GAMMA: Revisiting Template-based Automated Program Repair via Mask Prediction

Quanjun Zhang
State Key Laboratory for Novel
Software Technology
Nanjing University, China
quanjun.zhang@smail.nju.edu.cn

Chunrong Fang*
State Key Laboratory for Novel
Software Technology
Nanjing University, China
fangchunrong@nju.edu.cn

Tongke Zhang
State Key Laboratory for Novel
Software Technology
Nanjing University, China
201250032@smail.nju.edu.cn

Bowen Yu
State Key Laboratory for Novel
Software Technology
Nanjing University, China
201250070@smail.nju.edu.cn

Weisong Sun
State Key Laboratory for Novel
Software Technology
Nanjing University, China
weisongsun@smail.nju.edu.cn

Zhenyu Chen*
State Key Laboratory for Novel
Software Technology
Nanjing University, China
zychen@nju.edu.cn

Abstract—Automated program repair (APR) aims to fix software bugs without human intervention and plays a crucial role in software development and maintenance. Template-based APR has been widely investigated and shown promising results. However, it is challenging for template-based APR to select the appropriate donor code, which is an important repair ingredient for generating candidate patches. Inappropriate donor code may cause plausible but incorrect patch generation even with correct fix patterns, limiting the repair performance.

In this paper, we aim to revisit template-based APR, and propose GAMMA, to directly leverage large pre-trained language models for donor code generation. Our main insight is that instead of retrieving donor code in the local buggy file, we can directly predict the correct code tokens based on the context code snippets and repair patterns by a cloze task. Specifically, (1) GAMMA revises a variety of fix templates from state-of-the-art template-based APR techniques (i.e., TBar) and transforms them into mask patterns. (2) GAMMA adopts a pre-trained language model to predict the correct code for masked code as a fill-in-the-blank task. Although our idea is general and can be built on various existing pre-trained language models, we have implemented GAMMA as a practical APR tool based on the recent UniXcoder model. The experimental results demonstrate that GAMMA correctly repairs 82 bugs on Defects4J-v1.2, which achieves 20.59% (14 bugs) and 26.15% (17 bugs) improvement over the previous state-of-the-art template-based approach TBar and learning-based one Recoder. Furthermore, GAMMA repairs 45 bugs and 22 bugs from the additional Defects4J-v2.0 and QuixBugs, indicating the generalizability of GAMMA in addressing the dataset overfitting issue. We also prove that adopting other pre-trained language models can provide substantial advancement, e.g., CodeBERT-based and ChatGPT-based GAMMA is able to fix 80 and 67 bugs on Defects4J-v1.2, indicating the scalability of GAMMA. Overall, our study highlights the promising future of adopting pre-trained models to generate correct patches on top of fix patterns in practice.

Index Terms—Automated Program Repair, Fix Pattern, Pre-trained Model, LLM4SE

*Chunrong Fang and Zhenyu Chen are the corresponding authors.

I. INTRODUCTION

The complexity and size of modern software systems are continuously enlarging, leading to the soaring number of software bugs [1], [2]. Software bugs have detrimental effects on software development, as they give users an annoying experience, and sometimes can cause huge financial losses to developers [3]. A considerable amount of time and budget is spent on identifying and fixing such software bugs manually [4]. To facilitate the process of manual debugging, automated program repair (APR), which aims to generate correct patches for identified buggy code snippets automatically, is getting growing attention from both academia and industry [5], [6], such as Meta [7], Google [8] and Microsoft [9], [10].

In the literature, a variety of APR techniques have been proposed to generate patches, such as heuristic-based [11], [12], constraint-based [13], [14], template-based [15], [16]. Among these traditional APR techniques, template-based APR, which employs repair patterns hand-crafted by human experts to transform buggy code snippets into correct ones, has been widely investigated and recognized as state-of-the-art [17]–[19]. Candidate patches are usually generated by leveraging two kinds of repair ingredients (i.e., fix patterns and donor code) that are found in existing code bases. The repair pattern represents common code change actions (e.g., insertion of an If statement) and donor code represents code fragments (e.g., identifier tokens such as method names) to concretize patches guided by abstract patterns. A mass of studies has been dedicated to template extraction schemes, such as manually extracted templates and automatically mining templates [20]–[23]. For example, state-of-the-art template-based APR tool TBar [17] focuses on the local buggy file and leverages the context of buggy code to prune away irrelevant donor code. Previous works [24], [25] have shown a considerable number of bugs cannot be fixed because the relevant donor code is not available in the local file. Therefore, TBar may fail to generate

correct patches with inappropriate donor code although the fix pattern matches with correct code change actions.

In this paper, we propose a novel template-based APR tool called GAMMA by combining the advances of fix patterns and pre-trained language models. The key insight is that considering pre-trained models can acquire adequate general knowledge about programming language from all possible open-source projects in the wild, we can directly employ such models to retrieve relevant donor code from the fix pattern and surrounding code context. In particular, we first collect and summarize a super-set of fix patterns drawn from previous template-based work (e.g., TBar). We then transform these fix templates into hole-filling-based patterns, which replace donor code with several masked tokens to be filled. Finally, we perform a mask prediction task on the hole-filling-based fix patterns with the help of pre-trained models in a fill-in-the-blank manner, i.e., predicting the correct donor code for the masked tokens. Although GAMMA is conceptually generalizable to various pre-trained models, we have implemented GAMMA on top of one recent pre-trained language model, UniXcoder [26]. Unlike current template-based APR tools which usually retrieve fix ingredients in the local buggy file, GAMMA directly utilizes generic knowledge pre-trained with millions of code snippets from open-source projects, allowing it to provide a variety of donor code to fix different bugs.

We conduct extensive experiments to compare GAMMA with state-of-the-art APR approaches (including both traditional and learning-based ones) on the widely-adopted Defects4J-v1.2 benchmark. The experimental results demonstrate that GAMMA is able to outperform all existing APR approaches, improving the number of correctly-fixed bugs to 82 with a precision of 81.19%, and 14 unique bugs that no prior work can fix, which is a new frontier in the APR field. Besides, GAMMA fixes 45 and 22 bugs on the additional Defects4J-v2.0 and QuixBugs, 27 and 5 more than state-of-the-art learning-based technique Recoder, demonstrating that GAMMA can address the important dataset overfitting issue well. Moreover, we implement GAMMA with CodeBERT [27] and ChatGPT [28], and find 80 and 67 bugs are fixed correctly on Defects4J-v1.2. The results demonstrate that GAMMA with other pre-trained models can further provide substantial advancement, highlighting the generalizability of GAMMA.

To sum up, the contributions of this paper are as follows:

- **New Dimension.** We bridge the gap between the advances in recent pre-trained models and template-based APR. Different from existing template-based APR retrieving donor code from local buggy files and existing learning-based APR generating a patched code snippet from scratch, our work demonstrates that we can leverage pre-trained models to generate correct code tokens in a given fix pattern. More importantly, our work reveals the potential for leveraging pre-trained models to resolve the important fix ingredient problem in template-based APR.
- **Novel APR tool.** We propose GAMMA, which leverages the large pre-trained language model to generate correct code with the help of fix patterns without any additional

historical bug-fixing pairs for training. We define a set of fix patterns in a fill-in-the-blank format and leverage the original pre-training objective of pre-trained models to predict actual masked-out tokens. Considering the fill-in-the-blank task can leverage various pre-trained language models, GAMMA is general in concept and can be implemented with different pre-trained models in practice.
- **Extensive study.** We conduct an empirical study to investigate the effectiveness of GAMMA compared to state-of-the-art traditional and learning-based APR techniques. The results on the widely-adopted Defects4J-v1.2 show that GAMMA is able to fix 82 bugs and 14 of them cannot be fixed by existing APR tools, creating a new higher baseline of repair performance. More importantly, GAMMA fixes 45 and 22 bugs on the newly-developed Defects4J-v2.0 and QuixBugs, demonstrating that GAMMA can avoid the important dataset overfitting issue of existing APR techniques. Moreover, we adopt different pre-trained models (e.g., ChatGPT) to further investigate the generalization ability of GAMMA.
- **Available artifacts.** To support the open science community, we release the relevant materials (including source code, experimental results, and correct patches) in our experiment for replication and future research [29].

