SCHOLIA- An XAI Framework for APR

Nethum Lamahewage
University of Moratuwa, Sri Lanka
nethum.19@cse.mrt.ac.lk

Nimantha Cooray
University of Moratuwa, Sri Lanka
nimantha.19@cse.mrt.ac.lk

Ridwan Shariffdeen
National University of Singapore, Singapore
ridwan@comp.nus.edu.sg

Sandareka Wickramanayake
University of Moratuwa, Sri Lanka
sandarekaw@cse.mrt.ac.lk

Nisansa de Silva
University of Moratuwa, Sri Lanka
nisansadds@cse.mrt.ac.lk

***Abstract*—Automated Program Repair (APR) can assist developers by automatically generating patches for buggy code. However, as recent techniques leverage deep learning models, developers do not know why the model generated a particular patch. Existing Explainable AI (XAI) techniques, such as SHAP, can be applied to APR, however, their complexity raises questions about whether developers find such explanations understandable. In this work, we develop a novel framework SCHOLIA, with two extensions to feature attribution methods to make them more understandable to the developers. First generates a text explanation based on attribution scores. Second creates a visualization capturing the transformation of the patch based on the impact of code tokens, named patch transformation.**

We evaluated the proposed new two explanations types compared to SHAP, using a user survey. The survey received responses from 106 participants. Accordingly, 68.9% ($P < .05$) and 64.2% ($P < .05$) of participants agreed that text explanation and patch transformation methods are easy to understand, while only 17.9% ($P < .05$) agreed with the original SHAP explanation. The survey responses indicate, with statistical significance, that our extensions to SHAP are easier to understand than the original SHAP explanations.

***Index Terms*—Explainable Artificial Intelligence, Automated Program Repair, Deep Learning**

I. INTRODUCTION

Fixing software bugs manually is a monotonous and time-consuming task. Automated Program Repair (APR) addresses this issue by automatically fixing bugs [1]. While recent learning-based APR tools have achieved state-of-the-art results [2], [3] they are based on complex Deep Neural Networks (DNNs) such as transformers. These DNNs, particularly large-scale models like BERT [4] or GPT [5] are complex and black boxes [6], making it difficult for users to understand why such a model generates a particular patch for a given buggy code. In contrast, in manual program repair, the developer can explain why the patch was implemented to fix a bug.

Several empirical studies [7], [8] have demonstrated that the developers expect explanations for the automatically generated patches to understand the underlying reasons. Another study [9] shows that developer productivity isn't significantly improved by automatically generated patches alone, as developers must understand the defects and how the patches address them. Thus, when using APR tools, developers need explanations for the model's patch choices.

The eXplainable Artificial Intelligence (XAI) techniques developed in the AI research community can solve this problem by allowing users to gain insights into the rationale of AI systems' decisions. While there are various XAI techniques, the *feature attribution* methods explain the relationship between input and output of the model by indicating how much each input feature contributed to the model decision for a given instance (e.g., LIME [10], SHAP [11], Integrated Gradients [12]). When applied to APR tools, these methods will show how each token in the buggy code has influenced each token in the generated patch. However, existing research has shown that the explanations generated by these feature attribution methods, such as SHAP, are hard to comprehend for the users [13], [14]. Hence, it is imperative to develop a framework for APR tools that can generate more human-understandable explanations rationalizing decisions of APR tools.

This paper introduces SCHOLIA, a novel XAI framework for APR tools. It is developed based on existing feature attribution methods and offers two types of explanations: textual and patch transformation explanations, which are easily understood by the developers even if they do not have Machine Learning (ML) expertise. The textual explanations provide a natural language description for the rationale of the patch generated by an APR tool. To generate text explanations, SCHOLIA incorporates the syntactic structures and semantics in programming languages to generate more meaningful explanations for APR tool decisions. The patch transformation explanations demonstrate how the top-k influential tokens in the buggy code impact the patch. SCHOLIA can generate explanations based on any existing feature attribution methods for any learning-based APR tool, irrespective of the base model.

We evaluate the effectiveness of the explanations generated by the proposed XAI framework compared to existing feature attribution-based explanations and understand the developer's perspective of the explanations for patches generated by APR tools, we conduct a user survey. For this analysis, we apply SelfAPR [15] on bugs from the Defects4J dataset [16] and use SHAP[11] as the feature-attribution-based explanation method. The results of the survey indicate that the majority of the users find that text explanation (68.9%, $P < .05$) and patch transformation (64.2%, $P < .05$) are easier to understand than the original SHAP explanation (17.9%, $P < .05$).

To summarise, the contributions of this research are:

- To the best of our knowledge, this is the first work to apply XAI for learning-based APR tools and reports its usefulness.
- Develop a new XAI framework that extends existing feature-attribution-based explanation methods for learning-based APR tools.
- A user survey to evaluate the effectiveness of the explanations generated by the proposed XAI framework and understand the developer's perspective of the explanations

II. Motivational Example

Let's consider the bug `Cli 28`, from the Defects4J dataset [16], shown in Listing 1. For brevity, only the necessary parts of the code have been included here.

```
1   protected void processProperties(Properties properties) {
2     for (Enumeration e = properties.propertyNames(); e.hasMoreElements();) {
3       if (!cmd.hasOption(option)) {
4         if (opt.hasArg()) {
5         } else if (...) {
6           - break
7           + continue
8         }
9         ...
```

Listing 1: Developer Patch for Cli 28 (Defects4J)

This bug is from Apache Commons CLI[1], a library for parsing command line options passed to programs and it is in a class called `Parser`. This class builds an instance of `CommandLine`, a class that holds the parsed arguments. In `Parser::parse`, the method `processProperties` is called with the `properties` argument, which consists of the command line option key-value pairs. This method is supposed to set the values of `Options` using the values in the `properties` parameter. The `for` loop in line 2 iterates through the properties and processes each one. In the case that the current option has no argument, and the value is none of "yes", "true", or "1", then the current option should not be added to the `CommandLine` (as described in the comment on lines 10-11). However, in the buggy version of the program, this `Parser` stops parsing the arguments at this point, which is incorrect.

