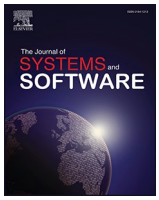
The Journal of Systems and Software 216 (2024) 112116

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss



Confix: Combining node-level fix templates and masked language model for automatic program repair

Jianmao Xiao [a,*], Zhipeng Xu [b], Shiping Chen [c], Gang Lei [a], Guodong Fan [d], Yuanlong Cao [e], Shuiguang Deng [f], Zhiyong Feng [d]

[a] School of Software, Jiangxi Normal University, Nanchang, China
[b] School of Digital Industry, Jiangxi Normal University, Shangrao, China
[c] CSIRO Data61, Sydney, New South Wales, Australia
[d] College of Intelligence and Computing, Tianjin University, Tianjin, China
[e] School of Computer and Information Engineering, Jiangxi Normal University, Nanchang, China
[f] College of Computer Science and Technology, Zhejiang University, Zhejiang, China

ARTICLE INFO

Editor: Prof Raffaela Mirandola

Keywords:
Automatic program repair
Fix templates
Masked language model

ABSTRACT

Automatic program repair (APR) is a promising technique to fix program defects by generating patches. In the current APR techniques, template-based and learning-based techniques have demonstrated different advantages. Template-based APR techniques rely on pre-defined fix templates, providing higher controllability but limited by the variety of templates and edit expressiveness. In contrast, learning-based APR techniques treat repair as a neural machine translation task, improving the edit expressiveness through training neural networks. However, this technique also faces the influence of quality and variety of training data, leading to numerous errors and redundant code generation. To overcome their limitations, this paper proposes an innovative APR technique called Confix. Confix first constructs a code information tree to assist in mining edit changes during historical repair. It then further enriches the types of fix templates using node information in the tree. Afterward, Confix defines masked lines based on node-level fix templates to control the scope of patch generation, avoiding redundant semantic code generation. Finally, Confix leverages the powerful edit expressiveness of the masked language model and combines it with fix strategies to generate correct patches more efficiently and accurately. Experimental results show that Confix exhibits state-of-the-art performance on the Defects4J 1.2 and QuixBugs benchmarks.

1. Introduction

In the context of comprehensive intelligence, software programs have become inseparable from our daily lives. Software defects can affect our quality of life in minor cases, but in severe cases, they can lead to the loss of large amounts of resources. To help developers reduce the time and effort spent on fixing software defects, automated program repair (APR) tools (Gazzola et al., 2019) have been constructed to utilize program-related information to generate potential fix patches automatically by identifying erroneous code segments. One of the main approaches currently used in APR technology is the Generate and Validate (G&V) method (Le Goues et al., 2011; Villanueva et al., 2020; Hua et al., 2018; Koyuncu et al., 2019; Xu et al., 2019; Kui Liu et al., 2019; Le and Bach, 2017; Mechtaev et al., 2018). In the initial phase, they use fault location techniques (Campos et al., 2012; Wong et al., 2016; Li et al., 2019; Wardat et al., 2021) to identify potential program locations that may lead to errors. After the error position is determined, the repair tool will generate a set of candidate patches, sort them, and modify the defective program. Afterward, the modified programs are compiled sequentially and verified by the test suite. Finally, they provide the patches that can pass compilation and all test cases to developers for selection, further checking the rationality of the patches to fix software vulnerabilities correctly.

Compared to other traditional APR technologies, template-based APR technology offers notable advantages, as suggested by previous studies (Kui Liu et al., 2019; Xia et al., 2023). Usually, these technologies utilize predetermined fix patterns designed by human specialists to search for reusable patches in the local codebase and replace erroneous

* Corresponding author.
E-mail address: jm_xiao@jxnu.edu.cn (J. Xiao).

https://doi.org/10.1016/j.jss.2024.112116
Received 11 January 2024; Received in revised form 28 May 2024; Accepted 31 May 2024
Available online 1 June 2024

code segments to complete the repair of particular types of defective programs. Although this technique is effective for fixing certain types of defects, it cannot be utilized to address undefined issues due to the constraints of the fix pattern. To address this issue, template mining-based technology (Soto and Goues, 2018; Liu and Zhong, 2018; Le et al., 2016; Long et al., 2017; Koyuncu et al., 2020; Kui Liu et al., 2019) has been proposed to achieve automatic program repair. These technologies utilize specific methods to identify operational changes from historical code modifications and convert them into fix templates. These fix templates are then applied to actual defects to resolve them. The use of template mining technology addresses the limitations of template types. However, the generation of candidate patches also relies on whether reusable code segments exist in the current code library. According to previous APR techniques (Yang et al., 2021), even if the fix templates accurately match the code change operations, generating the correct patch can still be impossible when related code segments in the code library are unavailable in local files. This is because template-based APR technology has low edit expressiveness (i.e., search space during patch generation), resulting in such issues.

To improve the edit expressiveness, researchers have proposed learning-based APR techniques for patch generation (Li et al., 2022; Chen et al., 2019; Jiang et al., 2021; Li et al., 2020; Lutellier et al., 2020; Zhu et al., 2021). These techniques typically treat program repair as a neural machine translation (NMT) task (Yang et al., 2020), training deep learning models to capture error context and generate patches for defective programs, transforming a buggy program into a fixed program. To create an effective deep learning model, having a training dataset that includes sets of erroneous and fixed code is crucial. These sets are often obtained by mining historical bug fixes from open-source repositories. Since the revised version code used for training may include unrelated code or unchecked errors, it introduces noise to the training dataset. Therefore, during the patch generation phase, learning-based APR techniques may result in many errors and unnecessary code, hindering the accurate repair of defective programs.

This paper combines the advantages of template-based and learning-based APR approaches, proposing a novel automatic program repair tool - Confix that aims to overcome the limitations of previous APR technology. Confix can provide applicable fix templates for defective programs, fixing the repair scope at the node level to avoid generating redundant code and making large-scale changes to the source code. To enhance the edit expressiveness, Confix utilizes a masked language model to retrieve relevant code from the context of defective code, repairing defective programs not in the training data based on selected fix templates. Additionally, Confix introduces fix strategies to assist the generation of patches, further improving the accuracy and efficiency of repairs.

Specifically, Confix first utilizes the abstract syntax tree (AST) analysis tool Gumtree (Falleri et al., 2014) to extract the modification sequences of preprocessed buggy and fixed versions code pairs. To obtain node-level fix templates, Confix extracts essential information based on the semantic context of AST nodes, including the corresponding node types and their hierarchical information, thus constructing a code information tree to assist in defining and selecting fix templates. Then, Confix filters all applicable templates from the fix template library based on the node information corresponding to the defective code in the code information tree constructed from the buggy program. Finally, Confix utilizes a masked language model and fix strategies to generate patches for processed masked inputs according to the fix templates, completing the repair of defective programs. Confix improves the security and stability of software and reduces potential risks and losses by promptly repairing program defects. Additionally, Confix can effectively enhance the efficiency and quality of software development, expanding the possibilities of software maintenance work.

The contribution of this paper is as follows:

1. Building a code information tree based on AST context semantic information assists in mining and expanding node-level fine-grained fix templates, providing applicable repairs for almost all types of nodes in defective programs.
2. Combined with pre-trained language models to generate patches, effectively solving the limitations of existing template-based APR techniques in obtaining code snippets from code repositories, greatly enhancing the edit expressiveness of APR techniques. Furthermore, Confix does not require additional fine-tuning or retraining with historical fixing data and applies to multiple programming languages.
3. By introducing fix strategies to assist pre-trained models in generating correct patches, experiments show that these fix strategies can avoid noise interference with the generation patches and improve the accuracy and efficiency of repairs.

2. Related work

2.1. Automatic program repair

Automatic Program Repair (APR) aims to generate patches for defective programs using automated tools, effectively reducing developers' debugging time. Therefore, APR techniques have developed rapidly and have become a research hotspot in software maintenance. The traditional APR process involves several steps. Firstly, use fault localization techniques to identify the arrangement of suspicious code lines in the defective program. Then, repair methods are applied to each suspicious code line to generate a set of candidate patches. Finally, validate the generated candidate patches with a test suite.

Traditional APR techniques can be categorized into heuristic-based (Le Goues et al., 2011; Villanueva et al., 2020), template-based (Hua et al., 2018; Koyuncu et al., 2019; Xu et al., 2019; Kui Liu et al., 2019), and semantic-constraint-based (Le and Bach, 2017; Mechtaev et al., 2018) according to the techniques used in the patch generation phase. Heuristic-based approaches typically guide the patch generation process by artificially defining heuristic rules. For example, GenProg (Le Goues et al., 2011) used a genetic algorithm to recombine existing code snippets through defined crossover and mutation operations to create patches that passed as many test cases as possible. GenProg iteratively applied these operations until a patch that passed all test cases was generated. Template-based approaches predefine some fix templates based on developers' or researchers' experiences and then retrieve reusable code snippets matching the fix templates from the local codebase to repair the defective program. For example, TBar (Kui Liu et al., 2019) integrated most of the fix templates from existing approaches and excluded irrelevant code snippets using bug code context to achieve more accurate repairs. Semantic constraint-based approaches infer the correct specification of programs and use it as a constraint to guide the patch generation process or verify the patch's correctness. For example, S3 (Le and Bach, 2017) collected constraint information from symbolic execution as input for the constraint solver. It ranked patches by comparing the syntactic and semantic similarity of code before and after modification, considering the grammatical and semantic features of the program to guide the patch generation process.