II. BACKGROUND AND MOTIVATION

A. Automated Program Repair

As a promising technique to shift the heavy manual debugging to efficient automated patch generation, APR has developed rapidly and received much attention from a broad of research communities, such as software engineering, software security, and artificial intelligence [6], [30]. The workflow of APR usually involves three phases: (1) *fault localization*, i.e., the off-the-shelf fault localization techniques are utilized to identify a ranked list of suspicious code elements, with whose help APR can focus on a small code region, thus reducing the workload [31]; (2) *patch generation*, i.e., candidate patches are generated by applying a set of transformation rules to the suspicious code snippets [32]; and (3) *patch validation*, i.e., the available test suites are utilized as the program specifications to check the correctness of candidate patches [33]. The candidate patches that pass all available test suites are considered plausible ones. The plausible patches that are semantically equivalent to the developer patch by manual inspection are considered correct ones; otherwise overfitting ones [34], [35].

In the literature, as the core component of APR research, a mass of research efforts are devoted to generating patches from different aspects, including traditional and learning-based ones. In particular, traditional APR techniques can be classified as heuristic-based [11], [12], constraint-based [13], [14], template-based [15], [16]. Among them, template-based APR is proven to achieve the best performance, which consists of two fix ingredients, i.e., fix patterns and donor code. Fix patterns are hand-crafted by human experts to denote the common code changes, and donor code is retrieved in buggy files to denote the actual correct code tokens. GAMMA aims

to revise the important donor code by employing pre-trained language models in a fill-in-the-blank manner.

Compared with traditional APR techniques, learning-based techniques handle the program repair problem as a neural machine translation (NMT) task, which translates a code sequence from a source language (i.e., buggy code snippets) into a target language (i.e., correct code snippets). Existing NMT repair models are typically built on the top of the *encoder-decoder* architecture [36]. The *encoder* extracts the hidden status of buggy code snippets with the necessary context, and the *decoder* takes the encoder's hidden status and generates the correct code snippets [22], [37], [38]. Thanks to the powerful ability of DL to learn hidden and intricate relationships from massive code corpora, learning-based APR techniques have achieved remarkable performance in the last couple of years. Although learning-based APR techniques have demonstrated their promising future, they are still limited by the quality and quantity of historical bug-fixing pairs for training [18]. We view GAMMA as a novel learning-based APR technique that attempts to boost traditional APR techniques by utilizing deep learning technology. However, different from most existing learning-based APR that treats patch generation as an end-to-end NMT task with a limited number of bug-fixing pairs as training data, GAMMA integrates pre-trained language models into template-based APR and only predicts masked code tokens with a zero-shot learning scenario.

B. Pre-trained Model

Recently, pre-trained language models (e.g., UniXcoder [26] and ChatGPT [28]) have significantly boosted performance across a wide range of code-related tasks [39], [40]. These models are pre-trained by self-supervised training on large-scale unlabeled corpora and then fine-tuned by supervised training on limited corpora to enhance performance on multiple downstream tasks. During the pre-training process, a masked language modeling objective is usually employed to derive generic language representations from the massive unlabeled training data [41], i.e., a small percentage of tokens are replaced by masked tokens, and the training objective is to predict the original values of the masked tokens.

Existing pre-trained models usually adopt the encoder-decoder architectures, where the former encodes an input sequence as a fixed-length vector representation, and the latter generates an output sequence based on the input representation. These models can be generally categorized into three architectures: encoder-only, decoder-only, and encoder-decoder ones [42]. Encoder-only models (e.g., CodeBERT [41]) usually pre-train a bidirectional transformer in which each token can attend to each other. Decoder-only models (e.g., GPT [43]) are pre-trained using unidirectional language modeling that only allows tokens to attend to the previous tokens and themselves to predict the next token. Encoder-decoder models (e.g., UniXcoder [26]) often make use of denoising pre-training objectives that corrupt the source input and require the decoder to recover them.

```
1  @@ -786,7 +786,7 @@ void replace() {
2      } else {
3  -      int indexOfDot = namespace.indexOf('.');
4  +      int indexOfDot = namespace.lastIndexOf('.');
5      if (indexOfDot == -1) {
6          compiler.getNodeForCodeInsertion(
       minimumModule)
```

Listing 1: The Defects4J bug Closure-92

In this work, we select UniXcoder to retrieve the doner code via a mask prediction task. UnixCoder is pre-trained using the MLM objective which can be used to directly generate masked code tokens from the appropriate fix pattern and surrounding code context. Besides, CodeBERT and ChatGPT are employed to investigate the generalization ability of GAMMA.

C. Fix Template

Fix templates are widely employed in the APR community [17]. A fix template is a pre-defined code transformation rule that represents a common code change in the bug-fixing process. The insight behind fix templates is that many software bugs are similar in nature [22]. Therefore, with fix templates summarized from previous bugs, it is possible to automatically fix some other flawed code [24].

In the literature, there are several strategies to access fix templates: (1) *manual template mining* [44], [45], i.e., through performing analysis on existing bugs as well as their relevant patches, experienced developers can identify similar code changes, and turn them into fix templates. (2) *machine learning* [15], [20], i.e., learning approaches are used so that fix templates can be generated automatically. (3) *static analysis* [16], [46], i.e., fix templates are generated from assorted types of warnings raised by static analysis tools.

The process of applying a fix template to patch generation typically involves two steps. First, the APR tool selects an appropriate fix template based on the abstract syntax tree (AST) representation of the buggy code. Fix templates are chosen according to the types of nodes in AST, and bugs are fixed by mutating the target nodes. Second, the APR tool generates a repaired version of the buggy code by searching and applying the relevant donor code to the fix pattern.

D. Motivation Example

To better illustrate the limitation of existing template-based APR, we further present a motivating example in this section. As shown in Listing 1, we use a real-world bug Closure-92 from the widely-used benchmark Defects4J-v1.2 as an example. Closure-92 denotes the 92nd buggy version of the Google Closure Compiler project in Defects4J-v1.2. This bug is fixed by GAMMA successfully, while TBar fails to generate a correct patch. To fix this bug, the method name "indexOf" is replaced by "lastIndexOf". The fix template used here is to mutate method names. TBar applies the selected fix patterns to the source code in a naive way. In this case, TBar searches all the methods that appear in the local file where the bug
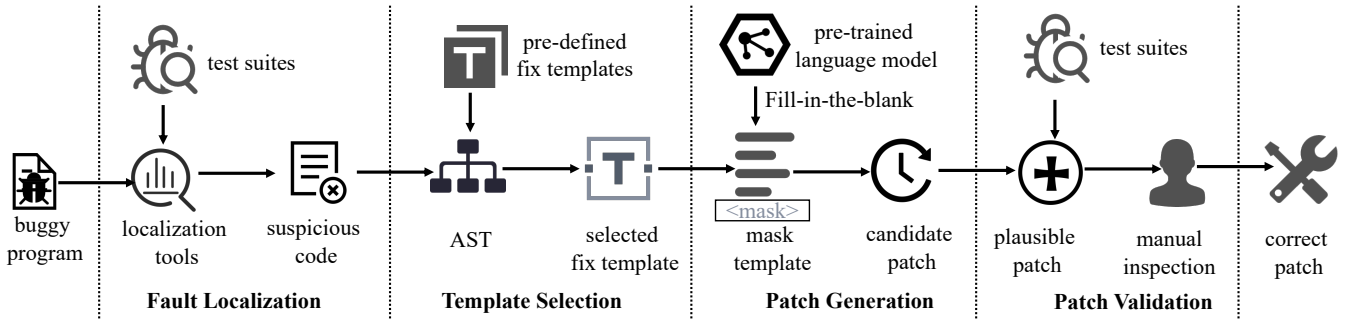
Fig. 1: The overall workflow of GAMMA

is localized, and replaces the buggy method with all the other methods with the same return type one by one. As a result, TBar is not able to generate method names that do not exist in the original file, like the "lastIndexOf" in this example, limiting its repair performance. Different from TBar, we replace the method name "indexOf" with a mask token (i.e. <mask>) instead, and query the pre-trained model UniXcoder to fill the mask with the fix pattern and corresponding context.