To fix this bug, suppose we use one of the state-of-the-art APR tools, SelfAPR [15]. SelfAPR generates the correct patch for this bug, replacing `break;` with `continue;` in Line 7 as shown in Listing 1. Applying this patch to the buggy code fixes the bug. Looking at the buggy code, we can see that the `break;` in line 7 terminates the loop as soon as the control reaches that conditional block. However, that would lead to the rest of the code not being processed, which is incorrect behavior. The generated patch fixes this by continuing to the next iteration with `continue;` instead of breaking out of the loop with the `break;`. That way, we do not skip processing the rest of the arguments and in fact, continue iterating through the properties until completion.

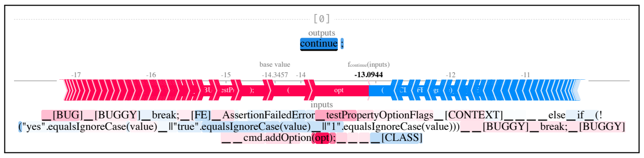
Next, we apply SHAP, an existing feature-attribution method, to understand SelfAPR's rationale for generating the patch. The output of SHAP is shown in Figure 1a. The SHAP explanation for text-to-text models is in the form of an interactive HTML page. At the top, we have the patch generated by the model. At the bottom, we have the input to the model, which is in the format expected by SelfAPR [15] requiring markers to indicate the buggy section (`[BUGGY]`), the context (`[CONTEXT]`), the class (`[CLASS]`), etc. The input and output are split into tokens based on the tokenization of the APR model. Figure 1a shows the view when we click on the `continue` token in the output, which shows us the attribution scores for that output token. The input tokens are highlighted based on their attribution scores. Positive attribution scores are shown in red, while negative ones are in blue. Positive attributions mean those input tokens have caused an increase in the probability that the APR tool generates the particular output token. In contrast, negative attributions mean those input tokens have caused a decrease in the probability. The darker the color, the higher the attribution. In our example, the highest positive attribution is for the `opt` near the end of the input. The line with arrows between the input and output shows these attribution scores. Tokens with positive attribution are on the right, and those with negative attribution are on the left. They are arranged in increasing order and decreasing order, respectively. Hovering over the tokens would also show the attribution score for that token. The line shows how all these attribution scores push the SHAP value from the base value to the final value. Figure 1a shows the `continue` input token was most impacted by the `opt` at the end of the context.

The main limitation of SHAP explanations is that users may find them hard to interpret due to their complicated representation, especially for text-to-text models. A developer must be instructed and trained to interpret a SHAP explanation. Even then, if the explanation in Figure 1a and the patch is shown to a developer who is working on this particular bug, they would likely not be able to grasp all the information that is presented in this explanation and make a decision.In our example, the patch contains just one token. However, for a multi-token patch, the SHAP explanation attributes each token in the buggy code to each token in the patch, resulting in $n \times m$ values, with n being the tokens in the buggy code and m those in the patch. This complexity increases the cognitive load for developers, who might prefer an overall explanation of why the APR tool generated the patch, rather than detailed attributions like those provided by SHAP.

In contrast, an explanation in natural language is easier to comprehend and places less mental burden on the developer. For example, consider the textual explanation shown in Figure 1b for the bug `Cli 28`.It offers a straightforward text explanation for the APR model's patch generation, identifying the specific part of the input buggy code—here, the condition in the `else if` statement—as the cause.

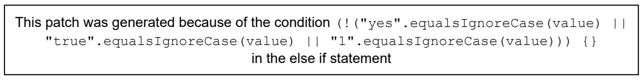
An alternative explanation that can be given is to show step-by-step how the top-k tokens impact the patch generation. The patch transformation-based explanation shown in Figure 1c indicates the impact on the patch from the top-k most impactful tokens in the buggy code. Initially, without the top 4 most impactful tokens, the patch is `continue;`. However, when we re-introduce the 3rd most impactful token, we can

[1]https://commons.apache.org/proper/commons-cli

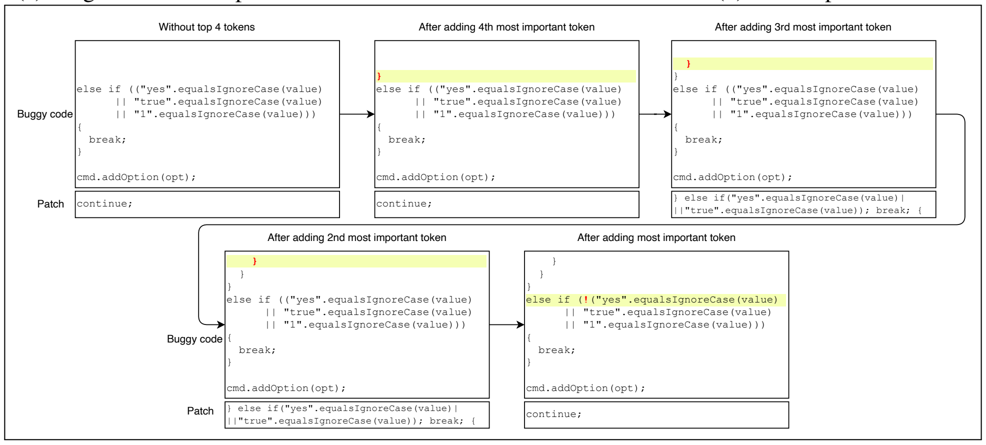


```
__[BUG]__[BUGGY]__break;__[FE]__AssertionFailedError__testPropertyOptionFlags__[CONTEXT]______else__if__(!
("yes".equalsIgnoreCase(value)__||"true".equalsIgnoreCase(value)__||"1".equalsIgnoreCase(value)))___[BUGGY]__break;__[BUGGY]
__cmd.addOption(opt);_____[CLASS]
```

