Hybrid Automated Program Repair by Combining Large Language Models and Program Analysis

FENGJIE LI, Tianjin University, China
JIAJUN JIANG*, Tianjin University, China
JIAJUN SUN, Tianjin University, China
HONGYU ZHANG, Chongqing University, China

Automated Program Repair (APR) has garnered significant attention due to its potential to streamline the bug repair process for human developers. Recently, LLM-based APR methods have shown promise in repairing real-world bugs. However, existing APR methods often utilize patches generated by LLMs without further optimization, resulting in reduced effectiveness due to the lack of program-specific knowledge. Furthermore, the evaluations of these APR methods have typically been conducted under the assumption of perfect fault localization, which may not accurately reflect their real-world effectiveness. To address these limitations, this paper introduces an innovative APR approach called GiantRepair. Our approach leverages the insight that LLM-generated patches, although not necessarily correct, offer valuable guidance for the patch generation process. Based on this insight, GiantRepair first constructs patch skeletons from LLM-generated patches to confine the patch space, and then generates high-quality patches tailored to specific programs through context-aware patch generation by instantiating the skeletons. To evaluate the performance of our approach, we conduct two large-scale experiments. The results demonstrate that GiantRepair not only effectively repairs more bugs (an average of 27.78% on Defects4J v1.2 and 23.40% on Defects4J v2.0) than using LLM-generated patches directly, but also outperforms state-of-the-art APR methods by repairing at least 42 and 7 more bugs under perfect and automated fault localization scenarios, respectively.

CCS Concepts: • **Software and its engineering → Software maintenance tools**; **Search-based software engineering**.

Additional Key Words and Phrases: Program Repair, Large Language Model, Program Synthesis

1 INTRODUCTION

As the scale and complexity of modern software systems continue to grow, the prevalence of software bugs has also risen, resulting in substantial financial and operational loss for organizations and end-users. Addressing these bugs requires a significant investment of time and effort from developers. As a result, Automated Program Repair (APR), which seeks to automatically generate correct patches for buggy code, has garnered considerable interest from both academia and industry.

In the past years, numerous APR approaches have been proposed with the goal of enhancing the quality of automatically generated patches and making them more practical for real-world use [1–18]. These approaches include generating patches through predefined repair templates [6, 11, 12, 16–18], heuristic rules [1, 7–10, 13, 14],

*Corresponding author.

Authors' addresses: Fengjie Li, fengjie@tju.edu.cn, College of Intelligence and Computing, Tianjin University, Tianjin, China; Jiajun Jiang, College of Intelligence and Computing, Tianjin University, Tianjin, China, jiangjiajun@tju.edu.cn; Jiajun Sun, College of Intelligence and Computing, Tianjin University, Tianjin, China, sjjtianjin@tju.edu.cn; Hongyu Zhang, School of Big Data and Software Engineering, Chongqing University, Chongqing, China, hyzhang@cqu.edu.cn.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2025 Copyright held by the owner/author(s).
ACM 1557-7392/2025/1-ART
https://doi.org/10.1145/3715004

Just Accepted

and constraint solving techniques [2–5]. While these methods have proven effective in addressing some real-world bugs, the number of correct fixes remains limited [19, 20]. The reason is that it is difficult for these methods to tackle the large search space of diverse patches in real applications. For example, the template-based methods rely on high expertise and manual efforts to construct the templates; the heuristic-based methods are less effective when facing the rapid growing of patch space; while the constraint-based methods suffer from scalability issues. Although deep-learning-based APR methods have significantly improved the repair ability by utilizing the latest advance of deep learning techniques [21–27], many previous studies [28–30] pointed out that their repair capacity relies on the quality of training data, making them hard to repair bugs that have not encountered during training.

Recently, Large Language Models (LLMs) have demonstrated promising results across various software engineering tasks, e.g., code search [31], program synthesis [32], defect detection [33], code summarization [34] and so on. Some recent studies [19, 20, 29, 35–40] have also explored the application of LLMs in automated program repair. The initial results demonstrate their ability to correctly repair real-world bugs, including those that were previously unrepairable by existing APR approaches. The promising outcomes suggest the potential of LLMs in developing more effective APR methods.

While recent studies [19, 20, 37–39] have explored the use of LLMs for automated program repair, there are still significant limitations that need to be addressed:

(1) Existing LLM-based APR approaches directly leverage the patches generated by LLMs, without further optimization or refinement. However, LLMs may struggle to generate patches that correctly incorporate program-specific elements like local variables and domain-specific method calls [41, 42]. This means that even if the LLM-generated patches are “close” to the desired solution, they may still fail to pass the test cases. How to effectively utilize these “incorrect” patches to improve the overall repair ability remains a largely unexplored question.
(2) Evaluations of LLM-based APR approaches have so far been conducted under the assumption of perfect fault localization, where the faulty locations are already known. This is an unrealistic scenario, as in practice, automated fault localization techniques are often inaccurate. The real-world performance of LLM-based APR approaches under the more realistic setting of automated fault localization is yet to be thoroughly investigated.

To address these limitations, a more comprehensive and practical evaluation of LLM-based APR approaches is required. This should involve exploring methods to better leverage the insights from “incorrect” patches generated by LLMs, as well as assessing the performance of these techniques under the more challenging scenario of automated fault localization. Addressing these limitations is crucial for understanding the true potential and practical applicability of LLM-based approaches in the field of automated program repair.

In this paper, we aim to address these two limitations. Specifically, to address the first limitation, we propose a novel automated program repair approach, named GiantRepair. The key insight behind GiantRepair is that patches generated by LLMs, although not always correct, can still provide valuable guidance for the patch generation process. Specifically, GiantRepair first leverages LLMs to efficiently generate a diverse set of candidate patches. It then abstracts these candidate patches into a set of *patch skeletons* that capture the core structures of the patches. These patch skeletons are then used to guide the subsequent context-aware patch generation process, where the patches are refined and instantiated to fit the specific program context. This two-step approach has several advantages. First, existing studies [19, 20, 29, 35–42] show that while LLMs demonstrate strong coding capabilities, they often lack precision when generating domain-specific code elements. Consequently, extracting *patch skeletons* from LLM-generated patches helps to confine the search space for possible patches (since LLMs can indicate a clear direction for the repair process) while avoiding unusable portions of the generated content. This makes the patch generation process more efficient and effective compared with generating patches from scratch. Second, by combining the strengths of LLMs (for initial patch generation) and context-aware refinement

Just Accepted

(for patch instantiation), GiantRepair is able to generate high-quality patches that can correctly fix real-world bugs. To address the second limitation and comprehensively evaluate the performance of GiantRepair, we assess its effectiveness not only under the assumption of perfect fault localization (as done in previous studies [19, 20, 37–39, 43]), but also with the more realistic scenario of automated fault localization. This allows us to better understand the practical applicability of GiantRepair in real-world settings.

We have conducted two large-scale experiments using the widely-used Defects4J benchmark [44] to evaluate GiantRepair in two different application scenarios.

(1) In the first scenario, we compared the repair results of individual LLMs with and without integrating GiantRepair. The results showed that GiantRepair improved the repair performance of individual LLMs by correctly repairing an average of 27.78% and 23.40% more bugs on Defects4J v1.2 and Defects4J v2.0.
(2) In the second scenario, we integrated GiantRepair with existing LLMs to form a standalone APR and compared its repair results with existing state-of-the-art APR approaches. Under the assumption of perfect fault localization, GiantRepair successfully repaired 171 bugs, outperforming the best state-of-the-art APR approaches by repairing at least 42 more bugs. When with the more realistic scenario of automated fault localization, GiantRepair can still repair at least 7 more bugs than the best-performing APR approaches.

Overall, the experimental results demonstrate the effectiveness and generality of GiantRepair, providing new insights for future research in the field of APR. The results highlight the potential for better utilization of LLM outputs for improved APR. To sum up, this paper makes the following major contributions.

- An innovative automated program repair technique that leverages the capabilities of LLMs and context-aware patch refinement.
- A novel patch generation method that extracts patch skeleton from LLM-generated patches to confine the patch space for better APR.
- A comprehensive evaluation in two application scenarios, and the experimental results confirm the effectiveness and generalizability of our approach.
- We have open-sourced our implementations and all experimental data to facilitate future research in this field. https://github.com/Feng-Jay/GiantRepair

2 MOTIVATING EXAMPLES

In this section, we show two real-world examples from our experiment to demonstrate how incorrect LLM-generated patches can be utilized to guide the patch generation process, thereby motivating the need for our context-aware patch generation method. Listing 1 shows the developer patch and LLM's patch for the bug JacksonDatabind-51 from Defects4J [44]. In this paper, a line of code starting with "+" denotes a newly added line while lines starting with "-" denote lines to be deleted.

The developer patch indicates the need to add an `if` condition to fix the bug. Specifically, the root cause of this bug is that the statement *type = ctxt.getTypeFactory().constructSpecializedType(...)* incorrectly overwrites the generic type information of `type`. Therefore, it is essential to verify if the current `type` includes generics before proceeding, where the method call `hasGenericTypes()` is used to achieve this goal. However, the LLM-generated patch, which uses `!type.equals(_baseType)` as the condition, is not semantically equivalent to the intended solution. This is because having `type` equal to `_baseType` does not necessarily imply that `type` includes generic parameters. For example, when both `type` and `_baseType` are `SimpleTypes` (e.g., `String`) without any generic types, the LLM-generated patch prevents the assignment within the `if` block from executing, whereas the desired behavior is the opposite. Addressing this bug presents several challenges for existing APR methods: (1) pinpointing the exact faulty line of code is difficult; (2) determining the necessity of introducing a new `if` statement is complex; (3) the specific conditional expression (`!type.hasGenericTypes()`) is domain-specific and may not be applicable

Just Accepted

elsewhere. These challenges result in a vast search space for potential patches. Consequently, none of the existing approaches in our experiment, including the latest LLM-based APR methods, were able to successfully address this issue.

```
JavaType type = _idResolver.typeFromId(ctxt, typeId);
...
if ((_baseType != null) && _baseType.getClass() == type.getClass()) {
  // Developer patch
+  if (!type.hasGenericTypes()) {
      type = ctxt.getTypeFactory()
        .constructSpecializedType(_baseType,
            type.getRawClass());
+  }
  // LLM's patch
+  if (!type.equals(_baseType)) {
      type = ctxt.getTypeFactory()
        .constructSpecializedType(_baseType,
            type.getRawClass());
+  }
}
```

Listing 1. Patches of JacksonDatabind-51 from Defects4J

```
// Developer patch
+ result[resultOffset] =
+     FastMath.atan2(y[yOffset], x[xOffset]);
// LLM's patch
+ for (int i = 0; i < tmp1.length; ++i) {
+     result[resultOffset + i] =
+         FastMath.atan2(y[yOffset + i],x[xOffset + i]);
+ }
```

Listing 2. Patches of Math-10 from Defects4J

Upon comparing the LLM's patch with the developer patch, it becomes evident that they are somewhat similar – both introduce a new `if` statement with a method call as the condition. Although the LLM's patch lacks domain-specific knowledge (i.e., the method call `hasGenericTypes()`), it still offers valuable guidance by providing a similar patch structure (e.g., the `if` statement), effectively narrowing down the search space for potential patches. However, effectively leveraging these LLM-generated patches for improving APR remains challenging due to the diverse and complex nature of real-world situations. For example, in Listing 2, another real-world bug repair example is presented. Here, the desired patch involves inserting a new `Assignment` for the variable `result[resultOffset]`, while the LLM's patch introduces a new `for` loop statement. In this case, only a portion of the LLM's patch may be useful, as reusing the entire `for` loop will not pass the test cases. Additionally, updating the indices of the array accesses to variables `x` and `y` is also necessary to construct the correct patch.