2.2. Fix template mining

It is crucial to obtain modification information related to defects from historical repair information, as program defects may inherently exhibit a high degree of similarity (Jiang et al., 2018), which can guide program repair. For instance, Liu and Zhong (2018) explored fix patterns from StackOverflow question and answer information and proposed a fix template mining approach called SOFIX. However, the effectiveness of SOFIX fix templates was suboptimal due to inadequate preprocessing of mining repair information. Le et al. (2016) proposed HistoricalFix for summarizing fix templates from historical repair information, but this

approach has limitations regarding the number and granularity of fix templates. Moreover, Long et al. (2017) introduced Genesis for Java language defects like null pointer exceptions, array out-of-bounds, and type conversions. This approach infers code transformations based on specific code contexts. However, this work relies on large amounts of the same type of training data, has limited applicability, and can only address specific deficiencies.

Researchers have been trying to improve the accuracy of fix templates by extracting the corresponding modification operations on Abstract Syntax Trees (AST) (Koyuncu et al., 2020; Bavishi et al., 2019; Shariffdeen et al., 2020). AST wiki (2023) is a tree representation of the abstract syntax structure corresponding to the text written in a formal language, where each node of the tree represents a construct that appears in the text, enabling syntactic and even semantic reasoning. Koyuncu et al. (2020) proposed Fixminer, which aims to mine AST-level fix templates from bug-fix patches in software repositories by adopting a diff analysis tool (Falleri et al., 2014) and using rich edit scripts for assistance. Fixminer considers specific tree representations in each iteration of fix template mining to discover similar changes. The process consists of the following stages: Firstly, collecting bug fix patches from software repository projects. Secondly, Rich Edit Scripts are constructed to capture more contextual information to describe the editing operations between the buggy and fixed versions of ASTs. Thirdly, building search indexes to quickly identify Rich Edit Scripts. Fourthly, compare two Rich Edit Scripts to determine if they are identical. Lastly, patterns are inferred by grouping identical trees into clusters using clustering and identifying clusters with more than two members as patterns. Similarly, Confix utilizes diff analysis tools to obtain fix templates at the AST node level and constructs a code information tree to assist in mining and selecting fix templates.

2.3. Large pre-trained code models

Large pre-trained code models (Lu et al., 2021; Feng et al., 2020; Guo et al., 2020; Wang et al., 2021) can provide powerful edit expressiveness for APR technology in the code generation phase, offering an alternative approach to the development of APR technology. Researchers often treat program repair as a neural machine translation (NMT) (Yang et al., 2020) task, training neural networks in supervised learning (Jiang et al., 2020) manner to generate patches for defective programs by capturing error contexts. For example, CoCoNuT (Lutellier et al., 2020) employed a multi-head attention mechanism (Vaswani et al., 2017) to enhance the amount of crucial information passed from the encoder to the decoder while also encoding the context and error lines separately. CURE (Jiang et al., 2021) pre-trained the NMT model on a large corpus of developer code and used a static checking strategy to generate patches with valid identifiers, improving grammatical correctness. Recoder (Zhu et al., 2021) introduced a grammar-guided decoder to modify the NMT model and generate syntactically correct edits for erroneous code snippets.

Although using supervised learning to train neural networks provides powerful editing expressiveness, it has strict requirements for training data to achieve high repair performance. The training data consists of an input and a label. The training data cannot cover all defect types, and many labels with redundant or erroneous codes cannot be excluded entirely, significantly affecting the model's performance. In order to utilize a large amount of open-source unlabeled training data to train neural networks, self-supervised learning (Liu et al., 2021) has been proposed to achieve self-monitoring goals during model training. Self-supervised learning eliminates the need for labels for training data, which significantly improves the model's number of parameters and performance. For example, the Masked Language Model (MLM) (Devlin et al., 2018) actively masks some actual data and trains by predicting the masked data. CodeBERT (Feng et al., 2020) is pre-trained based on large-scale code and natural language text data using the MLM training objective for automatic code completion. Recently, Xia and Zhang (2022) proposed AlphaRepair, which replaces defective code with mask tokens to fix the repair scope of the pre-trained language model and then generates patches using MLM. Confix chose to use CodeBERT to generate patches by capturing the surrounding context of defect code under the given fix templates. The benefit of using pre-trained code models with MLM as the training objective is that Confix does not need additional fine-tuning or retraining on the model.

3. Approach

By combining fix template mining techniques and pre-trained code models, Confix effectively integrates the advantages of template-based and learning-based APR techniques, bridging both limitations. Confix replaces bug-containing code snippets with spaces or masks according to the fix templates to constrain the generation scope of the pre-trained model. This approach ensures that the source code is not extensively modified, avoiding the generation of redundant semantic code while leveraging the powerful edit expressiveness of the pre-trained model to generate accurate and effective patches for defective programs. Fig. 1 illustrates the overall framework of Confix, and its specific workflow can be summarized as follows:

1) Fix templates mining: To mine fix templates from historical repair information, GumTree is first used to convert the source code into an AST, and differences between the buggy and fixed version's AST are identified. To obtain a more granular understanding of which node types have changed during the repair process, Confix constructs a code information tree based on the AST context semantic information to assist in template mining. Afterward, different nodes are captured by analyzing the surrounding contextual information and node types of code changes.
2) Fault localization and fix templates selection: Confix uses fault localization techniques for the introduced buggy program to locate suspicious lines of code but cannot determine the specific code elements that caused the defect. To obtain information on each code element in the suspicious lines of code, Confix constructs a code information tree by obtaining the AST corresponding to the buggy program. Each code element in the suspicious lines of code corresponds to a node in the code information tree, which includes the node type and hierarchical information in the AST. After obtaining the node type corresponding to the code element, Confix retrieves applicable fix templates from the fix template library and performs masking replacement operations on the suspicious lines of code.
3) Patches generation and validation: After performing masked preprocessing replacement on the suspicious lines of code using fix templates, the masked lines and their surrounding context constitute the input of CodeBERT. Since the input contains multiple mask tokens, Confix iteratively queries CodeBERT to predict the mask tokens and introduces fix strategies to generate candidate patches for the buggy program. Considering the large number of candidate patches, Confix ranks the candidate patches. It only retains a beam width number of patches for compilation testing to obtain the correct patches more efficiently and accurately.

3.1. Fix template mining

In order to mine fix templates at the AST node level, Confix uses GumTree to perform differential analysis on the buggy and fixed versions of the AST, obtaining a series of edit operations that transform one AST into another. The AST decomposes programming statements recursively, storing syntactic elements in nodes based on hierarchical information and rules, where the values of the nodes correspond to actual tokens in the code. Fig. 3 shows the AST corresponding to the Java code in Fig. 2. It is worth noting that when mining fix templates, Confix focuses on the variations of different types of nodes in historical fixes. The only requirement for historical repair is that it can pass

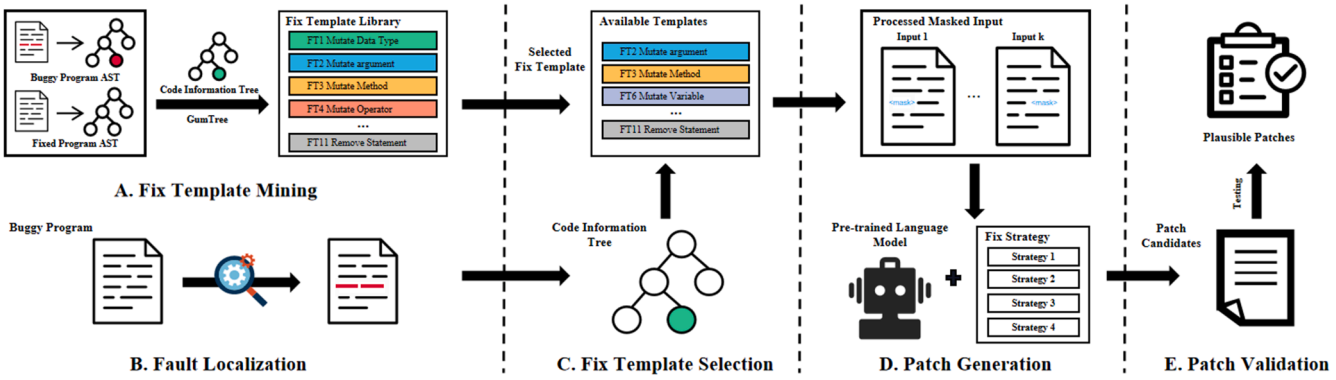


Fig. 1. The overall workflow of Confix.

```
public class SumCalculator {
    public int sum(int a, int b) {
        int c = a + b;
        return c;
    }
}
```

Fig. 2. Example of Java class.

compilation, which solves the limitations of learning-based APR technology due to unreliable historical repair.

The core ideas of GumTree are:

a) Mapping establishment: First, establish a mapping between isomorphic subtree nodes of the two ASTs through a greedy top-down algorithm, and then match the descendants of the two nodes through a bottom-up algorithm.

b) Deriving Edit Scripts: Utilizing a quadratic optimization algorithm (Chawathe et al., 1996), edit scripts that can be executed on the buggy version AST are inferred based on the mapping between nodes to obtain the fixed version AST. Due to GumTree's inability to effectively handle movement operations within the same parent node in the modification sequence, only the following editing operations are considered:

1) Replace (v1) at (p) to (v2): Replace the old value v1 with the new value v2 at position p.
2) Insert (v) at (p): Insert a new node with a value of v at position p.
3) Remove (v) at (p): Remove the node with the value of v at position p.