(a) Original SHAP Explanation


This patch was generated because of the condition `(!("yes".equalsIgnoreCase(value) || "true".equalsIgnoreCase(value) || "1".equalsIgnoreCase(value))) {}` in the else if statement

(b) Text explanation


Without top 4 tokens
After adding 4th most important token
After adding 3rd most important token
After adding 2nd most important token
After adding most important token
Buggy code
Patch
Buggy code
Patch

Without top 4 tokens — Buggy code:

```
else if (("yes".equalsIgnoreCase(value)
        || "true".equalsIgnoreCase(value)
        || "1".equalsIgnoreCase(value)))
{
  break;
}

cmd.addOption(opt);
```

Patch:

```
continue;
```

After adding 4th most important token — Buggy code:

```
}
else if (("yes".equalsIgnoreCase(value)
        || "true".equalsIgnoreCase(value)
        || "1".equalsIgnoreCase(value)))
{
  break;
}

cmd.addOption(opt);
```

Patch:

```
continue;
```

After adding 3rd most important token — Buggy code:

```
  }
}
else if (("yes".equalsIgnoreCase(value)
        || "true".equalsIgnoreCase(value)
        || "1".equalsIgnoreCase(value)))
{
  break;
}

cmd.addOption(opt);
```

Patch:

```
} else if("yes".equalsIgnoreCase(value)|
||"true".equalsIgnoreCase(value)); break; {
```

After adding 2nd most important token — Buggy code:

```
    }
  }
}
else if (("yes".equalsIgnoreCase(value)
        || "true".equalsIgnoreCase(value)
        || "1".equalsIgnoreCase(value)))
{
  break;
}

cmd.addOption(opt);
```

Patch:

```
} else if("yes".equalsIgnoreCase(value)|
||"true".equalsIgnoreCase(value)); break; {
```

After adding most important token — Buggy code:

```
    }
  }
}
else if (!("yes".equalsIgnoreCase(value)
        || "true".equalsIgnoreCase(value)
        || "1".equalsIgnoreCase(value)))
{
  break;
}

cmd.addOption(opt);
```

Patch:

```
continue;
```

(c) Patch transformation

Fig. 1: Comparison of SHAP explanation and our proposed explanations

see the patch changes to an incorrect patch. Finally, when we re-introduce the most impactful token, the patch changes to the correct one: `continue;`.

Hence, in this work, we focus on improving the understandability of existing feature-attribution-based methods (i.e. SHAP [11]) by generating additional explanations with consideration given to the syntactic and semantic structure of the code, in contrast to the original attribution based explanation.

III. SCHOLIA FRAMEWORK

This section describes the proposed SCHOLIA XAI framework for APR tools. The objective of the proposed framework is to generate explanations that are easier to understand and more useful for software practitioners by extending the existing feature-attribution-based explanation methods.

Suppose $\mathcal{M}$ is a learning-based APR tool. It accepts an input buggy code $X = x_1, x_2, .., x_n$ with n tokens and produces a patch $Y = y_1, y_2, ..., y_m$ with m tokens. Let $\mathcal{F}$ be a feature-attribution-based explanation method. SCHOLIA accepts $\mathcal{M}$ and $\mathcal{F}$ and produces two types of explanations *text explanations* denoted as $\mathcal{T}$ and *patch-transformation explanation* denoted as $\mathcal{P}$. The overview of the framework is shown in fig. 2. On the left, we show the text explanation generation. On the right, we show the patch transformation. Next, in Section III-A, we describe how we generate the text-based explanation, and in Section III-B, we explain how we generate patch-transformation explanations.

A. Text Explanation

The inputs to the text explanation generation algorithm are the APR tool $\mathcal{M}$, the explanation method $\mathcal{F}$ and the buggy code X. Given the buggy code and the APR tool, the feature-attribution-based explanation method outputs the attribution matrix $A \in \mathcal{R}^{n \times m}$ containing attribution of each input token $x_i \in X$ with respect to each output token $y_j \in Y$. Then, we aggregate (i.e. `aggregate()`) the feature attributions by taking the summation of attributions of each input token. Let the output be $\bar{A} \in \mathcal{R}^n$. $\bar{a}_i \in \bar{A}$ indicates the impact of x_i on the entire patch.

Algorithm 1: Text explanation algorithm

```
Input: Learning-based APR Tool $\mathcal{M}$, Buggy code $X$,
       Feature-attribution-based explanation method $\mathcal{F}$
Output: Text explanation $T$
1 $A \leftarrow \mathcal{F}(M, X)$ ;
2 $\bar{A} \leftarrow$ aggregate($A$) ;
3 $a_{max} \leftarrow$ max ($\bar{A}$);
4 $N_{max} \leftarrow$ getASTNodeWithToken($X, a_{max}$) ;
5 $T \leftarrow$ applyTextTemplate($N_{max}$) ;
6 return $T$;
```

Next, we extract the token with the highest attribution value, a_k. Using, use the Abstract Syntax Tree (AST) of the buggy code X, we then translate the token a_k into an AST node o_k. The method `get_AST_node_with_token()` takes as input the buggy code X and the interested token a_k. The buggy code X is parsed into an AST tree and using the positional information of a_k, we locate the corresponding AST node o_k. Finally, we apply the relevant text template to build the final text explanation T. Procedure `apply_text_template` takes as input an AST node, and based on the AST node type, a predefined template will be used to generate the text description.

