Boosting Redundancy-based Automated Program Repair by Fine-grained Pattern Mining

Abstract—**Redundancy-based automated program repair (APR), which generates patches by referencing existing source code, has gained much attention since they are effective in repairing real-world bugs with good interpretability. However, since existing approaches either demand the existence of multi-line similar code or randomly reference existing code, they can only repair a small number of bugs with many incorrect patches, hindering their wide application in practice. In this work, we aim to improve the effectiveness of redundancy-based APRs by exploring more effective source code reuse methods for improving the number of correct patches and reducing incorrect patches. Specifically, we have proposed a new repair technique named REPATT, which incorporates a two-level pattern mining process for guiding effective patch generation (i.e., token and expression levels). We have conducted an extensive experiment on the widely-used Defects4J benchmark and compared REPATT with nine state-of-the-art APR approaches. The results show that it complements existing approaches by repairing 9 unique bugs compared with the latest Large Language Model (LLM)-based and deep learning-based methods and 19 unique bugs compared with traditional repair methods when providing the perfect fault localization. In addition, when the perfect fault localization is unknown in real practice, REPATT significantly outperforms the baseline approaches by achieving much higher patch precision, i.e., 83.8%, although it repairs fewer bugs. Moreover, we further proposed an effective patch ranking strategy for combining the strength of REPATT and the baseline methods. The result shows that it repairs 124 bugs when only considering the Top-1 patches and improves the best-performing repair method by repairing 39 more bugs. The results demonstrate the effectiveness of our approach for practical use.**

Index Terms—**Automated program repair, Pattern mining, Program debugging**

I. INTRODUCTION

Automated program repair (APR) techniques have the potential to significantly reduce the debugging overhead of human developers and thus have been widely studied in the last decades. To date, a large number of APR approaches have been proposed and developed [1]–[11]. A typical APR approach takes a faulty program and a set of test cases, where at least one triggers the bug in the program, as inputs, and produces one or more (if possible) *plausible* patches that can make all the test cases pass. The repair process can be typically viewed as a search problem, where the search space is all possible programs. Therefore, the core challenge for APR techniques is how to effectively and efficiently locate the desired patches within limited computing resources and time budget. To facilitate this process, existing techniques have employed multiple data sources, e.g., similar code [8], [9], historical patches [12], [13], repair templates [4], [14], [15], etc., and leveraged diverse search algorithms, e.g., random search [16], genetic programming [6], [17], and advanced Large Language Model (LLM) and deep learning [18]–[24], aiming to effectively refine the search space and thus improve the repair ability and efficiency.

Among existing APR techniques, redundancy-based [25]–[28] techniques have attracted much attention and achieved considerable success. These techniques are on the basis of the **plastic surgery hypothesis** that "*changes to a codebase contain snippets that already exist in the codebase at the time of the change, and these snippets can be efficiently found and exploited* [29]". In other words, the code snippets for patch generation can be found within the buggy projects themselves. Therefore, redundancy-based APR approaches leverage the plastic surgery hypothesis to generate patches by searching and referencing existing code. For example, GenProg [6], [17], the pioneer of modern APR techniques, generates patches by directly reusing existing code under the guidance of genetic programming algorithms. Although the recent advance of LLMs has shown the promise to repair more bugs [20], [22]–[24], they still suffer from the interpretability issue due to the complexity and the random nature of deep neural networks as they are typically used in a black-box fashion. As reported by previous studies [30]–[32], the patches generated by SimFix, a representative redundancy-based APR approach, are of high quality under various testing scenarios. In addition, redundancy-based techniques also complement those learning-based techniques [18]–[20] (our evaluation results in Section V further prove it). Additionally, redundancy-based techniques tend to have better interpretability as they are on the basis of the *plastic surgery hypothesis*, and the patches are typically generated by referring human-written code snippets. In particular, they can repair a portion of relatively complex bugs [8], [9], [33].

Although existing redundancy-based techniques are demonstrated to be effective for repairing real-world bugs, they still face two major challenges: (1) *How to effectively and efficiently locate the reference code elements among a large-scale codebase?* Existing approaches identify similar snippets using structural [8], [33] or token-based [9] features, assuming similarity implies functional equivalence. However, as shown in our preliminary study (Section II-A), coarse-grained similar snippets are rare in practice, limiting their usability. (2) *How to reuse existing code for constructing new patches?* Existing approaches reuse code by comparing the difference between faulty and reference code via abstract syntax trees [8], [9] or program dependency graphs [33]. However, code elements under different contexts tend to vary greatly in structures [34], limiting the effectiveness of existing techniques, i.e., repairing a small number of real-world bugs with many incorrect patches.

As explained above, existing redundancy-based approaches reuse code either completely randomly (e.g., RSRepair [16] and GenProg [6]) or depending on the existence of similar code snippets of multiple lines (e.g., HDRepair [35], SimFix [8] and TransplantFix [33]). However, our preliminary study reveals that most reusable elements appear at a finer-grained level—e.g., 89.3% contain fewer than three tokens (Section II-A). This suggests that finer-grained code reuse is crucial. In this paper, we investigate the possibility of improving redundancy-based APR techniques by reusing code elements at a finer-grained level. Specifically, the basic idea of our approach is that **fine-grained code elements related to similar code semantics tend to co-appear nearby in the program**, which has been well-studied by existing research [36]–[39]. This phenomenon, which we refer to as the *locality property* of source code, suggests that semantically related code components—such as variables, functions, or expressions—are often found in close proximity within the same code block or module. Based on this property, we proposed a new APR approach that effectively guides patch generation by identifying the usage patterns of fine-grained code elements for confining the patch space and thus overcomes the first challenge of redundancy-based APR. To overcome the second challenge, i.e., reusing existing code under different contexts, we designed a new code representation method, i.e., S-TAC, which decomposes complex code expressions and statements into a unified form by ignoring context-specific features (e.g., code structures and operators). It can reduce the negative impact of code structures on code matching during patch generation but still preserve the *locality property* of source code.

To evaluate the performance of our approach, we have implemented it as an APR tool named REPATT and conducted an extensive study by comparing it with nine state-of-the-art APR approaches, including three best-performing traditional repair techniques and six latest LLM-based and deep learning-based techniques. The experimental results show that although our approach did not outperform all baselines considering all comparison aspects, it complements existing approaches by correctly repairing many unique bugs. Specifically, REPATT repairs 19 unique bugs that cannot be repaired by traditional methods and 9 unique bugs that neither LLM-based nor deep learning-based methods can repair. In particular, 5 bugs repaired by REPATT have never been repaired by all the baselines. Additionally, when the perfect fault localization is unknown, which is a more realistic scenario, REPATT can significantly outperform the baseline approaches by achieving 15.6%-51.7% higher patch precision, which is critical for practical use of APRs. Additionally, we also made the first attempt to combine the strength of REPATT and multiple existing approaches by further designing a patch ranking strategy. The evaluation result shows that the combined method can effectively repair 124 bugs when only considering Top-1 patches, 39 more bugs than the best-performing method (i.e., TBar). Our results demonstrate that our approach is indeed effective in improving the performance of existing APR techniques.

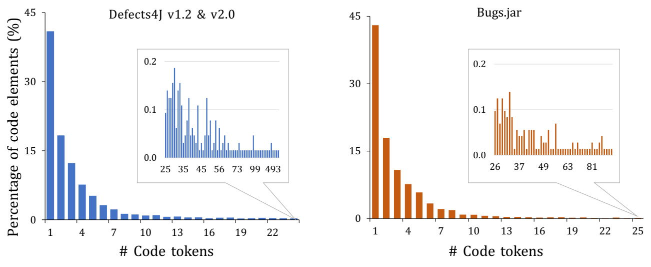


Fig. 1. The length distribution of code elements that can be found in the faulty program for repairing 564 and 609 bugs from benchmarks Defects4J and Bugs.jar.