Based on the example, we can observe that, although the correct fix pattern is selected, as a state-of-the-art traditional APR, TBar still fails to generate a correct patch with inappropriate donor code (i.e., "lastIndexOf"). The effectiveness of template-based APR is largely dependent on donor code, which refers to the code tokens (e.g., variable name) that can be combined with the fix template in order to produce a complete patch. Donor code can be accessed in different scopes of the buggy program (e.g. a method, a file, or a package), but for some bugs, the correct donor code cannot be found even if the whole program is searched. For example, previous work demonstrates that half of the bugs from the Defects4J benchmark cannot be fixed because the relevant donor code is not available in the search space [24]. Thus, these bugs cannot be correctly fixed by template-based APR tools like TBar [17], which only identifies the donor code within the local file. With a larger search space (e.g., searching donor code from other projects), there might be more chances to fix these bugs. However, such a strategy leads to the search space explosion problem and an unaffordable search time, reducing repair efficiency. In this paper, we utilize pre-trained language models to retrieve relevant donor code. These models have learned programming language knowledge from a great number of programs in the wild, making it possible to repair bugs that require donor code from outside the buggy program.

III. APPROACH

In order to assess the effectiveness of fix ingredients, we build GAMMA, a template-based APR tool that combines the recurrently-used fix patterns and pre-trained language models. Fig. 1 presents the workflow of GAMMA. Given a buggy program and a set of test suites that make the program fail, a list of suspicious code elements is returned by fault localization approaches (i.e., *fault localization phase*). On top of the existing fix pattern corpus, GAMMA then selects appropriate fix patterns to the suspicious elements (i.e., *pattern selection phase*) and queries pre-trained models to retrieve donor code via a mask prediction mask (i.e., *patch generation phase*). GAMMA finally employs the available test suites as the oracle to check the generated patches and returns plausible patches for manual inspection (i.e., *patch validation phase*). Considering that fault localization is usually developed as an independent field and existing APR techniques employ off-the-shelf fault localization tools in the repair pipeline, we do not discuss the fault localization below. We describe the role and operation of other phases as well as all necessary implementation details.

A. Mask Template Definition

In the literature, a variety of fix patterns are designed based on manual summarization or automatic mining. On top of the state-of-the-art template-based approach TBar [17], we manually inspect all fix patterns and transform them into hole-filling-based patterns. We show the related templates as well as how they are applied to buggy code, or how statements with mask tokens are generated based on these templates.

T1: Check Cast Expressions. Adding an instanceof check around a statement when it contains an unchecked cast expression.

```
+    if(exp instanceof T){
         var=(T) exp ...
+    }
```

T2: Mutate Conditional Expressions: Mutating an expression that returns a boolean value by removing part of the expression, replacing it with masks or adding new masks.

```
Removing expression:
-    condExp1 op condExp2
+    condExp1

Updating expression:
-    condExp1 op condexp2
+    condExp1 op <mask>

Adding expression:
-    condExp1
+    condExp1 <mask>
```

where *condExp* denotes conditional expressions and *Op* denotes the logical operator (i.e., ‖ or &&).

T3: Mutate Data Types: Using one or more masks to replace data types in the variable declaration or cast expression nodes.

```
-    T var = ...
+    <mask> var = ...

-    ... (T) exp ...
+    ... (<mask>) exp ...
```

where both *T* denotes a data type and *exp* denotes the being casted expression (e.g., variable).

T4: Mutate Literal Expressions: Replacing literal expressions, including number literals, string literals, boolean literals, etc. with masks.

```
-    ...literal...
+    ...<mask>...
```

T5: Mutate Method Invocations. Mutating method invocation expressions by changing either method names or arguments.

```
Method name replacement:
-    method(...)
+    <mask>(...)

Argument insertion:
-    method(arg)
+    method(arg,<mask>)

Argument removal:
-    method(arg1,arg2)
+    method(arg)

Argument replacement:
-    method(arg)
+    method(<mask>)
```

T6: Check Null Pointer: Adding a null check to a statement that contains an expression that is probably null.

```
Null point skip:
+    if (exp != null){
         ...exp...
+    }

Return insertion:
+    if (exp == null){
+        return <mask>;
+    }
     ...exp...

Continue:
+    if (exp == null){
+        continue;
+    }
     ...exp...

Exception throw:
+    if (exp == null){
+        throw new IllegalArgumentException();
+    }
     ...exp...

Re-assignment:
+    if (exp == null){
+        exp=<mask>
+    }
     ...exp...
```

T7: Mutate Operators: Replacing an operator in a statement with masks or changing the priority of operations.

```
Changing the priority:
-    (exp1 op1 exp2) op2 exp3
+    exp1 op1 (exp2 op2 exp3)

Replacing operator:
-    exp1 op exp2
+    exp1 <mask> exp2
```

T8: Check Array Range: Checking the range of index before accessing an element in an array.

```
+    if (index<array.length) {
+        ...array[index]...
+    }
```

T9: Mutate Return Statements: Replacing the expression (e.g., literals, variables, and conditional expressions) that is returned in a method with masks.

```
-    return exp;
+    return <mask>;
```

T10: Insert Statements: Inserting return statements, try catch statements, if statements, method invocations, or simply some masks to the existing statement.

```
Return statement:
+    return <mask>;
     statement;

Try-catch statement:
+    try{
         statement;
+    } catch(Exception e){}

If statement:
+    if (<mask>) {
         statement;
+    }

Simple statement:
+    <mask>;
     statement;
```

T11: Remove Statements: Directly deleting one or more buggy statements from the original code.

```
-    statement;
```

T12: Replace Variables: Replacing a variable in a buggy statement with masks.

```
-    ...var...
+    ...<mask>...
```

T13: Move statements: Moving a statement from its original position to a new one.

```
-    statement;
     ...
+    statement;
```

B. Template Selection

On top of the defined mask fix templates, GAMMA determines which template should be applied to the input buggy statements. Similar to TBar [17], GAMMA employs an AST-based matching approach with a depth-first strategy. We first generate an AST for the input, and then all the nodes in the AST are traversed. If the AST contains a type of node that is required in a specific template, we apply the template to

the bug. We give an example of the *Closure-10* bug from the Defects4J-v1.2 in Listing 2 and how we generate its patch in Fig. 1. In the AST of the input line, we find a *Method Invocation* node in the AST of the input line, so we choose the template *T5: Mutate Method Invocation*. For the template *Method name replacement* that alters the name of a method, after locating the method name *allResultsMatch*, we replace it with a mask token, which is going to be predicted in the next phase. It is worth noting that GAMMA is built on top of UniXCoder, which is able to predict a sequence of code tokens based on a masked token. Thus, we do not need to consider how many tokens should be masked during patch generation and only use one mask token in the selected repair template, which is different from other pre-trained models, such as CodeBERT used in AlphaRepair [18] (discussed in Section V-C). There might be multiple repair templates that are suitable for a piece of buggy code at the same time. In this case, we stop the selection of repair templates as soon as the first correct patch is generated.

```
1  @@ -1414,7 +1414,7 @@ static boolean mayBeString(
       Node n) {
2  static boolean mayBeString(Node n, boolean recurse)
       {
3     if (recurse) {
4  -     return allResultsMatch(n,
       MAY_BE_STRING_PREDICATE);
5  +     return anyResultsMatch(n,
       MAY_BE_STRING_PREDICATE);
6     } else {
7     return mayBeStringHelper(n);
8     }
```

Listing 2: The Defects4J bug Closure-10

C. Patch Generation with Mask Prediction

After selecting an appropriate fix template for a buggy code, we use UniXcoder [26] to generate the correct code tokens via a fill-in-the-blank format. To this end, we leverage the original pre-training training objective of masked language modeling in UniXcoder. UniXcoder is an advanced pre-trained model for programming languages that support code understanding and generation tasks. It contains a pre-training objective of Masked Language Modeling (MLM), which is designed to predict some tokens that have been masked out. We leverage this pre-training task to complete the clozes generated in the previous step with any fine-tuning, so that candidate patches for the buggy programs can be produced.

