Template-based Neural Program Repair

Xiangxin Meng
SKLSDE Lab, Beihang University
Beijing, China
mengxx@act.buaa.edu.cn

Xu Wang*†
SKLSDE Lab, Beihang University
Beijing, China
xuwang@buaa.edu.cn

Hongyu Zhang
Chongqing University
Chongqing, China
hyzhang@cqu.edu.cn

Hailong Sun
SKLSDE Lab, Beihang University
Beijing, China
sunhl@act.buaa.edu.cn

Xudong Liu*
SKLSDE Lab, Beihang University
Beijing, China
liuxd@act.buaa.edu.cn

Chunming Hu*
SKLSDE Lab, Beihang University
Beijing, China
hucm@buaa.edu.cn

***Abstract*—In recent years, template-based and NMT-based automated program repair methods have been widely studied and achieved promising results. However, there are still disadvantages in both methods. The template-based methods cannot fix the bugs whose types are beyond the capabilities of the templates and only use the syntax information to guide the patch synthesis, while the NMT-based methods intend to generate the small range of fixed code for better performance and may suffer from the OOV (Out-of-vocabulary) problem. To solve these problems, we propose a novel template-based neural program repair approach called TENURE to combine the template-based and NMT-based methods. First, we build two large-scale datasets for 35 fix templates from template-based method and one special fix template (single-line code generation) from NMT-based method, respectively. Second, the encoder-decoder models are adopted to learn deep semantic features for generating patch intermediate representations (IRs) for different templates. The optimized copy mechanism is also used to alleviate the OOV problem. Third, based on the combined patch IRs for different templates, three tools are developed to recover real patches from the patch IRs, replace the unknown tokens, and filter the patch candidates with compilation errors by leveraging the project-specific information. On Defects4J-v1.2, TENURE can fix 79 bugs and 52 bugs with perfect and Ochiai fault localization, respectively. It is able to repair 50 and 32 bugs as well on Defects4J-v2.0. Compared with the existing template-based and NMT-based studies, TENURE achieves the best performance in all experiments.**

***Index Terms*—automated program repair, fix templates, neural machine translation, deep learning**

I. Introduction

In recent years, two types of automated program repair (APR) methods have been extensively studied. One of them, known as *template-based repair method* [1]–[13], is to leverage the predefined fix templates to generate patch candidates. The other, known as *neural machine translation (NMT) based repair method* [14]–[22], adopts NMT models [23]–[25] to generate the fixed code snippets from the buggy ones based on historical bug-commit datasets. Both approaches have been shown good performance in some repair scenarios.

However, these two kinds of APR methods have their own disadvantages. For template-based methods [1]–[7], they cannot repair the bugs whose types beyond the capabilities of the templates [20]. Also, when selecting templates and donor code to produce patches, only the syntactic information is used to provide limited guidance on how to choose them, while the semantic information is not leveraged [26]. For NMT-based methods [14]–[22], although theoretically they can automatically extract any potential repair patterns by generating a large range (e.g., the entire method) of fixed code [27], the repair performance may be relatively low because long token sequences should be produced by NMT models [28]. In practice, many NMT-based APR approaches tend to generate the small ranges of fixed code, including one code line [15], [18], [19], [22] or sub-trees of abstract syntax trees [17], [20], [21]. If the generated code range is limited, some complex and multiple-line fix templates proposed in template-based APR methods may not be effectively captured, such as wrapping an if statement outside a specific code fragment, or moving a statement to another location. But these fix templates are usually summarized or extracted from a large amount of repair data with the help of expert experience, and the produced patches are similar to human-written ones with high quality and good readability [29]. Therefore, the knowledge of existing fix templates should be incorporated into NMT-based APR methods. In addition, NMT-based methods often face the OOV (Out-of-vocabulary) problem when project-specific and low-frequency tokens such as identifiers are required in the correct patches [20], [26]. In contrast, template-based methods are more effective at searching and using project-specific information to fill in the templates. Such project-specific information can also be effectively used to improve the patch generation in NMT-based methods.

In order to solve the above problems, we propose an approach TENURE (Template-based Neural Program Repair), which simultaneously absorbs the advantages of template-based and NMT-based APR methods. On the one hand, TENURE adopts the encoder-decoder neural models to learn the deep semantic features of the buggy methods for generating the formatted fix template representations and necessary program elements filling in them as the patch intermediate representations (IRs), where the fix templates include the 35

* Zhongguancun Laboratory, Beijing, P.R.China
† Corresponding author: Xu Wang, xuwang@buaa.edu.cn.

existing ones proposed by template-based APR methods and the single-line code generation (treated as one special fix template) in NMT-based APR methods. On the other hand, based on the project-specific knowledge, our method uses the optimized copy mechanism in the encoder-decoder neural models, the syntax-based replacement strategy for ⟨unk⟩ (i.e., unknown tokens) and static patch checking for compilation errors to alleviate the OOV problem and improve the repair efficiency. The approach mainly includes three components:

1) We construct two large-scale repair datasets (i.e., $Dataset_{FT}$ and $Dataset_{SL}$), including 35 fix templates (356,629 samples) collected from existing template-based APR methods and one fix template for single-line code generation from NMT-based APR methods (223,599 samples), respectively. Each sample is the pair of a buggy code method and the corresponding patch IR.

2) For each dataset, we build two LSTM (Long Short-Term Memory) based encoder-decoder models, one with and the other without copy mechanism. Both models are trained on the corresponding dataset to generate the token sequences of patch IRs. We also propose the joint inference strategy to better predict the OOV tokens from the input methods by averaging the probabilities of models with and without the copy mechanism.

3) Based on the combined patch IRs for different templates, three tools are developed at the post-processing phase to produce the real patches by leveraging the project-specific knowledge (method-level, file-level, project-level and third-party libraries information). PRT (patch recovery tool) restores a patch IR to the corresponding real patch based on the method-level information. URT (unknown tokens replacement tool) can search code elements (e.g., identifiers) from four different levels of project-specific information to replace ⟨unk⟩ tokens that are syntactically appropriate for the patches; PCT (patch checking tool) provides a static and lightweight patch checking service for 12 common compilation errors instead of the time-consuming runtime compilation for each generated patch, helping TENURE to filter the patches with compilation errors and improve the repair efficiency.

We conduct extensive experiments on 395 and 444 real-world software bugs from the widely used Defects4J-v1.2 [30] and Defects4J-v2.0 [30] benchmarks to evaluate our approach, respectively. For each benchmark, an comparative experiment under perfect localization setting is conducted for evaluate the upper limit of repair effectiveness, while another experiment under Ochiai [31] localization setting is designed to see the repair performance in the actual repair environment. We select one state-of-the-art template-based and six deep learning-based repair methods for comparison. The results show that, no matter which benchmark and which localization setting are chosen, our method outperforms all of the comparative methods. Specifically, for Defects4J-v1.2, TENURE successfully repairs 12 more bugs and 1 more bug than all other methods in perfect localization and Ochiai localization settings respectively. For Defects4J-v2.0, TENURE successfully repairs 5 more bugs and 8 more bugs than all others in the corresponding localization settings.

The main contributions of this paper are as follows:

- We propose the formatted template representations and build two large-scale repair datasets with different fix templates.
- We propose an approach TENURE by combining the template-based and NMT-based APR methods, which consists of the encoder-decoder models capable of generating the patch IRs and a tool set for patch recovery, ⟨unk⟩ replacement and patch checking to produce the real patches.
- We conduct extensive experiments on widely-used benchmarks Defects4J-v1.2 and Defects4J-v2.0 under different fault localization settings to evaluate our approach, and the experimental results show that our approach is effective and has good generalization performance.