In summary, we make the following major contributions.

- A new redundancy-based APR technique that can flexibly reuse source code at a finer granularities.
- A novel code representation method, which overcomes the diversity of code contexts, for better code search and reuse.
- An extensive study to evaluate the performance of our approach by comparing it with state-of-the-art approaches.
- The first attempt to combine the strength of multiple APR approaches, revealing its promise for practical use.
- We make all our experimental results and implementations publicly available to facilitate replication and comparison [40].

II. MOTIVATION

A. Preliminary Study

As previously mentioned, no existing studies have explored the potential of searching for and reusing finer-grained code elements for patch generation. To investigate the feasibility and necessity of this approach in APR, we conducted a preliminary study following the idea proposed by Barr et al. [29].

Specifically, our study aims to understand the granularity of reusable code elements for patch generation. Given a bug and its associated patch, we first extracted newly added code elements from the patch and searched for them in the faulty program, starting from coarse-grained elements and gradually decomposing them into finer-grained components based on the syntax tree structure. That is, if a code element (e.g., "`a + f(b,c)`") was not found, we would further decompose it into finer-grained code elements of the code (e.g., "`a`", "`+`", and "`f(b,c)`"). This process iterated until no further decomposition was possible. Finally, we analyzed the granularity of code elements that can be found in faulty programs. In particular, since AST structures cannot directly reflect granularity (e.g., an `expression` in the AST can be either a variable like "`a`" or a complex expression like "`f(a, b)`".), we measured granularity by the number of tokens in each code element.

We conducted our empirical study on two widely-used datasets of real-world bugs, i.e., Defects4J [41] and

Bugs.jar [42]. In particular, we filtered out bugs whose patches involve more than one Java file since they are still hard to repair at present [2], [8], [18], [20], [26], [43]–[45], resulting in 564 bugs from Defects4J and 609 bugs from Bugs.jar across 24 projects. Figure 1 presents the statistical results. The *x-axis* denotes the number of tokens in a code element and the *y-axis* denotes the percentages of code elements that can be reused for patch generation. The results show on average more than 89.3% code elements contain less than three tokens, and 42% elements only contain one token. In contrast, only a very limited portion of code elements exist at a coarse-grained granularity (i.e., more than three tokens). In particular, this is not a special case but holds over different programs in our study. The result indicates that the reusable code elements indeed exist at a fine-grained granularity, which will significantly limit the effectiveness of existing redundancy-based APR approaches. Moreover, it also reflects that designing APR approaches by reusing finer-grained code elements is promising.

However, achieving accurate and fine-grained code reuse for patch generation is challenging. The reasons are twofold: (1) It will significantly enlarge the search space of candidate patches, making it harder to efficiently and correctly find the desired code elements. (2) Due to weak test suites [46]–[48], plausible (i.e., can pass all the test cases) but incorrect patches are more likely to be generated, increasing the manual effort required for validation and significantly affect the practical usability of APR approaches. To overcome these challenges, we propose an effective APR approach that efficiently locates the desired code elements for patch generation leveraging the *locality property* of source code as its core idea. The details will be introduced in Section III.

B. Running Example

In this section, we use real-world bug examples to illustrate how the *locality property* of source code can guide patch generation. Listing 1 presents the patch code of Codec-3 from the Defects4J [41], where a line starting with "+" denotes newly added code while starting with "-" denotes deleted code.

```
452     if ((contains(value, 0 ,4, "VAN ", "VON ") || ...)){
453         //-- obvious germanic --//;
454         result.append('K');
455 -   } else if(contains(value, index + 1, 4, "IER")) {
    +   } else if(contains(value, index + 1, 3, "IER")) {
456         result.append('J');
457     } else {...
```

Listing 1. Patch code of Codec-3 in Defects4J

In this example, the constant parameter "4" was mistakenly used at line 455, and the desired parameter is "3". This bug cannot be fixed by existing redundancy-based APR techniques due to either the large search space (e.g., GenProg [6], [17], RSRepair [16], etc.) or the nonexistence of multi-line similar code snippets for reference (e.g., SimFix [8], Transplant-Fix [33], etc.). However, when it comes to the fine-grained token level, we will find that the constant number "3" usually co-appears with the tokens "value" and "index" elsewhere in the program. These tokens exhibit a strong correlation (i.e., the *locality property* of source code). Leveraging this property makes it both feasible and efficient to locate the correct code elements for patch generation.

```
    public Date read(JsonReader in) throws IOException {
-     if (in.peek() != JsonToken.STRING) {
-         throw new JsonParseException("The date ...");
+     if (in.peek() == JsonToken.NULL) {
+         in.nextNull();
+         return null;
      }
      Date date = deserializeToDate(in.nextString()); ...
```

Listing 2. Patch code of Gson-17 in Defects4J

```
if (in.peek() == JsonToken.NULL) {
    in.nextNull();
    return null;
}
in.beginArray();
```

Listing 3. The reference code for repairing Gson-17

In fact, the *locality property* of source code may not only exist at the token level but may also exist at higher levels, e.g., expression or statement levels. For example, Listing 2 presents the patch code of Gson-17 from the Defects4J benchmark, while Listing 3 presents a reference code that can be used for patch generation. According to the reference code, the expression "`in.peek()`" may be correlated with the expressions "`JsonToken.NULL`", "`in.nextNull()`" and "`return null`". It can be seen that, although the reference code can provide the required code elements for generating the desired patch in these examples, how to utilize them is still challenging since the possible combinations of code changes according to the reference code are still too many, such as replacing the `throw` statement with "`in.nextNull()`" or inserting "`in.nextNull`". In addition, replacing the conditional expression also does not fix the bug due to the incorrect operator "`!=`". In particular, the code structures of reference code and the faulty code may also be different in practice, making patch generation harder. To address this challenge, we propose a novel code representation method that decomposes complex expressions into a *unified* simple form, which ignores operators and complex code structures, e.g., `!=`, `for`, `while`, etc., but still preserves the *locality property* of source code.

III. Framework

In this section, we introduce the details of our approach (named REPATT). Figure 2 presents the overview of REPATT. In general, REPATT generates candidate patches by leveraging the *plastic surgery hypothesis* and the *locality property* of source code, utilizing both token-level pattern mining and expression-level code search to handle different granularities of code changes. Token-level pattern mining, an *offline* process, constructs a query-efficient code pattern database for repairing single-line bugs (*ref.* Listing 1). In contrast, expression-level code search, an *online* process, identifies reference code elements for multi-line fixes (*ref.* Listing 2). Their differing strategies stem from (1) token-level mining focusing on small-scale patterns within single lines, while expression-level patterns span multiple lines, significantly expanding the search space, and (2) token-level patterns often introducing numerous small changes, impacting efficiency, while the expression-level