To address the challenges outlined above, this paper introduces a context-aware and adaptive patch generation method aimed at effectively reusing patches generated by LLMs. As previously discussed, LLM's patches offer valuable patch structures. Therefore, the core concept of our approach is to construct patch skeletons from LLM's patches to limit the patch space, and then generate high-quality patches through context-aware skeleton instantiation using static analysis. This enables the generation of patches tailored to specific programs.

3 APPROACH

In this section, we provide a detailed explanation of our approach, i.e., GiantRepair. As previously mentioned, our approach is based on the insight that LLM-generated patches, while not always correct, can offer valuable guidance on patch structure for constraining the patch space. Therefore, our patch generation process consists of two key components: **skeleton construction** and **patch instantiation**. (1) The skeleton construction component

Just Accepted

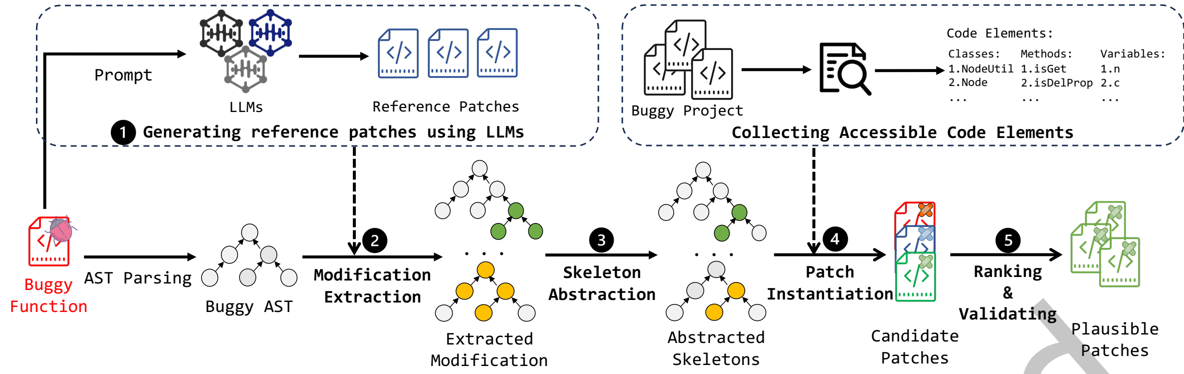

Fig. 1. Overview of GiantRepair

(Section 3.1) involves extracting a set of code modifications from the provided LLM-generated patches by comparing the buggy code and patched code through tree-level differencing over the abstract syntax tree. These modifications are then abstracted into patch skeletons using a set of abstraction rules. (2) The patch instantiation component (Section 3.2) takes these skeletons and instantiates them using valid (e.g., defined under certain contexts) and compatible (e.g., satisfying type constraints) program elements through static analysis, resulting in executable patches. Finally, GiantRepair evaluates the correctness of the patches by running test cases according to a patch ranking strategy (Section 3.3). Figure 1 provides an overview of our approach, and in the following sections, we will delve into each step in detail.

3.1 Skeleton Construction

As mentioned in Section 2, it is often the case that not all the code changes in an LLM-generated patch are desirable under different contexts. Therefore, it is necessary to disassemble the code changes in a patch into individual components, allowing them to be applied independently. To achieve this, GiantRepair incorporates a modification extraction process that aims to identify all concrete code changes through tree-level code differencing between the buggy code and the patched code. Each identified modification then will be abstracted into a patch skeleton for subsequent patch generation.

3.1.1 Modification Extraction. In order to extract concrete modifications, GiantRepair performs a tree-level code matching and differencing process. Specifically, GiantRepair endeavors to match the code elements from the buggy code and the patched code and then generates code modifications for those elements that are different between these two sides. In particular, we consider statement-level code matching rather than the expression-level. The reasons are twofold: (1) matching statements are more efficient and less likely to produce incorrect matching because different statements tend to be diverse while the finer-grained expressions have a larger possibility to be the same at different locations; (2) the search space of statement-level code changes is relatively small and should be efficient for skeleton construction and instantiation.

Algorithm 1 presents the matching algorithm in GiantRepair, which is inspired by GumTreeDiff [45]. In particular, we use the function *type(a)* to represent the AST node type (e.g., IfStatement) of node *a*, and use *children(a)* to return a set of statements that are child nodes of *a*. In general, it takes two AST nodes (one from the buggy code and one from the patched code) as inputs and recursively performs a top-down matching process following a greedy strategy – two AST nodes (i.e., statements) can match each other as long as their node types are the same (line 12). After this process, one statement from the buggy code may have more than one matched statement from the patched code (according to line 14). To obtain the best matching and extract the fewest

Just Accepted

Algorithm 1: AST Node Matching

```
1  func match(a: ASTNode, b: ASTNode):
2      mapping ← matchNode(a, b)
3      mapping.sortBySimilarity()
4      result ← ∅ , dups ← ∅
5      foreach ⟨a, b⟩ ∈ mapping do
6          if a ∉ dups and b ∉ dups then
7              dups ← dups ∪{a, b}
8              result ← result ∪{⟨a, b⟩}
9      return result
10 func matchNode(a: ASTNode, b: ASTNode):
11     result ← ∅
12     if type(a) = type(b) then
13         result ← result ∪{⟨a, b⟩}
           /* ×: Cartesian product of two sets */
14         foreach ⟨a′, b′⟩ ∈ children(a) × children(b) do
15             result ← result ∪ matchNode(a′,b′)
16     return result
```

modifications, GiantRepair removes the redundant matches and only preserves the best one by measuring the similarity between the matched statements (line 3). Formula 1 defines the computation of similarity between two statements. In the formula, we use "*atomic stmts*" to denote those statements that cannot be decomposed into finer-grained ones, e.g., ExpressionStatement, while use "*ensembled stmts*" to represent those that are ensembled by other statements, e.g., IfStatement. In addition, the function *editDistance(a,b)* computes the token-level edit distance [46, 47] between code snippets corresponding to node a and b, while C_a denotes the child statements of a.

$$sim(a,b) = \begin{cases} \frac{editDistance(a,b)}{length(a)} & \text{:atomic stmts.} \\ \frac{1}{|C_a|} \sum\limits_{c_1 \in C_a} \max\limits_{c_2 \in C_b} sim(c_1, c_2) & \text{:ensembled stmts.} \\ 0 & \text{:otherwise} \end{cases} \tag{1}$$

Once obtaining the matching results from Algorithm 1, GiantRepair extracts the concrete modifications at the statement level. Suppose that statements a and b are from the buggy and the patched code respectively. Then, GiantRepair may generate the following modifications according to the matching results:

Update(a,b) : replaces statement a with statement b if a matches b but they are not completely the same code;

Insert(b, i) : inserts the statement b into the buggy code as the i_{th} child statement of p' if b does not match any statement but its parent node p matches statement p'. i is the index of b in p;

Delete(a) : deletes the statement a from the buggy code;

Intuitively, after applying all extracted modifications to the buggy code, it will be transformed into the same one as the patched code.

3.1.2 Skeleton Abstraction. As aforementioned, the modifications extracted from LLM-generated patches contain valuable guidance. Specifically, they often make changes at the correct locations and possess AST structures similar to the correct fixes. However, directly applying these modifications may not necessarily produce correct patches as they may introduce incorrect program elements that are not applicable to certain contexts under repair. To overcome this issue, GiantRepair involves a code abstraction process. That is, after obtaining the

Just Accepted

Table 1. Abstraction rules for patch skeleton construction regarding different AST node types. The blue tokens in the skeleton are keywords that will be *extended* from the source code during abstraction, while the red tokens will be instantiated for patch generation via static analysis. In particular, EXPR[T] denotes the type of expression EXPR is constrained by T.

	AST Node Type	Example	Skeleton	Constraint Description
STMT	AssertStatement	assert a > 0;	assert EXPR[T];	T is boolean
	ConstructorInvocation	this(a);	this(EXPR$_1$[T$_1$],...,EXPR$_n$[T$_n$]);	n and T$_n$ should be compatible with the class.
	DoStatement	do{ S }while(a < 0);	do{STMT} while(EXPR[T]);	T is boolean
	ForStatement	for(;a<0;){ S }	for(;EXPR[T];){STMT}	T is boolean
	IfStatement	if(a < 0){ S }	if(EXPR[T]){STMT}	T is boolean
	ReturnStatement	return a;	return EXPR[T];	T is compatible with the return type
	SwitchStatement	switch(a){case b: f();}	switch(EXPR1[T$_1$]){case EXPR2[T$_2$]: STMT}	T$_1$ and T$_2$ are compatible
	ThrowStatement	throw a;	throw EXPR[T];	T is Exception
	VarDeclStatement	int a = b;	int EXPR[T];	T is compatible with int
	WhileStatement	while(a < 0){S}	while(EXPR[T]){STMT}	T is boolean
	ExpressionStatement	a = a + b;	EXPR[T];	No constraint on T
EXPR[T$_0$]	Assignment	a = a + b	VAR[T$_1$]=EXPR[T$_2$]	T$_1$ and T$_2$ are compatible
	CastExpression	(int) b	(int) EXPR[T]	T is compatible with int
	ClassInstanceCreation	new ClassA(a,b)	new CNAME(EXPR$_1$[T$_1$],...EXPR$_n$[T$_n$])	class CNAME is compatible with T$_0$; n and T$_n$ fit CNAME
	ConditionalExpression	a > b ? a : b	EXPR$_1$[T$_1$]?EXPR$_2$[T$_2$]:EXPR$_3$[T$_3$]	T$_1$ is boolean; T$_2$, T$_3$ are compatible with T$_0$
	FieldAccess	a.b	EXPR[T$_1$].VAR[T$_2$]	VAR is defined in T$_1$; T$_2$, T$_0$ are compatible
	InfixExpression	a + b	EXPR$_1$[T$_1$] INFIX_OP EXPR$_2$[T$_2$]	T$_1$, T$_2$ are compatible with INFIX_OP
	PrefixExpression	!a.isEmpty()	PREFIX_OP EXPR[T]	T, PREFIX_OP are compatible
	PostfixExpression	a++	EXPR[T] POSTFIX_OP	T, POSTFIX_OP are compatible
	MethodInvocation	a.method(b)	EXPR.FNAME(EXPR$_1$[T$_1$],...,EXPR$_n$[T$_n$])	Return type of FNAME is compatible with T$_0$; n and T$_n$ fit FNAME
	SimpleName	a	VAR[T]	T and T$_0$ are compatible
	SuperFieldAccess	super.a	super.EXPR[T]	T and T$_0$ are compatible
	SuperMethodInvocation	super.a(b)	super.FNAME(EXPR$_1$[T$_1$],...,EXPR$_n$[T$_n$])	Return type of FNAME is compatible with T$_0$; n and T$_n$ fit FNAME
	VarDeclExpression	int a = b	int a = EXPR[T]	T is compatible with int
	VarDeclFragment	a = b	a = EXPR[T]	T is compatible with T$_0$

concrete modifications according to the algorithm explained above, GiantRepair performs code abstraction that constructs patch skeletons by removing concrete program elements while preserving the code structure for confining the subsequent patch generation. Specifically, *GiantRepair performs the abstraction process only for the statement b appearing in modifications **Update(a,b)** and **Insert(b,i)***. In contrast, for modification **Delete(a)**, abstraction is not necessary as it will not introduce any new elements to the program.