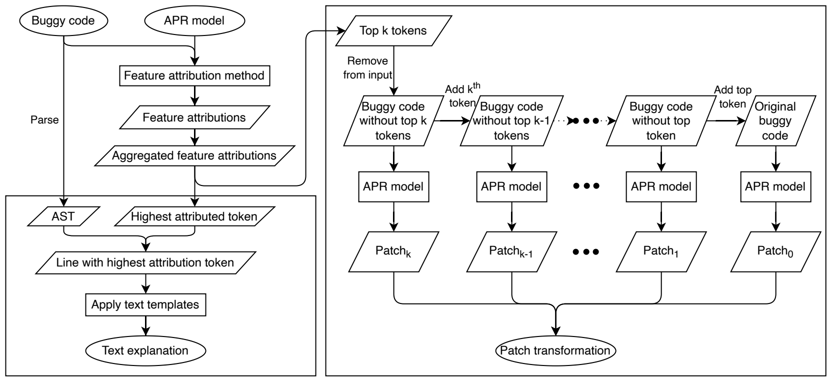


Fig. 2: Text explanation and patch transformation algorithm

```
public static boolean equal(GeneralPath p1, GeneralPath p2) {
  PathIterator iterator1 = p1.getPathIterator(null);
  - PathIterator iterator2 = p1.getPathIterator(null);
  + PathIterator iterator2 = p2.getPathIterator(null);
  ...
}
```

Listing 2: Chart 11 (Defects4J)

To illustrate further, we use `Chart 11` bug from the Defects4J [16] benchmark. Listing 2 shows the code for `Chart 11`. The bug is from a popular Java library JFreeChart. The `equal` method compares two polygons via the `GeneralPath` class. Here, the bug is in line 3, where instead of getting the path iterator for the second path, it gets the path iterator for the first path again. The correct patch is to change line 3 to `PathIterator iterator2 = p2.getPathIterator(null);`.

In our implementation, we use SelfAPR [15] as the state-of-the-art APR tool and SHAP [11] as the feature-attribution-based explanation method since it is a widely used XAI method. We chose SelfAPR as it is one of the state-of-the-art APR tools, it fixes a significant proportion of bugs [15], and it can generate a patch quickly (under a few seconds at most), which is essential for us to be able to apply an XAI method, as XAI methods such as SHAP query the model many times to build the final explanation [11]. For this particular bug, SelfAPR [15] generates the correct developer patch, and Listing 3 depicts our text-based explanation.

This patch was generated because of the method invocation `p1.getPathIterator`

Fig. 3: Text Explanation

Referring back to our motivational example `Cli 28` bug shown in Listing 1, based on the aggregation, we can identify the `!` in the `else if` condition (line 6) has the highest attribution. The AST node containing it is the condition node in the `else if`. Hence, the explanation from our algorithm is *This patch was generated because of the condition `(!("yes".equalsIgnoreCase(value) ——"true".equalsIgnoreCase(value) ——"1".equalsIgnoreCase(value)))` in the else if statement*. Compared to the complicated SHAP explanation shown in 1a, this explanation is much easier to digest and speaks in terms of code, which will be familiar to the developer using the APR model.

B. Patch Transformation

Similar to the text explanation generation described in Section III-A, given the APR tool $\mathcal{M}$, the explanation method $\mathcal{F}$ and the buggy code X, Algorithm 2 generates an explanation as a patch transformation.

Algorithm 2: Patch transformation algorithm

```
Input: APR Tool $\mathcal{M}$, Buggy code $X$, explanation method $\mathcal{F}$,
       Number of tokens $k$
Output: A sequence of buggy-fixed code pairs $(B, Y)$
1  $A \leftarrow \mathcal{F}(M, X)$ ;
2  $\bar{A} \leftarrow$ aggregate($A$) ;
3  $\bar{X} \leftarrow$ get_top_k($\bar{A}, X, k$) ;
4  $B_0 \leftarrow [X]$; $\bar{Y} \leftarrow []$ ;
5  for $i = 1$ to $k$ do
6      $B_i \leftarrow$ remove_token($B_{i-1}$, $\bar{x}_i$) ;
7  end
8  for $i = 0$ to $k$ do
9      $Y_i \leftarrow \mathcal{M}(B_i)$ ;
10 end
11 return reverse $B$, reverse $\bar{Y}$;
```

Algorithm 2 also calculates the aggregated attribution $\bar{a}_i$ for each token $x \in X$. Next, we extract the top-k tokens with the highest aggregated attributions. Suppose this set of tokens is denoted as $\bar{X} = \bar{x}_1, \bar{x}_2, ..., \bar{x}_k \subset X$, where the aggregated attribution decreases from $\bar{x}_1$ to $\bar{x}_k$. Next, we remove the tokens in $\bar{X}$ from the buggy code X one by one in the descending order of aggregated attribution value. This results in a set of new input buggy codes $B = B_1, B_2, ..., B_k$. For example, $B_1 = X - \bar{x}_1$ and $B_2 = X - \{\bar{x}_1, \bar{x}_2\}$ and so on. After that, we input these buggy codes to the APR tool $\mathcal{M}$ and generate the corresponding patches $\bar{Y} = Y_{B_1}, Y_{B_2}, ..., Y_{B_k}$. Finally, we formulate the sequence of these input-output pairs $(B_k, Y_{B_k}), (B_{k-1}, Y_{B_{k-1}}), ..., (X, Y)$ as the patch-transformation explanation. This sequence shows how adding top-k tokens one by one in the ascending order of feature attribution transforms the generated patch.