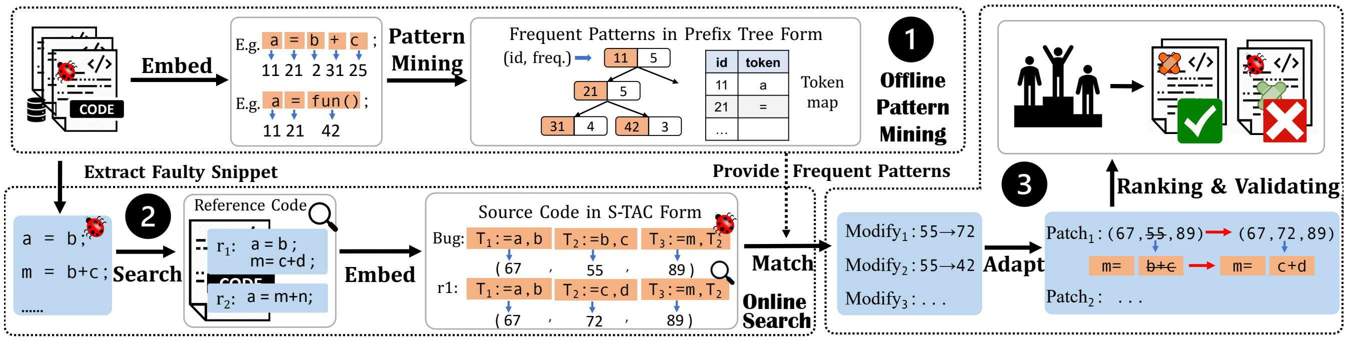


Fig. 2. Overview of our approach REPATT. Given a buggy project, ❶ REPATT first mines a set of token sequence patterns and represents them as prefix trees. ❷ Then, based on the faulty code snippet, REPATT online searches a set of similar code snippets and transforms them into S-TAC form as reference patterns. ❸ Finally, REPATT generates candidate patches by matching the faulty code with the reference patterns and validates them one by one.

patterns correlated to a certain location (i.e., the faulty code) will be very limited (refer to Figure 1). Therefore, we confine the token-level patterns to those that are frequent to balance efficiency.

A. *Offline Pattern Mining*

Based on our preliminary study (Section II-A), most reusable code elements for patch generation exist at a fine-grained level, typically one or two tokens. To leverage these elements, REPATT employs a novel token-level pattern mining algorithm guided by the *locality property* of source code. This process is conducted offline over the given faulty project, constructing a database of token usage patterns to support efficient online patch generation. As aforementioned, we only consider the token patterns within single lines to confine the search space and avoid involving too much noise. In particular, it is common that the tokens used in different code lines share both commonalities and diversities. For example, the token sequence in some code lines can be ("a", "b", "c"), while it may also be ("a", "d", "c") in some other code lines. To preserve the semantics of programs, we keep the token orders and design a *skip-fashion* pattern mining algorithm to address the problem of diversity among code lines. Additionally, each token is assigned a unique ID for efficient comparison, while separators (e.g., "," and "(") and structural keywords (e.g., "if" and "for") are removed as they do not contribute to token-level patch generation.

Specifically, to improve the pattern mining process, we design a data structure – *Prefix Embedding Tree*, which is defined as follows.

Definition 1: **(Prefix Embedding Tree)**: a prefix embedding tree constitutes a set of nodes, each of which is a tuple of $t = \langle tok, id, p, C, sup\rangle$, where id denotes a unique embedding of a certain token tok, p denotes the parent node of t, C denotes a set of child nodes of t, while sup represents the frequency of the token sequence from the root to the current node t.

Furthermore, we use $t.childNode(id)$ to obtain the child node of t with the embedding id, and use $t.setChildNode(id, chd)$ to set the node chd with embedding id as the child node of t. In addition, we use *findOrCreate(trees, tok)* to find the tree rooted at the token *tok* in *trees*. According to these notions, we present the pattern mining process in Algorithm 1. In general, given the faulty program, REPATT first decomposes all the code lines in the program into a set of token sequences (i.e., *ES*s), where each token sequence corresponds to one line of code. Then, it constructs the prefix embedding trees (i.e., *trees*), which stores the token patterns with corresponding usage frequencies.

Algorithm 1: Tree Building and Pattern Construction

```
   Input: ESs: sequences of code element embeddings (IDs)
   Output: trees: frequent pattern in the form of prefix tree.
1  Function build(ESs):
2      trees ← ∅
3      foreach S in ESs do
4          updated ← ∅ // Avoid counting duplicated tokens
5          buildTree(S, trees, updated) // Build trees
6      return trees // Embedding trees in programs
7  Function buildTree(seq, trees, updated):
8      start ← 0
9      while start < len(seq) do
10         root ← findOrCreate(trees, seq[start]) // Find node
11         root.sup ← root.sup + 1 // Increase the frequency
12         start ← start + 1
13         traverse(root, seq[start:], 0, 1, updated)
14         trees ← trees ∪ {root}
15 Function traverse(tree, seq, skip, length, updated):
16     if length ≥ MAX_LEN or len(seq) == 0 then
17         return
18     else if skip < MAX_SKIP then
19         traverse(tree, seq[1:], skip+1, length, updated)
20     node ← tree.childNode(seq[0]) // Not skip the token
21     if node is None then
22         node ← create a new node of id seq[0]
23         tree.setChildNode(seq[0], node)
24     if node is not in updated then
25         node.sup ← node.sup + 1 // Increase frequency
26         updated ← updated ∪ {node}
27     traverse(node, seq[1:], skip, length + 1, updated)
```

Specifically, for each token sequence $S \in ESs$ (line 3), REPATT either creates new trees using it or expands existing trees by updating pattern frequencies via *buildTree(*)* (Line 5). In this process, REPATT maintains a set *updated* to record the nodes whose frequencies have been calculated for avoiding duplicated counting. More specifically, given the token sequence *seq* and *trees* that have already constructed, REPATT performs a top-down tree building process, iterating over sub-sequences starting at different positions in *seq* (Line

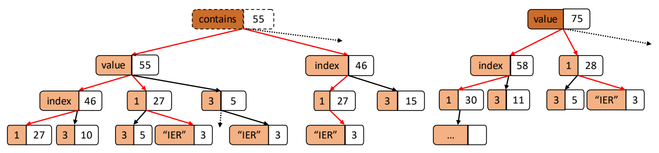


Fig. 3. Partial prefix embedding trees. We use concrete tokens to represent the token ids for ease of understanding.

9). That is, for each token sub-sequence starting at *start*, REPATT finds the tree (or creates a new one) whose root node is associated with the token *seq[start]* (Line 10) and updates its frequency (Line 11). The tree is then recursively expanded via *traverse()* using a *skip-fashion* pattern mining strategy, where a token can either be included (Lines 20-27) or skipped (Lines 18-19), constrained by MAX_SKIP (Line 18). For each token, REPATT locates an existing node (Line 20) or creates a new one if absent (Lines 21-23), updating its frequency as needed (Lines 24-26). A pattern is finalized upon reaching the maximum sequence length MAX_LEN. Our evaluation further examines the impact of MAX_LEN and MAX_SKIP on REPATT's performance.

Taking the faulty code Codec-3 (Listing 1) as an example, the token sequence of the faulty code line will be S=("contains", "value", "index", "+", "1", "4", "IER"), based on which the constructed prefix embedding trees are presented in Figure 3, where we use the red line to highlight the patterns related to the token sequence S. Consequently, the token sequence from the root node to the leaf node denotes a possible pattern and the number denotes its frequency. For example, the frequency of the pattern ("contains", "value", "1", "IER") is 3. In particular, each distinct token will be the root node of an individual prefix embedding tree. For instance, we present two prefix embedding trees in Figure 3 that are related to the token sequence S. As a result, when given a buggy sequence, we can only search a very limited number of embedding trees whose root tokens are included in the sequence, which can promote the searching process. During pattern mining, patterns with a frequency below a predefined threshold MIN_SUPPORT are discarded. A higher frequency suggests that a pattern is more common and thus more likely to be reusable for repair. In the repair phase, given a faulty code token sequence S' , REPATT searches for all matching patterns in the embedding trees and generates patches based on these references.

B. Online Code Search and Representation

The expression-level code search aims to find code usage patterns across multi-lines of code for reference, which may cause a large search space if it is also considered like the token-level pattern mining since there may be hundreds or thousands of lines of code even in a single method. As a result, the expression-level pattern mining process is designed as an online code search process, which automatically identifies a set of reference code when given the faulty code. Following existing work [8], REPATT extracts no more than three lines of code respectively before and after the given faulty line as the *faulty snippet*, and then finds the N most similar snippets via measuring the code similarity between the faulty and reference snippets. Specifically, REPATT extracts a vector from each snippet, where each element in the vector represents a feature and the feature value represents the number of corresponding AST node types in the snippet by following the study [8]. Then, we calculate the cosine similarity [49] between two vectors for code ranking.

