Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-Shot Learning

Chunqiu Steven Xia
University of Illinois Urbana-Champaign
chunqiu2@illinois.edu

Lingming Zhang
University of Illinois Urbana-Champaign
lingming@illinois.edu

ABSTRACT

Due to the promising future of Automated Program Repair (APR), researchers have proposed various APR techniques, including heuristic-based, template-based, and constraint-based techniques. Among such classic APR techniques, template-based techniques have been widely recognized as state of the art. However, such template-based techniques require predefined templates to perform repair, and their effectiveness is thus limited. To this end, researchers have leveraged the recent advances in Deep Learning to further improve APR. Such learning-based techniques typically view APR as a Neural Machine Translation problem, using the buggy/fixed code snippets as the source/target languages for translation. In this way, such techniques heavily rely on large numbers of high-quality bug-fixing commits, which can be extremely costly/challenging to construct and may limit their edit variety and context representation.

In this paper, we aim to revisit the learning-based APR problem, and propose AlphaRepair, the first *cloze-style* (or *infilling-style*) APR approach to directly leveraging large pre-trained code models for APR without any fine-tuning/retraining on historical bug fixes. *Our main insight is instead of modeling what a repair edit should look like (i.e., a NMT task), we can directly predict what the correct code is based on the context information (i.e., a cloze or text infilling task).* Although our approach is general and can be built on various pre-trained code models, we have implemented AlphaRepair as a practical multilingual APR tool based on the recent CodeBERT model. Our evaluation of AlphaRepair on the widely used Defects4J benchmark *shows for the first time that learning-based APR without any history bug fixes can already outperform state-of-the-art APR techniques.* We also studied the impact of different design choices and show that AlphaRepair performs even better on a newer version of Defects4J (2.0) with 3.3X more fixes than best performing baseline, indicating that AlphaRepair can potentially avoid the dataset-overfitting issue of existing techniques. Additionally, we demonstrate the multilingual repair ability of AlphaRepair by evaluating on the QuixBugs dataset where AlphaRepair achieved the state-of-the-art results on both Java and Python versions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3549101

CCS CONCEPTS

• **Computer systems organization** → *Neural networks*; • **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

Automated Program Repair, Deep Learning, Zero-shot Learning

ACM Reference Format:
Chunqiu Steven Xia and Lingming Zhang. 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 (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3540250.3549101

1 INTRODUCTION

Software systems are all-pervasive in everyday life from monitoring financial transactions [69], controlling transportation systems [73] to aiding healthcare tools [8]. Software bugs in these systems can affect people around the globe and cost billions in financial losses [10]. To fix such bugs, developers often need to invest significant manual effort, e.g., it is estimated that developers spend 35 to 50% of their time debugging software systems [56]. To reduce manual debugging efforts, Automated Program Repair (APR) techniques have been proposed to automatically generate patches to fix the bugs [24].

A popular approach for APR is Generate and Validate (G&V) [25, 29, 35, 38, 39, 44, 45, 49, 53, 61, 74]. To start off, fault localization [3, 40, 41, 58, 75, 79] is often used to reduce the search space by computing the suspicious program locations that likely caused the bug. Using these potential buggy locations, the G&V techniques will generate a list of candidate patches. Each candidate patch is compiled and validated against the test suite. Patches that successfully pass all tests are called *plausible patches*. However, tests often cannot cover all possible behaviors of the program [61], hence a plausible patch might still fail under other inputs. Therefore, plausible patches are further inspected by developers to determine the final *correct patches* that correctly fix the underlying bug.

Depending on how patches are generated, traditional APR techniques can be categorized into heuristic-based [38, 39, 74], constraint-based [17, 37, 55], and template-based [29, 35, 44, 45, 53]. Among all traditional techniques, template-based APR techniques, which leverage pre-defined fix patterns to transform buggy code snippets into correct ones, have been widely recognized as state of the art [7, 25, 44]. These fix patterns target specific types of bugs (e.g., null pointer exception) and patterns in the source code, and are often crafted by human experts. While effective, there is an upper limit to the number of candidate patches that such pre-defined templates can generate. Therefore, to increase the expressiveness of the edits,

researchers have recently utilized Machine Learning (ML) or Deep Learning (DL) techniques for patch generation [14, 32, 42, 51, 80].

Learning-based APR techniques often leverage the recent advances in DL to train a neural network that transforms the original buggy program into the fixed version. These techniques [14, 19, 32, 42, 51, 80] typically view the problem of program repair as a Neural Machine Translation (NMT) [66] problem and use NMT models from the field of Natural Language Processing (NLP), where the model input is a language sequence and the output is a translated sequence in another language. Researchers have used NMT models for program repair where instead of translating natural languages, the models aim to turn a buggy code into the fixed version. These NMT models are typically made up of an *encoder* and *decoder* pair where the encoder captures the buggy code elements with its context while the decoder takes in the encoded representation and generates a fixed version. To facilitate APR, such models *must* be trained using pairs of buggy and patched code. Despite being an effective APR approach, existing learning-based techniques face the following issues:

1) Quality of training data. Current learning-based APR tools require training or fine-tuning of the models by using historical bug fixes, i.e., pairs of buggy and patch code. This data is usually obtained by scraping open-source projects to find specific commits that are about bug fixes. However, this relies on various handcrafted heuristics. For example, to find the bug fixing commits, keywords such as *bug, fix, patch, solve, issue, problem* are often used to filter the commits [16, 30, 51, 80]. Individual bug-fixing commits can also include unrelated edits such as refactoring or new feature implementation [33]. As a result, the extracted data can contain various irrelevant commits and unrelated code changes within bug fixing commits, adding noise to the training dataset.