II. Related Work

A. Template-based APR methods

Template-based APR methods use fix templates derived from manual definition [32], [33] or code repository mining [1], [6], [34], [35] to generate code patches for specific types of bugs. Typical techniques include SimFix [7], Avatar [35], FixMiner [6], HERCULES [4], TBar [5] and so on. Avatar [35] leverages the fix patterns of static analysis violation. HERCULES [4] proposes an approach for collaboratively fixing multiple bugs. TBar [5] summarizes the previous studies and proposes 35 fix templates, which achieves the best repair effectiveness among these methods. For the bug types that the predefined fix templates can cover, template-based methods are generally able to provide the good repair performance. But for those beyond the template scope, they may not produce effective patches. In addition, during patch generation, the selection of fix templates and donor code is only guided by the syntactic information, while the semantic knowledge is not leveraged [26].

B. Deep learning-based APR methods

The early deep learning-based APR studies [36]–[38] adopt neural models to learn the code semantic for completing repair-related tasks, not directly generating the patches. For example, DeepRepair [38] learns the similarities between the buggy code and the repair ingredients to give a guidance for patch generation, and DeepFix [37] repairs syntax errors by learning the syntax rules. Recently a lot of NMT-based APR methods [15], [16], [18]–[22] are proposed, which use the encoder-decoder models to capture the program semantics of buggy code snippets for generating patches. For model input, some of them split the buggy code fragments into sequences of tokens by simple delimiters like spaces and underscores, such as SequenceR [15] and CoCoNuT [18], while the others represent code snippets as abstract syntax trees and leverage tree-based models (e.g., tree-based LSTMs [39] in DLFix [17] and TreeGen [40] in Recoder [20]) to learn the code structure information. RewardRepair [22] leverages the program execution information to further optimize the repair performance,

which can effectively improve the compilation pass rate of generated patch candidates as well.

In the above NMT-based APR methods, the existing fix templates presented in template-based APR methods are ignored even though they are proved to be helpful with a lot of expert experience [29]. A more recent work TRANSFER [26] uses a multi-classifier to learn the deep semantic features to give a guidance for the ranking of the predefined fix templates, which achieves better repair performance than many NMT-based APR methods. Thus it is desirable to incorporate existing fix templates into the NMT-based methods. In addition, it is usually difficult for NMT-based APR methods to predict project-specific and low-frequency tokens such as identifiers in the expected patches and may encounter the OOV problem [20], [26]. In contrast, such project-specific information are widely used in template-based APR methods to fill in the templates. Thus we can borrow the ideas from template-based methods and help NMT-based methods to tackle the OOV problem for better patch generation.

III. Proposed Approach

A. Overview

In this work we propose a novel APR method TENURE which simultaneously absorbs the advantages of template-based and NMT-based APR methods. As shown in Figure 1, TENURE mainly includes three components. The first component (Section III-B) proposes the formatted template representations and builds two large-scale repair datasets for 35 fix templates from template-based and 1 fix template from NMT-based APR methods. The second component (Section III-C) adopts the encoder-decoder neural models to learn the deep semantic features of the buggy methods for generating the patch intermediate representations (IRs) by combining the results of the two datasets. In addition, it also proposes the joint inference strategy to better predict the OOV tokens from the input methods. The third component (Section III-D) contains three tools for patch recovery (PRT), unknown tokens replacement (URT) and patch checking (PCT) to produce the real patches.

B. Dataset Construction for Fix Templates

To obtain sufficient repair data, we first download 2,000 Java projects with most stars on GitHub, and follow the same way as described in [26] to perform data pre-processing. First, to prevent data leakage, all projects in Defects4J-v1.2 and Defects4J-v2.0 are removed from the collected projects because they exist in the target benchmarks. Then, all commits related to bug fix are extracted. Specifically, a commit is considered bug-relevant only if its message contains at least one of the keywords: "bug", "fix", "error", "issue", "mistake", "defect", "incorrect", "fault", "flaw", "type" [41]. We keep the commits where only one method is modified, and about one million method pairs (the methods before and after the commit) are preserved in total.

Inspired by template-based APR methods [5]–[7], we believe that the code modifications of a considerable part of the collected method pairs can match some frequently-used human-defined fix templates. We use the same 15 categories of fix templates as in [5] and [26], which can be divided into 35 more fine-grained templates. Similar with previous work [26], the collected method pairs are divided into 35 groups, each of which corresponds to a certain fix template. And if no template matches, the method pair will be discarded.

TABLE I
Definitions of 4 Representative Fix Templates

No.	Fix Templates	Code Change Actions
1	Insert Null Pointer Checker 1	+ if (exp != null) { ... exp ... ; + }
2	Move Statement 1	- statement ; ... + statement ;
3	Remove Buggy Statement 1	... - statement ; ...
4	Mutate Variable 1	- ... var_1 ... ; + ... var_2 ... ;

Afterwards, we analyze and summarize the program elements which are necessary for patch synthesis of each fix template. Given the current buggy statement, the template-based patch synthesis will generate a patch skeleton with several holes, and then fill these holes with the required code elements (as called *backbone elements* in this paper) to produce the patch. Note that the backbone elements are abstract in templates, which can be replaced by actual program elements in specific projects. Table I provides four examples of fix templates, where the second column shows the template names and the third column shows the defined repair actions. The first fix template adds a null pointer checker before the expression *exp* is accessed. Since *exp* serves as the part of the newly added conditional expression, so it is the backbone element of this template. The second one moves a statement to another location, whose backbone element is a positive or negative integer used to represent the direction and code line number of movement. The third one does nothing but deletes the current buggy statement, which has no backbone element. The fourth fix template replaces an variable with another one. *var_2* is a backbone element since it is a newly added element in the fixed code snippet. Also, the patch needs to know which original variable (i.e., *var_1*) is replaced. Thus both *var_1* and *var_2* are considered as the backbone elements, which determine the overall picture of the patch together.

As shown in Table II, we list the backbone elements for all fix templates used in this paper. For each fix template, the abbreviated template name (e.g., *ICC1* in the first template) and the backbone elements (e.g., ⟨*exp*⟩ ⟨*T*⟩ in the first template) are combined as *formatted template representation*. This kind of fix template representation only retains the necessary program elements and reduces the redundancy of repair information. Besides the 35 artificial fix templates presented in the template-based APR methods [5], [26], we consider the