The prediction for the mask is largely dependent on the tokens surrounding the mask. If the correct token appears in the input for the model, it is more likely that the token will be chosen as one of the possible results. The precision of mask prediction is quite limited when only a single masked buggy line is given without any context of the code, where there may be some useful information for bug fixing. To get more context of the buggy line in each bug, we extract the method that contains the line and use the whole method with

the masked buggy line as the input for UniXcoder. Considering that some tokens in the buggy line have been replaced with masks but these tokens may also contain essential information for mask prediction, in the first line of our input, we add the original buggy line in the form of a comment (i.e., add a "//" in front of the line). The commented buggy line followed by the method the bug is in together forms the final input. For every input, N candidate patches are generated by the UniXcoder model. N is the beam size and is an adjustable parameter of GAMMA. Relatively large beam size increases the possibility of generating correct patches.

D. Patch Validation

After a candidate patch for a given bug is generated by GAMMA, we apply the corresponding changes to the buggy program. Following the practice in the APR community [18], [47], we first recompile the patched program and filter out any patches that fail to compile. We then execute the patched program against the available test suite to identify the plausible patches that successfully pass all the test suites. For those plausible patches, we examine them manually to ensure the programs are fixed correctly, i.e., whether the patches are semantically equivalent to developer patches.

IV. EXPERIMENTAL SETUP

A. Research Questions

In this paper, we study the following research questions:

RQ1: What is the performance of GAMMA compared to state-of-the-art APR approaches?

RQ2: What is the generalizability of GAMMA in repairing additional real-world bugs?

RQ3: What is the scalability of GAMMA when employing other advanced pre-trained models?

B. Benchmarks

To evaluate the repair performance, we use the standard benchmark of Defects4J-v1.2 [48] in the APR community. Defects4J-v1.2 is a collection of real-world bugs from open-source projects and is widely adopted by existing traditional [17], [49] and learning-based APR approaches [38], [39], [47], [50]. In particular, Defects4J-v1.2 contains 395 known and reproducible bugs, each of which contains a buggy version and a fixed version, as well as a corresponding test suite that triggers that bug for patch validation. Evaluation on Defects4J-v1.2 can reflect the performance of GAMMA in a real-world debugging scenario and provides sufficient comparison results against most of the existing APR techniques.

Besides, we choose Defects4J-v2.0 and QuixBugs as other bug benchmarks for evaluation, so as to investigate the generalizability of GAMMA. Defects4J-v2.0 provides 420 additional real-world bugs from 17 Java projects, which is adopted by some recent APR studies [18], [50]. QuixBugs [51] is a multi-lingual parallel bug-fixing dataset in Python and Java used in [18], [39]. QuixBugs contains 40 small classic algorithms with a bug on a single line, along with the bug-triggering test suite.

C. Baselines

To enable sufficient evaluations, we compare GAMMA against both traditional and learning-based APR approaches. We choose seven recent learning-based APR tools, i.e., AlphaRepair [18], Recoder [50], CURE [47], CoCoNuT [52], CIRCLE [39], DLFix [38], and SequenceR [53]. We also choose two state-of-the-art template-based APR tools TBar [17] and PraPR [54] as representatives of traditional APR. In total, we evaluate GAMMA against nine advanced APR tools from different categories. Although the fault localization configuration is a significant part of APR, we do not take it into consideration in our experiment because of potential deviations that fault localization may bring about. Following recent APR studies [38], [39], [47], [50], we apply perfect fault localization in the way of inputting the exact buggy lines into different APR techniques to standardize the impact of fault localization on repair performance, discussed in Section VI.

D. Evaluation Metrics

We use two common metrics to evaluate the performance of all involved APR approaches [5], [6], i.e., plausible patch and correct patch. The first one fixes the buggy functionality without harming other correct functionality (i.e., passing all available test suites), and the second one is semantically or syntactically equivalent to the developer patch (i.e., generalizing the potential test suite). We manually inspect each plausible patch to identify whether it is a correct patch by following the standard practice in APR research.

E. Implementation Details

At the stage of fix template selection, we apply Eclipse JDT to parse the input line into AST, and then the AST is traversed to examine if it contains any node that is required by a fix template. There are several templates that can fit all the input buggy lines. For example, the template *T10: Insert Statements* only requires adding statements around the buggy line and does not mutate any nodes in the AST. Such templates are directly applied to all the inputs without checking the AST.

In the mask prediction phase, we choose the UniXcoder model "unixcoder-base". This is a model pre-trained on natural language-programming language (NL-PL) pairs and is reported in the original UniXcoder paper [26]. We use the encoder-decoder mode of the model to give predictions for each mask and generate candidate patches. We set the beam size as 250 due to the limitation of our device, which is smaller than 1000 used in CURE [47] and CoCoNuT [52]. Following previous learning-based APR approaches [18], [50], we set a 5-hour running-time limit for fixing one bug to perform a fair comparison. w

All experiments are conducted on one Ubuntu 18.04.3 server with two Tesla V100-SXM2 GPUs.

```
1 @@ -409,7 +409,7 @@ public static double
      factorialLog(final int n) {
2 public static int gcd(int u, int v) {
3 -    if (u * v == 0) {
4 +    if ((u == 0) || (v == 0)) {
5          return (Math.abs(u) + Math.abs(v));
6    }
```

Listing 3: The Defects4J bug Math-94

V. EVALUATION AND RESULTS

A. Comparison with State-of-the-arts

Experimental Design. In this section, we aim to evaluate the performance of GAMMA. We employ the 395 real-world bugs presented in the Defects4J-v1.2 dataset and compare GAMMA with state-of-the-art APR techniques, including traditional and learning-based ones. We report the performance of all compared techniques under perfect fault localization (i.e., the ground-truth buggy statement is known to the techniques).

Results. Table I presents the number of bugs that different APR techniques successfully fix on the Defects4J-v1.2 dataset. Overall, we find that GAMMA substantially outperforms the compared APR techniques including both traditional and learning-based APR techniques. GAMMA is able to generate correct patches for 82 real-world bugs, 20.59% (14 bugs), 26.15% (17 bugs) and 14.8% (8 bugs) more than TBar, Recoder and AlphaRepair. In particular, GAMMA fixes 11, 24, 16, 25, 3, and 3 bugs for Chart, Closure, Lang, Math, Mockito, and Time projects, respectively, four of which are best-performing (bold in Table I). More importantly, we find that GAMMA achieves a correct rate of 81.19% (82/101) for plausible patches, 9.61% (68/95), 23.15% (65/112) and 13.30% (74/109) higher than TBar, Recoder and AlphaRepair, indicating that GAMMA is able to alleviate the long-standing patch overfitting problem in the community of APR.