Referring back to our motivational example `Cli 28`, the `!` in the code has a high attribution score, that should mean that removing it should have a significant effect on the patch generated by the model. We can take the original input, remove the most impactful tokens, and then get the model output. From here, we can add those tokens back, one by one, obtaining the model output at each step as shown in Algorithm 2. With this list of patches, we can see how the patch transforms by re-introducing these supposedly impactful tokens. Note that the tokens we consider are Java tokens and not the tokens that are used by the model itself. Most of these APR models use subword tokenization [15], [3], [17], so we combine those tokens to get Java tokens.

IV. User Survey

We conduct a user study to evaluate the effectiveness of the explanations generated by the proposed XAI framework, text explanations, and patch transformation explanations compared to the feature attribution-based explanations. In our evaluation, we use SHAP [11] as the baseline feature-attribution-based explanation method. The main research questions to which we try to find answers are as follows (Figure 4):

RQ1: According to our participants, which explanation type is suitable for understanding APR tool generated patch?
RQ2: How do different characteristics of explanation types affect the user preference?
RQ3: How do different demographic characteristics affect the user preference for explanation types?

Symbol	Question
Q1	gender
Q2	age group
Q3	designation
Q4	Java programming experience
Q5	machine learning experience
Q6, Q8, Q10	qualitative analysis of each explanation type
Q7, Q9, Q11	additional feedback for each explanation type
Q12	ranking of three explanations
Q13	suggestions for new type of explanation

TABLE I: User Survey Questions

User Survey Design. The user survey consisted 13 questions in total. The first part contained 5 questions to collect demographic information about participants. The second part was focused on capturing user preference on three explanation types. It included 4 closed-ended questions and 4 open-ended questions. For closed-ended questions, we used multiple-choice questions, Likert scales, and ranking questions. The question structure is shown in Table I. The first section included demographic questions (Q1-Q5) to confirm participant suitability and gather factors like gender, age, designation, Java experience, and ML experience that could influence the results. At the beginning of the second part of the survey, users were given instructions on how to read and understand each explanation type. Next, three Java programming bugs (*Chart-1*, *Chart-11* & *Cli-28* from Defects4J dataset) were shown along with the correct patch generated by an APR tool (i.e. SelfAPR [15]) and a small description of the bug. For each bug, three explanation types were shown.

The users had to rank the three explanation types according to the usefulness of understanding what caused the model to generate the patch (Q12). If the users had any other ideal explanation type other than the ones presented, it was also

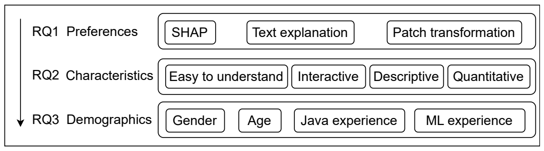


Fig. 4: Conceptual model for the user survey

asked (Q13). Instead of showing only the explanation types and asking about user preference, we showed three bugs and their patches to give the user more context and, by doing so, to simulate the real-world use of these explanations. Questions Q12 and Q13 will provide insights for RQ1.

A user survey conducted to assess the user perception of XAI in [18] reveals that the users expect the explanations to be Easy to understand, Interactive, Descriptive, and Quantitative. Hence, in our survey, then we asked the users (Q6, Q8, Q10) if they agreed that each explanation type possessed these characteristics. For these questions, 5-point Likert scales were used having a scale from "*Strongly disagree*" to "*Strongly agree*". Also, the user was asked to provide if there are any comments about each explanation type (Q7, Q9, Q11). The question set Q6-Q11 will provide insights for RQ2.

Participants. The main target group of this survey was developers experienced in Java programming. Using snowball sampling [19] we collected **106** survey responses. Multiple social media and email campaigns were initiated by the authors to attract more participants. Initial sampling started with contacts of the authors and using the snowball effect [19], the survey recorded answers from a diverse number of participants. We grouped all participants into 3 classes: a) *software practitioners*: those whose designation was related to software engineering industry (Software Engineer, Tech lead, Software Architect, QA Engineer/Lead, Project Manager & CTO) b) *Students*: those whose designation specified a student (Undergraduate, Masters, Graduate) c) *Others*: all other designations that are not captured in previous classifications (i.e. Researcher). Among the survey participants, 50% of our participants are *software practitioners*, while 41.5% are *students* and the rest of them, 8%, fall under *others*. The majority of the participants (43.4%) had a minimum of 2 years of experience, while 7.5% had no experience, and the rest of them with 1-2 years of experience.

Analysis. We manually went through the open-ended questions and identified points users have expressed. The closed-ended questions were quantitatively analyzed to obtain insights about user preferences. We used the chi-square goodness of fit test ($\alpha = 0.05$) to check if our results were statistically significant and not a random observation. The t-test for mean differences was used for analyzing the differences between demographic characteristics. For analyzing the ranking question (Q12), we used the Friedman test (α = 0.05) to check if there is a difference between the distribution of rankings of three explanation types. The results are presented in the Section V.