AST node types are crucial factors in obtaining fix templates and generating patches. Confix extracts the primary information of nodes, including node types and hierarchy information, based on the semantic context of AST nodes and constructs a code information tree to store this information. With the help of the code information tree, it is possible to obtain editing operations for specific types of nodes at a finer granularity, thereby defining specific fix templates for different types of code

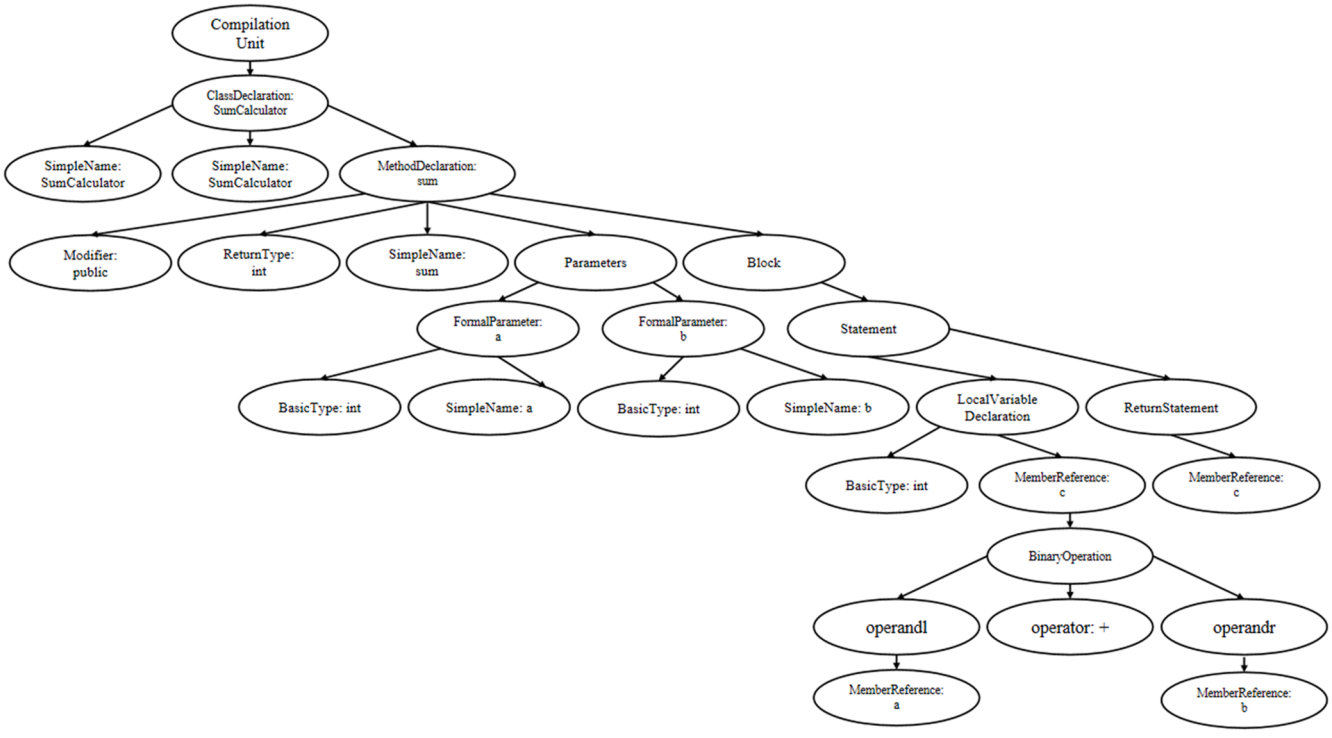


Fig. 3. AST representation of the SumCalculator class.

elements.

The primary process of constructing the code information tree is illustrated in Algorithm 1. Line 1: Define a semantic rule library for AST nodes to obtain specific node types; lines 2–5: Create a dictionary that stores the extracted node information and assigns values to the root node; line 6: Create a list to store child node information; lines 7–10: Traverse the AST using depth-first search to retrieve all node information iteratively; lines 11–19: Create dictionaries for child nodes to store their node information, where the node types are obtained based on semantic rules; lines 20–22: Store the child node information in a list and save it in the root node, finally returning the complete code information tree. Fig. 4 illustrates the serialized code information tree corresponding to the Java code in Fig. 2.

3.2. Fix template definition

In order to apply change operations for different types of nodes in historical repairs to defect program repairs, Confix organizes and summarizes modification sequences with the same structure based on node types after obtaining modification sequences with complete repair semantic information to derive fix templates. By constraining the repair scope of defect programs using fix templates, Confix reduces the changes made to the source program. Furthermore, it is possible to enrich the types of fix templates by considering the type and hierarchical information of nodes in the code information tree. Table 1 shows the fix template definitions after mask conversion and the fix templates related to different node types. Single mask token represent single-node repair, while multiple mask tokens represent multi-node repair.

3.3. Fix template selection

It is essential to select appropriate fix templates for defective programs as they can provide correct guidance for pre-trained language models. Confix uses fault localization techniques to locate suspicious lines of code and uses a code information tree to store node information for all code elements in the defective program. By traversing the code information tree using a depth-first strategy, Confix determines the corresponding nodes for each code element in the suspicious lines of code and retrieves their information. Fig. 5 presents an example from bug Chart 20 in Defects4J 1.2, where the red highlighted section represents the suspicious lines of code for that bug. Fig. 6 displays the node information in the code information tree that corresponds to the code elements belonging to the suspicious lines of code. It is worth mentioning that, in order to increase the generality of the fix template, we merged the node types in the original AST according to the AST context semantic rules during the construction of the code information tree. For example, the code element "super" in the suspicious line in Fig. 5 has a node type of "SuperConstructorInvocation" in the AST, while the node type of other code elements is "MemberReference". When constructing the code information tree, we unified these two types of nodes into "Method" and "Argument" respectively, so that one type of fix template can be applied to different node types to improve repair efficiency. Based on the node types and hierarchical information of the code elements in the code information tree, suitable fix templates are selected from the template library. In Fig. 5, the "Mask Line" represents the result of applying template FT2.2 to the suspicious lines of code after replacing them with mask tokens. By selecting the most suitable fix templates, it is possible to generate correct patches for defective programs to the greatest extent.

3.4. Patch generation

Confix improves its edit expressiveness by leveraging pre-trained language models to obtain a larger patch generation space. After selecting and applying appropriate fix templates to the suspicious lines of code, Confix combines them with the surrounding context as the input for CodeBERT. The CodeBERT tokenizer adopts Byte-Level Byte-Pair Encoding (BBPE) (Wang et al., 2020), effectively alleviating Out of Vocabulary issues (Radford et al., 2019). Taking bug Chart 20 as an example, Fig. 7 illustrates the token sequence composing the CodeBERT input. The maximum limit of input tokens for CodeBERT is 512. To provide a larger context for the pre-trained language model to infer from and maximize the number of input tokens, we expand the context range starting from the buggy lines of code. We continue to increase the number of context tokens until reaching the limit. As CodeBERT is trained on natural language text and code, including the buggy lines of code comments in the CodeBERT input sequence, it can provide references for replacement-type fix templates. However, for insertion-type fix templates, including the buggy lines of code as comments would introduce many noise tokens. Therefore, this operation is only performed for replacement-type fix templates.

After obtaining the complete input, Confix iteratively generates code

Algorithm 1
Code information tree construction.

```
Input: root: The root node of the abstract syntax tree
Input: level: The hierarchical information corresponding to the abstract syntax tree node
Output: codeInfoTree
1.  ruleLibrary <- define contextual semantic rule library for node types;
2.  codeInfoTree <- createDictionary();
3.  codeInfoTree.type <- root.get(type);
4.  codeInfoTree.value <- root.get(value);
5.  codeInfoTree.level <- level;
6.  codeInfoTree.children <- creatEmptyList();
7.  stack <- createStack();
8.  stack.push(root, codeInfoTree);
9.  while stack ≠ ∅ then
10.   (currentNode, currentCodeInfoTree) <- stack.pop();
11.   foreach childNode ∈ currentNode.children do
12.     childNodeInfo<-createDictionary();
13.     nodeRule <- getRule(childNode);
14.     if nodeRule ∈ ruleLibrary then
15.       childNodeInfo.type<-ruleLibrary.get(nodeRule);
16.     else
17.       childNodeInfo.type<-childNode.get(type);
18.     childNodeInfo.value <-childNode.get(value);
19.     childNodeInfo.level <- currentCodeInfoTree.get(level) + 1;
20.     currentCodeInfoTree.children.append(childNodeInfo);
21.     stack.push(childNode, childNodeInfo);
22. return codeInfoTree
```

level	type	value
0	CompilationUnit	
1	ClassDeclaration	SumCalculator
2	BasicType	int
2	MethodDeclaration	sum
3	BasicType	int
3	FormalParameter	a
3	BasicType	int
3	FormalParameter	b
3	LocalVariableDeclaration	
4	BasicType	int
4	VariableDeclarator	c
5	BinaryOperation	
6	MemberReference	a
6	BinaryOperation	+
6	MemberReference	b
3	ReturnStatement	return
4	MemberReference	c

Fig. 4. A serialized code information tree of the SumCalculator class.

tokens for each mask token using CodeBERT. CodeBERT was originally designed to achieve a training objective using the Masked Language Model (MLM). The MLM predicts the likelihood of a set of mask tokens $M = \{m1, m2, ..., mk\}$ based on the input sequence $X = \{x1, x2, ..., m1, m2, ..., mk, ..., xn\}$ that comprises the mask tokens and their surrounding context. The ultimate goal is to restore the original data of the masked parts. The MLM loss function is shown in Eq. (1). Since the training objective of CodeBERT is consistent with the generation objective of Confix, Confix does not need to perform additional retraining or fine-tuning on CodeBERT using datasets.

$$\ell_{MLM} = \sum_{i \in masked} -\log(p(xi|X_{masked})) \tag{1}$$

Fig. 8 shows the iterative generation process when there are multiple mask tokens. Firstly, starting from the initial input, N highest probability candidate results are generated for the first mask token and stored in a candidate list. N represents beam width, an adjustable parameter (N is set to 50 in the figure). Confix uses the temporary joint score (Xia and Zhang, 2022) to represent the current conditional probability of each candidate result, and the calculation of the temporary joint score is shown in Eq. (2). The higher the score, the higher the confidence of the generated result. Here, p represents the number of currently generated mask tokens, $T = \{t1, t2, ..., tn\}$ represents the set of mask tokens, and $C^*(T, ti)$ represents the CodeBERT conditional probability when generating ti. Then, the candidate token is selected by ranking to replace the first mask token, and CodeBERT is iteratively queried to generate code tokens corresponding to the second mask token. After that, the temporary joint score is calculated for all generated results of the first two tokens, and the top N candidates with the highest scores are stored in the candidate list. This process is repeated until code tokens corresponding to all mask tokens are generated.

$$temp\ joint\ score(T) = \frac{1}{p}\sum_{i=1}^{p} \log(C^*(T, ti)) \tag{2}$$

In addition, Confix introduces four fix strategies to assist CodeBERT in generating patches for defective programs more efficiently and accurately to avoid the influence of contextual noise during the generation process.