Overlap Analysis. To investigate what extent GAMMA complements existing APR techniques, we further calculate the number of overlapping bugs fixed by different techniques. One best-performing traditional technique (i.e., TBar) and three best-performing learning-based techniques (i.e., AlphaRepair, CURE, and Recoder) are selected. As shown in Fig. 2, GAMMA fixes 14 unique bugs that other APR approaches fail to fix, which is 11, 3, 8, and 10 more than TBar, AlphaRepair, CURE, and Recoder, respectively. More importantly, as a template-based APR technique, there are 22 correctly-fixed bugs unique to GAMMA compared with TBar, highlighting the benefits of mask prediction performed by UniXcoder. Overall, the results demonstrate that GAMMA is complementary to these best-performing APR techniques, to increase the number of correctly-fixed bugs in the Defects4J-v1.2 benchmark.

Case Study. We have demonstrated the superior performance of GAMMA over a state-of-the-art template-based tool TBar, which is most related to our work. To further investigate the effectiveness of GAMMA, we provide some examples of

TABLE I: Comparison with state-of-the-art APR techniques. Following the common practice in the APR community [39], [47], [52], we reuse the released results from the most recent work [18] instead of directly running the APR tools. Due to the APR community's subsequent validation of publicly released correct patches, the results of some APR tools may be different from the reported results in their published papers. (†) PraPR is evaluated with the results of Ochiai fault localization [31].

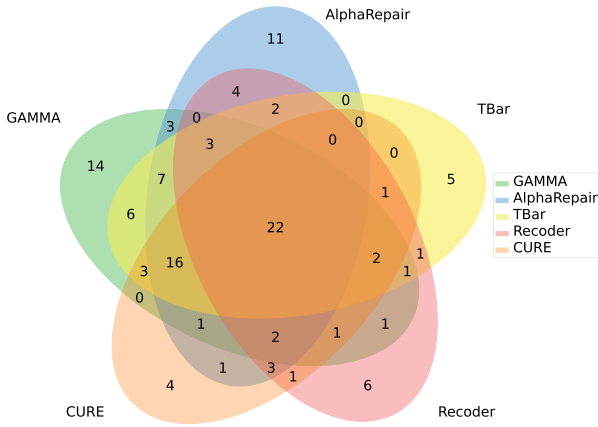| Project	SequenceR	CoCoNuT	CURE	DLFix	Recoder	AlphaRepair	CIRCLE	PraPR†	Tbar	GAMMA
Chart	3	7	10	5	10	9	7	7	11	**11**
Closure	3	9	14	11	21	23	17	12	16	**24**
Lang	2	7	9	8	11	13	10	6	13	**16**
Math	6	16	19	13	18	21	**27**	10	22	25
Mockito	0	4	4	1	2	**5**	1	3	3	3
Time	0	1	1	2	3	3	2	3	3	**3**
Total	14/19	44/85	57/104	40/68	65/112	74/109	64/182	41/146	68/95	**82/101**

Fig. 2: The overlaps of the bugs fixed by different approaches

```
1  @@ -742,7 +742,7 @@ static boolean isSimpleNumber(
       String s) {
2          return false;
3          }
4      }
5  -    return len > 0;
6  +    return len > 0 && s.charAt(0) != '0';
7      }
8      static double getSimpleNumber(String s) {
```

Listing 4: The Defects4J bug Closure-52

bugs that GAMMA is able to fix but TBar fails to. Listing 3 presents the bug Math-94 from the Defects4J-v1.2. Math-94 can be fixed with the fix template of mutating conditional expressions. The conditional expression "u * v == 0" within an if statement is replaced by "(u == 0) || (v == 0)". TBar deals with this template in the way of replacing the suspicious expression with other compatible ones collected from the same local file, while GAMMA directly replaces the expression with a mask token so that they can later be predicted by the mask prediction task from the pre-trained model, making it possible to generate new expressions that can correctly fix the bugs.

Listing 4 presents a similar example of the bug Closure-52 from the Defects4J-v1.2. Closure-52 denotes the 52nd buggy version of the Google Closure Compiler project in Defects4J.

To fix this bug, we need to insert a new sub-conditional expression into the original expression. TBar fails to generate a correct patch with improper operation and variables while GAMMA is able to directly predict the masked expression with corresponding code context.

B. Generalizability of GAMMA

Experimental Design. We have demonstrated that GAMMA achieves impressive performance to repair real-world bugs from the widely-adopted Defects4J-v1.2 benchmark on top of fix patterns. Durieux et al. [55] demonstrate that there exists a common benchmark overfitting phenomenon in APR evaluation, i.e., APR tools usually perform significantly better on Defects4J-v1.2 than on other benchmarks. In this section, according to prior work [18], [50], to evaluate the generalizability of GAMMA, we continue to conduct some extended experiments on additional projects for further evaluation.

Results. Table II presents the comparison results of GAMMA against baselines on Defects4J-v2.0 and QuixBugs. In Defects4J-v2.0, following some recent work [18], [56], we only focus on those bugs whose patches are confined to a single location. Overall, GAMMA generates 45 correct patches in the given 257 buggy programs, outperforming both traditional and learning-based approaches. We find that the performance achieved on the Defects4J-v2.0 dataset is commonly less than that achieved on the Defects4J-v1.2 dataset. For example, AlphaRepair fixes 18.73% (74/395) of bugs from Defects4J-v1.2 while only fixes 14.01% (36/257) of bugs from Defects4J-v2.0. Based on our analysis of the two datasets, the possible reason is that Defects4J-v2.0 contains a harder set of projects for APR with a different variety of fixes compared to Defects4J-v1.2. Despite that, GAMMA is able to generate 9, 34, and 37 more correct patches, which is the highest number among all approaches. We also find that as a template-based approach, TBar is able to generate a high amount of correct patches (68) for Defects4J-v1.2, while it only generates a limited number of correct patches (8) for Defects4J-v2.0. The possible reason may be that most fix patterns are designed to target Defects4J-v1.2, which may not generalize to other unseen projects, such as Defects4J-v2.0; Besides, learning-based approaches also suffer from moving to a harder evaluation dataset since the code transformation patterns are learned from training datasets

TABLE II: Comparison on additional datasets

Project	AlphaRepair	Recoder	CURE	CoCoNuT	CIRCLE	TBar	GAMMA
Defects4J 2.0	36	11	-	-	-	8	45
QuixBugs	28	17	26	13	19	-	22

which might not be present in Defects4J-v2.0. In contrast, GAMMA is able to address the generalizability issue without training on specific bug datasets, which makes it less prone to suffer from generalizability issues of traditional template-based or learning-based tools.

Apart from Defects4J-v2.0, we also try to validate our approach on QuixBugs, which extracts bugs from Quixey Challenge and translates them into both Java and Python languages. Since our fix templates are designed for Java, we only focus on Java programs in QuixBugs following previous work [17]. Table II shows that among 40 bugs in QuixBugs, 22 are correctly fixed by GAMMA, highlighting the competitive performance of GAMMA against state-of-the-art approaches. It is worth noting that most of the templates are summarized from Defects4J-v1.2, which may mean that some templates cannot be applied to any bugs except those from Defects4J-v1.2. Thus, GAMMA may be limited by the lack of more efficient fix templates when coming to other new bug datasets. For example, although various types of templates along with sub-templates are defined, some of the templates cannot be used to fix at least one bug from QuixBugs. As a result, we expect to explore more general fix templates in the future to further improve the performance of template-based APR.