To adapt the reference code to the buggy code that many have complex contexts in real practice, we propose a new code representation method, named "Simplified Three-Address Code" (*abbr.* S-TAC). It unifies the code representations by ignoring the code structures under different contexts but still preserves the *locality property* of source code.

Definition 2: **(Simplified Three-Address Code)**: an S-TAC is a triple of $\langle T, t_1, t_2 \rangle$, where T is an intermediate symbol to represent the expression correlated to t_1 and t_2, while t_1 and t_2 are two expressions that are either the intermediate symbols or simple items of variables, literal values, types, and function calls. In particular, t_1 and t_2 can be null if they are keywords, e.g., `break` and `return`.

Additionally, we use $T := t_1, t_2$ to represent $\langle T, t_1, t_2 \rangle$ when there is no ambiguity. Compared with traditional Three-Address Code (TAC) [50], S-TAC ignores both the coarse-grained code structure information, e.g., `if`, and the fine-grained operators, e.g., "+". The reason for ignoring the structure information is to make the source code from different contexts applicable for patch generation. For example, the conditional expression $a > b$ in `if` statements can match that either in `for` statements or in trinary conditional expressions, enabling a more flexible code match. Similarly, the fine-grained operators may also be specific to certain contexts and thus affect the matching and reuse of code. In other words, our S-TAC representation can better reflect the *locality property* of source code while reducing the noise induced by different contexts.

Taking the code shown in Listing 2 and 3 as examples, the S-TAC form for the if condition from the faulty code is $T_1 := in.peek()$, $T_2 := T_1, JsonToken.STRING$, while it will be $T_1 := in.peek()$, $T_2 := T_1, JsonToken.NULL$ for the reference code, where T_1 and T_2 are intermediate symbols. In this way, the two conditions can be better matched with each other. We will also evaluate the contribution of our S-TAC form in the experiment (Section V).

C. Reference Code Adaptation

When given the faulty code, we can obtain a set of reference code based on the previous two processes. Then for each reference code pattern, REPATT tries to generate candidate patches by matching the faulty code to the reference code. Algorithm 2 presents the details of the matching process. In general, REPATT performs a greedy match between the faulty code and the reference code. Then, it generates patches according to the difference between them. In other words, given the token (or S-TAC) sequences for both faulty (i.e., *BS*) and reference code (i.e., *RS*), the matching algorithm will return a set of pairs (i.e., *PS*), based on which REPATT generates candidate patches according to a set of predefined rules.

Given the token (or S-TAC) sequences of both faulty (*BS*) and reference code (*RS*), REPATT first computes their longest common sequence (*cs*, line 3) for greedy matching. It then establishes mappings between unmatched elements in the faulty and reference code based on these results (lines 5-13). Using *S.getNodes(a, b)*, REPATT retrieves elements between positions a and b in sequence S. As illustrated in Figure 4, *cs* for the S-TACs in Listings 2 and 3 is $[(b1, p1), (b5, p5)]$. From the figure we can observe that the S-TAC representation abstracts away concrete code content, mitigating its impact on pattern matching. The unmatched elements are divided into two lists, *os* ($[b2, b3, b4]$) and *ts* ($[p2, p3, p4]$), whose Cartesian product (i.e., $\bowtie$) forms all possible unmatched pairs (line 8). In particular, if no sibling AST nodes of a faulty code element are matched (line 21), REPATT attempts to map its parent nodes via *tryToMatchParent(*)* (lines 21-25). For instance, since the `throw` statement ($b4$) has no matched siblings, REPATT maps the faulty `if` statement to its reference counterpart.

Algorithm 2: Reference Code Matching

```
   Input: BS: a sequence of tokens or S-TACs that represent the bug.
          RS: a sequence of tokens or S-TACs that represent the
          reference code.
   Output: PS: a set of original and target node pairs.
1  Function matchElement(BS, RS):
2      PS ← [], bf ← −1, rf ← −1
3      cs ← LCS(BS, RS) // The longest common sequence
4      if cs.size() > 0 then
5          foreach c in cs do
6              os ← BS.getNodes(bf, BS.indexOf(c.fst))
7              ts ← RS.getNodes(rf, RS.indexOf(c.snd))
8              PS.append(os⋈ts) // All unmatched pairs
9              bf ← BS.indexOf(c.fst)
10             rf ← RS.indexOf(c.snd)
11         os ← BS.getNodes(bf, BS.length)
12         ts ← RS.getNodes(rf, RS.length)
13         PS.append(os⋈ts) // Unmatched pairs at the end
14     PS ← PS ∪ tryToMatchParent(PS)
15     return PS
16 Function tryToMatchParent(PS):
17     result ← []
18     foreach ⟨a, b⟩ in PS do
19         parent ← a.getParent()
20         children ← parent.getChildren()
21         if PS.FirstSet().containsAll(children) then
               // All sibling AST nodes are not matched
22             ts ← []
23             foreach c in children do
24                 ts.append(PS.getSecond(c).getParent())
25             result.append([parent] ⋈ ts)
               // Add mappings of parent AST nodes
26     return result
```

Finally, according to the constructed mapping (i.e., *PS*), REPATT will generate candidate patches from each pair $(a, b) \in PS$ according to the following rules (we use a to represent the associated AST node in what follows for ease of presentation): (1) replace a with b if the value type of b is compatible with that of a; (2) insert b before and after a if b is a standalone statement, e.g, expression statement; (3) insert b before a if b is a conditional expression. Both token-level and expression-level patch generations follow this mapping algorithm, based on the types of a and b (e.g., replacing variable a with another variable b generates a token-level patch). In particular, we disable delete operations for patch generation since they tend to generate incorrect patches [5], [8]. Moreover, during this process, REPATT will automatically check the validity of the new code b via static analysis for efficiency, such as whether all variables used by b are valid and usable in the faulty location by checking their scopes.

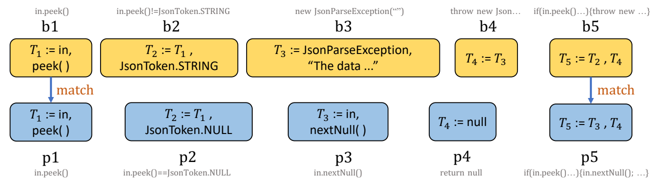

in.peek()
b1
T_1 := in, peek( )
in.peek()!=JsonToken.STRING
b2
$T_2 := T_1$, JsonToken.STRING
new JsonParseException("")
b3
T_3 := JsonParseException, "The data ..."
throw new Json...
b4
$T_4 := T_3$
if(in.peek()...){throw new ...}
b5
$T_5 := T_2 , T_4$
match
match
T_1 := in, peek( )
p1
in.peek()
$T_2 := T_1$, JsonToken.NULL
p2
in.peek()==JsonToken.NULL
T_3 := in, nextNull( )
p3
in.nextNull()
T_4 := null
p4
return null
$T_5 := T_3 , T_4$
p5
if(in.peek()...){in.nextNull(); ...}

Fig. 4. Matching results of partial S-TACs in Gson-17 shown in Listing 2. b_i represents the S-TAC form of the faulty `if` statement while the p_i corresponds to the `if` statement in the reference code. In particular, we also present the source code corresponding to each T-SAC.

D. Patch Ranking

After generating candidate patches, they will be evaluated against the test suites associated with the faulty project. To make the correct patches be evaluated as early as possible and reduce incorrect patches, we have proposed a hierarchical patch ranking strategy. As demonstrated in our preliminary study, most of the reusable code elements exist at the fine-grained granularity. Furthermore, previous study [51] also shows that correct patches tend to involve small code changes. Therefore, REPATT ranks all the patches generated by the finer-grained token-level code changes higher than those generated by the coarse-grained expression-level code changes.