1) Strategy 1: Add shielded tokens. Since CodeBERT generates code tokens for mask tokens by referencing the context, the candidate results may contain many incorrect characters. For fix templates targeting different node types, corresponding fix patches should be generated. By adding shielded tokens, incorrect options can be filtered out when generating candidate code tokens for mask tokens. For example, the patch generated by FT6.1 node-level variable replacement can only be composed of letters, numbers, and underscores. If other types of characters are generated, they should be removed from the candidate list to reduce the impact of generating incorrect characters.
2) Strategy 2: Limit the number of mask tokens and remove the same modification action inputs. For fix templates targeting different node types, the corresponding tokens' quantity range should be set instead of a fixed range for mask tokens. By properly configuring the number of mask tokens and removing the same modification action generated by different fix templates, the number of preprocessed masked lines can be effectively reduced, thus reducing the failure cases of being unable to use the correct fix changes due to timeouts. In addition, to avoid making extensive modifications to the source program, Confix sorts the preprocessed masked lines in ascending order of repair scope based on the number of mask tokens.

Table 1
Node-level fix template definition.

Fix Template		Fix Template Definition	Code Change Action
FT1 Mutate Data Type	FT1.1 Replace data type declaration	- T var ...; + <mask> var ...;	Use mask tokens to replace the code element and generate a new data type declaration or the expression data type declaration as the target.
	FT1.2 Replace the expression data type declaration	- ... (T) Exp ...; + ... (<mask>) Exp ...;	
FT2 Mutate argument	FT2.1 Replace a single parameter	- ...method(arg1, arg2, ...)... + ...method(<mask>, arg2, ...)...	By changing one or more parameters in a method invocation, the method invocation expression can be altered. It should be noted that it is possible to choose to replace any number of consecutive parameter nodes at the same node level, thereby generating more repair types.
	FT2.2 Replace multiple parameters	- ...method(arg1, arg2, ...)... + ...method(<mask><mask>...<mask>, ...)...	
	FT2.3 Remove a single parameter	- ...method(arg1, arg2, ...)... + ...method(arg1, arg3, ...)...	
	FT2.4 Remove multiple parameters	- ...method(arg1, arg2, ...)... + ...method(arg3, ...)...	
	FT2.5 Insert a single parameter	- ...method(arg1, arg2, ...)... + ...method(arg1, arg2, ..., <mask>)...	
	FT2.6 Insert multiple parameters	- ...method(arg1, arg2, ...)... + ...method(arg1, arg2, ..., <mask><mask>...<mask>)...	
FT3 Mutate Method	FT3.1 Replace method	- ...method(args)... + ...<mask>(args)...	Update the method name by replacing the code element of the method node with mask tokens. In addition, the hierarchical information of the method node is extended downwards, and the mask tokens are used to replace its parameters together, updating the entire method expression.
	FT3.2 Replace method expression	- ...method(args)... + ...<mask><mask>...<mask>...	
FT4 Mutate Operator	FT4.1 Replace operator	- ...Exp1 Op Exp2... + ...Exp1 <mask> Exp2...	Modify the operation expression by replacing the operator with mask tokens. In addition, the hierarchical information of the operation node is further expanded to replace part or the entire operator expression.
	FT4.2 Replace part of the operation expression	- ...Exp1 Op Exp2... + ...Exp1 <mask><mask>...<mask>...	
	FT4.3 Replace the entire operation expression	- ...Exp1 Op Exp2... + ...<mask><mask>...<mask>...	
FT5 Mutate Literal	FT5.1 Replace node-level literal	- ...literal... + ...<mask>...	For Literal nodes, replace their code elements with masks, including boolean literals, numeric literals, and string literals. In historical fixes, boolean literals may be replaced with conditional expressions, resulting in node-level or statement-level outcomes.
	FT5.2 Using statement-level expressions for replacement	- ...literal...; + ...<mask><mask>...<mask>...	
FT6 Mutate Variable	FT6.1 Replace node-level variable	- ...var... + ...<mask>...	Update the variable name by replacing the code element in the variable node with mask tokens. In an assignment statement, replacing a single variable with an expression is possible, resulting in either node-level or statement-level outputs.
	FT6.2 Using statement-level expressions for replacement	- ...var... + ...<mask><mask>...<mask>...	
FT7 Mutate Conditional Expression	FT7.1 Insert conditional expression	- ...condExp...; + ...condExp <mask><mask>...<mask>...;	Insert or remove specific conditional statement nodes based on the hierarchical information of the nodes in the conditional expression. Furthermore, partial or entire conditional expressions can be updated by replacing some or all conditional expression nodes with mask tokens.
	FT7.2 Remove conditional expression	- ...condExp1 Op condExp2...; + ...condExp1...;	
	FT7.3 Replace conditional expression	- ...condExp...; + ...<mask><mask>...<mask>...;	
FT8 Mutate Return Expression	FT8.1 Replace expression	- return Exp; + return <mask><mask>...<mask>;	Insert or remove specific return expression nodes based on the hierarchical information of the nodes in the return expression. Furthermore, partial or entire return expressions can be updated by replacing some or all return expression nodes with masks.
	FT8.2 Remove expression	- return Exp1 Op Exp2; + return Exp1;	
	FT8.3 Insert expression	- return Exp; + return Exp <mask><mask>...<mask>;	
FT9 Insert Statement	FT9.1 Insert an if statement	+ if (<mask><mask>...<mask>) { statement; ..... +}	Fix a buggy program by inserting the specified expression around the defective code snippet, where the expression can be an if statement, return statement, method invocation statement, etc. In this process, crucial expressions are replaced with mask tokens.
	FT9.2 Insert an while statement	+ while (<mask><mask>...<mask>){ statement; ..... +}	
	FT9.3 Insert a return statement	+ return <mask><mask>...<mask>;	
	FT9.4 Insert a simple statement	+ <mask><mask>...<mask>; statement;	
FT10 Insert Null Pointer Checker	FT10.1 Null pointer check	+ if (Exp != null) { ...Exp...; ..... +}	Insert a null check to the bug statement that is probably null using a null pointer checker.
	FT10.2 Replace expression	+ if (Exp == null) { - ...Exp...; + ...<mask><mask>...<mask>...; +}	
	FT10.3 Exception throw	+ if (Exp == null) { + throw newIllegalArgumentException (...); +}	
FT11 Remove Statement	FT11.1 Remove Statement	- statement;	Remove all bug statements from the buggy program.

```
    public ValueMarker(double value, Paint paint, Stroke stroke,
                       Paint outlinePaint, Stroke outlineStroke, float alpha) {
-           super(paint, stroke, paint, stroke, alpha);
+           super(paint, stroke, outlinePaint, outlineStroke, alpha);
            this.value = value;
        }
```

Code Change Action:
Replace the arguments "paint" and "stroke" with mask tokens.
Mask Line:
super(paint, stroke, <mask><mask>...<mask>, alpha);

Fig. 5. Fixes and code change action for bug Chart-2.

level	type	value
4	SuperConstructorInvocation	super
5	MemberReference	paint
5	MemberReference	stroke
5	MemberReference	paint
5	MemberReference	stroke
5	MemberReference	alpha

Fig. 6. A serialized code information tree of the Chart-20 buggy line.

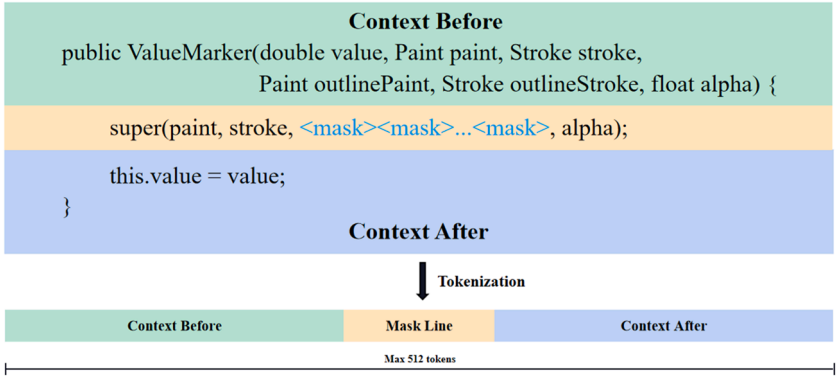


Fig. 7. Token sequence composition of bug Chart-20.