C. Scalability of GAMMA

Experimental Design. To further investigate whether the performance of GAMMA is affected by different pre-trained models, we apply two other advanced models to perform the mask prediction task: CodeBERT and ChatGPT. CodeBERT [27] is a pre-trained model for programming and natural languages, and mask prediction is one of its pre-training tasks. ChatGPT is a state-of-the-art language model that has shown impressive ability in conversations with human beings. We also use Defects4J-v1.2 as a benchmark but replace UniXcoder with these two models in the process of filling masks to find out to what extent pre-trained models influence the effectiveness of template-based program repair.

Similar to UniXcoder, CodeBERT can also generate predictions for a mask token "<mask>" in the given code snippet. The difference between them is that UniXcoder can predict several continuous tokens for a single mask while CodeBERT can only give one token for a mask. However, there is usually more than one token under a mask, so when using CodeBERT, we have to use different numbers of successive masks to mask the initial code and then predict them sequentially. We do not know the exact number of masks we should use (i.e., the number of tokens in the fixed code) as there could be a great many possibilities in the patch. So while masking, we naively try all the mask numbers from 1 to 20, which is a range suitable to most cases. In every iteration, a mask is predicted

and a joint score for each prediction is calculated. Those predictions with the highest scores will be chosen to replace the mask, and the next mask will be predicted according to the previous predictions. We set the beam size as 250, the same as that used in UniXcoder, so in each iteration, CodeBERT will give 250 most possible predictions for the mask. Taking the process of fixing the Defects4J Closure-10 bug as an example (shown in Listing 2), the patch of the bug involves a change in the method name. To fix this bug, the method name *allResultsMatch* is replaced with masks, and then CodeBERT is asked to give 250 predictions for the first mask. Among the predictions, the token "any" has a relatively high score, so it replaces the first mask and CodeBERT will continue to predict the next mask until all masks are filled.

Different from UniXcoder, ChatGPT is fine-tuned from GPT-3.5 and close-sourced. We can access ChatGPT with ChatGPT's API of gpt-3.5-turbo-0301, which is the latest version available. We interact with ChatGPT through natural language conversations, i.e., sending requests to ChatGPT or receiving responses from ChatGPT. To fill the masks with ChatGPT, we first give it some prompts, instructing it to return back predictions for the mask. Following the prompts we then add the masked buggy line along with its context to form the complete query for ChatGPT. In our experiment, the input for ChatGPT starts with a prompt "*Next token prediction task, the first line is a comment to help prediction, just return 250 possible predictions for <mask> with highest probability:* ", and then the bug context we give is the same as the input for UniXcoder, which consists of a commented buggy line and the whole method where the buggy line belongs. Besides, due to the benefits of the designed prompt, we do not set the number of masked tokens in the buggy code, which is the same as UniXcoder.

Results. Fig. 3 presents the repair results of different pre-trained language models. Overall, the combination of the three models is able to fix 93 bugs from Defects4J-v1.2, demonstrating these models can be used together by GAMMA to further increase the number of correct patches that can be generated. In particular, we find when using CodeBERT to perform the mask prediction task, 80 bugs in total are fixed by GAMMA correctly, only two less than the bugs that GAMMA with UniXcoder fixes. However, it takes much more time for CodeBERT to generate correct patches, as the number of masks that should be used in fixing a bug is unpredictable, and we have to run the mask prediction program on the same bug and the same fix template for a lot of times, each with a different mask number. In contrast, UniXcoder circularly predicts the next token for a mask until an EOF token is generated, so only one mask is required to fix the bug. We
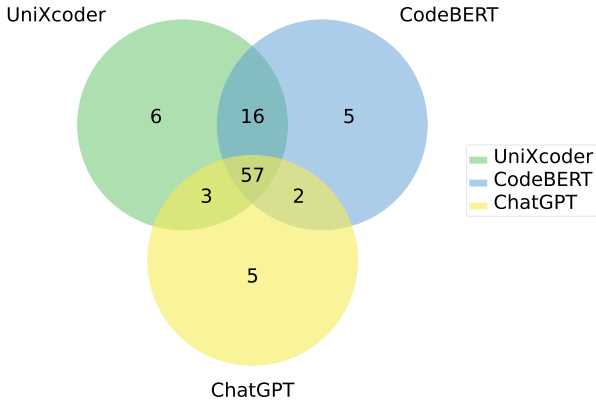
Fig. 3: The overlaps of the bugs fixed by different pre-trained language models

also find that GAMMA with ChatGPT only fixes 67 bugs correctly, performing not well as GAMMA with UniXcoder and CodeBERT in mask prediction. The possible reason lies in that UniXcoder is pre-trained with a masked language modeling objective, where some training text is artificially masked and the training objective is to predict the real text. However, ChatGPT is designed for natural language conversations and it is unclear how ChatGPT is pre-trained due to it being close-sourced. Thus, it is natural to employ UniXcoder to recover masked code tokens for buggy code snippets in our approach. Future researchers should further explore how to betters utilize ChatGPT (e.g., designing other prompts) for mask prediction.

VI. Threats to Validity

The first threat to validity comes from the manual inspection of correct patches. To alleviate the influence of potential bias, following previous works [39], [47], three authors manually verify all plausible patches (i.e. patches that successfully pass the test) based on ground truth patches (i.e., developer patches). A plausible patch is considered to be correct if all three authors identify it as equivalent to a ground truth patch semantically. To facilitate the replication and verification of our experiments, we also release the relevant materials (including all source code and correct patches) publicly [29].

The second threat to validity is the fault localization setting. We evaluate GAMMA and baselines on perfect fault localization (i.e., ground-truth buggy location is known) due to two reasons. First, off-the-shelf fault localization approaches affect the performance of APR techniques significantly, introducing a bias in comparison results [57]. The perfect fault localization mitigates the influence of differences in different fault localization approaches on the repair results and enables a fair assessment of repair performance independently of the fault localization approach used. Second, the most recent APR techniques [39], [47], [52], [58], [59] are only evaluated with perfect fault localization, which makes our work also use perfect localization to ensure direct comparison. However, this comparison setting may bring bias in repair performance since

the perfect fault localization results are usually unavailable in real practice. Despite that, we believe that perfect fault localization has little impact on our results, as perfect localization can show the pure performance of different APR approaches. In the future, we attempt to report the repair performance of GAMMA and baselines with both the automated fault localization (e.g., Ochiai [31]) and perfect fault localization.

The third threat to validity comes from the potential of data leakage of pre-trained models. In our experiment, we implement GAMMA on top of UniXCoder and evaluate GAMMA on the widely-adopted benchmark Defects4J-v1.2. Considering that UniXCoder is pre-trained with millions of code snippets, there may exist some bugs in the evaluation benchmark of Defects4J-v1.2 that appear in the pre-training dataset of UniXcoder. We perform a manual inspection to check whether the fixed bugs by GAMMA are leaked into the pre-training dataset. In particular, we query the pre-training datasets including 2.3M functions paired with comments and 4.1M unimodal code from CodeSearchNet. The manual inspection is performed by two authors independently and confirmed by a third author. We find there are three bugs leaked into the pre-training set, i.e., Closure-73, Closure-126, and Time-19. For the three bugs, we manually perturb the buggy code (e.g., changing variable names, adding dead code) and find GAMMA is still able to generate the correct patches for all three bugs. We also find that if we exclude the three overlapping bugs, GAMMA still outperforms state-of-the-art APR techniques (79 vs. 68 for TBar, 79 vs. 74 for AlphaRepair). Thus, we are confident that the data leakage is not a key point to our conclusion.

VII. Related Work

A. Automated Program Repair

Existing APR techniques can be divided into four categories, i.e., heuristic-based [11], [12], constraint-based [13], [14], [60], template-based [15]–[17] and learning-based repair techniques [38], [50], [52]. Our work is related to template-based and learning-based APR, discussed as follows.