Then, regarding the patches generated by expression-level code changes, REPATT simply takes the similarity in code search (refer to Section III-B) as the ranking score by following previous studies [8]. For patches generated by token-level code changes, we consider two important factors: the frequency of the reference code pattern and the similarity between the faulty and fixed code by following existing studies [8], [51]. The ranking score is defined by Formula 1,where $freq$ denotes the frequency of a pattern and max_freq denotes the max $freq$ of patterns referenced for generating patches. L_{orig} and L_{fixed} represent the length of source code (i.e., number of tokens) before and after applying a patch, while *LevenshteinDist* represents the token-level Levenshtein distance [52] between the faulty and the fixed code. As a result, a patch with a higher score (i.e., referencing a more frequent pattern and applying finer-grained modifications) will rank higher and thus will be validated earlier by running the test cases in the faulty project.

$$score = 0.5 * \frac{freq}{max\_freq} + 0.5 * \left(1 - \frac{LevenshteinDist}{max(L_{orig}, L_{fixed})}\right) \quad (1)$$

IV. EXPERIMENT CONFIGURATION

We address the following research questions in the study.

- **RQ1:** How effective is REPATT in repairing real-world bugs?

- **RQ2:** Can our approach improve the state-of-the-art APRs?
- **RQ3:** What is each component's contribution in REPATT?
- **RQ4:** What is the impact of the pattern frequency in REPATT?

A. *Subjects and Baselines*

In our evaluation, we employed the widely-used Defects4J benchmark [41], following existing studies [2]–[4], [8], [9], [18], [20], [33]. In particular, we used both Defects4J v1.2 and v2.0 to demonstrate the generality of our approach. Specifically, v1.2 includes 395 bugs from six large-scale real-world projects and v2.0 includes 440 additional bugs from 12 real-world projects.

Furthermore, to show the effectiveness of REPATT, we compared it with nine state-of-the-art APR approaches from different categories, i.e., SimFix [8], TransplantFix [33], TBar [4], Recoder [18], SelfAPR [19], ITER [53], AlphaRepair [22], Repilot [23] and GAMMA [24]. Specifically, SimFix and TransplantFix are the two latest and representative **redundancy-based** methods that repair bugs by referencing similar code. TBar is the best-performing **template-based** repair technique that generates candidate patches by a set of manually defined patch templates. Recoder, ITER and SelfAPR are the latest and best-performing **deep learning methods** that are specially designed for program repair. Finally, AlphaRepair, Repilot, and GAMMA represent the most recent advance in APR techniques by **adopting LLMs**. These baselines are the best-performing APR approaches using different technologies and their complete experimental results are available. By comparing our approach with these diverse baselines, we would like to analyze the overall effectiveness of our approach from different perspectives and make the conclusions reliable.

B. *Configuration and Metrics*

For each bug, REPATT first constructs the token-level code patterns based on the complete faulty program under repair before the online repair (see Section III-A). The construction is actually efficient and on average took about three minutes in our experiment. Then, given the faulty line of code, REPATT first generates at most 200 patches based on the constructed token-level code patterns and then generates at most 1000 patches based on the expression-level code patterns due to its large search space. Finally, all the patches will be ranked based on the ranking strategy introduced in Section III-D. For baselines, we adopt their published experimental results directly in their open-source repositories.

In our experiment, REPATT generates at most 3 candidate patches that can pass all the test cases (i.e., plausible patches) for each bug since some baseline approaches did not report the ranking of patches, e.g., SelfAPR and LLM-based methods. Following previous study [4], [8], [9], [18], a patch is deemed to be correct *iff* it is semantically equivalent to the developer patch by manual check (We also published all our results for further check). Finally, we report the number of correctly repaired bugs (equivalent to the well-known **recall**) and the **precision** of patches (the ratio of correctly repaired bugs to all the bugs that have plausible patches).

All experiments were conducted on a server with Ubuntu 18.04, equipped with 128GB RAM and a processor of Intel(R) Xeon(R) E5-2640.

V. RESULT ANALYSIS

A. *Overall Effectiveness of* REPATT *(RQ1)*

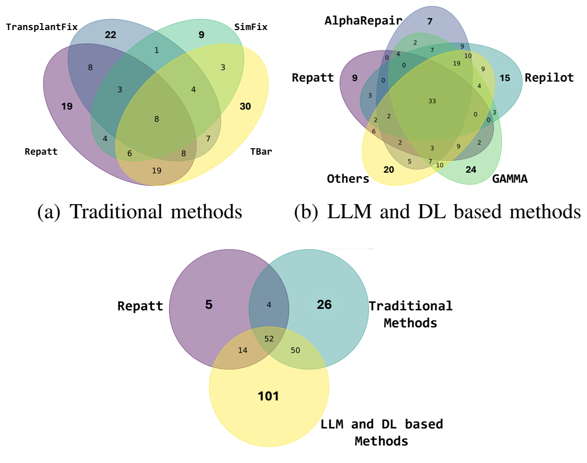
1) Perfect fault localization: As explained in Section IV-A, we evaluated the effectiveness of our approach by comparing it with eight state-of-the-art APR approaches since ITER was only evaluated under the automated fault localization setting (*ref.* Section V-A3). The repair results are presented in Table I when given the actual faulty locations by following existing studies [4], [18]–[20], [33]. From the table we can see that REPATT successfully repaired 75 bugs while generating incorrect patches for other 63 bugs, yielding a patch precision of 54.3%. **Notably, 66 bugs were correctly repaired by the first plausible patch**, suggesting that developers could focus primarily on the first patch generated by REPATT to reduce manual validation efforts. The results also show that our approach outperforms the state-of-the-art redundancy-based methods SimFix and TransplantFix by repairing 97.4% and 23.0% more bugs, indicating the effectiveness of our approach in reusing code at different granularities for patch generation. Compared to SelfAPR, which repaired 108 bugs when considering Top-50 generated patches but only 34 when restricted to Top-1, REPATT shows a more favorable trade-off between precision and recall.

Similarly, LLM-based repair methods also did not report the ranking of their correct patches. Specifically, AlphaRepair [22] and Repilot [23] generated up to 5,000 patches per bug, while GAMMA [24] even did not limit the number of candidate patches. Although these methods may achieve a higher number of correct fixes, they require developers to spend significant time manually reviewing patches, reducing their practicality [46], [48]. In contrast, REPATT enforces strict limits on the number of generated patches, effectively mitigating this issue. Furthermore, we analyzed REPATT's performance on repairing multi-location bugs. Among 75 correctly repaired bugs, 30 involved multiple locations, highlighting the effectiveness of REPATT's fine-grained repair strategy.

2) Degree of complementary: We also analyzed the overlaps of bugs repaired by both REPATT and the baselines. Specifically, we classified the baselines into traditional, LLM-based and deep-learning-based methods according to the patch generation techniques, and then compared REPATT with these two methods separately. Figure 5 presents the results. In summary, REPATT successfully repaired 19 unique bugs that the traditional APRs cannot fix, and 9 unique bugs that the LLM-basd and deep learning-based methods cannot fix. In particular, when compared with all the eight baselines, our approach still can repair 5 unique bugs. For example, to repair the bug shown in Listing 1, the constant value "4" should be replaced by "3". Without the guidance of the fine-grained reference code, it is nearly impossible for existing APRs to get the correct

TABLE I
NUMBER OF BUGS REPAIRED BY DIFFERENT METHODS WITH PERFECT FAULT LOCALIZATION. IN THE TABLE, X/Y DENOTES THE CORRESPONDING APPROACH GENERATES CORRECT PATCHES FOR X BUGS AND GENERATES PLAUSIBLE PATCHES FOR Y BUGS.