V. Survey Results

A. RQ1: Preferred Explanation

The average rankings of each explanation type are SHAP (2.14), Text (1.99), and Transformation (1.87). Accordingly, the most preferred explanation is *patch transformation* while the least preferred is *original SHAP explanation*. Our proposed extensions, *text explanation* and *patch transformation*, are preferred to the original SHAP explanation. Friedman test (p-value = 0.1366) for rankings suggests there is not enough evidence to say that there is a significant difference between the rankings for explanation types at α = 0.05. However, there is statistical significance for the observation of "original SHAP explanation is least preferred by users" ($P < 0.02$). Figure 5 shows the rankings given to explanation types by all participants. According to the average ranks, patch transformation is ranked first, text explanation is ranked second, and original SHAP explanation is the last.

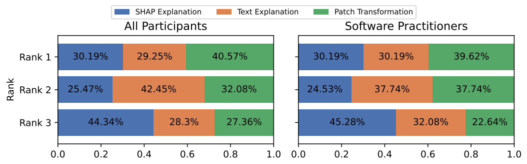


Fig. 5: Ranking of explanation types by profession

For the open-ended question on the ideal explanation type (Q13), a few comments mentioned they prefer a combination of text explanation and patch transformation.

- "It will be useful [sic] if the Explanation type 2 would be more descriptive since it is easy to understand. Another suggestion is combining Explanation type 2 and Explanation type 3." - Software Engineer (ID83)
- "…Certain type[sic] of explanations would make sense in certain scenarios. So, beat[sic] would be to strike a balance. I'd like a combination of 2 and 3." - Tech Lead (ID128)
- "I would like a hybrid of type 2 and type 3. A simpler explanation is[sic] natural language with a step-by-step interactive explanation.…" - Software Engineer (ID305)
- "A combination of interactive show case of where the bug accompanied with a brief explanation." - Software Engineer (ID306)

User Preference (RQ1): Original SHAP explanation is least preferred by our participants ($P < 0.02$)

B. RQ2: Explanation Characteristics

Figure 6 captures how users have responded to questions on characteristics of explanation types (Q6, Q8, Q10). These questions can be divided into four areas: easy to understand, interactive, descriptive, and quantitative.

Easy to Understand. The majority of the participants agrees that text explanation (68.9%, $P < .05$) and patch transformation (64.2%, $P < .05$) are *easy to understand* while only 17.9% ($P < .05$) state SHAP explanation to be so. Furthermore, among the software practitioners, 64.2% ($P < .05$) and 56.6% ($P < .05$) have, respectively, agreed that text explanation and patch transformation are *easy to understand* while only 20.8% ($P < .05$) agree for SHAP explanation.

Interactive. 80.2% of the participants ($P < .05$) agree patch transformation is interactive. While 46.2% ($P < .05$) and 28.3% ($P < .05$) of the participants agree that SHAP explanation and text explanation are interactive. Among software practitioners, 75.5% ($P < .05$) agree that patch transformation is interactive while 41.5% and 30.2% agree for SHAP explanation and text explanation.

Descriptive. Among all participants only 52.8% ($P < .05$) agreed that both text explanation and patch transformation are *descriptive*. 51.9% ($P < .05$) agree that the SHAP explanation is descriptive. Among software practitioners, 50.9% ($P < .05$) agree that the text explanation is descriptive while 43.4% ($P < .05$) and 41.5% agree for the patch transformation and the SHAP explanation.

Quantitative. For the quantitative characteristic, 58.5% ($P < .05$) of the participants agree that the SHAP explanation is quantitative while 41.5% and 29.2% agree for patch transformation and text explanation. This is obvious since there is no quantitative element in the text explanation and the patch transformation by design.

For the open-ended questions, we received several responses. The user comments on SHAP (Q7) highlighted the difficulty in understanding the explanation given (user ID is given in parentheses):

- "it's not very straightforward to understand…" - Tech Lead (ID128)
- "Less readable and need experience to understand the suggestions" - Tech Lead (ID248)
- "…hard to understand because it has too much information and data representation is hard to comprehend." - Software Engineer (ID312)

Users have highlighted that the text explanation is easier to understand, but lacks descriptiveness (Q9):

- "…definitely the easiest to understand." - Tech Lead (ID128)
- "…better than the Type 1, straightforward, understandable, but may not be descriptive enough." - Software Engineer (ID196)
- "…Simple and easy to understand but might lack more information …" - Software Engineer (ID312)

For the patch transformation method (Q11), mixed comments were received regarding interactivity and easy-to-understand properties.

- "it's interactive. But doesn't really make much sense to me. I don't see the usefulness[sic] of this replay of tokens." - Tech Lead (ID128)
- "Gives more interactive impact of patch implementation steps" - Tech Lead (ID248)
- "So far the most balanced one with more information and easy to understand." - Software Engineer (ID312)

Explanation Characteristics: majority of the participants agrees that text explanation (68.9%, $P < .05$) and patch transformation (64.2%, $P < .05$) are *easy to understand* and 52.8% of the participants ($P < .05$) agreed that both text explanation and patch transformation are *descriptive*.

C. RQ3: Impact of Demographics

We further analyze the preferences among different demographic groups to validate if our findings are consistent. A t-test was used to check if there was a difference between the average ranking between those groups. Table II summarizes this analysis. We re-compute the average ranking for each demographic group: gender, age, Java experience, and ML experience. For the gender group, we only received binary responses (despite providing non-binary options) hence we divided among male and female. For the age group, we divided

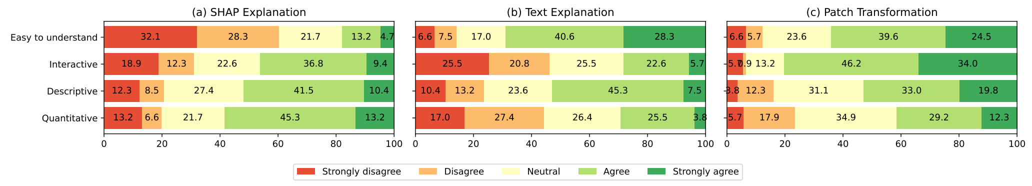


Fig. 6: Responses for questions on characteristics of explanation types

the participants as greater than or equal to 30 years and below. For the Java experience group we divided the participants as less than equal to 2 years and above. For ML experience group we had three classes (N - Novice, B - Beginner, O - Others).