More specifically, we have defined a set of code abstraction rules by following the AST node definition in the Java Development Toolkit [48]. Table 1 presents the details of the rules. In the table, the first column presents the abstracted notations that can be further abstracted according to their actual AST node types as shown in the second column. That is, the abstraction process is a recursive process in a top-down fashion by following the abstract syntax tree structure of the code until it cannot be further abstracted by any rules, e.g., individual variables or operators. Particularly, we provide a simple example (the 3rd column) for each type of AST node for better understanding of the skeleton construction rules (the 4th column). The last column describes the constraints that have to be satisfied when instantiating the skeleton for patch generation. Taking the AssertStatement "`assert a>0;`" as an example, the abstracted skeleton will be "`assert EXPR[boolean]`", where the "`EXPR`" will be further abstracted into "`VAR INFIX_OP 0`" according to the rule for InfixExpression. Please note that we do not abstract constant values (e.g., 0) in the code since they are not program-specific elements and thus can be reused directly. Consequently, the ultimate skeleton will be "`assert VAR INFIX_OP 0;`", where `VAR` has to be a variable of number type (e.g., `int` and `float`) and the operator `INFIX_OP` has to be logical comparators (e.g., > and <). In this way, the structure of the LLM-generated patches can be preserved for effectively constraining the patch space, and the abstracted tokens (colored red) can be instantiated via analyzing the contexts under repair for tailoring to certain programs.

3.2 Patch Instantiation

According to the constructed patch skeletons introduced above, the patch instantiation process becomes straightforward – replacing the abstracted tokens in the skeleton with concrete program elements that satisfy the given constraints. Specifically, there are in total four types of abstracted tokens (see Table 1) in the ultimate skeletons

Just Accepted

for instantiation, i.e., variables (VAR), classes (CNAME), method calls (FNAME), and operators (INFIX_OP, PREFIX_OP, and POSTFIX_OP). While the skeleton can already effectively constrain the patch space, randomly generating patches from them may still encounter a large search space. Therefore, it is less efficient and easy to fail due to a limited time budget. Therefore, GiantRepair incorporates a context-aware patch generation strategy during skeleton instantiation, which involves three optimizations:

- **Element selection:** All used elements have to be usable under certain contexts and meet the constraints of used patch skeletons. For operators, GiantRepair enumerates all type-compatible ones belonging to each type during skeleton instantiation since the number of them is very small (<10), while for the other three types of tokens, GiantRepair includes a static analysis process for collecting all usable ones. Specifically, for variables, GiantRepair records their types and scopes; for classes, it records their inheritance relations and accessible fields; for method calls, it records their complete signatures, including the required arguments, return types and classes which they belong to. Such information will determine the usability of program elements by checking the constraints associated to the skeleton.
- **Context similarity:** GiantRepair considers two kinds of similarities, one of which is between the instantiated patch and the buggy code while the other is between instantiated patch and the LLM-generated patch. The first similarity is inspired by previous studies [19, 20, 29, 35–40], which reported that the desired patches in realistic scenarios often involve small code changes. The second similarity is inspired by our insights and the prominent results of LLMs as they can provide valuable material for patch generation. To achieve these goals, GiantRepair first preserves the common code elements (e.g., variables) used in both the buggy code and the LLM patch as much as possible. For different elements, GiantRepair prefers patches that are "close" to the LLM patch. Specifically, it uses the general token-level edit distance [46, 47] to measure the closeness between patches. The reason is that the instantiated patches share the same skeleton with the LLM patches, and thus the major differences between them mainly lie in the variable and function names. In addition, by "closeness", we mean whether the patches use **the same** variables or function calls as the LLM patch. We believe that this distinction can be effectively measured using the token-level edit distance.
- **Adaptive application:** As explained in Section 2, it is possible that only a part of the LLM patch is desirable. Therefore, if the patches with all code changes failed to repair the bug, GiantRepair adaptively applies a subset of the extracted modifications. Specifically, GiantRepair endeavors to apply at most three individual modifications in one patch (producing a reasonable and manageable number of patches). In particular, as shown by existing literature [19, 20, 37–40, 49] and our motivating examples outlined in Section 2, LLMs have demonstrated powerful capabilities in coding related tasks and the patches generated by LLMs often contain valuable modifications. To fully leverage the guidance information within the LLM-generated patches, GiantRepair prefers to select the most complex modifications for combination, as they can maximize the use of valuable information within the patches, as explained above.

Based on the above patch instantiation process, given an LLM-generated patch, GiantRepair generates candidate patches on top of the abstracted patch skeletons, which can effectively constrain the search space of patches.

3.3 Patch Ranking and Validation

To make the most probable patches to be evaluated early, we have developed a patch ranking strategy. As reported in previous studies [19, 20, 29, 31–40], LLMs have shown powerful capabilities in code understanding and generation. Therefore, when given a set of candidate patches generated by LLMs, GiantRepair favors those that can offer more new resources for patch generation. In this way, we can make best use of the code generation capability of LLMs by maximizing the repair-relevant information available for generating effective fixes. Specifically, GiantRepair assesses the number of **Insert(*)** and **Update(*)** modifications involved in a

Just Accepted

patch since they can bring new program elements/structures while **Delete(*)** cannot. The higher the count of these modifications, the higher the rank of the patch. Subsequently, for each LLM-generated patch, GiantRepair constructs candidate patches using the patch instantiation process introduced earlier. Finally, GiantRepair executes the associated test cases after applying each patch and identifies the ones that pass all the test cases as plausible patches, in line with existing studies [20, 27, 29, 38, 39]. In this process, GiantRepair utilizes the ExpressAPR [50] framework for managing the test running. Consistent with existing studies [20, 27, 29, 38, 39], a patch is deemed correct only if it is semantically equivalent to the developer patch, as determined through manual inspection.

4 EXPERIMENTAL SETUP

4.1 Research Questions

In this paper, we aim to answer the following research questions for evaluating the effectiveness of GiantRepair.

- **RQ1:** ***How effective is GiantRepair for improving LLMs in repairing real-world bugs?*** In this RQ, we explore whether GiantRepair can improve the effectiveness of existing LLMs in the task of program repair. Specifically, we integrate GiantRepair with different LLMs and check whether it can correctly repair more bugs than using the LLM-generated patches directly.
- **RQ2:** ***How effective is GiantRepair compared to the state-of-the-art APR tools?*** In this RQ, we integrate GiantRepair with existing LLMs to form a standalone APR tool by following existing study [20, 38, 39], and then compare its performance with a set of state-of-the-art APR tools.
- **RQ3:** ***What is the contribution of each component in GiantRepair?*** In this RQ, we investigate the contribution of each component in GiantRepair to its effectiveness. Specifically, as introduced in Section 3, GiantRepair incorporates four major components, including patch skeleton construction, context-aware patch instantiation, adaptive application of modifications and patch ranking. For the first component, we explore the contribution of each abstraction rule (see Table 1) for skeleton construction to the generation of correct patches. For the remaining three components, we develop a set of variants of GiantRepair for analyzing their effectiveness. The details will be introduced in Section 5.3.

4.2 Subjects

In our experiment, we employed the widely-studied Defects4J [44] benchmark. In particular, we adopted both version 1.2 and version 2.0 of the benchmark for evaluating the generality of GiantRepair. Specifically, Defects4J v1.2 consists of 391 bugs from six real-world projects, while Defects4J v2.0 includes another 438 bugs from 11 real-world projects. By following existing studies [19, 37], we leveraged LLMs to generate candidate patches when providing the buggy function. The reasons are twofold: (1) The code length of a single function is suitable for the input and output of current LLMs, and the entire method can offer local contexts for LLMs; (2) Function-level fault localization is more precise than the line-level fault localization, and thus the APR tools depending on the former can be more practical for real use. Therefore, we removed the bugs that require cross-function modifications. Consequently, we use all 255 single-function bugs from Defects4J v1.2 and 228 single-function bugs from Defects4J v2.0 in our evaluation.

4.3 Baselines and Metrics

To answer RQ1, we selected four commonly-used LLMs as the baselines, including two general-purpose LLMs (GPT-3.5 Turbo [51, 52] and Llama-2 [53]) and two code-specific LLMs (StarCoder [54] and CodeLlama [55]), all of which have been used in diverse software engineering tasks [56, 57] and demonstrated to be effective, including automated program repair [40, 43].

Just Accepted

To answer RQ2, we compared GiantRepair with the **11** latest and best-performing APR tools due to the space limit, including *four LLM-based* (FitRepair [20], Repilot [38], GAMMA [39], and AlphaRepair [29]), *five specially-designed deep-learning-based* (Tare [27], ITER [58], CURE [25], Recoder [24], and Hanabi [59]), *one template-based* (TBar [15]), and *one heuristic search-based* (SimFix [14]). These APR tools cover most of the SOTA techniques used in recent APR research. In particular, we also offer a more comprehensive comparison regarding unique fixes between our approach and existing APR tools in Section 6.1.