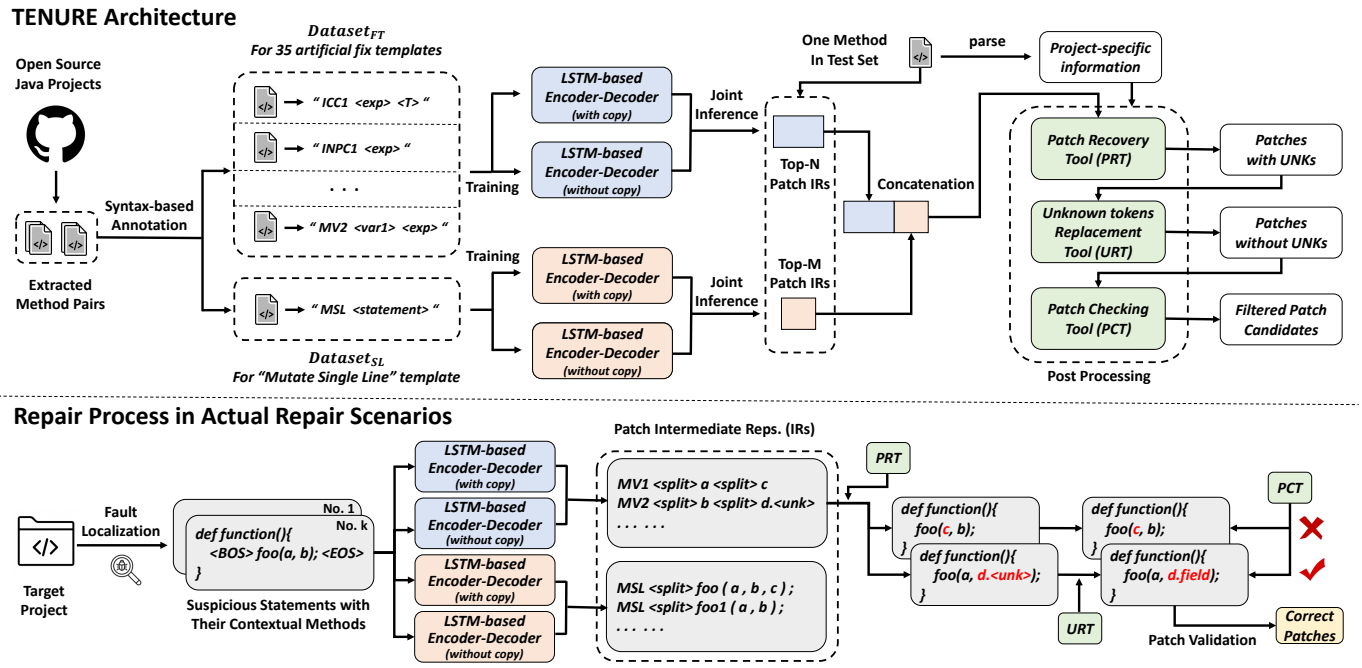


Fig. 1. An overview of TENURE, where the suspicious statement is marked with ⟨BOS⟩ and ⟨EOS⟩, and ⟨split⟩ is used as the delimiter to split the fix template name and each backbone element in patch IRs.

TABLE II
35 TEMPLATES FROM TEMPLATE-BASED AND 1 TEMPLATE (IN BOLD) FROM NMT-BASED APR METHODS, WHERE REP. MEANS REPRESENTATION

Fix Template	Formatted Template Rep.	Fix Template	Formatted Template Rep.
1. Insert Cast Checker 1	ICC1 ⟨exp⟩ ⟨T⟩	19. Mutate Data Type 1	MDT1 ⟨T1⟩ ⟨T2⟩
2. Insert Null Pointer Checker 1	INPC1 ⟨exp⟩	20. Mutate Data Type 2	MDT2 ⟨T1⟩ ⟨T2⟩
3. Insert Null Pointer Checker 2	INPC2 ⟨exp⟩ ⟨default⟩	21. Mutate Integer Division Op. 1	MIDO1 ⟨divisor⟩
4. Insert Null Pointer Checker 3	INPC3 ⟨exp⟩ ⟨exp1⟩	22. Mutate Integer Division Op. 2	MIDO2 ⟨dividend⟩
5. Insert Null Pointer Checker 4	INPC4 ⟨exp⟩	23. Mutate Integer Division Op. 3	MIDO3 ⟨dividend⟩ ⟨divisor⟩
6. Insert Null Pointer Checker 5	INPC5 ⟨exp⟩	24. Mutate Literal Expression 1	MLE1 ⟨literal1⟩ ⟨literal2⟩
7. Insert Range Checker 1	IRC1 ⟨exp⟩ ⟨index⟩	25. Mutate Literal Expression 2	MLE2 ⟨literal1⟩ ⟨exp⟩
8. Insert Range Checker 2	IRC2 ⟨exp⟩ ⟨index⟩	26. Mutate Method Invocation Exp. 1	MMIE1 ⟨method1⟩ ⟨method2⟩
9. Insert Missed Statement 1	IMS1 ⟨expression_statement⟩	27. Mutate Method Invocation Exp. 2	MMIE2 ⟨buggy_args⟩ ⟨patch_args⟩
10. Insert Missed Statement 2	IMS2 ⟨default⟩	28. Mutate Method Invocation Exp. 3	MMIE3 ⟨deleted_args⟩
11. Insert Missed Statement 3	IMS3	29. Mutate Method Invocation Exp. 4	MMIE4 ⟨point_args⟩ ⟨insert_args⟩
12. Insert Missed Statement 4	IMS4 ⟨conditional_exp⟩	30. Mutate Operators 1	MO1 ⟨op1⟩ ⟨op2⟩
13. Remove Buggy Statement 1	RBS1	31. Mutate Operators 2	MO2 ⟨b_infix_exp⟩ ⟨p_infix_exp⟩
14. Move Statement 1	MS1 ⟨move_step⟩	32. Mutate Operators 3	MO3 ⟨exp⟩ ⟨T⟩
15. Mutate Conditional Exp. 1	MCE1 ⟨cExp1⟩ ⟨cExp2⟩	33. Mutate Return Statement 1	MRS1 ⟨exp1⟩ ⟨exp2⟩
16. Mutate Conditional Exp. 2	MCE2 ⟨op⟩ ⟨cExp2⟩	34. Mutate Variable 1	MV1 ⟨var1⟩ ⟨var2⟩
17. Mutate Conditional Exp. 3	MCE3 ⟨cExp1⟩ ⟨op⟩ ⟨cExp2⟩	35. Mutate Variable 2	MV2 ⟨var1⟩ ⟨exp⟩
18. Mutate Class Instance Creat. 1	MCIC1	***36. Mutate Single Line***	***MSL ⟨statement⟩***

single-line code generation from NMT-based APR methods [15], [18], [19], [22] as a special fix template *Mutate Single Line* (i.e. the 36^{th} template *MSL* in Table II). The backbone element of this fix template is the entire code line to be generated. We treat this template as a supplement to the existing 35 artificial fix templates, and filter out a subset from the remaining method pairs (i.e., the method pairs not divided into the previous 35 artificial fix templates) that modify only one code line. Due to space limitation, we put the detailed definition of each fix template at our GitHub web page[1].

For each collected method pair, we develop a syntax-based annotation tool to produce a *patch intermediate representation (IR)* by matching the template type and replacing the backbone elements in the formatted template representation with the corresponding actual program elements, such as replacing *⟨var_1⟩* and *⟨var_2⟩* in the formatted template representation "*MV1 ⟨var_1⟩ ⟨var_2⟩*" with the variables *a* and *b*,

[1]https://github.com/mxx1219/TENURE

to generate the corresponding patch IR "*MV1 a b*". In this way, we construct two large-scale repair datasets $Dataset_{FT}$ and $Dataset_{SL}$, including 356,629 samples for 35 artificial fix templates and 223,599 samples for *MSL* fix template, respectively. Each sample is the pair of a buggy code method and the corresponding patch IR.

C. Template-based Neural Patch IR Generation

Given a buggy source code method, we use NMT-based encoder-decoder models to generate patch IRs. Compared with existing NMT-based APR methods [17]–[22], [42] that take the entire method [27], the single code line [15], [18], [22], or sub-trees of abstract syntax trees (ASTs) [20], [21] as the generation targets, we take the patch IRs as the generation target in our work. In addition, we propose the joint inference strategy to enhance the copy mechanism of our models for better predicting the low-frequency and project-specific tokens in the input methods.

1) Encoder-decoder Model for Patch IR Generation: As mentioned in Section III-B, we parse the patch to the forms of formatted template representations which only include the names of the fix templates and the backbone elements, thus the patch IRs are usually more concise and shorter in token numbers than source code or AST subtrees used in previous studies [17]–[22], [42] and can lead to better performance for NMT-based models [28]. For the fix templates with only repair actions in single-line code (e.g., *MV1*, *MO1*) if NMT models directly generate the fixed code line, the output token sequences are usually longer than the corresponding patch IRs since our patch IRs can reuse the input code line and only specify the changes. For the fix templates including multiple-line repair actions (e.g., *INPC1*, *ICC1*, *MS1*), it may produce a large range of fixed code with multiple lines if we use the source code as the target. For example, suppose a line of code needs to be moved down 5 lines to synthesize a correct patch, this cross-line repair action will involve at least 5 lines of code and a large number of tokens. However, only two tokens are needed to represent the full repair semantics when patch IR is used (i.e., "*MS1 5*").