Template-based APR, which generates patches with the help of fix patterns, represents state-of-the-art among traditional APR techniques. TBar [17] systematically collects and summarizes fix patterns from previous literature and investigates the effectiveness of patch generation based on these templates. Some other techniques explore fix patterns in various ways. For example, PAR [44] manually extracts fix patterns from 60,000 human-written patches. FixMiner [15] mines fix patterns with an iterative clustering strategy. AVATAR [16] leverages fix patterns from static bug detection tools to generate patches. Different from most existing template-based APR techniques that focus on mining fix patterns, GAMMA is the first work that aims to address the donor code issue by integrating pre-trained models with a fill-in-the-blank task.

With large available open-source code corpora, learning-based APR, which applies machine learning to the bug-fixing objective, is getting growing attention. For example, DLFix [38] uses a tree-based recurrent neural network (RNN) model to learn from bug fixes and surrounding contexts in the

form of an abstract syntax tree. CoCoNuT [52] introduces a novel context-aware neural machine translation (NMT) architecture that separately represents the buggy code and context. CURE [47] attempts to break the limit of existing NMT-based techniques by pre-training a programming language model on a large codebase, introducing a new code-aware search strategy, and using subword tokenization to narrow the search space. Recoder [50] is a syntax-guided edit decoder with placeholder generation, which provides a novel provider/decider architecture to guarantee that patches with correct syntax are generated. Different from existing learning-based APR techniques generating patches from scratch with bug-fixing data training, GAMMA aims to directly predict correct code tokens with the help of fix patterns in a zero-shot scenario.

Recently, there exists an increasing number of APR techniques on top of pre-trained models. For example, Yuan et al. [39] propose CIRCLE, a T5-based program repair framework equipped with continual learning ability across multiple languages. Xia et al. [18] propose AlphaRepair, a cloze-style APR approach based on CodeBERT without fine-tuning on historical bug-fixing data. In our work, we include CIRCLE and AlphaRepair as baselines in the experiment. Sobania et al. [61] investigate the performance of ChatGPT on the QuixBugs benchmark. Mashhadi et al. [62] investigate the performance of fine-tuning CodeBERT to fix software bugs from ManySStuBs4J. Jiang et al. [58] explore the performance of pre-trained models with and without fine-tuning for the program repair domain. Xia et al. [59] further present an extensive evaluation of recent pre-trained models for fixing real-world projects and find state-of-the-art pre-trained models are able to fix a considerable number of bugs. For example, CodeX, the most effective one, fixes 99 bugs in Defects4J-v1.2 with a total combination of three repair settings. We exclude CodeX as a baseline in our experiment due to the uncertainty of training data in such black-box large pre-trained models.

B. Pre-trained Language Models and Applications

In this section, we introduce some typical pre-trained language models and then discuss the applications of pre-trained language models to some code-related tasks, e.g., code search.

1) Pre-trained Language Models: Pre-trained language models have shown promising results on NLP tasks. BERT [63] is a model to condition on left and right contexts in all layers so as to pre-train deep bidirectional representations from unlabeled text. GPT-3 [43] is an autoregressive language model having 175 billion parameters, significantly outnumbering the parameters in previous language models. ChatGPT [28] is the currently most popular language model fine-tuned from GPT-3 and is receiving attention from both scientific and industrial fields. The most remarkable feature of ChatGPT is that it can generate human-like responses and communicate with human beings like what a real human can do.

Inspired by the success of pre-trained models in NLP, many researchers apply the pre-trained model to code-related tasks. Feng et al. [27] propose a bimodal pre-trained model (CodeBERT) for both programming language and natural lan-

guage. CodeBERT is developed on Transformer-based neural architecture and pre-trained with the task of masked language modeling, which is to predict tokens, and replaced token detection. Guo et al. [26] present UniXcoder, a unified cross-modal pre-trained model for programming language. UniXcoder utilizes mask attention matrices with prefix adapters to control the behavior of the model and leverages cross-modal contents such as AST and code comment to enhance code representation. Different from these studies designing novel pre-train models from scratch, we attempt to boost template-based APR on top of these pre-trained models.

2) Applications of Pre-trained Models: In addition to the above-mentioned typical pre-trained models, researchers have also applied such pre-trained models to some code-related domains (e.g., code completion, and program repair). Mastropaolo et al. [64] present an empirical study to investigate the usage of pre-trained models for four code-related tasks, including program repair, mutants injection, assertion generation, and code summarization. A similar strategy combining mutation patterns and pre-trained models is adopted in mutation testing. For example, Degiovanni et al. [65] introduce μBERT, a CodeBERT-based mutation testing tool by masking a token from the expression and replacing the masked token with the predicted one from CodeBERT. Richter et al. [66] propose a contextual mutation operator by employing CodeBERT to produce a context-dependent distribution over feasible token replacements. Recently, Zhang et al. [67] conduct an extensive empirical study to investigate the performance of pre-trained models in repairing security vulnerabilities and propose a enhanced approach with bug fixing transfer learning. Although there exist some SE tasks (e.g., mutation testing and program repair) benefitting from pre-trained models, in this work, we perform the first work to employ pre-trained models to directly predict the correct code with the help of fix patterns.

VIII. CONCLUSION

In this work, we present GAMMA, an innovative template-based APR tool that assimilates the advances of fix templates and pre-trained models. GAMMA first defines a set of mask fix templates by masking buggy code tokens with corresponding code context. GAMMA then uses the off-the-shelf pre-trained models to directly recover the correct code with a mask prediction task. More importantly, GAMMA can be built on various pre-trained models under a zero-shot learning setting and we implement it as a practical APR tool using the recent UniXcoder model. The experimental results on the popular Defects4J-v1.2 dataset have shown promising performance, e.g., 82 bugs are fixed by GAMMA, outperforming all state-of-the-art APR techniques. We also demonstrate that GAMMA is able to address the dataset overfitting well, e.g., 45 and 22 bugs are fixed in Defects4J-v2.0 and Quixbugs. We further demonstrate that GAMMA is generalizable to different pre-trained language models, such as CodeBERT and ChatGPT.

REFERENCES

[1] M. Monperrus, "Automatic software repair: A bibliography," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–24, 2018.

[2] D. Hao, L. Zhang, L. Zang, Y. Wang, X. Wu, and T. Xie, "To be optimal or not in test-case prioritization," *IEEE Transactions on Software Engineering*, vol. 42, no. 5, pp. 490–505, 2015.

[3] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering (TSE)*, vol. 45, no. 1, pp. 34–67, 2019.

[4] S. Benton, X. Li, Y. Lou, and L. Zhang, "On the effectiveness of unified debugging: An extensive study on 16 program repair systems," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE'20)*. IEEE, 2020, pp. 907–918.

[5] X. Gao, Y. Noller, and A. Roychoudhury, "Program repair," *arXiv preprint arXiv:2211.12787*, 2022.

[6] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, "A survey of learning-based automated program repair," *arXiv preprint arXiv:2301.03270*, 2023.

[7] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, "Sapfix: Automated end-to-end repair at scale," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP'19)*. IEEE, 2019, pp. 269–278.

[8] A. Mesbah, A. Rice, E. Johnston, N. Glorioso, and E. Aftandilian, "Deepdelta: Learning to repair compilation errors," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESE/FSE'19)*, 2019, pp. 925–936.

[9] D. Drain, C. Wu, A. Svyatkovskiy, and N. Sundaresan, "Generating bug-fixes using pretrained transformers," in *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming (MAPS'21)*, 2021, pp. 1–8.

[10] M. Allamanis, H. Jackson-Flux, and M. Brockschmidt, "Self-supervised bug detection and repair," *Advances in Neural Information Processing Systems (NeurIPS'21)*, vol. 34, pp. 27 865–27 876, 2021.

[11] M. Martinez and M. Monperrus, "Astor: A program repair library for java," in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA'16)*, 2016, pp. 441–444.