In this paper, a patch is **plausible** if it can pass all the test cases, and a plausible patch is **correct** if it is semantically equivalent to the developer patch. For result analysis, we mainly compare the number of bugs that can be correctly repaired by each baseline by following previous studies [14, 15, 20, 24, 25, 27, 29, 38, 39, 59]. Furthermore, we also calculate the **precision** of the generated patches, which denotes the ratio of bugs with correct patches to the bugs with plausible patches, and the **recall** of the generated patches, which is the ratio of bugs with correct patches to the total number of fixable bugs.

```
//Provide a fix for the buggy function
//Buggy function
{Example_bug_1}
//Fixed function
{Example_fix_1}
//Provide a fix for the buggy function
//Buggy function
{Example_bug_2}
//Fixed function
{Example_fix_2}
//Provide a fix for the buggy function
//Buggy function
{bug}
```

Fig. 2. The input prompt for the function-level APR

Just Accepted

4.4 Implementation and Configuration

Foundation models. We selected four widely-used models in previous studies[60–62], including three open-source models – StarCoderBase (i.e., StarCoder-15.5B), CodeLlama-7B, Llama-2-13B, and one close-source model – GPT-3.5, which have demonstrated impressive performance on many tasks. The first three open-source models were downloaded from HuggingFace [63] and then deployed on our local machines while the last GPT-3.5 online model was accessed via API requests [64]. For each model, we reused the *prompt* proposed by Xia et al. [19], and adopted the model default settings for patch generation – Top-p Necleus Sampling [65] with p = 0.95 and temperature = 0.8. Figure 2 shows the template of the prompt used in our experiment. Specifically, we use a 2-shot method. The first example demonstrates the repair task and the expected output format to the LLM. The second example, selected from the same project where the bug originates, provides the LLM with the relevant coding style. Lastly, the specific bug to be fixed is included in the prompt. Finally, for each bug, one LLM generates at most 200 patches. For other baseline APR tools, we reused their experimental results from the corresponding publications directly.

GiantRepair. We implemented GiantRepair in Java with approximately 22k lines of code. During the repair process, GiantRepair at most generates 200 candidate patches based on one patch skeleton from LLM-generated patches. Besides, following prior work [14, 15, 27], we offer a 5-hour time budget for repairing a single bug.

Fault localization. As mentioned in Section 1, we comprehensively evaluated the performance of our approach under both perfect and imperfect (i.e., automated) fault localization. In the first scenario, when given the perfect fault localization results (i.e., the specific faulty code lines) [15, 24, 27, 29], we initially mapped these lines to their respective enclosing functions. This step is necessary because, as shown in Figure 2, our approach uses the entire faulty function as input rather than isolated faulty lines. Subsequently, we directly fed each faulty function to the LLMs for patch generation. While in the second scenario, we followed prior studies [14, 15, 24] and employed the spectrum-based algorithm, Ochiai [66], implemented in GZoltar [67], to obtain a list of faulty code lines. Similarly, based on existing work [68], we further mapped these faulty lines to their respective enclosing functions to achieve function-level fault localization results, which were fed to our approach. Then, following the function ranking, we tried to generate patches for each one until exceeding the time limit. Please note that some baseline APR tools used a finer-gained line-level perfect fault localization (i.e., offering the buggy line) in their experiments, such as FitRepair and Repilot. Although it can be more accurate as it confines the patch space into a single line, we do not further unify this configuration in this paper because the baselines cannot work with the function-level fault localization. Nevertheless, this difference may underestimate the effectiveness of our approach when compared with the baselines.

Experimental environment. Our experiments were conducted on a local machine equipped with dual Intel Xeon 6388 CPUs, 512GB RAM, and four A800 GPUs, running Ubuntu 20.04.6LTS.

5 RESULT ANALYSIS

5.1 RQ1: Overall Effectiveness for Improving LLMs in APR

As explained in Section 4.1, to evaluate whether our approach can better utilize the LLM-generated patches in program repair, we compare the results of GiantRepair with the repair results when using the LLM-generated patches directly. Specifically, we selected four diverse LLMs for comparison, aiming to display the generality of our approach (see Section 4.3). In this experiment, we offer LLMs the function-level perfect fault localization by following existing studies [19, 20, 25, 29, 37, 38]. Table 2 presents the number of bugs that can be correctly repaired by each method. In the table, we use GiantRepair$_{GPT-3.5}$ to represent the repair results when GiantRepair takes the patches generated by GPT-3.5 as inputs.

From the table, we can observe that GiantRepair can effectively increase the number of correct fixes compared with using the LLM-generated patches directly. Specifically, on Defects4J v1.2, GiantRepair increases the number of correct fixes from 43, 42, 40, 19 to 53, 55, 51, 25 respectively compared with the four LLMs. *The relative improvement is up to 31.58%, with an average increase of 27.78%.* On Defects4J v2.0, GiantRepair increases the number of correct fixes from 45, 44, 34, 18 to 53, 54, 43, 24. *The relative improvement is up to 33.33%, with an average increase of 23.40%.* This results demonstrate the generalizability of GiantRepair in enhancing repair performance of LLMs since it achieved relatively close effectiveness when comparing with diverse models over different benchmarks.

Furthermore, the repair performance of the selected LLMs in this paper is also consistent with prior work [57]: *GPT-3.5>StarCoder>CodeLlama>Llama-2.* This reflects that training purposes tend to have a larger influence on the LLM's performance than LLM's sizes. For example, GPT-3.5, despite having a much larger model size than StarCoder, shows very close repair effectiveness in our experiment. In contrast, CodeLlama, which is fine-tuned from Llama-2 on code, demonstrates a significant improvement in patch generation. We leave the further exploration of this situation in a broader range to our future work.

Just Accepted

Table 2. Result comparison with different LLMs

Tool	Defects4J v1.2					Defects4J v2.0				
	#Correct	Impv.(%)	#Plausible	Precision (%)	Recall (%)	#Correct	Impv.(%)	#Plausible	Precision (%)	Recall (%)
GPT-3.5	43		75	57.33	16.86	45		62	72.58	19.74
GiantRepair$_{GPT-3.5}$	53	23.26	102	51.96	20.78	53	17.78	82	64.63	23.25
StarCoder	42		72	58.33	16.47	44		60	73.33	19.30
GiantRepair$_{StarCoder}$	55	30.95	104	52.88	21.57	54	22.73	86	62.79	23.68
CodeLlama	40		65	61.54	15.69	34		58	58.62	14.91
GiantRepair$_{CodeLlama}$	51	27.50	96	53.13	20.00	43	26.47	77	55.84	18.86
Llama-2	19		29	65.52	7.45	18		29	62.10	7.89
GiantRepair$_{Llama-2}$	25	31.58	61	40.98	9.80	24	33.33	47	51.06	10.53
Average$_{LLMs}$	36		60.25	59.75	14.12	35.25		52.25	63.80	15.46
Average$_{GiantRepair_{LLMs}}$	**46**	**27.78**	90.75	50.69	18.04	**43.25**	**23.40**	73	59.25	18.97

To analyze the additional overhead introduced by GiantRepair in exploring the patch space, we conducted analyses from two perspectives: **1) Balance between precision and recall.** As shown in Table 2, although GiantRepair leads to a decrease in precision for each LLM, this decrease is relatively modest compared to the increase in the number of correct fixes. The most substantial drop in precision occurs with Llama-2, which we attribute to its poor coding capabilities, resulting in less effective guidance information. On average, the precision of patches generated on Defects4J v1.2 and Defects4J v2.0 decreased from 59.75% and 63.80% to 50.69% and 59.25%, respectively. This decrease is relatively minor and is comparable to the improvement in recall, which increased from 14.12% and 15.46% to 18.04% and 18.97% on Defects4J v1.2 and Defects4J v2.0, respectively, demonstrating that GiantRepair strikes a favorable balance between precision and recall. In addition, as reported in the previous study [69], plausible patches will not largely sacrifice the debugging efficiency of developers with the help of debugging tools. Moreover, patch filtering tools [70–73] can be further incorporated to improve patch precision. **2) Token and time consumption.** To explore the overhead of our approach, we further analyzed the token and time consumption required by GiantRepair to generate all correct fixes. As shown in Table 3, GiantRepair on average requires only an additional 2517.33 output tokens to generate one correct fix compared to using the LLMs directly, which results in on average 0.025 more dollars for repairing one bug according to the pricing of the GPT-4o model [74]. Regarding the time consumption required by GiantRepair, about 71.67% of the correct fixes were generated within 0.5 hours, and the vast majority (about 93.25%) could be produced within 2 hours. This execution time is also acceptable compared with many existing approaches [13–15], which may even require about five hours to repair a bug.

We also analyzed the complementary of different LLMs in this task. Figure 3 presents the number of bugs that are uniquely repaired when integrating GiantRepair with each LLM. The results reveal that, although different LLMs performed diversely, they tend to complement each other as each individual LLM can offer the valuable repair guidance for some unique bugs. For example, Llama-2, which achieved the fewest correct fixes, contributed 3 unique fixes. This indicates that both code-specific and general-purpose LLMs should be considered in APR. In addition, the results also inspired us to explore whether GiantRepair can still improve the repair performance when taking the complementary among different LLMs into consideration. Consequently, we combined the repair results of all the four LLMs, and then compared it with GiantRepair that takes all their patches as inputs. The results show that the combination of the four LLMs successfully repaired 141 bugs on the two benchmarks, while GiantRepair repaired 171 bugs (will be further discussed in Section 5.2), demonstrating that GiantRepair is indeed effective as it can effectively utilize the LLM-generated patches for better APR. In fact, when comparing with the latest GPT-4, GiantRepair can still contribute unique fixes. We will discuss this result in Section 6.2.

To show the necessity and effectiveness of our approach, we present two examples that the studied LLMs failed to repair under our experimental setting, whereas GiantRepair generated the correct fixes. Listing 3 and

Just Accepted

Table 3. Token consumption of GiantRepair in generating correct fixes

Tool	Token Consumption			
	#min	#max	#median	#average
GPT-3.5	19	66751	244	4765.93
$\text{GiantRepair}_{GPT-3.5}$	19	101508	282	6934.11
StarCoder	22	60828	258	3293.81
$\text{GiantRepair}_{StarCoder}$	13	64253	335	5390.38
CodeLlama	26	255798	299	9328.30
$\text{GiantRepair}_{CodeLlama}$	17	255798	482	11881.89
Llama-2	30	76127	286	7694.51
$\text{GiantRepair}_{Llama-2}$	30	76127	598	11688.57
Total_{LLM}	19	255798	260.50	5874.41
$\text{Total}_{\text{GiantRepair}}$	13	255798	346.50	8391.74

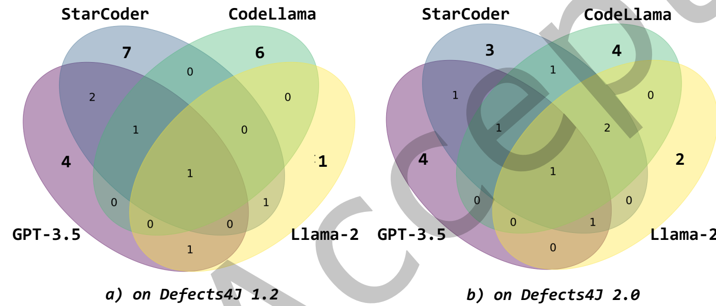

Fig. 3. GiantRepair uniquely repaired bugs when integrating with different LLMs.