Similar to many previous studies [16], [18], [19], [22], we use the encoder-decoder model to generate patch IRs from buggy lines and their contextual methods. The buggy lines are marked by the symbols $\langle$BOS$\rangle$ and $\langle$EOS$\rangle$. In the encoder-decoder model, we leverage a BiLSTM (Bidirectional LSTM) [43] based encoder and an LSTM [44] based decoder. For the encoder, suppose the input is a buggy method *c* containing *n* tokens $x_1, ..., x_n$, an embedding layer is used to initialize them by vectors. The encoder can extract code semantic features from forward and backward directions, which are combined later as h_t. For the decoder, a global attention mechanism [45] is adopted to produce different context vectors for output tokens which are calculated as follows:

$$v_i \;=\; \sum_{j=1}^{n} a_{ij} h_j \tag{1}$$

where h_j is the j^{th} encoder hidden state, and a_{ij} is the attention weight, which is further computed by:

$$a_{ij} \;=\; \frac{exp(s_{ij})}{\sum_{k=1}^{n} exp(s_{ik})} \tag{2}$$

$$s_{ij} \;=\; f(q_{i-1}, h_j) \tag{3}$$

Here, q_{i-1} means the previous decoder hidden state. Then, an MLP (Multi-Layer Perceptron) is adopted to compute the correlation scores between encoder and decoder hidden states (i.e., f in Equation 3). Simultaneously, the current state is updated from q_{i-1} to q_i. Then, another MLP with *softmax* activation function is used as the token generator (i.e., g_a in Equation 4), and computes the probability of each token in the vocabulary $\mathbb{V}$:

$$p_v(y_i|y_1, ..., y_{i-1}, c) \;=\; g_a(y_{i-1}, q_i, v_i) \tag{4}$$

Finally, a loss function $\mathcal{L}$ is defined to optimize this model:

$$\mathcal{L}(\theta) \;=\; -\sum_{i=1}^{N}\sum_{t=1}^{M} logP(y_t^{(i)}|y_1^{(i)}, ..., y_{t-1}^{(i)}, c) \tag{5}$$

where θ indicates the trainable parameters, *N* is the number of training samples, and *M* records the length of each target sequence. In the inference phase, we use beam search algorithm [42] to keep multiple patch IRs.

In order to combine the advantages of the template-based and NMT-based APR methods, we train encoder-decoder models on the two large-scale repair datasets $Dataset_{FT}$ and $Dataset_{SL}$ constructed in Section III-B, respectively, and then merge their generated patch IRs as the final outputs. Such combination offers three advantages. First, the semantic knowledge learned by the models trained on $Dataset_{FT}$ is beneficial for better selecting fix templates and donor code, which is not achieved by existing template-based APR methods [5], [6], [35]. Second, the models trained on $Dataset_{FT}$ make it easier to learn how the 35 artificial fix templates synthesize correct patches than existing NMT-based APR methods [18]–[22] since the concise patch IRs are used. Third, the types of bugs in $Dataset_{FT}$ and $Dataset_{SL}$ are different, so the models trained on $Dataset_{SL}$ are able to provide additional help to repair some specific single-line bugs which cannot be fixed by the models trained on $Dataset_{FT}$. Note that our approach can be flexibly extended with additional bug types, including more artificial fix templates and novel designs in code generation methods.

2) Joint Inference Strategy for Copy Mechanism Enhancement: NMT-based repair methods often face the challenge of OOV problem, which is mainly due to the presence of low-frequency and project-specific tokens. Even if our method use patch IRs as the generation target, it still faces this problem, because many of the backbone elements of patch IRs are project-specific code elements. The copy mechanism [46] is widely-used to alleviate OOV problem [15] in NMT-based repair methods, which copies low-frequency tokens from the input sequence to the corresponding patch IR. Specifically, the

copy mechanism computes the copy probabilities with the help of the intermediate results produced in the calculation of the attention weights in Section III-C:

$$p_{copy} = g_c(y_{i-1}, q_i, v_i) \quad (6)$$

where g_c is another MLP layer. After that, since the attention weight in Equation 2 can reflect the correlation between the input token x_j and the output token y_i, it can be served as the probability of selecting x_j as the copy element. The probability distributions of the output tokens are calculated as follows:

$$p_c(y_i) = (1 - p_{copy}) \cdot p_v(y_i) + p_{copy} \cdot \sum_{j:x_j=y_i} a_{ij} \quad (7)$$

where $y_i \in \mathbb{V} \cup \{x_1, ..., x_n\}$, and $p_v(y_k) = 0$ if $y_k \notin \mathbb{V}$. In this way, the special token $\langle$unk$\rangle$ in $\mathbb{V}$ can only be selected when p_{copy} is small and $p_v(<unk>)$ is large enough.

Nevertheless, since NMT-based APR methods often take a suspicious statement and its contextual method as input [18], [20], [22], the project-specific tokens outside this method cannot be generated by the copy mechanism. Therefore, we consider designing an unknown token replacement tool (i.e., URT in Section III-D2) to search syntax-correct project-specific code elements that do not exist in this method. In addition, we find that after the copy mechanism is leveraged, the model rarely generates target sequences with $\langle$unk$\rangle$ tokens, which leaves little room for the subsequent replacement task. Therefore, an approach is expected to not only take advantage of the copy mechanism to copy tokens for the elements within the method, but also preserve $\langle$unk$\rangle$ tokens as much as possible for the elements outside the method, which can be better solved in the subsequent replacement task. To achieve this goal, for each large-scale dataset, we train two encoder-decoder models, i.e., one with ($model_{(w)}$) and the other without ($model_{(wo)}$) copy mechanism. During inference, both trained models are loaded, and the final probability of the output token y_i is calculated as follows:

$$p_f(y_i) = \frac{p_c^{(w)}(y_i) + p_v^{(wo)}(y_i)}{2} \quad (8)$$

where $p_c^{(w)}(y_i)$ and $p_v^{(wo)}(y_i)$ indicate the probabilities that $model_{(w)}$ and $model_{(wo)}$ generate y_i, respectively. Since $model_{(wo)}$ can only generate tokens from $\mathbb{V}$, which may not cover the tokens in the input sequence (i.e., $p_v^{(wo)}(y_i) = 0$ if $y_i \notin \mathbb{V}$), the probabilities of tokens in $\mathbb{V}$ will become relatively larger after the combination. In this way, the project-specific tokens in the input code sequence are copied only if $model_{(w)}$ gives a high copy probability, otherwise the tokens in $\mathbb{V}$ including $\langle$unk$\rangle$ will be selected as output.

D. Real Patch Generation with Project-specific Information

Based on the generated patch IRs, three tools are developed to recover real patches from the patch IRs, replace the unknown tokens, and filter the patch candidates with compilation errors by leveraging the project-specific information.

Project-specific information. Four categories of project-specific information are used in this paper. For a suspicious statement and its contextual method, the first category is *method-level information*, which consists of the method parameters and local variables defined in the current method. Note that only the variables defined before the current statement are reserved. The second category is *file-level information*, which contains the necessary attributes of the current class, including class name, package name, class fields, method signatures, super class, etc. The third category is *project-level information*, which contains the same types of attributes as the second category but for other classes in the target project. The last category is *third-party libraries information*, which includes the same attributes but for the third-party packages and JDK (Java Development Kit) libraries that the current class depends on. For third-party packages, an out-of-the-box tool jd-cli[2] is adopted to decompile the binary code to the corresponding source code. For JDK libraries, we parse the source code of the specified JDK version in the target project. Some existing NMT-based APR methods [20], [21] also consider project-specific information to replace $\langle$unk$\rangle$ tokens, but they all ignore the parsing of the third-party packages and JDK libraries. They also ignore some advanced language features such as the inheritance from the super class, the interfaces, the effects of different modifiers on the element accessibility, and the parsing of some special types such as enum classes. These ignored project-specific information may be important for some repair scenarios.