	Project	**TBar**	**SimFix**	**TransplantFix**	**SelfAPR**	**Recoder**	**AlphaRepair**	**Repilot**	**GAMMA**	**REPATT**	**COMBINE**
Defects4J v1.2 (395 bugs)	Mockito	3/3	0/0	3/3	3/3	2/-	4/-	0/-	2/-	**1/2**	**2/5**
	Closure	16/24	5/6	10/19	19/23	23/-	22/-	21/-	23/-	**10/12**	**20/31**
	Chart	11/13	4/8	6/10	7/10	10/-	8/-	5/-	10/-	**6/7**	**11/21**
	Lang	13/18	8/13	4/10	10/13	10/-	12/-	14/-	15/-	**6/11**	**16/27**
	Math	22/35	14/26	12/25	21/25	18/-	20/-	20/-	24/-	**15/25**	**22/51**
	Time	3/6	1/1	1/2	3/3	3/-	2/-	1/-	2/-	**2/3**	**4/7**
Defects4J v2.0 (440 bugs)	Closure	0/0	1/1	0/3	1/1	0/-	0/-	0/-	0/-	**1/1**	**2/5**
	Cli	1/7	0/1	4/8	8/9	3/-	5/-	6/-	9/-	**4/9**	**4/15**
	Codec	3/6	1/1	2/4	8/9	2/-	6/-	6/-	3/-	**4/6**	**3/8**
	Collections	0/0	0/0	0/0	1/1	0/-	0/-	1/-	0/-	**0/0**	**0/0**
	Compress	2/12	0/4	4/10	6/8	3/-	1/-	3/-	4/-	**4/13**	**7/18**
	Csv	2/6	0/2	1/2	1/1	3/-	1/-	3/-	0/-	**1/3**	**2/7**
	Gson	1/4	2/2	1/2	1/1	0/-	2/-	1/-	3/-	**2/3**	**2/5**
	JacksonCore	0/5	0/2	1/6	3/3	0/-	3/-	3/-	3/-	**3/7**	**3/9**
	JacksonDatabind	2/17	1/10	7/20	8/10	0/-	8/-	8/-	10/-	**10/18**	**11/33**
	JacksonXml	0/1	0/0	0/0	1/1	0/-	0/-	0/-	0/-	**0/0**	**1/1**
	Jsoup	6/17	1/2	3/8	6/8	7/-	9/-	18/-	11/-	**5/13**	**10/25**
	JxPath	1/6	0/0	2/7	1/1	0/-	1/-	1/-	2/-	**1/5**	**4/12**
Total		85/180	38/79	61/139	108/130	84/-	104/-	111/-	121/-	**75/138**	**124/280**
Precision(%)		47.2	48.1	43.9	83.1	-	-	-	-	**54.3**	**44.3**



(a) Traditional methods (b) LLM and DL based methods

(c) All methods

Fig. 5. Overlaps of bugs repaired by different approaches.

patch due to the large search space. In general, compared with traditional APRs, REPATT can more effectively utilize fine-grained reusable code elements for patch generation, while compared with LLM-based and deep learning-based methods, REPATT only depends on a small number of reusable samples rather than the large-scale training data, which enables REPATT to repair infrequent bugs, especially those requiring domain-specific knowledge. In conclusion, the results reflect that our approach indeed complements existing approaches.

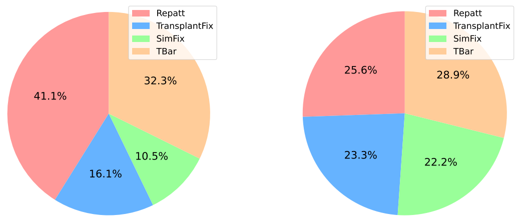
3) Automated fault localization: In addition, some of the baseline approaches [4], [8], [18], [33], [53] were also evaluated in a more realistic scenario where the perfect fault localization was unknown. To ensure a fair comparison, we evaluated our approach under the same conditions. Specifically, we utilized the commonly-used Ochiai [54] algorithm, implemented by the GZoltar toolset [55], to obtain a list of candidate faulty locations like existing APR tools [4], [8], [18]. Table II summarizes the experimental results. From the table, we can see that the number of correctly repaired bugs drops sharply for all the repair approaches due to the inaccurate fault localization results. Moreover, almost all baseline approaches experienced a significant decline in patch precision. For instance, the precision of SimFix, TBar and TransplantFix drops 7.8%, 19.1%, and 26.9%, respectively. In contrast, the precision of our approach was improved by 54.3%. One of the reasons is that fine-grained code changes focus on fixing local code without making extensive changes to the original content, thereby preserving program logic and maintaining the effectiveness of test cases, leading to more correct patches. In contrast, coarse-grained modifications often involve larger code changes, which may unintentionally alter program functionality, resulting in many patches that pass the tests but are semantically incorrect due to the problem of weak tests [46]–[48]. In summary, our approach significantly outperforms the baselines by achieving 15.6%-51.7% higher patch precision. Notice that the high precision of patches is very important since it affects the usability of APR techniques [46], [48], especially in the real-world repair scenarios where perfect fault localization is unknown – high precision denotes less wasted human effort for manual validation.

TABLE II
REPAIR RESULT WHEN USING SBFL (#CORRECT/#PLAUSIBLE).

	SimFix	**TBar**	**TransplantFix**	**Recoder**	**ITER**	**REPATT**	**COMBINE**
Total	36/81	50/131	44/137	70/140	45/66	**31/37**	**90/236**
Precision (%)	44.4	38.2	32.1	50.0	68.2	**83.8**	**38.1**

B. Improvement over SOTA APRs (RQ2)

As presented in Section V-A2, REPATT complements existing APR approaches. In this research question, we aim to explore the possibility of our approach to improving existing APRs. Specifically, we combine REPATT with existing APRs by proposing a post patch ranking strategy and see whether it can further improve the best-performing APR. Given the patches generated by different APRs for a bug, we will rank



(a) Using perfect FL (b) Using SBFL

Fig. 6. Source of correct patches in COMBINE

the ones with smaller code changes higher by following the insights from both our study and the previous research [51]. In particular, we use GumTree [56], a fine-grained and mature code differencing tool, to measure the change sizes of patches. If two patches have the same number of code changes, we use the patch precision to break the tie. In this study, we combined our approach with the three traditional APR approaches, i.e., Simfix, TBar, and TransplantFix. In this way, the combined method shares a similar repair pipeline to the individuals, i.e., not demanding large-scale training data. Therefore, the patch ranking will be REPATT>SimFix>TBar>TransplantFix based on their patch precision in Tables I and II for patches having the same change size.

The experimental results are also presented in Tables I and II (i.e., **COMBINE**) when using the perfect fault localization and the SBFL, respectively. The results show that by combining the strength of different APR tools, COMBINE correctly repaired 124 bugs by the **first patches**, 39 more bugs compared with the best-performing APR (TBar for Top-1 patches), leaving the improvement as 45.9%. Moreover, even compared to LLM-based methods, which do not report detailed patch rankings and impose loose constraints on the number of generated patches (e.g., producing 5,000 or more patches per bug), COMBINE still outperforms these 3 LLM-based approaches, repairing up to 20 more bugs. Similarly, when using the SBFL results, COMBINE also outperforms the best-performing Recoder by repairing 20 more bugs, and improves the individuals by at least 80% (*vs* Tbar). Our results show the promise to improve the repair capability of APRs by combining the strength of individuals. In particular, to analyze the contribution of our approach in the combination, we present the percentages of correct patches generated by each APR in Figure 6. About 41.1% and 25.6% correct patches were contributed by REPATT under different repair settings, indicating its large contribution to the overall effectiveness of the combined method. However, though the combined method is effective in repairing much more bugs, the patch precision is still low, i.e., less than 45%. To improve the patch precision, existing patch filtering approaches [5], [47], [48] can be further incorporated.