We computed the mean, standard deviation, and t-test values to check for significant differences between subgroups. Overall, the results align with the ranking preferences, except in two cases where the text-based explanation was preferred: (i) participants aged 30 or older and (ii) those with beginner-level ML experience. Furthermore, the average ranking of patch transformation differs on ML experience (between Novice & Beginner) at a confidence level of 95%.

Impact of Demographics: the lowest average ranking is consistently observed where *original SHAP explanation* being ranked lowest among different demographic groups.

VI. Limitations and Threats to Validity

Limitations: Our text-based explanation highlights the most important section using constructs like if conditions but lacks finer granularity and overlooks other impactful tokens. In our patch transformation explanation, we focus on changes tied to the most impactful tokens, but the resulting series of patches may not provide sufficient clarity for developers to understand why the model generated a specific patch.

Threats to Validity: Despite using a snowball sampling method to collect diverse survey responses, we cannot guarantee that our findings apply to all software developers. To address this, we've made our research artifacts, including survey results, publicly available for replication. Another potential validity threat is the mix of software practitioners and students. While students may not reflect the expertise of practitioners, they are commonly used in developer studies [20]. To mitigate this, we present findings for all participants and separately for practitioners, noting no significant differences between the two groups.

The survey may not fully reflect real-world scenarios, as it only includes three bugs. To address this, we selected bugs from well-known, real-world projects in the Defects4J benchmark. Additionally, participants might misinterpret the survey questions. To minimize this risk, we conducted a pilot survey with 5 experienced programmers to ensure clarity. We found some users misunderstood the explanations as explanations of the bug itself rather than why the APR model generated the patch. In the final survey, we clarified that the explanations were meant to explain the model's patch generation.

VII. Related Work

Learning-based Automated Program Repair aims to fix software bugs automatically using Deep Learning techniques. Examples of learning-based APR tools include CURE [21] RewardRepair [3], DEAR [22] and SelfAPR [15]. These state-of-the-art learning-based APR tools utilize the advancements in Natural Language Processing. For example, CURE [17] is based on the GPT model. The authors pre-trained a programming language model on a large software code base to learn developer-like source code. RewardRepair [3] based on the T5 model, was trained based on rewarding the network to produce patches that compile and that do not over-fit. Finally, the SelfAPR [15], which has employed a self-supervised training approach, is based on T5.

Explainable Artificial Intelligence (XAI) methods for explaining the rationale behind Deep Learning models can be categorized along various dimensions. Post-hoc methods explain the decisions of already trained models operating external to the model [11], [12]. On the other hand, ante-hoc explanations are built into the deep learning model, e.g., CCNN [23]. Local explanations explain individual predictions [11], [24], whereas global explanations, such as feature importance rankings, explain the overall model tendencies. Moreover, XAI can be categorized into: 1) feature attribution explanations, such as gradient-based methods (e.g., Integrated gradients [12]), which pinpoint influential features towards the model decision, and 2) concept-based explanations like linguistic explanations [23], which provides explanations in terms of higher-level concepts. This work proposes a framework that extends feature-attribution-based explanation methods such as SHAP to generate concept-based explanations.

XAI for APR: Explaining the rationale behind the generated patch is crucial for APR to build trust and confidence in AI-generated code, and it directly impacts how fast a generated patch is merged into the codebase [25]. However, the incorporation of XAI for APR tools has been explored less. To the best of our knowledge, the only work in this domain is utilizing causal inference with sequence-to-sequence models by [26]. One key difference compared to our methods is the explanation format. Our explanations are textual and in the form of patch transformation, while this research generates a graph showing the dependencies between input and output tokens. Further, similar to other XAI techniques, such as SHAP, this work does not consider syntactic structures of code.

TABLE II: Demographic effect on the average rank for each explanation type

Explanation type		Gender		Age		Java Experience		ML Experience					
		Male	Female	<30yrs	>=30yrs	<=2yrs	>2yrs	N	B	N	O	B	O
SHAP explanation	Mean	2.15	2.09	2.11	2.23	2.10	2.20	2.23	2.04	2.23	2.15	2.04	2.15
	Std	0.86	0.87	0.87	0.82	0.86	0.86	0.84	0.85	0.84	0.87	0.85	0.87
	p-value	0.757		0.490		0.571		0.350		0.683		0.611	
Text explanation	Mean	1.99	2.00	2.05	1.83	2.03	1.93	2.09	1.85	2.09	1.94	1.85	1.94
	Std	0.74	0.87	0.75	0.79	0.76	0.77	0.78	0.77	0.78	0.75	0.77	0.75
	p-value	0.948		0.183		0.512		0.210		0.389		0.658	
Patch transformation	Mean	1.86	1.91	1.84	1.93	1.87	1.87	1.67	2.11	1.67	1.91	2.11	1.91
	Std	0.84	0.75	0.82	0.83	0.83	0.81	0.75	0.85	0.75	0.84	0.85	0.84
	p-value	0.792		0.607		0.986		0.027		0.203		0.361	

VIII. Conclusion

We proposed SCHOLIA framework that generates text and patch-transformation explanations, for the patches generated by the learning-based APR tools, utilizing an existing feature-attribution-based method. In implementing the framework, we have used SHAP as the feature attribution method and SelfAPR as the APR model. We have conducted a user survey with 106 participants to compare the effectiveness of our new framework against the original SHAP explanations. The survey results indicate that users find text explanation and patch transformation easier to understand compared to the original SHAP explanation. The most preferred explanation type is patch transformation, while the least preferred is the SHAP explanation.