After the project-specific information is collected, three tools are developed at the post-processing phase for real patch recovery, unknown tokens replacement and patch checking.

1) Restore Real Patches from Patch IRs: As mentioned in Section III-C, patch IRs are generated from the encoder-decoder models. In this section, we develop a patch recovery tool (PRT) based on javalang library[3]. For each fix template, a specific patch recovery mode is designed to match the buggy method context and use the method-level information if needed. This is actually the inverse process of building the formatted template representations in Section III-B.

Now, we are going to introduce the patch recovery process by taking a real-world bug *Lang_38* in Defects4J-v1.2 dataset as an example. As shown in Table III, the second row indicates the buggy code snippet before fix, where the buggy line is marked with the keywords $\langle$BOS$\rangle$ and $\langle$EOS$\rangle$. The entire method is not shown due to the space limitation. Then, the encoder-decoder model takes the buggy line and its contextual method as input, and outputs a patch IR shown in the fourth row, which can be divided into two parts by the keyword $\langle$split$\rangle$. The first part is *IMS1*, indicating the fix template to be used (i.e., *Insert Missed Statement 1*), while the second part is the backbone element, referring to the statement that should be inserted before the current line. We use the keyword $\langle$split$\rangle$ to make separations to distinguish the potential spaces

[2]https://github.com/intoolswetrust/jd-cli

[3]https://github.com/c2nes/javalang

TABLE III
AN EXAMPLES OF PATCH RECOVERY PROCESS

Bug Version	Lang_38 (Defects4J-v1.2)

Buggy Code Snippet	if (mTimeZoneForced) {
	⟨BOS⟩ calendar = (Calendar) calendar.clone(); ⟨EOS⟩
	calendar.setTimeZone(mTimeZone);
	}

Fix Template	Insert Missed Statement 1
Patch IR	IMS1 ⟨split⟩ calendar . ⟨unk⟩ ( ) ;

Restored Patch Candidate	if (mTimeZoneForced) {
	calendar.⟨unk⟩();
	calendar = (Calendar) calendar.clone();
	calendar.setTimeZone(mTimeZone);
	}

inside the backbone elements. PRT first determines the repair action according to *IMS1*, that is, to add a statement before the buggy line. However, the statement has not been determined yet. Then, the backbone element gives a instance of the statement to uniquely determine the content of the patch. The restored patch is shown in the last row. Since the generated patch contains ⟨unk⟩ tokens, it requires further processing to generate the final patch. Next, we will describe how to replace ⟨unk⟩ tokens with syntax-correct program elements.

2) Replace Unknown Tokens with Project-specific Information: The unknown tokens replacement tool (URT) is proposed to search for the syntactically correct code elements from the collected project-specific information by analyzing the syntactic constraints of the program context. The joint inference strategy proposed in Section III-C2 retains more ⟨unk⟩ tokens than the original copy mechanism, which gives URT more opportunities to perform better. Compared with previous work [20], [21], URT performs more effectively since it considers more project-specific information especially for the third-party packages, JDK libraries as well as the advanced language features as mentioned above.

Also taking the bug version *Lang_38* in Tabel III as an example, the component represented by ⟨unk⟩ is the name of a method call, which is expected to meet the following two requirements: 1) it is defined in the class determined by the declared type of the instance *calendar*, and 2) the number of parameters of the target method is zero. URT first determines the class type of *calendar*, and then searches for all method declarations defined in this class which has no parameter. The filtered replacement tokens are sorted in the descending order by the number of times they appear in the current method. In the cases where the numbers are the same, the replacement tokens are further sorted in the order they are declared. Since TENURE is expected to generate and validate patches in a limited time, URT only keeps the top-M candidates. The preserved token number M of ⟨unk⟩ replacements is configurable. In many cases the replacement token candidates may exceed 100, so keeping all tokens will seriously hurt the repair efficiency. Also, keeping only 1 replacement token may miss the correct patches. Thus we choose a moderate value of top-3 by default, which gives a balance between repair effectiveness and efficiency. Finally, for *Lang_38*, the correct element *getTime* is found and kept, so this bug can be successfully fixed.

3) Filter Patch Candidates with Compilation Errors: Generally, the existing APR methods use out-of-the-box syntax analyzers (e.g., javalang library) to check if the generated patches are syntactically correct before patch validation. Nevertheless, we find that even though some patches pass the check, they still contain errors leading to compilation failures, such as "cannot find symbol", "no suitable method found", "incompatible types", etc. Actually, these errors cannot be caught by the syntax analyzers which only take a method or a class as input and lack the analysis for the more complex inter-procedural propagation. Since four levels of project-specific information have been collected, it is feasible to design a patch checking tool (PCT) to filter out common compilation errors for the improvement of repair efficiency.

Specifically, PCT performs the static and lightweight analysis on the patch candidates at the expression and statement granularities. It first extracts the expressions located in the newly added or modified part of the patch and then infers their types or analyze related statements based on the four levels of project-specific information. For different types of expressions, different checking rules are designed, such as checking if the conditional expression is boolean type, and the array index is a positive integer. In addition, some checkers are performed on the statements, such as checking if the continue/break statement is in a loop statement, and checking if the types on the left and right sides of a declaration statement are compatible.

We have grouped these checkers into 12 categories, including *Check Identifiers*, *Check Binary Operations*, *Check Return Types*, *Check Array Indices*, *Check Assignments*, etc. The details of these categories are listed at our GitHub web page. After checking, a large number of patch candidates with compilation errors are filtered out, which not only significantly improves the compilation pass rate of the remaining patches, but also effectively reduces the repair time. Note that PCT has no impact on the repair performance under perfect fault localization. But it can speed up the time for TENURE to reach the actual buggy statements when the fault localization results are not so accurate (e.g., the results from Ochiai [31]), and may fix more bugs if the time cost is limited, which is validated in our experiments.

IV. EVALUATION

A. Benchmark Dataset

We use Defects4J-v1.2 [30] and Defects4J-v2.0 [30] in our experiments, which are widely used in automated program repair studies [5], [18]–[22], [26]. Defects4J-v.1.2 contains 395 real-world bugs that belong to 6 open-source Java projects, while Defects4J-v2.0 contains 444 extra bugs compared with the former, which belong to 12 open-source Java projects.

B. Experimental Settings

1) Training Encoder-Decoder Models: During training, for each of the two large-scale datasets $Dataset_{FT}$ and $Dataset_{SL}$, the duplicate samples are removed and the remaining ones are divided into three sets for training, validation and test. We randomly select 10k samples for validation, and another 10k samples for test, while the rest for training. The hyper-parameters are determined by the grid-search algorithm based on the validation set, and the test set is used to double check the performance of the saved models. The encoder-decoder models used in our approach are implemented based on the open-source framework OpenNMT [47]. We set the input length as 500, the vocabulary size as 30k, the length of the word vectors initialized in the embedding layer as 128, the number of the hidden units in the recurrent layers as 128, and the number of the recurrent layers as 2. The widely used optimizer Adam [48] is adopted and the learning rate is set as 0.001. The models are trained for 200k iterations on the batches of size 32.