Listing 4 show the LLM-generated patches and the developer patches for two different bugs. The root cause of the bug in Listing 3 is that inlining singleton getter method is not permitted. Therefore, it is necessary to verify whether the inlined method is a singleton or not (i.e., calling `getSingletonGetterClassName`). However, without the domain-specific knowledge regarding the used API, LLMs cannot generate the desired patch. In contrast, the LLM's patch provide a meaningful repair guidance by inserting an `IfStatement` to check `callNode`. GiantRepair successfully leveraged this insight by constructing a patch skeleton and performing static analysis to instantiate the skeleton via considering the type constraint and contexts. Ultimately, GiantRepair successfully repaired this bug.

```
private boolean canInline(Reference declaration,Reference initialization,Reference reference){
  ...
  Node callNode = reference.getParent();
  CodingConvention convention = compiler.getCodingConvention();
  ...
  //Developer patch: handle a special case by inserting a new if statement
+  if (convention.getSingletonGetterClassName(callNode) != null) {
+      return false;
+  }
  //LLM's patch
+  if (convention.getGetterMethod(callNode) != null) {
+      return false;
```

Just Accepted

```
+ }
  ...
}
```

Listing 3. Patch code of Closure-36 from Defects4J.

```
protected int findWrapPos(String text, int width, int startPos) {
  int pos;
  ...
  // look for the last whitespace character before (startPos+width)
  pos = startPos + width;
  while(...) (...)
  ...
  // Developer patch
  // if we didn't find the last whitespace, simply chop at (startPos+width)
  pos = startPos + width;
-  while ((pos <= text.length()) && ((c = text.charAt(pos)) != ' ') && (c != '\n') && (c != '\r')){
-      ++pos;
-   }
  // LLM's patch
-  pos = startPos + width;
-  while ((pos <= text.length()) && ((c = text.charAt(pos)) != ' ') && (c != '\n') && (c != '\r')) {
-      ++pos;
-  }
   return pos == text.length() ? -1 : pos;
}
```

Listing 4. Patch code of Cli-32 from Defects4J.

Regarding the faulty method shown in Listing 4, its purpose is to return the position of the last whitespace in the input `text`, starting at `startPos` and ending at `(startPos+width)`. If no whitespace is found, it should return the position of `(startPos+width)`. However, the faulty code erroneously continues searching for whitespace in the text beyond the length of `(startPos+width)` due to an incorrect while loop. To fix this bug, the entire while loop statement should be removed, as demonstrated. Unfortunately, without sufficiently understanding the context, the LLM-generated patch also removed the line `(pos=startPos+width;)`, resulting in errors. In contrast, GiantRepair, utilizing its adaptive patch instantiation method, successfully generated a patch that only removed the `while` loop statement, thereby fixing the bug. This is noteworthy since deletion can often lead to incorrect patches and is usually avoided by existing APR approaches [14, 72]. From these examples, we can see that GiantRepair effectively leverages program-specific knowledge (such as requiring `getSingletonGetterClassName()` in Closure-36) and contextual information (such as assigning a value to `pos`) to address bugs that are challenging for LLMs to repair.

5.2 RQ2: Effectiveness Compared with Baselines

(1) Performance with perfect localization. We compare our approach with the state-of-the-art APR tools that were also evaluated under the assumption of perfect fault localization. As explained in Section 4.4, GiantRepair takes a buggy function as input and leverages LLMs to generate the initial patches, while the baseline APR tools may take a buggy line as input. Table 4 displays the number of bugs that can be correctly fixed by each APR on both Defects4J v1.2 and v2.0. In particular, FitRepair [20] concurrently runs four LLMs to generate patches. For a fair comparison, we define two configurations: $\text{GR}_{\text{LLM}\times 4_{all}}$ and $\text{LLM} \times 4_{all}$. Configuration $\text{GR}_{\text{LLM}\times 4_{all}}$ denotes that GiantRepair takes the patches generated by the four LLMs as inputs (i.e., 200×4=800) for subsequent patch generation. Configuration $\text{LLM} \times 4_{all}$, on the other hand, uses the LLM-generated patches directly without the aid of GiantRepair. This configuration obviously requires more computing resources. To offer a fair comparison with other baselines, we developed another variant of $\text{GR}_{\text{LLM}\times 4_{part}}$, which only takes the first quarter of patches from each LLM as inputs (i.e., 50×4=200), resulting in the same number of candidate patches as using one LLM. Similarly, $\text{LLM} \times 4_{part}$ represents adopting the same quarter of LLMs' patches directly.

Just Accepted

Table 4. Repair results with perfect fault localization. In the table, GR represents our approach GiantRepair.

Project	#Bugs	$GR_{LLM\times4_{all}}$	$LLM\times4_{all}$	$GR_{LLM\times4_{part}}$	$LLM\times4_{part}$	FitRepair	Repilot	Tare	GAMMA	AlphaRepair	CURE	Recoder	TBar
Chart	16	8	7	7	7	8	6	11	9	8	9	10	9
Closure	93	32	20	30	20	29	21	22	20	22	13	20	15
Lang	42	14	11	11	10	17	15	13	10	11	9	10	10
Math	72	26	23	26	23	23	20	20	19	19	16	16	16
Time	16	1	1	1	1	3	2	3	1	3	1	3	2
Mockito	16	6	6	6	6	4	0	2	2	4	4	2	2
Defects4J v1.2	255	**87**	68	**81**	67	85	64	71	61	67	52	61	54
Closure	12	2	1	1	1	0	0	1	0	0	1	-	-
Cli	23	7	6	7	6	6	6	6	8	5	2	-	-
Codec	11	8	7	8	7	5	5	4	2	5	4	-	-
Collections	1	0	0	0	0	1	1	0	0	0	0	-	-
Compress	33	12	9	12	9	2	3	4	4	1	1	-	-
Csv	11	6	6	6	6	2	2	5	0	1	0	-	-
Gson	9	6	6	6	6	1	1	1	3	2	0	-	-
JacksonCore	13	8	5	8	5	3	3	2	2	3	2	-	-
JacksonDatabind	51	15	15	14	14	10	8	0	9	8	2	-	-
JacksonXml	4	1	0	1	0	0	0	0	0	0	0	-	-
Jsoup	53	18	17	18	17	13	17	14	10	9	4	-	-
JxPath	7	1	1	1	1	1	1	3	1	1	2	-	-
Defects4J v2.0	228	**84**	73	**82**	72	44	47	40	39	35	18	-	-
Total	483	**171**	141	**163**	139	129	111	111	100	102	70	-	-

Table 5. Repair results without perfect fault localization. X/Y denotes X correct and Y plausible patches.

Project	$\text{GiantRepair}_{LLM\times4_{all}}$	$\text{GiantRepair}_{LLM\times4_{part}}$	Tare	ITER	TBar	SimFix	Hanabi
Chart	7/10	7/10	11/14	8/12	7/10	4/5	1/3
Closure	16/33	15/33	12/23	15/20	6/10	5/5	-/-
Lang	12/19	11/18	12/19	7/7	4/11	6/9	1/1
Math	22/40	7/38	18/34	13/24	12/26	11/20	13/15
Time	1/3	1/3	2/3	2/3	1/2	1/1	2/2
Mockito	6/6	6/6	2/2	-/-	1/2	-/-	-/-
Total	**64/111**	**62/108**	57/95	45/66	31/61	27/40	17/21
P(%)	57.66	57.41	60.00	68.18	50.82	67.50	80.95

From the table, we can see that GiantRepair can not only significantly improve the correct fixes compared with using the LLM-generated patches directly, it also significantly outperforms all baseline APR tools. For example, compared with the best-performing FitRepair, GiantRepair successfully repaired 42 more bugs (171 *vs* 129). In particular, even only using the first quarter of LLM patches, GiantRepair can still outperform all the baseline methods by repairing as least 34 more bugs (163 *vs* 129). Specifically, GiantRepair performs consistently well on different benchmarks. On the contrary, the baseline APR tools tend to achieve better performance on Defects4J v1.2 than v2.0, indicating that our approach is more general and less likely to overfit certain benchmarks. As it will be presented in Section 6.3, when using an another new benchmark that was never used by previous studies, GiantRepair can still effectively repair a number of bugs.

```
static boolean isReduceableFunctionExpression(Node n) {
// Developer patch
+   return NodeUtil.isFunctionExpression(n)
+     && !NodeUtil.isGetOrSetKey(n.getParent());
-    return NodeUtil.isFunctionExpression(n);
// LLM's patch
+   return NodeUtil.isFunctionExpression(n)
+     && !NodeUtil.isNameDeclaration(n.getParent());
```

Just Accepted

```
-     return NodeUtil.isFunctionExpression(n);
}
```

Listing 5. Patch code of Closure-55 from Defects4J.

Similarly, we further analyzed the complementary between our approach and the baseline APR tools. In particular, since not all the 24 baselines were evaluated on Defects4J v2.0, we only compared their results on Defects4J v1.2. Figure 4 shows the overlaps of correct fixes by different APR tools. Specifically, we first compare our approach with the top-4 baseline APR tools (see Table 4) in the left figure. Then, we compare our approach with these 11 SOTA APR tools (explained in Section 4.3) in the right figure. From the figures we can see that GiantRepair has the capability to repair more new bugs compared with existing APR tools. In particular, compared with all 11 SOTA baselines, GiantRepair can still repair 22 unique bugs, indicating its high effectiveness. For instance, besides the two bugs introduced in Section 2, the bug shown in Listing 5 is an another example which can be correctly repaired by GiantRepair but cannot by all the baseline APR tools. To repair this bug, a new condition should be inserted. As a result, the patch skeleton (i.e., inserting a method call as the condition) generated from the LLM's patch can offer the valuable insights and effectively confine the search space.