C. Contribution of Each Component (RQ3)

REPATT incorporates two major components for patch generation, i.e., offline token-level pattern mining and online expression-level code search. To evaluate the effectiveness of each component, we have repeated our experiment by removing each component, respectively. Table III presents the experimental results. The offline token-level pattern mining contributed 42 correct fixes and expression-level code search contributed 33 correct fixes, demonstrating that both components largely contributed to the overall effectiveness of REPATT. However, from the table, we can also observe that the expression-level code search tends to generate more plausible but incorrect patches compared with the token-level pattern mining, which is consistent with the conclusions in prior studies [51]. Our further analysis of the results shows that most of incorrect patches are generated due to referencing infrequent code. In contrast, token-level pattern mining takes the frequency into consideration, which potentially can effectively filter out many incorrect patches. In summary, the two components in REPATT complement each other.

TABLE III
BUGS FIXED BY EACH COMPONENT (#CORRECT/#PLAUSIBLE).

Components	Offline Mining	Online Search	No Skip	No S-TAC
Total	42/64	33/85	24/54	0/22
Precision (%)	65.6	38.8	44.4	0

In addition, as introduced in Section III, our token-level pattern mining process incorporates a *skip-fashion* pattern mining process, which can find more potential reference patterns for patch generation. Therefore, we further conducted an experiment, where we only use the token-level repair but disable the *skip-fashion* in pattern mining (i.e., MAX_SKIP=0 in Algorithm 1). The column of "No Skip" in Table III presents the results. It shows that this *skip-fashion* mining process is effective since its removal caused 42-24=18 fewer bugs to be repaired. Moreover, the patch precision is also decreased from 65.6% to 44.4%, demonstrating the value of this process in REPATT. Additionally, we further evaluated the performance of the S-TAC form in our expression-level repair. Specifically, we did not transform the reference code into S-TAC. Instead, they will be decomposed into the traditional TAC format without removing the code structures based on the abstract syntax tree. The experimental results are also presented in Table III (i.e., "No S-TAC"). After removing the S-TAC form, REPATT failed to repair any bugs by searching reference code online. We further analyzed the results and found that the contexts of most reference code from online search are different from the faulty code, making the code adaptation fail to generate correct patches. On the contrary, 22 incorrect patches were generated. In summary, the experimental results demonstrate that both the *skip-fashion* online pattern mining and the S-TAC form in the online search process are effective.

D. The Impact of Pattern Frequency (RQ4)

As introduced in Section IV-B, the threshold of pattern frequency is 3 by default. To investigate its impact on the effectiveness of our approach, we have conducted an additional experiment when using different frequency settings for

MIN_SUPPORT, i.e., 2, 3, and 5. As a consequence, when adopting different settings, the performance of REPATT was not largely affected. Specifically, REPATT correctly repaired 43/91, 42/64, and 32/68 bugs by the token-level patterns when using the threshold as 2, 3, and 5, respectively. In other words, our approach is not very sensitive regarding the number of correct patches when the threshold is small (i.e., 2 or 3). However, when the threshold is too large (such as 5), REPATT tends to repair much fewer bugs. The reason is evident as the large threshold will cause fewer patterns that can be used for patch generation. On the contrary, a smaller threshold (i.e., 2) tends to produce much more incorrect patches (i.e., 91-43=48) since the referenced patterns may be not general enough for reuse, which will affect the usability of REPATT. In summary, the recall and precision of program repair indeed contradict each other. Effective patch generation techniques are always in an urgent need. In this paper, we have moved forward towards this direction. In general, setting the threshold as 3 can produce relatively better results.

VI. DISCUSSION

Limitation: In this study, we have explored the feasibility of reusing finer-grained code elements for patch generation. For complex bugs that require multi-line changes, the reference code may not exist, where our approach will be less effective. In this case, our approach may potentially be combined with others since they complement each other according to our experimental results.

Threats: Like existing studies, we face the *internal* threat of manually verifying plausible patches. To mitigate this, we rigorously compared them with developer patches for semantic equivalence and publicly released all generated patches for further check. The threats to *external* validity mainly lie in the subjects used in our study. Since the Defects4J benchmark has been widely used by existing studies and contains more than 800 real-world bugs from 17 diverse projects, we believe the results can be reliable. However, the effectiveness of our approach on a broader range remains to be investigated.

VII. RELATED WORK

There are many automated program approaches have been proposed and achieved good results [2], [3], [6], [8], [10]–[12], [20], [21], [51], [57]–[61], among which redundancy-based APR approaches have been attracted much attention, such as GenProg [6], [17], AE [58], [62], RSRepair [16], ARJA [62], ssFix [9], SimFix [8], CapGen [1], SearchRpepair [63], HERCULES [64] and so on. All these approaches are based on the "plastic surgery hypothesis" [29]. The major difference among them lies in the adopted patch generation strategies. For example, GenProg reuses existing code randomly, ssFix searches similar code as references from a large codebase, while SimFix further incorporates history patches to refine the patch space. Similarly, CapGen considers the frequency of tokens for patch generation while HERCULES leverages the co-evolution of program elements. Our approach enhances redundancy-based APR by enabling finer-grained code reuse, differentiating it from existing methods. Unlike SimFix and ssFix, which rely on AST differences, our method operates at the token level, capturing more granular code patterns. Compared to CapGen, which considers token frequency without their contextual relationships, our approach mines structured code patterns for patch generation. Additionally, unlike SearchRepair, we avoid costly semantic code searches and uniquely support nonconsecutive token usage patterns, which have been shown to be effective but are not utilized by existing methods. Other APR techniques, such as PAR [14] and TBar [4], improve patch quality through predefined repair templates, while constraint-solving-based techniques [3], [10], [51], [60], [65]–[68] rely on symbolic execution and constraint solving, often facing scalability challenges. Different from these approaches, this work aims to improve the performance of redundancy-based APR techniques and does not face the scalability issue.

With the rapid development of deep learning techniques, the latest APR approaches also leverage such techniques for patch generation. Early studies use statistical machine learning algorithms to sort or pick patch ingredients. For example, Prophet [7], ACS [2], Elixir [15], Hanabi [69], and LIANA [70] all use different learning models for patch ingredient selection. More recently, state-of-the-art deep learning techniques have been employed in APR techniques, such as SEQUENCER [44], CoCoNut [71], CURE [21], DLFix [45], Recoder [18], RewardRepair [72], SelfAPR [19], and many others [43], [73]. These approaches suffer from interpretability issues and mostly can repair simple bugs (e.g., single-line bugs). In contrast, our approach generates patches by reusing existing code, which complements them based on our experimental results. The latest LLM-based APR methods further improved repair performance [20], [22], [43], [74]. For example, Repilot [23] integrates CodeT5 [75] with a completion engine to enhance repair performance, while GAMMA [24] utilizes CodeBERT and UniXcoder [76] to fill predefined repair templates. Unlike REPATT, these methods treat LLMs as black boxes and are still facing challenges related to interpretability [77], data leakage [78], and generalizability [79]. In addition, according to our evaluation results, our approach also complements these LLM-based methods, and thus can be further combined with them for better APR.

VIII. CONCLUSION

To enhance redundancy-based APR, this paper proposes REPATT, a novel repair technique using a two-level pattern mining process for precise patch generation via fine-grained code reuse. Evaluations on Defects4J v1.2 and v2.0 show that REPATT complements existing methods by repairing unique bugs and improving patch precision in real-world repair scenarios. Furthermore, we present the first attempt at combining different approaches to improve the SOTA APRs. Our promising results encourage further research, and we have open-sourced our data and implementations for replication and exploration [40].