3) Strategy 3: Introduce allocation coefficients. For long mask token sequences, the generation of subsequent mask tokens depends mainly on the results of the previous mask tokens. When the temporary joint score calculated for the correct tokens generated by the current few mask tokens is very low, it often leads to the correct repair being excluded from the candidate list prematurely. By introducing allocation coefficients to reduce the impact of the temporary joint score of the first few mask tokens, as much repair possibility as possible can be obtained, ensuring that the correct repair can appear in the final candidate list. The allocation coefficient is calculated by Formula 3, and Formula 4 represents the joint score after considering the allocation coefficient. Here, Pi represents the allocation coefficient of each mask token; Fi corresponds to the weight of the mask tokens.

$$Pi = Fi \Big/ \sum_{i=1}^{n} Fi \tag{3}$$

$$joint\ score(T) = \sum_{i=1}^{n} Pi * \log(C^*(T, ti)) \tag{4}$$

4) Strategy 4: Gradient beam width. Although introducing allocation coefficients can increase the possibility of obtaining the correct patch, it can also result in a small number of incorrect patches occupying the candidate list. Since the correct repair is generated step by step, given the correct generation of the first few mask tokens, the conditional probability of the subsequent mask tokens is also high. Therefore, when generating patches for long mask token sequences, it is necessary to appropriately increase the capacity of the candidate list for the first few mask tokens to ensure that the correct repair always exists in the candidate list. To improve the accuracy of the repair and ensure its efficiency, it is necessary to gradually reduce the beam width and finally retain only N candidates.

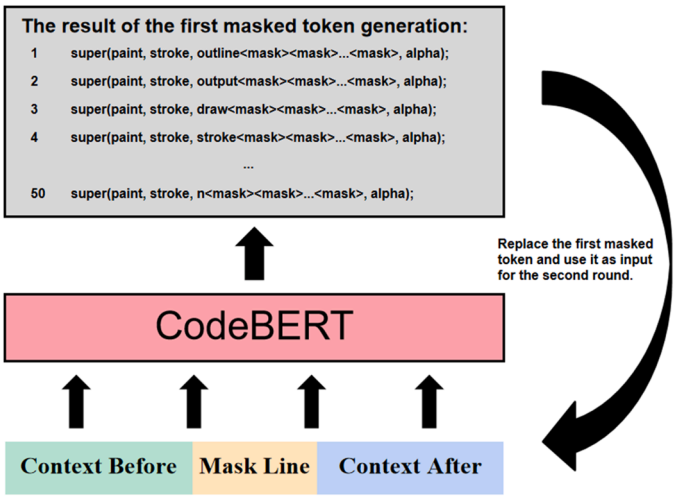


Fig. 8. Patch generation example.

3.5. Patch validation

Confix generates candidate patches for a defective program and applies the corresponding changes. The patches that do not compile are filtered out, and the patched program is then tested using the test suite to identify plausible patches that pass all test cases. Finally, an automated tool is used to compare the consistency of plausible patches with ground-truth patches, and a thorough manual inspection ensures that the program is correctly repaired.

4. Experimental setup

4.1. Research questions

In this paper, the effectiveness of Confix is demonstrated by designing the following research questions:

RQ1: How does Confix perform compared to the latest APR techniques?

RQ2: How generalizable is Confix across multiple programming languages?

RQ3: How effective are the proposed fix strategies in the patch generation phase?

In order to compare the effectiveness of the patch generation phase with the latest APR techniques, we adopt perfect fault localization to identify suspicious lines of code. Perfect fault localization is the preferred comparative setting for recent learning-based APR approaches (Xia et al., 2023), which eliminates the influence of different defect localization techniques adopted by different APR techniques and further demonstrates the effectiveness of Confix in patch generation.

4.2. Benchmarks

In order to evaluate the performance of Confix in repairing defective programs, we compare it with other APR techniques using the Defects4J 1.2 (Just et al., 2014) benchmark dataset widely adopted by the APR community to answer research question 1. Defects4J 1.2 is an open-source toolkit for software defect analysis and testing, which contains a collection of 395 real-world, verified, and documented reproducible bug sets. Each defective program includes a buggy version, a fixed version, and a corresponding test suite. To answer research question 2, we use QuixBugs (Lin et al., 2017) to evaluate the generality of Confix's repairs across different programming languages. QuixBugs consists of 40 buggy programs containing single-line defects, and each buggy program has both a Python version and a Java version for investigating the cross-language performance of repair tools. To answer research question 3, we conducted ablation experiments on the QuixBugs benchmark to investigate the improvement of fix strategies in terms of performance and efficiency.

4.3. Baselines

Confix is an approach that combines template-based and learning-based APR techniques. To fully evaluate the effectiveness of Confix, we compare it with state-of-the-art template-based APR techniques and learning-based APR techniques. We choose the template-based APR tool TBar (Kui Liu et al., 2019) for comparison to verify the improvement of the repair performance of the pre-trained model in terms of editing expressiveness. It should be noted that Confix has fewer fix templates than TBar. In addition, we compared Confix with five other APR tools that use NMT models for repairing buggy programs, including SequenceR (Chen et al., 2019), CURE (Jiang et al., 2021), DLFix (Li et al., 2020), CoCoNuT (Lutellier et al., 2020), Recoder (Zhu et al., 2021). To verify the improvement of Confix by node-level fix templates and fix strategies, we compare with the recent advanced MLM-based APR tool AlphaRepair (Xia and Zhang, 2022). Furthermore, we compare with several well-performing APR tools (Jiang et al., 2021; Lutellier et al., 2020; Zhu et al., 2021; Drain et al., 2021) on the QuixBugs benchmark to verify Confix's generality across different programming languages.

4.4. Implementation

Confix can generate code tokens for mask tokens by reusing the parameters of the pre-trained CodeBERT model without requiring additional training or fine-tuning. Confix chooses a reasonable number of mask tokens when selecting fix templates based on the node type and hierarchy information of the code elements in the code information tree. Since CodeBERT requires tokenized vector representation as input, in the fix template selection phase, the suspicious code lines are masked and replaced according to the applicable template. In the patch generation phase, we need to set the beam width for each preprocessed mask line to limit the number of candidate patches. For Defects4J 1.2, the correct patches of some defective programs are ranked between 40 and 50, and a small number are ranked after 50. When the beam width exceeds 50, some defective programs cannot be repaired due to timeout. When the beam width is less than 50, some defective programs cannot be repaired because the correct patch is not in the candidate list. In order to make Confix the best repair performance, we set the beam width to 50, which means that only 50 candidate patches are retained for each preprocessed masked line input. After generation, candidate patches are validated based on their joint scores. To ensure fairness, we set a time limit of 5 h, similar to other APR techniques (Li et al., 2020; Lutellier et al., 2020; Xia and Zhang, 2022), for bug fixing. All experiments were conducted on a Ubuntu server with an NVIDIA GeForce RTX 3090 GPU for patch generation and validation.

4.5. Evaluation metrics

We adopt two widely used performance evaluation metrics in the APR community: plausible patches and correct patches. If the program can pass all test cases after applying the candidate patch, the candidate patch is considered reasonable. Correct patches must pass all test suites and be semantically equivalent to the developer's patch. After obtaining plausible patches, an automated tool is first used to determine whether it is consistent with the ground-truth patch provided by the benchmark data set. Inconsistent patches will be further manually checked for equivalence with ground-truth patches. The plausible patch is considered correct if it is semantically equivalent to the ground-truth patch. Otherwise, it is merely considered plausible.

5. Result analysis

5.1. RQ1: performance evaluation

To evaluate the performance of Confix, we compared it with template-based and learning-based APR techniques on the Defects4J 1.2 under perfect fault localization conditions. Table 2 shows the number of bugs successfully fixed by different APR techniques on the Defects4J 1.2 dataset (correct patch / plausible patch). The results showed that Confix successfully fixed more bugs than the state-of-the-art template-based APR technology Tbar, fixing 15 more bugs (22.1 %). It should be noted that Confix has fewer fix templates than TBar, which shows that even with more fix templates, better repair performance cannot be obtained under the limitation of low edit expressiveness. Moreover, Confix outperforms Recoder and AlphaRepair for models trained on NMT and MLM, fixing 18 (27.7 %) and 9 (12.2 %) more defective programs, respectively. Overall, Confix performed the best, fixing 11, 26, 16, 22, 5, and 3 bugs in Chart, Closure, Lang, Math, Mockito, and Time projects on Defects4J 1.2, respectively.

To investigate the effectiveness of Confix further, we evaluated its performance in uniquely fixing bugs compared to other APR techniques. As shown in Fig. 9, Confix fixed 17 unique bugs that other APR techniques could not fix. Confix correctly fixed 30 unique bugs compared to the template-based APR technique TBar, indicating that enhancing edit expressiveness using pre-trained language models improves repair performance under the premise of fix templates. We further illustrate by providing unique bug examples that only Confix can fix. Fig. 10 shows the case Closure-52 from Defects4J 1.2, where although TBar has a template "Mutate Return Statement" for fixing this bug, it cannot generate the correct patch due to the lack of applicable code snippets in its codebase. Confix inserts mask tokens using the template FT8.3 to add the missing conditional expression and utilizes CodeBERT to predict the mask, thereby generating the correct repair result.

Compared to learning-based APR techniques, Confix correctly fixed 34 and 28 more unique bugs than CURE and Recoder, respectively, indicating that selecting the correct fix template for preprocessing buggy lines can generate correct patches for defective programs more efficiently and accurately. As shown in Fig. 11, we chose bug Lang-57 from Defects4J 1.2 for illustration. Confix performed masked replacement on "cAvailableLocaleSet" using template FT6.1, generating the correct code tokens "availableLocaleList()" at the node level. Unlike other APR techniques that treat program repair as a NMT task, Confix focuses on generating the required repair results at the node level. This approach reduces the likelihood of generating erroneous or irrelevant semantic code by pre-trained language models.