2) Patch Generation and Validation: The patch IRs generated by the models trained on $Dataset_{FT}$ and $Dataset_{SL}$ are concatenated in a ratio of 3:2, since the proportion of the sample number in $Dataset_{FT}$ to $Dataset_{SL}$ approximates 3:2, which generally reflects the real distribution of the bugs fixed by the 35 artificial fix templates and the single-line fix template. To better validate the repair effectiveness of our approach, we conduct extensive experiments under perfect and Ochiai [31] localization settings. The Ochiai algorithm is one of the most frequently-used spectrum-based fault localization methods [31], [49]–[53] in existing APR studies [5], [7], [20]–[22], and we use GZoltar-v1.7.2 [54] to generate the fault localization results. Existing studies use 100, 200, 500, 1000 as the beam size to keep multiple patch candidates [18]–[20], [22]. In this paper, we use 500 in both localization settings to balance the effectiveness and efficiency of the repair process. Following the previous work [7], [20], [21], we set the running time as 5 hours for each bug version, and only the top-1 plausible patch candidate is saved. That is to say, when a generated patch candidate can pass all the test cases, the repair process will be terminated at once. Finally, the saved plausible patches will be manually checked to determine whether they are semantically correct or not.

All the experiments are conducted on Ubuntu 18.04 server with 20 cores of 2.4GHz CPU, 384GB RAM and NVIDIA Tesla V100 GPUs with 32 GB memory.

C. Results and Discussion

RQ1: How does TENURE perform in program repair task? To answer this RQ, we compare the repair effectiveness of TENURE with one state-of-the-art template-based APR method (i.e., TBar [5]) and six deep learning-based APR methods (i.e., CoCoNuT [18], CURE [19], DEAR [21], TRANSFER-PR [26], RewardRepair [22] and Recoder [20]). We adopt two widely used benchmark datasets Defects4J-v1.2 and Defects4J-v2.0 to validate the repair performance, and conduct experiments on each of them under perfect localization and Ochiai [31] localization settings, respectively. As in the previous studies, perfect localization means that the actual buggy lines are directly given, while a spectrum-based fault localization method Ochiai [31] is frequently used to simulate the repair process in an actual environment.

As shown in Table IV, no matter which benchmark and which fault localization setting are chosen, our method outperforms all of the comparative methods. Specifically, On Defects4J-v1.2 dataset, TENURE successfully repairs 12 more bugs than all other methods in perfect localization setting (79 in total). Since the fix templates used in this paper are all derived from the existing comparative methods (i.e., 35 artificial fix templates from TBar [5], and *MSL* from NMT-based APR methods [15], [16], [18], [22]), the experimental results prove that the idea of incorporating knowledge contained in the artificial fix templates into the existing NMT-based APR methods is able to improve the repair performance. In contrast, the improvements under Ochiai fault localization setting are relatively small (i.e. from 51 to 52 bugs). The reason is that, on the one hand, the fault localization results generated by the Ochiai algorithm are not accurate enough, which makes TENURE take a long time to reach the actual buggy statements for some bug versions. On the other hand, TENURE keeps top-3 replacement tokens for each $\langle$unk$\rangle$ by default, so the number of the generated patch candidates will increase accordingly, also making it harder to reach the actual buggy statements in a limited time. Generally when the fault localization is more accurate, our method can access the buggy statements sooner to fix more bugs.

To validate the generalization performance of TENURE on the difficult-to-fix bugs, we also conduct experiments on Defects4J-v2.0. TENURE can fix 5 more bugs under perfect localization setting and 8 more bugs under Ochiai localization setting than all other methods. The previous study [20] mentioned that the 35 artificial fix templates given by TBar [5] are overfitted on Defects4J-v1.2, so the repair results on Defects4J-v2.0 may be not good. However, the experimental results of TENURE prove that using patch IRs for model training and the additional *MSL* fix template can achieve much better repair effectiveness. The details of the repaired bug versions on Defects4J-v2.0 under Ochiai localization setting for TENURE and other two state-of-the-art repair methods (i.e., RewardRepair [22] and Recoder [20]) are shown in Figure 2. We can see that there are 17 bugs exclusively fixed by TENURE which are not fixed by either of the other two methods.

RQ2: How effective are the main components of TENURE? In this RQ, we explore the effectiveness of the main components of TENURE. Three groups of comparative experiments are conducted on Defects4J-v1.2 under perfect localization setting. The first group is to analyze the impact of the used fix templates on the repair results. As shown in Table V, the complete model (i.e., the sixth row) adopts two parts

[1]The experimental results of Recoder are obtained from its open source repository (https://github.com/pkuzqh/Recoder), which are slightly different from those reported in their paper [20].

TABLE IV
EXPERIMENTAL RESULTS COMPARED WITH STATE-OF-THE-ART PROGRAM REPAIR METHODS IN DIFFERENT FAULT LOCALIZATION SETTINGS

Techniques	Fault Localization Settings			
	Perfect		Ochiai	
	D4J-v1.2	D4J-v2.0	D4J-v1.2	D4J-v2.0
TBar [5]	62	8	41	8
CoCoNuT [18]	44	-	33	-
CURE [19]	57	-	36	-
DEAR [21]	53	-	47	-
TRANSFER-PR [26]	67	-	42	-
RewardRepair [22]	45	45	29	24
Recoder [20][1]	66	-	51	19
TENURE	**79**	**50**	**52**	**32**

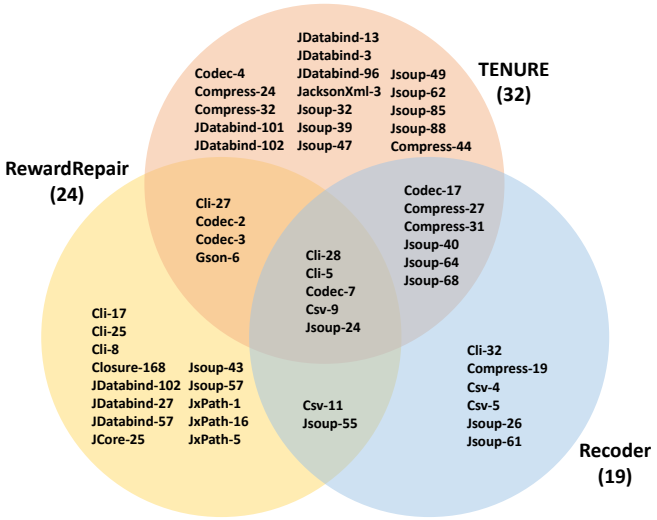


Fig. 2. Overlapping analysis for repair experiments on Defects4J-v2.0.

of fix templates, the first of which consists of the 35 artificial fix templates, while the second contains the fix template *MSL*. Therefore, we build two model variants for each part (i.e., the first and the second row). It can be seen that no matter which part is removed, the repaired bugs decrease. Obviously, the 35 artificial fix templates have better repair effectiveness than *MSL* (i.e., 73 and 32) since they can cover more repair scenarios than the single-line code generation.

The second group is to analyze the impact of using joint inference strategy on the repair results. This strategy is designed to better predict the OOV tokens by averaging the probabilities of the models with and without the copy mechanism, which is expected to copy the tokens inside the method and preserve the ⟨unk⟩ placeholders for the tokens outside the method. We construct two model variants, one of which only uses copy mechanism while the other even does not use copy mechanism (i.e., the third and the fourth row). Compared with the complete model (i.e., the sixth row), the repaired bugs of each variant decrease obviously (i.e., 66 and 56). When only the copy mechanism is used, it works well for the scenarios where the OOV tokens can be replaced with the inside-method code elements. However, such model tends to over-copy. That

TABLE V
COMPARATIVE ANALYSIS WITH DIFFERENT COMPONENTS, WHERE TP MEANS TEMPLATE AND JOINT MEANS THE JOINT INFERENCE

Model Variants	#Bugs
TP_{35} + JOINT + URT + PCT	73
TP_{MSL} + JOINT + URT + PCT	32
TP_{35} + TP_{MSL} + Only copy + URT + PCT	66
TP_{35} + TP_{MSL} + No copy + URT + PCT	56
TP_{35} + TP_{MSL} + JOINT + PCT	63
TP_{35} + TP_{MSL} + JOINT + URT + PCT (TENURE)	79

is, for OOV tokens that should be replaced with the outside-method code elements, the model will still try hard to choose the inside-method tokens, thus leaving a very small number of ⟨unk⟩ placeholders to URT. If the copy mechanism is not used, the OOV tokens that can be effectively replaced with the inside-method elements are still kept as ⟨unk⟩s and more ⟨unk⟩ placeholders are left. In this way, The benefit of the copy mechanism during the model training phase is missing, and may excessively increase the burden on URT. Therefore, according to the experimental results, we conclude that the combination of the joint inference strategy and URT is a better solution to the OOV problem.