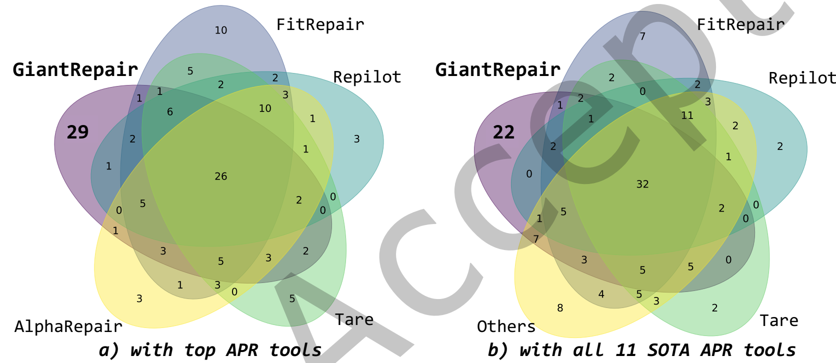

Fig. 4. Unique bug repairs compared to SOTA baselines on Defects4J v1.2

In contrast, previous APR tools often struggled to find modifications similar to the correct fixes in the vast search space. Additionally, many existing LLM-based tools are designed to fix only single-line or single-hunk bugs, limiting their ability to address complex issues that require modifications across multiple lines. To demonstrate GiantRepair's capability in fixing more complex bugs, we compared the number of lines of code modified in patches generated by GiantRepair with those generated by other LLM-based APR methods. Figure 5 shows the number distribution of changed code lines in the patches generated by different approaches. We observe that patches from existing LLM-based APR tools typically involve fewer than 3 lines of code. In contrast, patches generated by GiantRepair modify an average of 4.9 lines of code, nearly 1.94 times that of Repilot (changes on average 2.53 lines of code). This result further demonstrates the promise of combining the strengths of LLM and static analysis techniques.

(2) Performance with imperfect localization. As mentioned in Section 1, existing studies usually evaluated the performance of LLM-based APR tools under the assumption of perfect fault localization. In this experiment, we further evaluate GiantRepair in a more realistic application scenario with imperfect fault localization. Specifically, we used the automated fault localization results as introduced in Section 4.4. Therefore, we compared it with baseline approaches that were also evaluated under the same setting. Table 5 shows the repair results of

Just Accepted

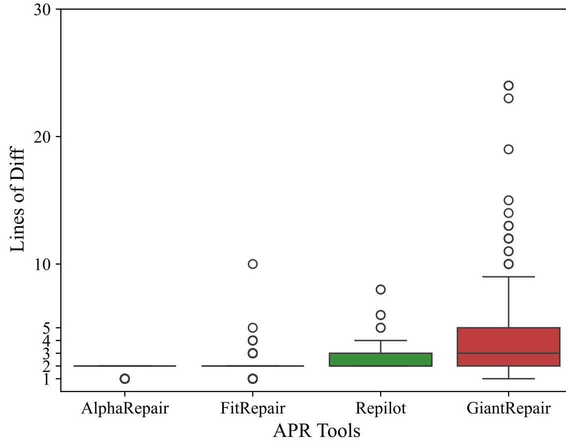

Fig. 5. Numbers of changed lines in correct patches.

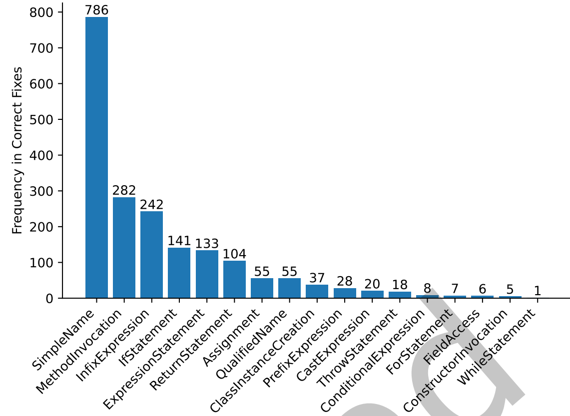

Fig. 6. Rules used in GiantRepair's correct patches.

different methods on Defects4J v1.2 because all the baselines consistently used this benchmark. Moreover, we also report their patch precision like the baselines did.

Table 6. Component contribution of GiantRepair

Based LLM	Variant	#Plausible Patches ($\mathcal{P}$)	#Correct Patches ($\mathcal{C}$)	%Precision (=$\mathcal{P}/\mathcal{C}$)
GPT-3.5	GiantRepair$_{w/o-context}$	93	47	50.54%
	GiantRepair$_{w/o-adaptive}$	99	51	51.52%
	GiantRepair$_{w/o-rank}$	93	47	50.54%
	GiantRepair$_{GPT-3.5}$	102	53	51.96%
StarCoder	GiantRepair$_{w/o-context}$	105	46	43.81%
	GiantRepair$_{w/o-adaptive}$	107	49	45.79%
	GiantRepair$_{w/o-rank}$	101	51	50.50%
	GiantRepair$_{StarCoder}$	104	55	52.88%

From the result we can conclude that GiantRepair can also achieve better repair performance when using the imperfect fault localization than the best-performing APR. Specifically, GiantRepair successfully repaired 64 bugs, which is even close to existing LLM-based APR tools with perfect fault localization, such as Repilot and AlphaRepair. This is partially attributed to the reason that GiantRepair relies on a coarse-grained function-level fault localization, which is much easier than that at the line level. However, the results also show that the powerful code generation ability of LLMs may also increase the risk of generating incorrect patches (i.e., low patch precision) due to the issue of weak tests [71, 73, 75].

5.3 RQ3: Contribution of Each Component in GiantRepair

In this section, we experimentally analyze the contribution of each component in GiantRepair. As introduced in Section 4.1, we have developed a set of variants of GiantRepair for systematically conducting this ablation study:

- GiantRepair$_{w/o-context}$ replaces the context-aware patch instantiation in GiantRepair with a random method. Specifically, to address a more practical scenario, rather than randomly selecting code elements

Just Accepted

from the entire project to fill the code skeletons, we randomly selected from all *accessible* code elements at the current position. This strategy fills the skeletons without considering the type constraints present within them.
- GiantRepair$_{w/o-adaptive}$ will randomly select modifications from LLM patches, rather than preferring the coarse-grained modifications.
- GiantRepair$_{w/o-rank}$ evaluates the generated patches in the order of generation, rather than ranking by the similarities with original buggy code.

In particular, in this experiment, we adopted GPT-3.5 (a closed-source LLM) and StarCoder (an open-source model, which demonstrated the best performance among other selected open-source LLMs in Table 2) as the representative LLMs. Besides, we conducted this ablation study on Defects4J 1.2 (including 255 single-function bugs) to save time. Table 6 presents the experimental results of the original GiantRepair (i.e., GiantRepair$_{GPT-3.5}$ and GiantRepair$_{StarCoder}$ when adopting GPT-3.5 and StarCoder, respectively) and their variants introduced above. According to the experimental results, we observe that: **1) Every component in GiantRepair significantly contributed to its overall effectiveness.** The results manifest that the absence of any one component will lead to a reduced number of both plausible and correct patches. Specifically, the context-aware patch instantiation, adaptive modification application and patch ranking contributed 9, 6 and 4 more correct fixes for StarCoder, and 6, 2, 6 more correct fixes for GPT-3.5, respectively. **2) The context-aware patch instantiation is the most effective component in GiantRepair.** We find that considering the context constraints for patch instantiation significantly enhances the repair capabilities of GiantRepair, which again demonstrates that our approach is effective for advancing LLMs' repair ability by incorporating static analysis for patch generation.

Finally, to investigate the contribution of each abstraction rule in the repair process of GiantRepair, we analyzed the frequency of each rule that was involved in all the correct fixes of GiantRepair. The results is shown in Figure 6. Note that one fix may involve multiple same or different rules. For example, abstracting an `if` statement may also need to abstract an infix expression in its condition. The *x-axis* presents the associated AST node types of the rules. From the figure we can see that most rules contributed to the final correct fixes. In particular, MethodInvocation, InfixExpression and IfStatement are the most frequently used ones except for SimpleName, which aligns with existing conclusions [14, 15, 76]. Moreover, benefited from the patch skeleton, GiantRepair is able to repair bugs that require complex modifications, such as inserting a completely new `for` statement. In summary, all components are essential for the overall performance of GiantRepair. And the abstraction rules in GiantRepair are most effective.

6 DISCUSSION

6.1 Unique Repairs Compared with 24 Existing APR Tools

To investigate whether our approach can significantly advance the research of APR and repair unique bugs, we further compared GiantRepair with 24 diverse APR tools in this section. Specifically, besides the 11 APR tools explained in Section 4.3, the other 13 APR tools are as follows: one LLM-based method (Fine-tuned UniXcoder [77]), two deep learning-based methods (KNOD [78], SelfAPR [79]), five heuristic search-based methods (PraPR [80], CapGen [13], jGenProg [81], jKali [6], jMutRepair [6]), three template-based methods (FixMiner [47], AVATAR [16], SketchFix [12]), and two constraint solving-based methods (JAID [82], NOPOL [2]). These baseline methods encompass the majority of methodologies employed in recent APR tools. Comparing GiantRepair against these varied baselines strengthens the reliability of the evaluation conclusions. The details of all baselines are presented in Table 7.

In particular, we compared the bugs that were correctly repaired by different APR tools on the Defects4J v1.2 dataset since all the compared baselines were consistently evaluated on it. Figure 7 presents the comparing results.

Just Accepted

Table 7. Details of APR Tools for comparing unique repairs

APR Tool	Year	Methodology	APR Tool	Year	Methodology
ITER	2024	Deep Learning-based	FixMiner	2020	Template-based
KNOD	2023	Deep Learning-basd	AVATAR	2019	Template-based
Tare	2023	Deep Learning-based	TBar	2019	Template-based
Fine-tuned UniXcoder	2023	LLM-based	PraPR	2019	Heuristic Search-based
FitRepair	2023	LLM-based	CapGen	2018	Heuristic Search-based
GAMMA	2023	LLM-based	SimFix	2018	Heuristic Search-based
Repilot	2023	LLM-based	SketchFix	2018	Template-based
AlphaRepair	2022	LLM-based	JAID	2017	Constraint Solving-based
Hanabi	2022	Deep Learning-based	jGenProg	2017	Heuristic Search-based
SelfAPR	2022	Deep Learning-based	jKali	2016	Heuristic Search-based
CURE	2021	Deep Learning-based	jMutRepair	2016	Heuristic Search-based
Recoder	2021	Deep Learning-based	NOPOL	2016	Constraint Solving-based

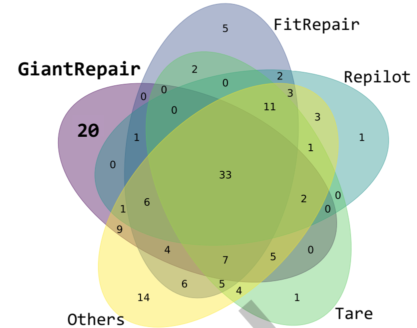

Fig. 7. Unique repairs

According to this figure, even when compared to all the 24 baselines, GiantRepair still offers 20 unique repairs, highlighting its effectiveness.