As shown in Fig. 12, compared with AlphaRepair, an APR technique based on MLM for defect program repair, Confix successfully fixes 17 additional unique bugs. This can be attributed to Confix's ability to mine and extend templates based on node types and hierarchies in the code information tree, resulting in a more comprehensive set of fix template definitions than AlphaRepair. Moreover, by introducing fix strategies, Confix mitigates the impact of noise introduced by training data from multiple programming languages on patch generation. Overall, by combining fix templates and pre-trained language models to enhance edit expressiveness during patch generation, Confix improves the probability of generating correct patches, providing a new approach for APR techniques.

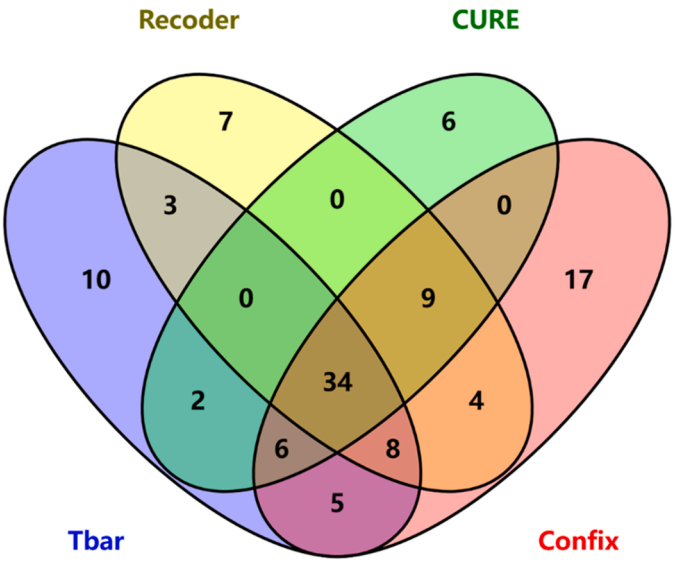


Fig. 9. Unique bugs fixed by different approaches on Defects4J 1.2.

5.2. RQ2: Generalizability of Confix

In order to demonstrate that the performance of Confix is not a result of overfitting on the Defects4J 1.2 benchmark, we chose to evaluate its performance and cross-language capabilities on the QuixBugs dataset. Table 3 presents the performance of Confix on the QuixBugs dataset, which includes Java and Python version defective programs. CodeBERT is trained jointly in multiple programming languages, enabling Confix to be directly applied to datasets in different programming languages without additional training. In the repair of Java version defective programs, Confix fixes 13 (76.5 %), 4 (15.4 %), and 2 (7.1 %) more bugs compared to Recoder, CURE, and AlphaRepair, respectively. In the repair of Python version defective programs, Confix fixes 14 (73.7 %), 12 (57.1 %), and 6 (22.2 %) more bugs compared to CoCoNuT, DeepDebug, and AlphaRepair, respectively.

In the same function but different versions of defective programs, Confix fixes 3 more bugs in the Python version than the Java version, which is related to the fact that Python language is more concise in syntax. Repairing Python version defective programs requires single-node repair, while corresponding Java version defective programs require multi-node or multi-statement repair. This is why Confix performs better in Python version defective programs.

Overall, Confix fixes most bugs in Java and Python (30 and 33, respectively), demonstrating its excellent performance and cross-programming language repair capability. The cross-language repair capability means that Confix is no longer limited to the programming language used by the defective program to be repaired. After the user provides the defective program code, fault location, fix template selection, and patch generation can be performed in one stop. It should be

Table 2
Baseline comparisons on Defects4J 1.2.

Project	SequenceR	CoCoNuT	CURE	DLFix	Recoder	Tbar	AlphaRepair	Confix
Chart	3	7	10	5	10	11	9	**11**
Closure	3	9	14	11	21	16	23	**26**
Lang	2	7	9	8	11	13	13	**16**
Math	6	16	19	13	18	22	21	**22**
Mockito	0	4	4	1	2	3	5	**5**
Time	0	1	1	2	3	3	3	**3**
Total	14/19	44/85	57/104	40/68	65/112	68/95	74/109	**83/105**

```
                return false;
            }
        }
-       return len > 0;
+       return len > 0 && s.charAt(0) != '0';
    }
    static double getSimpleNumber(String s) {
```

Code Change Action:
Insert an expression in the Return statement.
Mask Line:
return len > 0 <mask><mask>...<mask>;

Fig. 10. Fixes and code change action for bug Closure-52.

```
public static boolean isAvailableLocale(Locale locale) {
-       return cAvailableLocaleSet.contains(locale);
+       return availableLocaleList().contains(locale);
}
```

Code Change Action:
Replace the method "cAvailableLocaleSet" with mask tokens.
Mask Line:
return <mask><mask>...<mask>.contains(locale);

Fig. 11. Fixes and code change action for bug Lang-57.

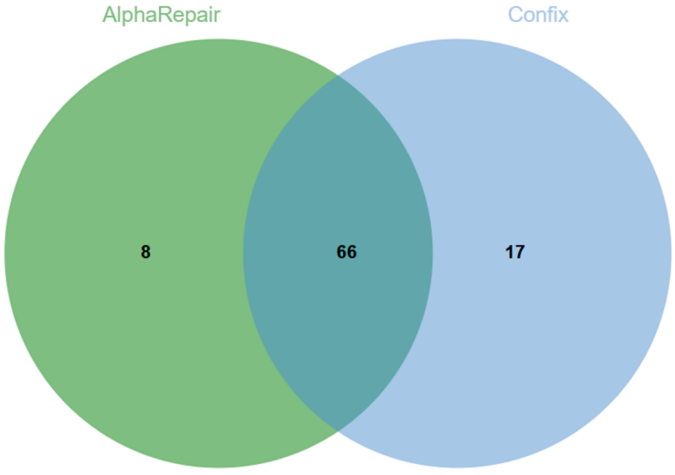


Fig. 12. Unique bugs fixed by Confix compared to AlphaRepair on Defects4J 1.2.

noted that some bugs require fix strategies to improve the effectiveness of CodeBERT in patch generation, thereby enabling Confix to achieve optimal results compared to the baselines. The reasons for the improvement brought by fix strategies to Confix will be discussed in Research Question 3.

5.3. RQ3: Fix strategies effectiveness

We performed ablation experiments on the QuixBugs dataset to evaluate the effectiveness of various fix strategies. The aim was to demonstrate how different fix strategies affect the performance and efficiency of Confix during the patch generation phase.

1) Strategy 1: Add shielded tokens

As shown in Table 4, fix strategy 1 contributed to 4 repairs (4.8 %) in Defects4J 1.2 and 3 repairs (4.8 %) in QuixBugs. In addition, when combined with several other fix strategies (Introduce allocation coefficients and Gradient beam width), fix strategy 1 can also contribute an additional 4 (4.8 %) and 2 (3.1 %) repairs. In repairs involving strategy 1, 6 / 8 (75 %) bugs (Defects4J 1.2) and 4 / 5 (80 %) bugs (QuixBugs) are insertion-type repairs, demonstrating the effectiveness of strategy 1 for insertion-type repairs. As CodeBERT is influenced by context when generating code tokens for each mask token, invalid characters that are non-node types are included in the candidate list. For insertion-type fix templates, since only mask tokens are inserted between statements without providing constrained tokens, CodeBERT generates comment-type code tokens (e.g., "#" and "//") for the first mask token. This is not the desired generation result, as it makes the subsequent code tokens part of the comment

Table 4
Component contribution on Defects4J 1.2 and QuixBugs.

Component	Defects4J 1.2	QuixBugs
Fix templates + CodeBERT	72	56
Add shielded tokens	4	3
Introduce allocation coefficients	2	1
Gradient beam width	1	1
Add shielded tokens + Introduce allocation coefficients + Gradient beam width	4	2
Total correct patches	83	63

Table 3
Baseline comparisons on QuixBugs.

Project	CoCoNuT	Recoder	DeepDebug	CURE	AlphaRepair	Confix
Java	13	17	/	26	28	**30**
Python	19	/	21	/	27	**33**

lines. Additionally, non-corresponding node-type tokens should be excluded when using specific node-type fix templates. For example, When replacing operators, tokens such as "->" and ":" in the candidate list should be excluded.

Fig. 13 shows the ranking of correct patches in the candidate list with and without strategy 1, further demonstrating the effectiveness of strategy 1. When the beam width is set to 50, patches ranking more than 50 cannot be repaired. Without fix strategy 1, the correct patches for five defective programs are not in the candidate list. After using fix strategy 1, the maximum ranking of the five defective programs in the candidate list was significantly reduced from 143 to 15. In addition, the average ranking of all correct patches is from 15 to 6, which allows the correct patches to be found in advance. Fix strategy 1 improves the ranking of correct patches while also improving repair efficiency.

2) Strategy 2: Limit the number of mask tokens and remove the same modification action inputs

When replacing code elements with mask tokens, the number of mask tokens can be determined based on the node type, as the fix template is node-type-based. For example, in the case of FT6.1 node-level variable replacement, the number of tokens obtained by tokenizing all the defined variables from the entire project context can be used as the upper limit for replacing mask tokens. Reducing the number of masked lines defined by an incorrect number of mask tokens prevents repair timeouts caused by excessive input of masked lines. Additionally, when applying fix templates to buggy lines, different fix templates may generate the same modification action. For instance, an expression in a control statement could be replaced simultaneously by FT4.3 and FT7.3. To avoid duplicate generation, we remove masked lines with identical modifications.