Artifacts Link: DOI 10.5281/zenodo.11125367

References

[1] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[2] N. Jiang, T. Lutellier, Y. Lou, L. Tan, D. Goldwasser, and X. Zhang, "Knod: Domain knowledge distilled tree decoder for automated program repair," in *Proceedings of the 45th International Conference on Software Engineering*, 2023, p. 1251–1263.

[3] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1506–1518.

[4] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Minneapolis, Minnesota, 2019, pp. 4171–4186.

[5] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," 2018, publisher: OpenAI. [Online]. Available: https://www.mikecaptain.com/resources/pdf/GPT-1.pdf

[6] P. Lopes, E. Silva, C. Braga, T. Oliveira, and L. Rosado, "Xai systems evaluation: A review of human and computer-centred methods," *Applied Sciences*, vol. 12, no. 19, p. 9423, 2022.

[7] E. R. Winter, V. Nowack, D. Bowes, S. Counsell, T. Hall, S. Haraldsson, J. Woodward, S. Kirbas, E. Windels, O. McBello *et al.*, "Towards developer-centered automatic program repair: findings from bloomberg," in *Proceedings of the 30th ACM Joint ESEC and Symposium on the FSE*, 2022, pp. 1578–1588.

[8] Y. Noller, R. Shariffdeen, X. Gao, and A. Roychoudhury, "Trust enhancement issues in program repair," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, p. 2228–2240.

[9] J. P. Cambronero, J. Shen, J. Cito, E. Glassman, and M. Rinard, "Characterizing Developer Use of Automatically Generated Patches," in *2019 IEEE Symposium on Visual Languages and Human-Centric Computing*, Memphis, TN, USA, Oct. 2019, pp. 181–185.

[10] M. T. Ribeiro, S. Singh, and C. Guestrin, ""Why Should I Trust You?": Explaining the Predictions of Any Classifier," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, San Francisco California USA, Aug. 2016, pp. 1135–1144.

[11] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," *Advances in neural information processing systems*, vol. 30, 2017.

[12] M. Sundararajan, A. Taly, and Q. Yan, "Axiomatic attribution for deep networks," in *International conference on machine learning*. PMLR, 2017, pp. 3319–3328.

[13] H. Kaur, H. Nori, S. Jenkins, R. Caruana, H. Wallach, and J. Wortman Vaughan, "Interpreting interpretability: understanding data scientists' use of interpretability tools for machine learning," in *Proceedings of the 2020 CHI conference on human factors in computing systems*, 2020, pp. 1–14.

[14] S. Wickramanayake, S. Rasnayaka, M. Gamage, D. Meedeniya, and I. Perera, "Explainable artificial intelligence for enhanced living environments: A study on user perspective," 2023.

[15] H. Ye, M. Martinez, X. Luo, T. Zhang, and M. Monperrus, "Selfapr: Self-supervised program repair with test execution diagnostics," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2023.

[16] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, p. 437–440.

[17] N. Jiang, T. Lutellier, and L. Tan, "CURE: Code-Aware Neural Machine Translation for Automatic Program Repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering*, Madrid, ES, May 2021, pp. 1161–1173.

[18] S. Wickramanayake, S. Rasnayaka, M. Gamage, D. Meedeniya, and I. Perera, "Explainable artificial intelligence for enhanced living environments: A study on user perspective," in *Internet of Things: Architectures for Enhanced Living Environments*, ser. Advances in Computers, G. Marques, Ed. Elsevier, 2024, vol. 133, pp. 1–32.

[19] Y. Y. V. . R. C. P. Dusek, G. A., "Using social media and targeted snowball sampling to survey a hard-to-reach population: A case study," in *International Journal of Doctoral Studies*, 2015, p. 279–299.

[20] B. Kitchenham, S. Pfleeger, L. Pickard, P. Jones, D. Hoaglin, K. El Emam, and J. Rosenberg, "Preliminary guidelines for empirical research in software engineering," *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 721–734, 2002.

[21] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *Proceedings of the 43rd International Conference on Software Engineering*, 2021, p. 1161–1173.

[22] Y. Li, S. Wang, and T. N. Nguyen, "DEAR: a novel deep learning-based approach for automated program repair," in *Proceedings of the 44th International Conference on Software Engineering*, Pittsburgh Pennsylvania, May 2022, pp. 511–523.

[23] S. Wickramanayake, W. Hsu, and M. L. Lee, "Comprehensible convolutional neural networks via guided concept learning," in *2021 International Joint Conference on Neural Networks*. IEEE, 2021, pp. 1–8.

[24] ——, "Flex: Faithful linguistic explanations for neural net based model decisions," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, 2019, pp. 2539–2546.

[25] M. Monperrus, "Explainable software bot contributions: Case study of automated bug fixes," in *2019 IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE)*, 2019, pp. 12–15.

[26] J. Wang, S. Si, Z. Zhu, X. Qu, Z. Hong, and J. Xiao, "Leveraging causal inference for explainable automatic program repair," in *2022 International Joint Conference on Neural Networks*, 7 2022, pp. 1–6.