6.2 Comparing GiantRepair with GPT-4

As reported in RQ1, GiantRepair can significantly increase the correct fixes compared with directly using the LLM-generated patches, including many unique fixes. However, since the repair ability of LLMs may also increase over the time. To investigate whether or not GiantRepair is still effective for repairing unique bugs when comparing to the most advanced LLMs, we conducted another experiment with a more powerful LLM. For cost-efficiency, we selected GPT-4o-mini-2024-07-18 [83] (According to the evaluation results from OpenAI [84], this LLM scored 87.2 on the HumanEval dataset [85] and has competitive coding abilities comparable to GPT-4 and GPT-4o [86]). For each of the 483 single-function bugs from Defects4J, we used GPT-4o-mini via API requests to generate 200 patches. In this process, we used the same configuration and *prompt* introduced in Section 4.4. The experimental results reveal that GPT-4o-mini can repair 105 bugs (51 from Defects4J v1.2 and 54 from Defects4J v2.0), while GiantRepair, utilizing only StarCoder, can repair 109 bugs (55 from Defects4J v1.2 and 54 from Defects4J v2.0). We further examined the 30 bugs that GiantRepair successfully fixed but were not resolved by the four studied LLMs. It turns out that GPT-4o-mini could fix only six of these, leaving 24 bugs unresolved. This suggests that GiantRepair remains valuable even when compared to the latest LLM.

6.3 Data leakage

Data leakage is a common concern for LLM-based APR tools. To investigate its impact on our conclusion, we first performed a manual analysis by following prior studies [20, 29, 38]. Specifically, we analyzed whether the patched code has been used as the training data of the LLMs. We selected StarCoder as the representative as it is the only one that published its training data. The results show that among the 109 (=55+54) correct patches generated by $\text{GiantRepair}_{StarCoder}$, 23 of them were included in StarCoder's training data. That is, a large majority of the correct patches 86/109 were not seen by StarCoder previously, and thus the effectiveness should come from the abilities of StarCoder and our patch generation method themselves, rather than the data leakage.

Moreover, to offer a more strong evidence of GiantRepair's effectiveness while avoiding data leakage, we conducted two extra experiments. In the first experiment, we adopted the GrowingBugs [87–89] dataset by removing the projects involved in StarCoder's training data (34/250 projects left) and the bugs that require cross-function modifications in these 34 projects (51/122 bugs left). Finally, we employed GiantRepair with StarCoder to repair them. The results demonstrate that GiantRepair correctly repaired 10 out of the 51 bugs.In the second experiment, we employed the HumanEval-Java [37] dataset, which consists of 163 bugs, created by manually converting Python code from HumanEval dataset [85] to Java and intentionally injecting bugs into the

Just Accepted

converted code. Additionally, it does not temporally overlap with the chosen LLMs, thereby preventing the data leakage issue. Similarly, we also employed GiantRepair with StarCoder to repair these 163 bugs. The results demonstrate that GiantRepair correctly repaired 143 out of the 163 bugs. These results further confirm the effectiveness of our approach while effectively mitigating the risk of data leakage impacting its performance.

6.4 Limitation

First, our experiments involved four LLMs (GPT-3.5-turbo, Llama-2, StarCoder and CodeLlama) and one programming language (Java). While this provides valuable insights, it still represents a limited scope in demonstrating the full capabilities of GiantRepair, which is theoretically capable of utilizing any generative LLMs' generated patches in a wide range of programming languages. A second limitation of GiantRepair lies in its time efficiency. The time LLMs take to generate patches is not accounted for in the patch generation process of GiantRepair. Third, GiantRepair currently abstracts and instantiates code skeletons from single LLM-generated patch, while during our experiments, we found some bugs' correct fixes may be distributed across multiple patches. Investigating how to integrate these meaningful fixes from various patches could be a valuable focus for future development. Finally, while there are existing approaches that explore the patch search space, the performance of GiantRepair compared to these methods has yet to be evaluated. We intend to address this in future work.

6.5 Threats to validity

Internal threats to validity. Manually reviewing all plausible patches to identify correct patches that are semantically consistent with the reference patch is an internal threat to the validity of our work. Following common APR practice, we perform a careful analysis of each plausible patch and have published our full set of correct and plausible patches. Another internal threat to validity is the LLMs used in our paper may trained on open-source code from GitHub, potentially overlapping with Defects4J dataset. To address this, we conduct a detailed discussion in Section 6.3 and employ a new dataset to demonstrate the effectiveness of GiantRepair.

External threats to validity. The primary external threat to validity comes from the evaluation datasets we used, and the performance of GiantRepair may not be generalized to other datasets. To address this, we use two different datasets to evaluate GiantRepair: Defects4J v1.2, Defects4J v2.0 and demonstrate that GiantRepair is still effective and able to achieve state-of-the-art results. In the future, we plan to evaluate GiantRepair on more datasets across multiple programming languages to address this threat.

7 RELATED WORK

In this section, we introduce the most related works to this paper.

7.1 Automated Program Repair

Numerous APR approaches have been proposed to address the vast search space of bug fixes, aiming to improve the quality of generated patches. Traditional APR tools can be categorized into three types: heuristic-based [1, 7–10, 13, 14], template-based [6, 11, 12, 16–18] and constraint-based [2–5]. Among these, several heuristic-based methods are most related to GiantRepair, as they also utilize contextual information for patch generation, such as RETE [90], FixMiner [47], SimFix [14] and CapGen [13].

RETE [90] abstracts variables within patch templates produced by existing APR tools and predicts embeddings for these variables. Candidate variables are ranked based on the similarity between their embeddings and the predicted embedding, and they are subsequently inserted into the template to create patches. In contrast, GiantRepair utilizes LLMs to generate patch templates and can abstract a broader range of code elements, such as variable names, class names, and operators. It then instantiates patches by calculating the similarity with LLM-generated modifications. This broader abstraction capability enables GiantRepair to handle more

Just Accepted

complex code structures and adaptively produce more varied and contextually appropriate patches. FixMiner [47] incorporates contextual information to search useful patterns from historical changes, generating patches by filling and applying these patterns. GiantRepair differs from FixMiner in several key aspects. Firstly, FixMiner derives repair guidance by searching for similar code in historical changes, GiantRepair directly extracts modifications from patches generated by LLMs. Secondly, while FixMiner utilizes code elements from the file containing the bug to fill the pattern during patch generation, GiantRepair imposes strict constraints on type and AST node type during the construction of skeletons and searches for code elements across the entire project. Thirdly, unlike FixMiner, which applies only one pattern per patch, GiantRepair allows the combination of multiple modifications. Unlike SimFix [14] and CapGen [13], which reduce the search space by identifying similar or frequent code elements within the context of the buggy project, GiantRepair adopts a distinct approach by extracting modifications from patches generated by LLMs. Furthermore, GiantRepair offers greater flexibility in extracting patch skeletons, which are subsequently filled through a context-aware patch generation process, as opposed to simply filling pre-defined patterns or directly reusing referenced code.

Recently, many works also employ pre-trained models for APR tasks, treating program repair as a task of code generation. AlphaRepair [29] is the first to employ LLMs for infilling-style APR, masking the buggy code and utilizing CodeBERT [91] to directly replace the masked tokens with correct tokens to generate patches. Several studies [19, 36, 37] have explored the efficacy of applying various types of LLMs directly to APR tasks. While these works highlight the potential of LLMs in APR, they typically treat the LLM as a black box and directly utilize the patches generated by these models. However, it has become evident that LLMs may not always produce correct patches without a comprehensive understanding of the project's full context under repair. To address this issue, FitRepair [20] leverages the plastic surgery hypothesis by fine-tuning two CodeT5 models specifically on the buggy project and incorporates relevant identifiers from the buggy project into the prompt. It engages four CodeT5 models simultaneously to generate patches. Similarly, Repilot [38] fuses CodeT5 [92] with a completion engine to improve repair performance. Additionally, GAMMA [39] employs CodeBERT [91] and UniXcoder [31] to fill a set of predefined repair templates. All these methods differ from GiantRepair as none of them modify the patches generated by LLMs like GiantRepair whereas reuse the LLM-generated patches directly.

7.2 Automated Code Template Extraction

7.2.1 Extracting Templates from Code Changes. Extracting templates from code changes has many potential uses, such as systematic program editing, refactoring and program repair. Many existing works [11, 93–98] extract templates from multiple examples with similar code changes. Spdiff [93, 94] and LASE [95] extract transformation templates from a set of examples and take the most common part of templates as a transformation pattern. REFAZER [96] searches for a transformation template that is consistent with all provided examples. Genesis [11] extracts AST templates from existing patches that can cover all examples. Phoenix [97] extracts repair tempaltes via clustering to fix bugs reported by static analyzers. CPATMINER [98] extracts semantic code change graphs from a large number of repositories. All these approaches requires the duplication of very similar examples, which are often difficult to obtain in practice. Different from them, GiantRepair generates patch skeleton by comparing with only one LLM generated patch.

To overcome the dependency on duplicate examples, SYDIT [99] and GenPat [18] were proposed. Specifically, SYDIT extracts code change templates relying on the predefined rules, i.e., removing all variable and method names while preserving the code structures. In contrast, GenPat extracts code change templates by analyzing the distribution of certain code elements in a code corpus. Different from these approaches, GiantRepair removes all the concrete code elements away (including operators) and preserves the type information of variables and expressions, whereas these existing methods do not. Then, GiantRepair incorporates a context-aware patch instantiation process for patch generation. Please note that these differences are critical since the type information

Just Accepted

can effectively refine the search space and promote the repair ability as demonstrated in our evaluation (*ref.* GiantRepair$_{w/o-context}$). Nevertheless, these existing approaches are orthogonal to GiantRepair and potentially can be combined with GiantRepair for optimizing the constructed patch skeletons.

7.2.2 Extracting Templates from LLM. Recently, some works [100, 101] in the field of program synthesis have also explored extracting templates with LLMs. Specifically, these works adjust the weights of various grammar rules used for synthesizing programs by analyzing the content generated by LLMs. In essence, the templates extracted from the LLM-generated programs are grammar rules applied for program generation. Different from them, GiantRepair derives program repair templates by directly abstracting the specific modifications made by LLMs. It then generates executable patches by instantiating these templates with concrete program elements.

8 CONCLUSION

In this paper, we have proposed GiantRepair, a novel automated program repair approach. Specifically, GiantRepair leverages LLM-generated patches for patch skeleton construction and constraining the patch space, and then incorporates a context-aware skeleton instantiation process for generating high-quality patches tailored to specific programs. We have conducted two large-scale experiments for evaluating the effectivenes of GiantRepair. The results demonstrated that it not only improved the correct fixes compared with pure LLMs, but also outperformed the latest state-of-the-art APR tools.

ACKNOWLEDGMENTS

We thank the editors and anonymous reviewers for their constructive suggestions to help improve the quality of this paper. This work was supported by the National Natural Science Foundation of China under Grant Nos. 62202324.