The third group is to analyze the impact of URT on the repair results. After removing URT (i.e., the fifth row). The comparative experiment results show that the number of the repaired bugs for this model variant is 16 fewer than the complete model, since the semi-finished patch candidates with ⟨unk⟩ placeholders cannot fix any bugs.

We do not discuss the PCT component in this RQ, because it only filters out the patch candidates containing compilation errors to make the repair method reach the actual buggy statement sooner. The repair results of all the model variants and the complete model under the perfect localization setting are not affected no matter whether the PCT component is used or not. Objectively speaking, PCT is able to speed up the repair process under the perfect localization. But on Defects4J (v1.2/v2.0), the setup of 5 hours is enough for patch validation even without PCT. We will discuss PCT in RQ5, which focuses on the impact derived from PCT on the repair results under the Ochiai localization setting.

RQ3: How does each fix template contribute to the final repair results? In order to quantify the contributions of the fix templates used in this paper to the repair results, we further analyze the 79 bugs successfully fixed by TENURE on Defects4J-v1.2 and the 50 bugs on Defects4J-v2.0 under perfect localization setting. Since some bug versions (e.g., Math_22, Math_35, etc.) need multiple fixes on different statements, there are overall 89 and 51 correct patches on the two datasets. The distributions of the fix templates used for these correct patches are shown in Figure 3, where we keep 24 fix templates and ignore the other 12 fix templates that cannot generate any correct patches. The x-axis indicates the abbreviations of the fix templates (i.e., 23 artificial fix templates and the *MSL* fix template), while the y-axis indicates

the number of the generated correct patches. Overall, There are 24 out of 36 fix templates generating at least one correct patch. Among them, *MSL* contributes 20 correct patches in total, which is tied for the first place with *RBS1 (Remove Buggy Statement 1)*, proving that *MSL* is able to provide a good complement to the 35 artificial fix templates. In addition, we find that the contributions may vary with different datasets for some fix templates. For example, the template *MO1* generates 10 correct patches on Defects4J-v1.2, while there is only 1 patch on Defects4J-v2.0. On the contrary, the template *INPC1* only generates 2 correct patches on Defects4J-v1.2, while there are 5 patches on Defects4J-v2.0. For the 12 out of 36 fix templates that cannot generate any correct patches, we find their corresponding numbers of the collected samples in $Dataset_{FT}$ are small. Thus they may not be effectively leveraged in our experiments.

It also should be noted that, in our experiments, we assume that we have no priori knowledge about the target projects in Defects4J(v1.2/v2.0) to follow the industrial practice, thus we keep all templates even if some of them may not generate any correct patches. However, developers can add or delete the fix templates if they know which templates are helpful or not before using TENURE.

RQ4: How does the project-specific information we provide alleviate the OOV problem? The copy mechanism is one of the most frequently used approaches to reduce the performance loss caused by the OOV problem in NMT-based repair methods [15], [16], which copies a token from the input sequence to the output. However, since most of the NMT-based approaches take a single method in the source code as model input [18]–[20], [22], the copy scope is limited to the tokens within the method. Thus it is less effective for the cases where the inter-procedural and project-specific program elements are needed for patch generation. In this work we propose a technique for combining copy and non-copy mechanisms for joint inference, and using URT to replace ⟨unk⟩ tokens in the post-processing phase. To verify the necessity of this implementation, we first identify 27 out of the 89 correct patches on Defects4J-v1.2 and 13 out of the 51 correct patches on Defects4J-v2.0, which contain ⟨unk⟩ placeholders before replacements by URT. Afterwards, the source of the correct replacement tokens provided by URT are analyzed. Specifically, on Defects4J-v1.2, there are 12 correct patches relying on the code elements from the same method, 11 from the same class, and 4 from the other classes; on Defects4J-v2.0, the corresponding numbers are 3, 5 and 5. Since the replacement tokens from the same class or the other classes are beyond the scope of the current method, the copy mechanism cannot effectively handle these cases. These two cases account for 62.5% in total, indicating that our method is a better solution for the OOV problem.

RQ5: How effective and efficient is our patch checking tool? In this RQ, we answer three questions related to PCT in detail. The first question is to analyze the impact of PCT on the repair results. We conduct comparative experiments on benchmarks under the Ochiai localization setting. Specifically, a model variant is constructed which only removes the PCT component. The numbers of the repaired bugs are 46 and 31 on Defects4J-v1.2 and Defects4J-v2.0, respectively, which are 6 and 1 fewer than the results from the complete model. Before using PCT, the list of patch candidates has been already determined. Thus PCT can reduce the time for the repair process to reach the real buggy statements, but does not affect the repair effectiveness. However, since TENURE combines template-based and NMT-based methods, the time to reach the buggy statements will be slightly longer for the inaccurate fault localization methods (e.g., Ochiai). In this way, TENURE without PCT may not access a small part of buggy statements for patch validation within the specified time, while other APR methods such as Recoder[20] can reach. On Defects4J-v2.0, TENURE can fix 31 bugs under the Ochiai localization even without PCT, which is far more than 19 (Recoder [20]) and 24 (RewardRepair [22]).

TABLE VI
AVERAGE COMPILATION RATES OF THE TOP-K PATCH CANDIDATES IN DEFECTS4J-V1.2

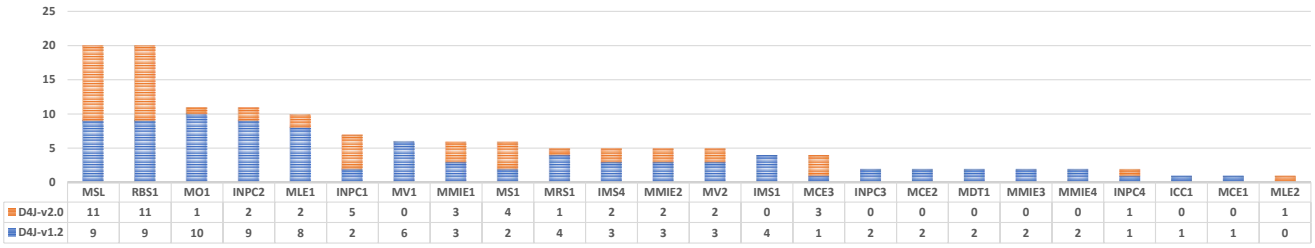
Techniques	Top-30	Top-100	Top-200
SequenceR [15]	33%	-	-
CoCoNuT [18]	24%	15%	6-15%
CURE [19]	39%	28%	14-28%
RewardRepair [22]	45.3%	37.5%	33.1%
TENURE w/o PCT	47.3%	38.0%	33.0%
TENURE w/ PCT	**70.0%**	**65.7%**	**62.8%**

The second question is the impact of PCT on the compilation pass rate of the patch candidates. As shown in Table VI, we list the compilation pass rates of patch candidates for several recent NMT-based repair methods where the beam size is set to 30, 100 and 200, respectively. Among them, RewardRepair [22] improves the compilation pass rate by adding the information on whether the current code snippet can pass the compilation check or not to the model training. The experimental results of TENURE without and with PCT are both listed in the table (the last two rows). It can be seen that the compilation pass rates of the patch candidates generated by TENURE are nearly the highest ones even PCT is not used. This proves that the processing of first generating patch IRs and then restoring them to the real patches is beneficial for generating compilable patch candidates. And more importantly, when PCT is used, the pass rates increase greatly in all beam size setups.

