearchnet challenge: Evaluating the state of semantic code search,” *arXiv preprint arXiv:1909.09436*, 2019.

[38] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient estimation of word representations in vector space,” *arXiv preprint arXiv:1301.3781*, 2013.

[39] M. Fischer, M. Pinzger, and H. Gall, “Populating a release history database from version control and bug tracking systems,” in *Proc.2003 ICSM*, IEEE, 2003, pp. 23–32.

[40] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, “An empirical study on learning bug-fixing patches in the wild via neural machine translation,” *ACM-TOSEM*, vol. 28, no. 4, pp. 1–29, 2019.

[41] C.-Y. Lin and F. J. Och, “Automatic evaluation of machine translation quality using longest common subsequence and skip-bigram statistics,” in *Proc.42nd Annual (ACL-04)*, 2004, pp. 605–612.

[42] N. Reimers *et al.*, “Sentence-bert: Sentence embeddings using siamese bert-networks,” in *Proc.2019 EMNLP*, ACL, 2019, pp. 671–688.

[43] M. Allamanis, “The adverse effects of code duplication in machine learning models of code,” in *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2019, pp. 143–153.

[44] R. Sennrich *et al.*, “Nematus: A toolkit for neural machine translation,” in *Proc.Software Demonstrations of the 15th ACL*, 2017, pp. 65–68.

[45] H. B. Mann and D. R. Whitney, “On a test of whether one of two random variables is stochastically larger than the other,” *TAMS*, pp. 50–60, 1947.

[46] M. M. Rahman, F. Khomh, and M. Castelluccio, “Why are some bugs non-reproducible?:–an empirical investigation using data fusion–,” in *2020 ICSME*, IEEE, 2020, pp. 605–616.

[47] L. N. Q. Do and E. Bodden, “Explaining static analysis with rule graphs,” *TSE*, 2020.

[48] A. J. Ko and B. A. Myers, “Designing the whyline: A debugging interface for asking questions about program behavior,” in *Proc.ACM-SIGCHI*, 2004, pp. 151–158.

[49] C. M. University. “Alice – Tell Stories. Build Games. Learn to Program.” (2021), [Online]. Available: https://www.alice.org/ (visited on 01/18/2022).

[50] B. Y. Lim, A. K. Dey, and D. Avrahami, “Why and why not explanations improve the intelligibility of context-aware intelligent systems,” in *Proc.ACM-SIGCHI*, 2009, pp. 2119–2128.

[51] S. Zhang, C. Zhang, and M. D. Ernst, “Automated documentation inference to explain failed tests,” in *2011 26th ASE 2011*, IEEE, 2011, pp. 63–72.

[52] M. T. Befrouei, C. Wang, and G. Weissenbacher, “Abstraction and mining of traces to explain concurrency bugs,” *FMSD*, vol. 49, no. 1, pp. 1–32, 2016.

[53] M. Bragaglio, N. Donatelli, S. Germiniani, and G. Pravadelli, “System-level bug explanation through program slicing and instruction clusterization,” in *2021 IFIP/IEEE 29th VLSI-SoC*, IEEE, 2021, pp. 1–6.

[54] J. Liang, Y. Hou, S. Zhou, J. Chen, Y. Xiong, and G. Huang, “How to explain a patch: An empirical study of patch explanations in open source projects,” in *2019 IEEE 30th ISSRE*, IEEE, 2019, pp. 58–69.

[55] C. Pornprasit, C. Tantithamthavorn, J. Jiarpakdee, M. Fu, and P. Thongtanunam, “Pyexplainer: Explaining the predictions of just-in-time defect models,” in *2021 36th ASE*, 2021, pp. 407–418. DOI: 10.1109/ASE51524.2021.9678763.

[56] P. W. McBurney and C. McMillan, “Automatic documentation generation via source code summarization of method context,” in *Proc.22nd ICPC*, 2014, pp. 279–290.

[57] G. Sridhara, L. Pollock, and K. Vijay-Shanker, “Automatically detecting and describing high level actions within methods,” in *2011 33rd ICSE*, IEEE, 2011, pp. 101–110.

[58] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, “On the use of automated text summarization techniques for summarizing source code,” in *2010 17th WCRE*, IEEE, 2010, pp. 35–44.

[59] E. Wong, J. Yang, and L. Tan, “Autocomment: Mining question and answer sites for automatic comment generation,” in *2013 28th ASE*, IEEE, 2013, pp. 562–567.

[60] M. Allamanis, H. Peng, and C. Sutton, “A convolutional attention network for extreme summarization of source code,” in *ICML*, PMLR, 2016, pp. 2091–2100.

[61] B. Wei, Y. Li, G. Li, X. Xia, and Z. Jin, “Retrieve and refine: Exemplar-based neural comment generation,” in *2020 35th ASE*, IEEE, 2020, pp. 349–360.

[62] A. Mastropaolo *et al.*, “Studying the usage of text-to-text transfer transformer to support code-related tasks,” in *Proc.Proc. ICSE*, IEEE, 2021, pp. 336–347.

[63] S. Xu, Y. Yao, F. Xu, T. Gu, H. Tong, and J. Lu, “Commit message generation for source code changes,” in *28th IJCAI 2019*, IJCAI, 2019, pp. 3975–3981.