References

[1] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *ICSE*, 2018.

[2] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *ICSE*, 2017.

[3] J. Xuan, M. Martinez, F. Demarco, M. Clément, S. Lamelas, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *TSE*, 2017.

[4] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: Association for Computing Machinery, 2019, p. 31–42. [Online]. Available: https://doi.org/10.1145/3293882.3330577

[5] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair," in *FSE*, 2016.

[6] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *ICSE*, 2009, pp. 364–374.

[7] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 298–312.

[8] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *ISSTA*, 2018.

[9] Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair," ser. ASE, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155644

[10] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *ICSE*, 2013, pp. 772–781.

[11] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *ISSTA*, 2016, pp. 177–188.

[12] X.-B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *SANER*, 2016, pp. 213–224.

[13] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: syntax- and semantic-guided repair synthesis via programming by examples," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 593–604.

[14] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *ICSE*, 2013, pp. 802–811.

[15] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object oriented program repair," in *ASE*. IEEE Press, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155643

[16] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *ICSE*, 2014, pp. 254–265.

[17] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *TSE*, vol. 38, no. 1, pp. 54–72, Jan 2012.

[18] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *ESEC/FSE*, 2021, p. 341–353. [Online]. Available: https://doi.org/10.1145/3468264.3468544

[19] H. Ye, M. Martinez, X. Luo, T. Zhang, and M. Monperrus, "Selfapr: Self-supervised program repair with test execution diagnostics," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.

[20] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1430–1442.

[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, pp. 1161–1173.

[22] C. S. Xia and L. Zhang, "Less training, more repairing please: revisiting automated program repair via zero-shot learning," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 959–971.

[23] Y. Wei, C. S. Xia, and L. Zhang, "Copiloting the copilots: Fusing large language models with completion engines for automated program repair," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 172–184.

[24] Q. Zhang, C. Fang, T. Zhang, B. Yu, W. Sun, and Z. Chen, "Gamma: Revisiting template-based automated program repair via mask prediction," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 535–547.

[25] Z. Chen, "The essence of similarity in redundancy-based program repair," 2018.

[26] M. White, M. Tufano, M. Martinez, M. Monperrus, and D. Poshyvanyk, "Sorting and transforming program repair ingredients via deep learning code similarities," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 479–490.

[27] C. Yang, "Accelerating redundancy-based program repair via code representation learning and adaptive patch filtering," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021, 2021, p. 1672–1674.

[28] Z. Chen and M. Monperrus, "The remarkable role of similarity in redundancy-based program repair," *arXiv preprint arXiv:1811.05703*, 2018.

[29] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 306–317. [Online]. Available: https://doi.org/10.1145/2635868.2635898

[30] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. Bissyandé, D. Kim, P. Wu, J. Klein, X. Mao, and Y. L. Traon, "On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20, 2020, p. 615–627.

[31] M. Motwani, M. Soto, Y. Brun, R. Just, and C. Le Goues, "Quality of automated program repair on real-world defects," *IEEE Transactions on Software Engineering*, vol. 48, no. 2, pp. 637–661, 2022.

[32] S. Wang, M. Wen, L. Chen, X. Yi, and X. Mao, "How different is it between machine-generated and developer-provided patches? : An empirical study on the correct patches generated by automated program repair techniques," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2019, pp. 1–12.

[33] D. Yang, X. Mao, L. Chen, X. Xu, Y. Lei, D. Lo, and J. He, "Transplantfix: Graph differencing-based code transplantation for automated program repair," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22, 2023.

[34] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.

[35] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, pp. 213–224.

[36] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," ser. ICSE '12, 2012, pp. 837–847.

[37] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the" naturalness" of buggy code," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 428–439.

[38] A. Khanfir, M. Jimenez, M. Papadakis, and Y. Le Traon, "Codebert-nt: code naturalness via codebert," in *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2022, pp. 936–947.

[39] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, 2014, p. 269–280.

[40] Homepage, 2025, available at: https://zenodo.org/records/14997209.

[41] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *ISSTA*, 2014, pp. 437–440.

[42] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad, "Bugs.jar: A large-scale, diverse dataset of real-world java bugs," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18, 2018, p. 10–13.

[43] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proceedings of the 45th International Conference on Software Engineering*, 2023.

[44] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2019.

[45] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 602–614.

[46] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," ser. ISSTA 2015, 2015, pp. 24–36.

[47] Q. Xin and S. P. Reiss, "Identifying test-suite-overfitted patches through test case generation," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 226–236.

[48] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *ICSE*, 2018.

[49] G. Salton, Ed., *Automatic Text Processing*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1988.

[50] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques and tools*, 2020.

[51] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," in *ICSE*, 2015, pp. 448–458.

[52] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, vol. 10, no. 8, 1966, pp. 707–710.

[53] H. Ye and M. Monperrus, "Iter: Iterative neural repair for multi-location patches," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[54] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *TAICPART-MUTATION*, 2007, pp. 89–98.

[55] J. Campos, A. Riboira, A. Perez, and R. Abreu, "Gzoltar: An eclipse plug-in for testing and debugging," ser. ASE '12, 2012, p. 378–381.

[56] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ASE*, 2014, pp. 313–324.

[57] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *ESEC/FSE*, 2017, pp. 727–739.

[58] W. Weimer, Z. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *ASE*, 2013, pp. 356–366.

[59] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," *IEEE Transactions on Software Engineering*, vol. 40, no. 5, pp. 427–449, 2014.

[60] S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury, "Semantic program repair using a reference implementation," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 129–139.

[61] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 19–30.

[62] Y. Yuan and W. Banzhaf, "Arja: Automated repair of java programs via multi-objective genetic programming," *IEEE Transactions on software engineering*, vol. 46, no. 10, pp. 1040–1067, 2018.

[63] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search," in *ASE*, 2015, pp. 295–306.

[64] S. Saha *et al.*, "Harnessing evolution for multi-hunk program repair," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 13–24.

[65] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *ICSE*, 2016.

[66] X. Gao, B. Wang, G. J. Duck, R. Ji, Y. Xiong, and A. Roychoudhury, "Beyond tests: Program vulnerability repair via crash constraint extraction," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 2, pp. 1–27, 2021.

[67] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts," in *ASE*, 2017.

[68] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Towards practical program repair with on-demand candidate generation," in *ICSE*, 2018.

[69] Y. Xiong and B. Wang, "L2s: A framework for synthesizing the most probable program under a specification," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 3, 2022.

[70] L. Chen, Y. Pei, M. Pan, T. Zhang, Q. Wang, and C. A. Furia, "Program repair with repeated learning," *IEEE Transactions on Software Engineering*, vol. 49, no. 2, pp. 831–848, 2022.

[71] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 101–114.

[72] 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.

[73] Z. Fan, X. Gao, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models," in *Proceedings of the 45th International Conference on Software Engineering*, 2023.

[74] J. A. Prenner, H. Babii, and R. Robbes, "Can openai's codex fix bugs? an evaluation on quixbugs," in *Proceedings of the Third International Workshop on Automated Program Repair*, 2022, pp. 69–75.

[75] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[76] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," *arXiv preprint arXiv:2203.03850*, 2022.

[77] Q. Zhang, C. Fang, Y. Xie, Y. Ma, W. Sun, Y. Yang, and Z. Chen, "A systematic literature review on large language models for automated program repair," *arXiv preprint arXiv:2405.01466*, 2024.

[78] J. Sallou, T. Durieux, and A. Panichella, "Breaking the silence: the threats of using llms in software engineering," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, 2024, pp. 102–106.

[79] F. Li, J. Jiang, J. Sun, and H. Zhang, "Evaluating the generalizability of llms in automated program repair," 2025. [Online]. Available: https://arxiv.org/abs/2503.09217