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

[13] T. Durieux and M. Monperrus, "Dynamoth: Dynamic code synthesis for automatic program repair," in *Proceedings of the 11th International Workshop on Automation of Software Test (AST'16)*, 2016, pp. 85–91.

[14] 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 (TSE)*, vol. 43, no. 1, pp. 34–55, 2016.

[15] 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 (ESE)*, vol. 25, no. 3, pp. 1980–2024, 2020.

[16] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Avatar: Fixing semantic bugs with fix patterns of static analysis violations," in *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'19)*, 2019, pp. 1–12.

[17] 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 (ISSTA'19)*, 2019, pp. 31–42.

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

[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] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo, and Y. Le Traon, "Mining fix patterns for findbugs violations," *IEEE Transactions on Software Engineering*, vol. 47, no. 1, pp. 165–188, 2018.

[21] 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 IEEE/ACM International Conference on Software Engineering (ICSE'18)*, 2018, pp. 1–11.

[22] 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 (ISSTA'18)*, 2018, pp. 298–309.

[23] 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 (ICSE'18)*, 2018, pp. 12–23.

[24] D. Yang, K. Liu, D. Kim, A. Koyuncu, K. Kim, H. Tian, Y. Lei, X. Mao, J. Klein, and T. F. Bissyandé, "Where were the repair ingredients for defects4j bugs? exploring the impact of repair ingredient retrieval on the performance of 24 program repair systems," *Empirical Software Engineering*, vol. 26, pp. 1–33, 2021.

[25] K. Liu, J. Zhang, L. Li, A. Koyuncu, D. Kim, C. Ge, Z. Liu, J. Klein, and T. F. Bissyandé, "Reliable fix patterns inferred from static checkers for automated program repair," *ACM Transactions on Software Engineering and Methodology*, 2023.

[26] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL'22)*, 2022, pp. 7212–7225.

[27] 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," in *Findings of the Association for Computational Linguistics (EMNLP'20)*, 2020, pp. 1536–1547.

[28] OpenAI, "Chatgpt: Optimizing language models for dialogue," https://openai.com/blog/chatgpt, 2023, Lasted accessed 2023-05-01.

[29] Q. Zhang and T. Zhang, "The gamma project website," https://github.com/iSEngLab/GAMMA, 2023, Lasted accessed 2023-05-01.

[30] Y. Yang, X. Xia, D. Lo, and J. Grundy, "A survey on deep learning for software engineering," *ACM Computing Surveys (CSUR)*, vol. 54, no. 10s, pp. 1–73, 2022.

[31] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering (TSE)*, vol. 42, no. 8, pp. 707–740, Aug. 2016.

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

[33] Q. Zhang, C. Fang, W. Sun, Y. Liu, T. He, X. Hao, and Z. Chen, "Boosting automated patch correctness prediction via pre-trained language model," *arXiv preprint arXiv:2301.12453*, 2023.

[34] S. Benton, Y. Xie, L. Lu, M. Zhang, X. Li, and L. Zhang, "Towards boosting patch execution on-the-fly," in *Proceedings of the 44th International Conference on Software Engineering (ICSE'22)*, 2022, pp. 2165–2176.

[35] Q. Zhang, X. Zhai, S. Xu, W. Huang, J. Zhang, and Y. Fan, "Interactive patch filtering via test generation," in *2022 9th International Conference on Dependable Systems and Their Applications (DSA'22)*. IEEE, 2022, pp. 42–53.

[36] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems (NeurIPS'17)*, 2017, pp. 5998–6008.

[37] 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 (POPL'16)*, 2016, pp. 298–312.

[38] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE'20)*, 2020, pp. 602–614.

[39] W. Yuan, Q. Zhang, T. He, C. Fang, N. Q. V. Hung, X. Hao, and H. Yin, "Circle: Continual repair across programming languages," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'22)*, 2022, pp. 678–690.

[40] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. Fu *et al.*, "Automating code review activities by large-scale pre-training," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESE/FSE'22)*, 2022, pp. 1035–1047.

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

[42] C. Niu, C. Li, V. Ng, J. Ge, L. Huang, and B. Luo, "Spt-code: Sequence-to-sequence pre-training for learning the representation of source code," in *Proceedings of the 44th International Conference on Software Engineering (ICSE'22)*, 2022, pp. 2006–2018.

[43] 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," in *Proceedings of the Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 1877–1901.

[44] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th International Conference on Software Engineering (ICSE'13)*. IEEE, 2013, pp. 802–811.

[45] K. Pan, S. Kim, and E. J. Whitehead, "Toward an understanding of bug fix patterns," *Empirical Software Engineering*, vol. 14, pp. 286–315, 2009.

[46] X. Liu and H. Zhong, "Mining stackoverflow for program repair," in *2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER'18)*. IEEE, 2018, pp. 118–129.

[47] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE'21)*, 2021, pp. 1161–1173.

[48] 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 23rd International Symposium on Software Testing and Analysis (ISSTA'14)*, 2014, pp. 437–440.

[49] Q. Zhang, Y. Zhao, W. Sun, C. Fang, Z. Wang, and L. Zhang, "Program repair: Automated vs. manual," *arXiv preprint arXiv:2203.05166*, 2022.

[50] 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 (ESEC/FSE'21)*, 2021, pp. 341–353.

[51] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge," in *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion'17)*, 2017, pp. 55–56.

[52] 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 (ISSTA'20)*, 2020, pp. 101–114.

[53] 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 (TSE)*, vol. 47, no. 9, pp. 1943–1959, 2019.

[54] 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 (ISSTA'19)*, 2019, pp. 19–30.

[55] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, "Empirical review of java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts," in *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*, 2019, pp. 302–313.

[56] H. Ye, J. Gu, M. Martinez, T. Durieux, and M. Monperrus, "Automated classification of overfitting patches with statically extracted code features," *IEEE Transactions on Software Engineering (TSE)*, vol. 48, no. 8, pp. 2920–2938, 2022.

[57] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. Le Traon, "You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems," in *Proceedings of the 12th IEEE conference on software testing, validation and verification (ICST'19)*, 2019, pp. 102–113.

[58] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," in *Proceedings of the 45th International Conference on Software Engineering (ICSE'23)*, 2023, pp. 1430–1442.

[59] 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'23)*, 2023, pp. 1482–1494.

[60] 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 (ICSE'16)*, 2016, pp. 691–701.

[61] D. Sobania, M. Briesch, C. Hanna, and J. Petke, "An analysis of the automatic bug fixing performance of chatgpt," in *Proceedings of the Fouth International Workshop on Automated Program Repair (APR'23)*, 2023, pp. 1–8.

[62] E. Mashhadi and H. Hemmati, "Applying codebert for automated program repair of java simple bugs," in *Proceedings Companion of the 18th IEEE/ACM International Conference on Mining Software Repositories (MSR'21)*, 2021, pp. 505–509.

[63] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT'19)*. Association for Computational Linguistics, 2019, pp. 4171–4186.

[64] A. Mastropaolo, N. Cooper, D. N. Palacio, S. Scalabrino, D. Poshyvanyk, R. Oliveto, and G. Bavota, "Using transfer learning for code-related tasks," *IEEE Transactions on Software Engineering (TSE)*, 2022.

[65] R. Degiovanni and M. Papadakis, "μbert: Mutation testing using pre-trained language models," in *2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW'22)*. IEEE, 2022, pp. 160–169.

[66] C. Richter and H. Wehrheim, "Learning realistic mutations: Bug creation for neural bug detectors," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST'22)*. IEEE, 2022, pp. 162–173.

[67] Q. Zhang, C. Fang, B. Yu, W. Sun, T. Zhang, and Z. Chen, "Pre-trained model-based automated software vulnerability repair: How far are we?" *IEEE Transactions on Dependable and Secure Computing*, 2023.