REFERENCES

[1] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, “Genprog: A generic method for automatic software repair,” *Ieee transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2011.
[2] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, “Nopol: Automatic repair of conditional statement bugs in java programs,” *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2016.
[3] 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*, 2017, pp. 593–604.
[4] F. Long and M. Rinard, “Staged program repair with condition synthesis,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 166–178.
[5] S. Mechtaev, J. Yi, and A. Roychoudhury, “Angelix: Scalable multiline program patch synthesis via symbolic analysis,” in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 691–701.
[6] M. Martinez and M. Monperrus, “Astor: A program repair library for java,” in *Proceedings of the 25th international symposium on software testing and analysis*, 2016, pp. 441–444.
[7] X.-B. D. Le, D. Lo, and C. L. Goues, “History driven program repair,” *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, pp. 213–224, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:8844190
[8] 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.
[9] Q. Xin and S. P. Reiss, “Leveraging syntax-related code for automated program repair,” in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 660–670.
[10] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, “Precise condition synthesis for program repair,” in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 416–426.
[11] F. Long, P. Amidon, and M. Rinard, “Automatic inference of code transforms for patch generation,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 727–739.
[12] J. Hua, M. Zhang, K. Wang, and S. Khurshid, “Towards practical program repair with on-demand candidate generation,” in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 12–23.

Just Accepted

[13] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 1–11.

[14] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, 2018, pp. 298–309.

[15] 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*, 2019, pp. 31–42.

[16] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandè, "Avatar: Fixing semantic bugs with fix patterns of static analysis violations," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 2019, pp. 1–12.

[17] 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.

[18] J. Jiang, L. Ren, Y. Xiong, and L. Zhang, "Inferring program transformations from singular examples via big code," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 2019, pp. 255–266.

[19] 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 (ICSE 2023). Association for Computing Machinery*, 2023.

[20] C. S. Xia, Y. Ding, and L. Zhang, "The plastic surgery hypothesis in the era of large language models," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 2023, pp. 522–534.

[21] 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.

[22] 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.

[23] 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.

[24] 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 *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2021, pp. 341–353.

[25] 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 (ICSE).* IEEE, 2021, pp. 1161–1173.

[26] 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.

[27] Q. Zhu, Z. Sun, W. Zhang, Y. Xiong, and L. Zhang, "Tare: Type-aware neural program repair," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE).* IEEE, 2023, pp. 1443–1455.

[28] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, "Vulrepair: a t5-based automated software vulnerability repair," in *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*, 2022, pp. 935–947.

[29] C. Xia and L. Zhang, "Less training, more repairing please: revisiting automated program repair via zero-shot learning," *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:250627519

[30] S. Feng and C. Chen, "Prompting is all you need: Automated android bug replay with large language models," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[31] 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.

[32] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[33] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. Hoi, "Codet5+: Open code large language models for code understanding and generation," *arXiv preprint arXiv:2305.07922*, 2023.

[34] T. Ahmed and P. Devanbu, "Few-shot training llms for project-specific code-summarization," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–5.

[35] S. D. Kolak, R. Martins, C. Le Goues, and V. J. Hellendoorn, "Patch generation with language models: Feasibility and scaling behavior," in *Deep Learning for Code Workshop*, 2022.

[36] 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.

[37] 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.

Just Accepted

[38] 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.
[39] 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.
[40] A. Silva, S. Fang, and M. Monperrus, "Repairllama: Efficient representations and fine-tuned adapters for program repair," *arXiv preprint arXiv:2312.15698*, 2023.
[41] X. Gu, M. Chen, Y. Lin, Y. Hu, H. Zhang, C. Wan, Z. Wei, Y. Xu, and J. Wang, "On the effectiveness of large language models in domain-specific code generation," *ACM Transactions on Software Engineering and Methodology*, 2024.
[42] K. Jin, C.-Y. Wang, H. V. Pham, and H. Hemmati, "Can chatgpt support developers? an empirical evaluation of large language models for code generation," in *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. IEEE, 2024, pp. 167–171.
[43] C. S. Xia and L. Zhang, "Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt," *arXiv preprint arXiv:2304.00385*, 2023.
[44] 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, pp. 437–440.
[45] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 313–324.
[46] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 295–306.
[47] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *Empirical Software Engineering*, vol. 25, pp. 1980–2024, 2020.
[48] "Eclipse JDT Core," https://www.eclipse.org/jdt/core/, Eclipse Foundation, 2024, accessed: 2023-08.
[49] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1469–1481.
[50] Y.-A. Xiao, C. Yang, B. Wang, and Y. Xiong, "Expressapr: Efficient patch validation for java automated program repair systems," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, 2023, pp. 1–4.
[51] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.
[52] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.
[53] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.
[54] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.
[55] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.
[56] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Transactions on Software Engineering*, 2023.
[57] Z. Zheng, K. Ning, Y. Wang, J. Zhang, D. Zheng, M. Ye, and J. Chen, "A survey of large language models for code: Evolution, benchmarking, and future trends," *arXiv preprint arXiv:2311.10372*, 2023.
[58] 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.
[59] Y. Xiong and B. Wang, "L2s: A framework for synthesizing the most probable program under a specification," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–45, 2022.
[60] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," *Advances in Neural Information Processing Systems*, vol. 36, 2024.
[61] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Evaluating large language models in class-level code generation," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.
[62] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand, "Lost in translation: A study of bugs introduced by large language models while translating code," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.
[63] "Hugging face," https://huggingface.co, 2023, accessed: 2023-10-06.
[64] "Gpt3.5 api," https://platform.openai.com/docs/models/gpt-3-5-turbo, 2023, accessed: 2023-05.
[65] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The curious case of neural text degeneration," *arXiv preprint arXiv:1904.09751*, 2019.

Just Accepted

[66] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06).* IEEE, 2006, pp. 39–46.
[67] A. Riboira and R. Abreu, "The gzoltar project: A graphical debugger interface," in *International Academic and Industrial Conference on Practice and Research Techniques.* Springer, 2010, pp. 215–218.
[68] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 332–347, 2019.
[69] J. Liang, R. Ji, J. Jiang, S. Zhou, Y. Lou, Y. Xiong, and G. Huang, "Interactive patch filtering as debugging aid," in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 239–250.
[70] 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. [Online]. Available: https://doi.org/10.1145/3092703.3092718
[71] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 789–799.
[72] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 727–738.
[73] H. Tian, Y. Li, W. Pian, A. K. Kaboré, K. Liu, A. Habib, J. Klein, and T. F. Bissyandé, "Predicting patch correctness based on the similarity of failing test cases," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 4, 2022.
[74] "Pricing of openai's llms," https://openai.com/api/pricing/, 2024, accessed: 2024-10-15.
[75] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015, 2015, p. 24–36.
[76] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object-oriented program repair," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 648–659.
[77] K. Huang, X. Meng, J. Zhang, Y. Liu, W. Wang, S. Li, and Y. Zhang, "An empirical study on fine-tuning large language models of code for automated program repair," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 2023, pp. 1162–1174.
[78] N. Jiang, T. Lutellier, Y. Lou, L. Tan, D. Goldwasser, and X. Zhang, "Knod: Domain knowledge distilled tree decoder for automated program repair," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE).* IEEE, 2023, pp. 1251–1263.
[79] 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*, 2022, pp. 1–13.
[80] A. Ghanbari and L. Zhang, "Prapr: Practical program repair via bytecode mutation," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 2019, pp. 1118–1121.
[81] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset," *Empirical Software Engineering*, vol. 22, pp. 1936–1964, 2017.
[82] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE).* IEEE, 2017, pp. 637–647.
[83] "Gpt4o-mini api," https://platform.openai.com/docs/models/gpt-4o-mini, 2024, accessed: 2024-10-15.
[84] "simple-eval," https://github.com/openai/simple-evals, 2024, accessed: 2024-10-15.
[85] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.
[86] "Gpt4 api," https://platform.openai.com/docs/models, 2024, accessed: 2024-10-15.
[87] Y. Jiang, H. Liu, N. Niu, L. Zhang, and Y. Hu, "Extracting concise bug-fixing patches from human-written patches in version control systems," in *IEEE/ACM 43rd International Conference on Software Engineering (ICSE 2021).* Los Alamitos, CA, USA: IEEE Computer Society, may 2021, pp. 686–698. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICSE43902.2021.00069
[88] Y. Jiang, H. Liu, X. Luo, Z. Zhu, X. Chi, N. Niu, Y. Zhang, Y. Hu, P. Bian, and L. Zhang, "Bugbuilder: An automated approach to building bug repository," *IEEE Transactions on Software Engineering*, pp. 1–22, 2022.
[89] Y. Jiang, H. Liu, Y. Zhang, W. Ji, H. Zhong, and L. Zhang, "Do bugs lead to unnaturalness of source code?" in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 1085–1096. [Online]. Available: https://doi.org/10.1145/3540250.3549149
[90] N. Parasaram, E. T. Barr, and S. Mechtaev, "Rete: Learning namespace representation for program repair," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE).* IEEE, 2023, pp. 1264–1276.
[91] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.
[92] 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.

Just Accepted

[93] J. Andersen and J. L. Lawall, "Generic patch inference," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 337–346.
[94] J. Andersen, A. C. Nguyen, D. Lo, J. L. Lawall, and S.-C. Khoo, "Semantic patch inference," in *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 382–385.
[95] N. Meng, M. Kim, and K. S. McKinley, "Lase: locating and applying systematic edits by learning from examples," in *2013 35th International Conference on Software Engineering (ICSE).* IEEE, 2013, pp. 502–511.
[96] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann, "Learning syntactic program transformations from examples," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE).* IEEE, 2017, pp. 404–415.
[97] R. Bavishi, H. Yoshida, and M. R. Prasad, "Phoenix: Automated data-driven synthesis of repairs for static analysis violations," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 613–624.
[98] H. A. Nguyen, T. N. Nguyen, D. Dig, S. Nguyen, H. Tran, and M. Hilton, "Graph-based mining of in-the-wild, fine-grained, semantic code change patterns," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE).* IEEE, 2019, pp. 819–830.
[99] N. Meng, M. Kim, and K. S. McKinley, "Sydit: Creating and applying a program transformation from an example," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 440–443.
[100] Y. Li, J. Parsert, and E. Polgreen, "Guiding enumerative program synthesis with large language models," in *International Conference on Computer Aided Verification.* Springer, 2024, pp. 280–301.
[101] S. Barke, E. A. Gonzalez, S. R. Kasibatla, T. Berg-Kirkpatrick, and N. Polikarpova, "Hysynth: Context-free llm approximation for guiding program synthesis," *arXiv preprint arXiv:2405.15880*, 2024.

Just Accepted