The third question is about the efficiency of PCT. We use PCT and *javac* to conduct analysis on 500,000 Java files simultaneously, and find that *javac* takes an average of 1,201 seconds to compile 1000 Java files, while PCT only takes 30 seconds. Thus, compared with *javac*, PCT provides a fast parsing solution with 40x speed, which is of high practice value in APR studies.

V. THREATS TO VALIDITY

One threat to external validity is that only the Java language is used in this paper. In fact, our method is not bundled



	MSL	RBS1	MO1	INPC2	MLE1	INPC1	MV1	MMIE1	MS1	MRS1	IMS4	MMIE2	MV2	IMS1	MCE3	INPC3	MCE2	MDT1	MMIE3	MMIE4	INPC4	ICC1	MCE1	MLE2
D4J-v2.0	11	11	1	2	2	5	0	3	4	1	2	2	2	0	3	0	0	0	0	0	1	0	0	1
D4J-v1.2	9	9	10	9	8	2	6	3	2	4	3	3	3	4	1	2	2	2	2	2	1	1	1	0

Fig. 3. The contribution of each fix template to the number of the correct patches under perfect localization on Defects4J-v1.2 and Defects4J-v2.0.

with Java. Due to the similarities in functional design, other programming languages can also be represented by abstract syntax trees, which are then used in the similar processes of dataset construction and model training. Another threat to external validity is the used benchmarks. Since Durieux et al. [55] shows that program repair techniques may overfit on Defects4J-v1.2, we additionally conduct the comparative experiments on Defects4J-v2.0 to reduce this threat, which contains extra 444 bugs in 12 projects (11 are different from Defects4J-v1.2). In the future, we will evaluate our approach on more datasets.

One threat to internal validity is that the three tools implemented in the post-processing phase may still not take into account the full programming language features. However, the experiments have showed the effectiveness of them for APR tasks. In the future, we will maintain these tools to help improve the repair performance for more APR methods. Another internal threat is the standard for manual checking of the plausible patches. To minimize this threat, all plausible patches are double-checked by two authors, and a patch is only be considered correct if both authors acknowledge its correctness. Simultaneously, we also refer to the published results of the previous studies.

VI. Conclusion

In this work we propose a novel template-based neural program repair approach to combine the advantages of both template-based and NMT-based methods. We construct two large-scale datasets, which include 35 fix templates from template-based method and one fix template from NMT-based method, respectively. We then train encoder-decoder models on the datasets to generate patch IRs. An optimized copy mechanism is also used to better solve the OOV problem. With the combined patch IRs and project-specific information, we develop three tools to recover real patches from the patch IRs, replace the unknown tokens, and filter the patch candidates with compilation errors. On the commonly-used benchmark Defects4J-v1.2, TENURE can successfully fix 79 bugs under perfect fault localization and 52 bugs under Ochiai fault localization. While on another more difficult-to-fix benchmark Defects4J-v2.0, TENURE is able to fix 50 and 32 bugs, respectively. Compared with all existing automated program repair methods, TENURE achieves the best performance in all of the above settings.

Data availability: our source code and experimental data are publicly available at: **https://github.com/mxx1219/TENURE**.

Acknowledgement

This work was supported partly by National Key Research and Development Program of China (No.2021YFB3500700), partly by National Natural Science Foundation of China (No. 62072017, 62141209) and Australian Research Council Discovery Projects (DP200102940, DP220103044), and the Ministry of Industry and Information Technology of the PRC.

References

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

[2] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Towards practical program repair with on-demand candidate generation," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 12–23.

[3] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object-oriented program repair," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 648–659.

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

[5] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 31–42.

[6] 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*, pp. 1–45, 2020.

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

[8] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 1–11.

[9] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury, "Repairing crashes in android apps," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 187–198.

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

[11] 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 *2019 12th IEEE conference on software testing, validation and verification (ICST)*. IEEE, 2019, pp. 102–113.

[12] T. Durieux, B. Cornu, L. Seinturier, and M. Monperrus, "Dynamic patch generation for null pointer exceptions using metaprogramming," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 349–358.

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

[14] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 25–36.

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

[16] S. Chakraborty, M. Allamanis, and B. Ray, "Codit: Code editing with tree-based neural machine translation," *arXiv preprint arXiv:1810.00314*, 2018.

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

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

[19] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1161–1173.

[20] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, p. 341–353.

[21] Y. Li, S. Wang, and T. N. Nguyen, "Dear: A novel deep learning-based approach for automated program repair," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, p. 511–523.

[22] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE, 2022, pp. 1506–1518.

[23] N. Kalchbrenner and P. Blunsom, "Recurrent continuous translation models," in *Proceedings of the 2013 conference on empirical methods in natural language processing*, 2013, pp. 1700–1709.

[24] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[25] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in neural information processing systems*, vol. 27, 2014.

[26] X. Meng, X. Wang, H. Zhang, H. Sun, and X. Liu, "Improving fault localization and program repair with deep semantic features and transferred knowledge," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1169–1180.

[27] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 4, pp. 1–29, 2019.

[28] P. Koehn and R. Knowles, "Six challenges for neural machine translation," *arXiv preprint arXiv:1706.03872*, 2017.

[29] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1936–1964, 2017.

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

[31] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. IEEE, 2006, pp. 39–46.

[32] K. Pan, S. Kim, and E. J. Whitehead, "Toward an understanding of bug fix patterns," *Empirical Software Engineering*, vol. 14, no. 3, pp. 286–315, 2009.

[33] 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)*. IEEE, 2013, pp. 802–811.

[34] R. Rolim, G. Soares, R. Gheyi, T. Barik, and L. D'Antoni, "Learning quick fixes from code repositories," *arXiv preprint arXiv:1803.03806*, 2018.

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

[36] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 87–98.

[37] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Thirty-First AAAI conference on artificial intelligence*, 2017.

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

[39] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," *arXiv preprint arXiv:1503.00075*, 2015.

[40] Z. Sun, Q. Zhu, Y. Xiong, Y. Sun, L. Mou, and L. Zhang, "Treegen: A tree-based transformer architecture for code generation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 05, 2020, pp. 8984–8991.

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

[42] S. Wiseman and A. M. Rush, "Sequence-to-sequence learning as beam-search optimization," *arXiv preprint arXiv:1606.02960*, 2016.

[43] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[44] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[45] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[46] A. See, P. J. Liu, and C. D. Manning, "Get to the point: Summarization with pointer-generator networks," *arXiv preprint arXiv:1704.04368*, 2017.

[47] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush, "Opennmt: Open-source toolkit for neural machine translation," *arXiv preprint arXiv:1701.02810*, 2017.

[48] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[49] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 2007, pp. 89–98.

[50] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 2005, pp. 273–282.

[51] W. E. Wong, V. Debroy, Y. Li, and R. Gao, "Software fault localization using dstar (d*)," in *2012 IEEE Sixth International Conference on Software Security and Reliability*. IEEE, 2012, pp. 21–30.

[52] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Transactions on software engineering and methodology (TOSEM)*, vol. 20, no. 3, pp. 1–32, 2011.

[53] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," *Acm Sigplan Notices*, vol. 40, no. 6, pp. 15–26, 2005.

[54] J. Campos, A. Riboira, A. Perez, and R. Abreu, "Gzoltar: an eclipse plug-in for testing and debugging," in *Proceedings of the 27th IEEE/ACM international conference on automated software engineering*, 2012, pp. 378–381.

[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 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 302–313.