Fig. 14 shows the number of preprocessed masked lines for each defective program with and without strategy 2. It can be seen that after using strategy 2, the number of preprocessed masked lines is significantly reduced, with the maximum number decreasing by 21 %, from 390 to 308. In addition, the average number of preprocessed masked lines for defective programs in QuixBugs is reduced by 18.7 %, from 224 to 182. Confix may need to generate and verify many error patches when there are many preprocessed masked lines, which is not conducive to the repair efficiency of the APR tool. By using fix strategy 2, reducing the number of preprocessing mask lines can improve the repair efficiency while reducing the likelihood of repair failure due to timeouts where the correct patch is not used.

3) Strategy 3: Introduce allocation coefficients

As shown in Table 4, fix strategy 3 contributes 2 (2.4 %) and 1 (1.2 %) repairs in Defects4J 1.2 and QuixBugs, respectively. In addition, when combined with several other fix strategies (Add shielded tokens and Gradient beam width), fix strategy 3 contributes an additional 4 (4.8 %) and 2 (2.4 %) repairs. The allocation coefficient can solve the problem of failed repairs caused by a low joint score of previous mask tokens. When Confix generates code tokens for previous mask tokens in a long sequence, if the correct repair has already been excluded from the candidate list, subsequent mask tokens will fail to obtain the correct patch.

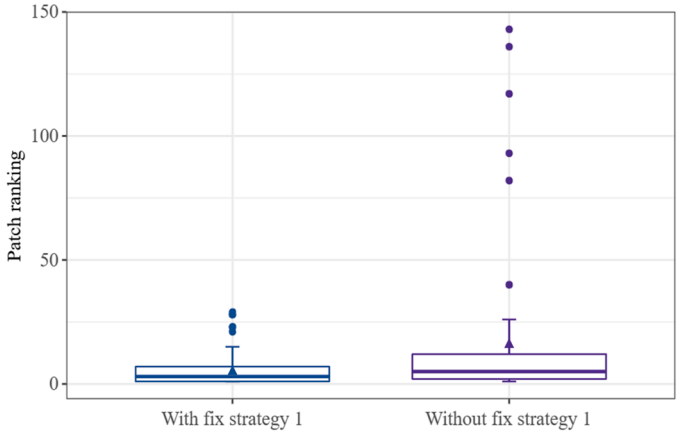


Fig. 13. Patch ranking of 63 correct fixes with and without fix strategy 1 on QuixBugs.

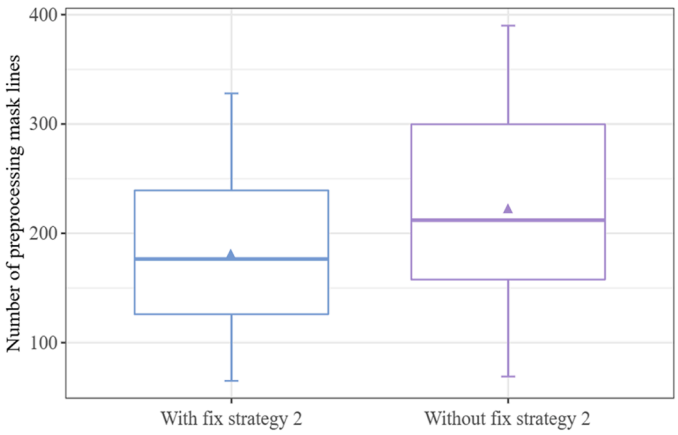


Fig. 14. Number of preprocessing mask lines with and without fix strategy 2 on QuixBugs.

During the experimental process, we found that when given partial correct code tokens, the conditional probability of the subsequent mask tokens corresponding to the code tokens being correctly repaired in CodeBERT is close to 1 and consistently ranks first in the candidate list. Inspired by this, we introduced allocation coefficients to reduce the influence of joint scores of previous mask tokens, ensuring that the correct repair is not prematurely excluded from the candidate list. As the joint score improves, the ranking of the correct repair candidate rises continuously and remains in the final candidate list. Fig. 15 shows an example defect program in QuixBugs, where after the fourth round of mask token generation, the correct repair ranks 72nd in the candidate list. Under the condition of a beam width set to 50, Confix cannot generate the correct patch for this buggy program. However, after using allocation coefficients, the correct repair ranks 37th in the candidate list and 19th in the final candidate list.

4) Strategy 4: Gradient beam width

During the generation process of the first few mask tokens, allocation coefficients can be used to keep the correct patch in the candidate list. However, this strategy may also include some incorrect patches in the list. To prevent any negative impact on the final repair, the gradient beam width is set to increase the space of the candidate patch list, ensuring that the correct repair can always exist in the candidate patch list. As shown in Table 4, fix strategy 4 contributes 1 (1.2 %) and 1 (1.2 %) repairs in Defects4J 1.2 and QuixBugs, respectively. In addition, when combined with several other fix strategies (Add shielded tokens and Introduce allocation coefficients), fix strategy 4 contributes an additional 4 (4.8 %) and 2 (2.4 %) repairs. Among them, the combined use of strategy 3 and strategy 4 accounted for the majority, indicating that strategy 4 effectively solved the negative impact of strategy 3.

6. Threats to validity

Confix utilizes node-based edits to mine fix templates, which may pose a validity threat. To construct the code information tree and further merge and expand template types, it is necessary to predefine AST node semantic context rules and merge fix templates based on node types. This paper only defines the rules required for Java and Python versions.

```
        return True
    else:
+       nodesvisited.add(node)
        return any(
            search_from(nextnode) for nextnode in node.successors
        )
```

Code Change Action:
Insert a simple statement represented by mask tokens.
Mask Line:
<mask><mask>...<mask>

Fig. 15. Fixes and code change action for bug depth_first_search.

For different programming languages, the structure of the corresponding AST will change. Therefore, defining syntax rules for each programming language to construct a code information tree is very time-consuming. Building an automated tool allows code elements and their hierarchical information to be directly extracted without relying on AST. First, the predefined rules are used to parse the code, and the corresponding lexical and syntax analyzer is selected according to the syntactic structure and characteristics of the programming language to generate a grammatical information tree. Then, the hierarchical structure is determined by analyzing the relationship between the code and element structures. Finally, the identified code elements and their corresponding hierarchical relationships are indexed to form a code information tree. This tool can automatically identify the program's language and generate a code information tree, significantly improving the efficiency and versatility of the APR tool. This paper has demonstrated the effectiveness of Confix by building code information trees for Java and Python versions.

Another threat arises from the evaluation of plausible patches to determine their correctness. Although the plausible patches pass all test cases, they may not be semantically equivalent to the developer's patch. Therefore, after obtaining plausible patches, an automated tool is first used to determine whether it is consistent with the ground-truth patch provided by the benchmark data set. Inconsistent patches will be further manually checked for equivalence with ground-truth patches. The plausible patch is considered correct if it is semantically equivalent to the ground-truth patch. Since Confix limits the repair scope of defective programs through fix templates, the plausible patches generated are almost always consistent with ground-truth patches. In Defects4J 1.2 and Quixbugs, 3 (3.6 %) and 1 (1.6 %) plausible patches were considered correct, respectively. In addition to manual inspection by the authors, the results of previous repair tools were also used to evaluate the correctness of these four patches to ensure fairness.

In addition, CodeBERT's training set consists of more than 6 million code functions, so there will inevitably be some data overlap with the evaluation benchmark Defects4J 1.2. Therefore, we checked the number of functions in Defects4J 1.2 included in the CodeBERT training data. In total, 65 functions overlapped with training data. Confix can repair 83 defective programs in Defects4J 1.2, 6 (7.2 %) of which are in the CodeBERT training data. We modified these six defective programs to prove that CodeBERT performs retrieval based on context and generates corresponding patches. Without changing the program function, the input of CodeBERT is interfered with by modifying the original variable names and adding redundant statements. This interference increases the repair difficulty to a certain extent, but it can fully prove that Confix does not simply overfit the training data. The experiments showed that Confix could still generate correct patches for all bugs despite using modified versions. Moreover, in order to further demonstrate the effectiveness of Confix, we conducted performance comparison experiment on the QuixBugs dataset. The QuixBugs dataset does not overlap with CodeBERT's training data, but Confix also achieves state-of-the-art performance compared to the benchmark approaches.

7. Conclusion

For automated program repair techniques, template-based approaches rely on predefined templates to generate patches for buggy programs in a controllable manner. However, they are limited by the types of templates and edit expressiveness. Learning-based approaches can generate fix patches more flexibly, but there is some uncontrollability in the patch generation phase. To fully leverage the advantages of these two approaches, this paper proposes Confix. Confix constructs a code information tree to assist in mining node-level fix templates and expanding template types, eliminating the limitations of traditional template-based approaches and the uncontrollability of learning-based approaches. Additionally, Confix utilizes CodeBERT to predict masked lines defined by the fix templates and introduces fix strategies to generate more reasonable patches. It is important to note that CodeBERT does not require additional training data sets for training or fine-tuning, making Confix more convenient and efficient. Confix demonstrates state-of-the-art performance on Defects4J 1.2, which is widely used by the APR community. On the QuixBugs dataset, Confix successfully repaired many bugs in both Java and Python, demonstrating its cross-language repair capabilities. In future work, we will attempt to integrate large language models and thinking chains to improve repair performance further.