2) Quantity of training data. Compared to large amount of open-source code snippets that are available in the wild, the amount of bug fixes is limited. To reduce the effect of the aforementioned issue of a commit containing irrelevant changes from bug fixes, learning-based APR tools usually limit the commits in their dataset to ones with few lines of changes [32, 42, 51, 80], further limiting the amount of training data. By training on such limited historical fixes, current learning-based tools might restrict the edit variety of their approach only on what is in their training data.

3) Context representation. To provide a correct fix to a buggy code snippet, the context before and after are crucial in providing useful syntactic/semantic information. Current learning-based APR tools first pass the context including the buggy code elements into an encoder as plain texts [32, 51] or structured representations [42, 80]. The encoded context is then used directly or combined with a separate encoding of the buggy code snippet as input to the decoder. However, this process is *unnatural* since it is challenging for the models to distinguish the patch location within the context, or effectively merge the separate bug/context encodings. As a result, such techniques may miss intricate relations between a patch and its context, such as the proximity of each code element that provides important syntax/semantic information.

Our Work. We present AlphaRepair – the first *cloze-style* (or *infilling-style*) APR approach that uses large pre-trained code models under a *zero-shot learning* setting [36] to directly generate patches, i.e., without any additional training or fine-tuning on bug-fixing datasets. Different from all existing learning-based APR techniques, our main insight is that *instead of modeling what a repair edit should look like, we can directly model/predict what the correct code is based on the context information* (like a *cloze* [1, 67] or "text infilling" task). In this way, our cloze-style APR can avoid all above limitations of the existing techniques: 1) it completely frees APR from historical bug fixes, 2) it can simply get trained on all possible open-source projects in the wild for massive training, 3) it is directly pre-trained to model patches based on the surrounding context, and thus can effectively encode the intricate relations between patches and their contexts. Furthermore, while it is non-trivial to adapt prior APR techniques for a new language (due to a huge amount of code/data engineering work for preparing historical bug fixes), under our cloze-style APR, extending APR to a new language can be as simple as mining a new code corpus in the new language!

While our cloze-style APR is generalizable to various pre-trained models, in this paper, we implement AlphaRepair with one recent pre-trained code model, CodeBERT [23]. Unlike current learning-based APR tools which use limited numbers of bug fixes as training data, CodeBERT is directly pre-trained using millions of code snippets from open-source projects, allowing it to provide a variety of edit types to fix different bugs. We directly leverage the initial training objective of Masked Language Model (MLM) [18] - predicting/recovering randomly masked out tokens used in CodeBERT to perform zero-shot learning for APR. We first prepare model inputs where each potentially buggy line is replaced with a mask line. We then query CodeBERT to fill the mask line with replacement tokens to produce candidate patches for a buggy program input. This allows us to directly perform zero-shot learning (with no fine-tuning) since we use the same tasks as defined in MLM– instead of predicting random mask tokens, we generate predictions by masking out only the buggy code snippet. Note that to improve the patch search space, the mask lines are designed to systematically reuse parts of the buggy line. Furthermore, the bidirectional nature of CodeBERT (and MLM) also allows us to naturally capture both the contexts before and after the buggy location for effective patch generation.

Contribution. This paper makes the following contributions:

- **Direction/Dimension** This paper opens a new dimension for cloze-style APR to directly query large pre-trained code models under a zero-shot learning setting. Compared with existing learning-based APR techniques, our approach does not need any additional fine-tuning/retraining using historical bug fixes and can be easily adopted for multiple programming languages. Additionally, we demonstrate the efficacy of directly applying large pre-trained models for generating code fixes for real-world systems, which previously were mainly tested on generating simple/small programs [5, 13].
- **Technique** While our idea is general and can leverage various existing pre-trained code models, we have built AlphaRepair as a practical APR tool based on the recent CodeBERT model. We leveraged the original training objective of CodeBERT using specific inputs of mask lines for direct program repair. We used a variety of different mask line templates and further propose probabilistic patch ranking to boost APR.