CRediT authorship contribution statement

Jianmao Xiao: Writing – review & editing, Validation, Supervision, Methodology, Investigation. **Zhipeng Xu:** Writing – original draft, Validation, Methodology, Investigation. **Shiping Chen:** Writing – review & editing, Validation, Supervision, Conceptualization. **Gang Lei:** Validation, Supervision. **Guodong Fan:** Writing – review & editing, Validation. **Yuanlong Cao:** Validation, Supervision. **Shuiguang Deng:** Validation. **Zhiyong Feng:** Validation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

This work was supported by the Natural Science Foundation of Jiangxi Province (Grant No. 20224BAB212015, 20224ACB202007), Jiangxi Provincial 03 Special Project and 5G Project (20224ABC03A13, 20232ABC03A26), the National Natural Science Foundation of China under grant No. 62067003, 62262030 and 62363015.

References

AST wiki page. https://en.wikipedia.org/wiki/Abstract_syntax_tree. 2023.

Bavishi, Rohan, Yoshida, Hiroaki, Prasad, Mukul R., 2019. 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.

Campos, José, et al., 2012. Gzoltar: an eclipse plug-in for testing and debugging. In: Proceedings of the 27th IEEE/ACM international conference on automated software engineering.

Chawathe, Sudarshan S., et al., 1996. Change detection in hierarchically structured information. Acm. Sigmod. Record. 25.2, 493–504.

Chen, Zimin, et al., 2019. Sequencer: sequence-to-sequence learning for end-to-end program repair. IEEE Trans. Software Eng. 47.9, 1943–1959.

Devlin, Jacob, et al. "Bert: pre-training of deep bidirectional transformers for language understanding." arXiv preprint arXiv:1810.04805 (2018).

Drain, Dawn, et al. "Deepdebug: fixing python bugs using stack traces, backtranslation, and code skeletons." arXiv preprint arXiv:2105.09352 (2021).

Falleri, Jean-Rémy, et al., 2014. Fine-grained and accurate source code differencing. In: Proceedings of the 29th ACM/IEEE international conference on Automated software engineering.

Feng, Zhangyin, et al. "Codebert: a pre-trained model for programming and natural languages." arXiv preprint arXiv:2002.08155 (2020).

Gazzola, Luca, Micucci, Daniela, Mariani, Leonardo, 2019. Automatic Software Repair: a Survey. IEEE Trans. Software Eng. 45 (01), 34–67.

Guo, Daya, et al. "Graphcodebert: pre-training code representations with data flow." arXiv preprint arXiv:2009.08366 (2020).

Hua, Jinru, et al., 2018. Sketchfix: a tool for automated program repair approach using lazy candidate generation. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.

Jiang, Jiajun, et al., 2018. Shaping program repair space with existing patches and similar code. In: Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis.

Jiang, Nan, Lutellier, Thibaud, Tan, Lin, 2021. Cure: code-aware neural machine translation for automatic program repair. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE.

Jiang, Tammy, Gradus, Jaimie L., Rosellini, Anthony J., 2020. Supervised machine learning: a brief primer. Behav. Ther. 51.5, 675–687.

Just, René, Jalali, Darioush, Ernst, Michael D., 2014. 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.

Koyuncu, Anil, et al., 2019. iFixR: bug report driven program repair. In: Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering.

Koyuncu, Anil, et al., 2020. Fixminer: mining relevant fix patterns for automated program repair. Empirical Software Engineering 25, 1980–2024.

Le, Xuan, Bach, D., et al., 2017. S3: syntax-and semantic-guided repair synthesis via programming by examples. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering.

Le, Xuan, Bach, D., Lo, David, Goues, Claire Le, 2016. History driven program repair. In: 2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER), 1. IEEE.

Le Goues, Claire, et al., 2011. Genprog: a generic method for automatic software repair. In: Ieee Transactions On Software Engineering, 38.1, pp. 54–72.

Li, Xia, et al., 2019. Deepfl: integrating multiple fault diagnosis dimensions for deep fault localization. In: Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis.

Li, Yi, Wang, Shaohua, Nguyen, Tien N., 2020. Dlfix: context-based code transformation learning for automated program repair. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering.

Li, Yi, Wang, Shaohua, Nguyen, Tien N., 2022. Dear: a novel deep learning-based approach for automated program repair. In: Proceedings of the 44th International Conference on Software Engineering.

Lin, Derrick, et al., 2017. 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.

Liu, Kui, et al., 2019b. 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.

Liu, Kui, et al., 2019a. TBar: revisiting template-based automated program repair. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis.

Liu, Xiao, et al., 2021. Self-supervised learning: generative or contrastive. IEEE Trans. Knowl. Data Eng. 35.1, 857–876.

Liu, Xuliang, Zhong, Hao, 2018. Mining stackoverflow for program repair. In: 2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER). IEEE.

Long, Fan, Amidon, Peter, Rinard, Martin, 2017. Automatic inference of code transforms for patch generation. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering.

Lu, Shuai, et al. "Codexglue: a machine learning benchmark dataset for code understanding and generation." arXiv preprint arXiv:2102.04664 (2021).

Lutellier, Thibaud, et al., 2020. 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.

Mechtaev, Sergey, et al., 2018. Semantic program repair using a reference implementation. In: Proceedings of the 40th International Conference on Software Engineering.

Radford, Alec, et al., 2019. Language models are unsupervised multitask learners. OpenAI Blog 1 (8), 9.

Shariffdeen, Ridwan Salihin, et al., 2020. Automated patch transplantation. ACM Transactions on Software Engineering and Methodology (TOSEM) 30.1, 1–36.

Soto, Mauricio, Goues, Claire Le, 2018. Using a probabilistic model to predict bug fixes. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE.

Vaswani, Ashish, et al., 2017. Attention is all you need. Adv. Neural. Inf. Process Syst. 30.

Villanueva, Omar M., Trujillo, Leonardo, Hernandez, Daniel E., 2020. Novelty search for automatic bug repair. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference.

Wang, Changhan, Cho, Kyunghyun, Gu, Jiatao, 2020. Neural machine translation with byte-level subwords. In: Proceedings of the AAAI conference on artificial intelligence, 34.

Wang, Yue, et al. "Codet5: identifier-aware unified pre-trained encoder-decoder models for code understanding and generation." arXiv preprint arXiv:2109.00859 (2021).

Wardat, Mohammad, Le, Wei, Rajan, Hridesh, 2021. Deeplocalize: fault localization for deep neural networks. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE.

Wong, W.Eric, et al., 2016. A survey on software fault localization. IEEE Trans. Software Eng. 42.8, 707–740.

Xia, Chunqiu Steven, Wei, Yuxiang, Zhang, Lingming, 2023. 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.

Xia, Chunqiu Steven, Zhang, Lingming, 2022. 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.

Xu, Xuezheng, et al., 2019. VFix: value-flow-guided precise program repair for null pointer dereferences. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE.

Yang, Deheng, et al., 2021. 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 26, 1–33.

Yang, Shuoheng, Yuxin Wang, and Xiaowen Chu. "A survey of deep learning techniques for neural machine translation." arXiv preprint arXiv:2002.07526 (2020).

Zhu, Qihao, et al., 2021. 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.

Jianmao Xiao received his Ph.D. from the College of Intelligence and Computing, Tianjin University. He is an assistant professor at Jiangxi Normal University. His-main research interests include service computing and intelligent software engineering.

Zhipeng Xu is currently pursuing an M.S. degree in software engineering at the Jiangxi Normal University. His-research interests include automatic program repair and intelligent software engineering.

Shiping Chen is a principal research scientist at CSIRO Data61. He also holds an adjunct associate professor title with the University of Sydney through teaching and supervising PhD/Master students. He has been working on distributed systems for over 20 years with focus on performance and security. He has published over 100 research papers in these research areas. He is actively involved in computing research community through publications, journal editorships and conference PC services, including WWW, EDOC, ICSOC and IEEE ICWS/SCC/CLOUD. His-current research interests include secure data storage & sharing and secure multi-party collaboration. He is a senior member of the IEEE and a fellow of Institution of Engineering and Technology (IET).

Gang Lei is currently the Dean, associate professor and master tutor of Software College of Jiangxi Normal University. His-research fields are data mining, machine learning and computer linguistics. He has presided over or participated in more than 20 projects, published more than 20 academic papers and obtained 4 national software copyrights.

Guodong Fan is currently pursuing a Ph.D. degree in Computer Science and Technology with the College of Intelligence and Computing, Tianjin University. He is working on Cognitive Services. His-main research interests are Representation Learning and Service Computing.

Yuanlong Cao received the B.S. degree in computer science and technology from Nanchang University, China, in 2006, the M.S. degree in software engineering from the Beijing University of Posts and Telecommunications (BUPT), Beijing, China, in 2008, and the Ph. D. degree in communication and information system from the Institute of Network Technology, BUPT, in 2014. He was an Intern/Software Engineer with BEA TTC, IBM CDL, and DT Research, Beijing, China, from 2007 to 2011. He is currently a Professor with the School of Software, Jiangxi Normal University, Nanchang, China. His-research interests

include multimedia communications, network security and next-generation Internet technology.

Shuiguang Deng (Senior Member, lEEE) received the BS and PhD degrees in computer science, from the College of Computer Science and Technology in Zhejiang University, China, in 2002 and 2007 respectively, where he is currently a full professor. He previously worked at the Massachusetts Institute of Technology, in 2014 and Stanford University in 2015 as a visiting scholar. His-research interests include edge computing, service computing, mobile computing, and business process management. He serves as the associate editor for the journal lEEE Access and lET Cyber-Physical Systems: Theory & Applications. Up to now, he has published more than 100 papers in journals and refereed conferences. In 2018, he was granted the Rising Star Award by IEEE TCSVC. He is a fellow of lET.

Zhiyong Feng, born in 1965. Professor and Ph.D. supervisor in the School of Software, Tianjin University. His-main research interests include service computing, software engineering, Internet of things, etc.