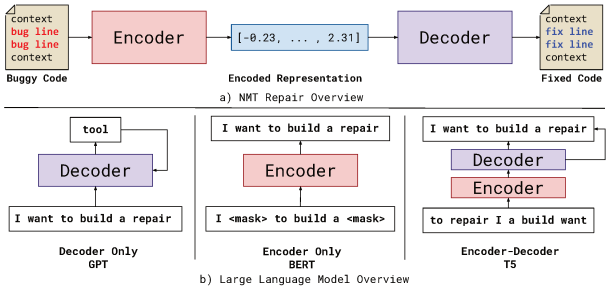


Figure 1: NMT and large language model overview

- **Extensive Study** We have compared AlphaRepair with state-of-the-art Java APR tools (both traditional and learning-based) on Defects4J [34]. The results show that AlphaRepair can outperform all existing tools on the widely studied Defects4J 1.2, improving the number of fixed bugs from 68 to 74 and fixing 8 unique bugs that no prior work can fix. More surprisingly, AlphaRepair even fixes 3.3X more bugs than the best baseline on the newly included bugs in Defects4J 2.0, demonstrating that AlphaRepair can avoid the dataset-overfitting issue of existing APR techniques. Moreover, we have also studied AlphaRepair on both the Java and Python versions of the widely studied QuixBugs dataset [43]. The results not only confirm that AlphaRepair can outperform all existing APR techniques (in both Java and Python), but also demonstrate the multilingual capability of AlphaRepair.

2 BACKGROUND

2.1 Learning-based APR

Deep Learning (DL) [26] is a powerful learning mechanism to learn from large amounts of data used in many domains of Machine Learning. Researchers have leveraged DL techniques for APR by viewing the problem of program repair as a Neural Machine Translation (NMT) [66] task of turning a buggy code snippet into a patched version. These learning-based APR tools are typically built using the encoder-decoder architecture. Figure 1a shows the workflow of these learning-based repair tools. The encoder takes in the buggy line and its surrounding context as input and outputs an encoded representation. The decoder uses the encoded representation to generate a new code line to replace the buggy line. SequenceR [14] is a sequence-to-sequence network built using a Long Short-Term Memory (LSTM) [28] encoder and decoder network for program repair. DLFix [42] encodes the input code as an Abstract Syntax Tree then uses a tree-based Recurrent Neural Network [64] as a part of the encoder and decoder network to generate patches. CoCoNuT [51] proposes a new NMT architecture which encodes the context and buggy line separately. It also leverages multi-stage attention to increase the amount of important information passed on from the encoder to the decoder. In order to improve the syntactic correctness, before training on code fix pairs, CURE [32] first pre-trains the NMT model on a large corpus of developer code. Furthermore, CURE uses a static checking strategy to only generate patches with valid identifiers. More recently, Recoder [80] modifies the NMT model by using a syntax-guided decoder designed to generate syntactically correct edits on an input buggy code snippet.

Recent study has shown that learning-based APR tools can achieve the state of the art in APR [32, 80]. However, they are still limited by their need for pairs of buggy and fixed versions as training data. These buggy and fixed code pairs are challenging to obtain as handcrafted heuristics are used to identify bug fixing commits and individual commits can contain other code changes apart from bug fixing, adding noise to the dataset. Additionally, learning-based APR tools learn bug patterns and corresponding patch fixes from the training data in order to automatically generate patches for unseen buggy code. This means it is hard for learning-based APR tools to generalize to fix patterns that are not present in their training dataset. Furthermore, it can be tricky for current learning-based APR tools to encode the context surrounding buggy code elements causing the generated patches to miss important syntax and semantic information. Interestingly, although pre-trained code models have also been adopted for APR recently [19, 32, 54], they are still leveraged to learn from a large number of historical bug fixes, thus still suffering from the above limitations of existing learning-based techniques.

In this paper, we address these issues by using large pre-trained code model, trained on massive amount of open-source code, directly for cloze-style APR without the need to train or fine-tune on any smaller dataset of buggy and fixed code.

2.2 Large Pre-trained Code Models

Recent popularity in Natural Language Processing (NLP) has led to development of large pre-trained models that use large amounts of data. To make use of the massive unlabeled training data, pre-trained models apply self-supervised objectives, e.g., **Masked Language Model (MLM)** - where some training data is artificially masked and the training objective is to predict/recover the real data. A common component of large pre-trained language models is a Transformer [70]. It contains an encoder made up of multiple differentiable self-attention layers in order to learn representation and also a decoder used to generate output. Figure 1b shows the three categories of large pre-trained language models. GPT [62] is a large generative model which uses only the *decoder* component to predict the next token output given all previous tokens. This type of decoder is autoregressive where a sequence is generated by iteratively inputting all previous tokens in order to generate the next one. BERT [18] is another type of large pre-trained model which contains only the *encoder* component. BERT is designed to learn a representation of the data and is trained using the MLM objective. A small percentage of tokens in the training data will be replaced by a mask token, where the goal is to train BERT to predict the true value of the mask token. To combine the usage of both encoder and decoder, *encoder-decoder* models have also been used to build large pre-trained language models. Models such as T5 [63] are designed for sequence-to-sequence tasks where the training objective aims to recover the correct output sequence given the original input (English to French, corrupted to uncorrupted, etc). These large pre-trained language models can be fine-tuned for downstream NLP tasks such as text summarization [46], text classification [77], as well as question and response text generation [15].

Researchers have extended encoder, decoder and encoder-decoder models to build large pre-trained models for various *programming*

language tasks. CodeGPT [50] adopts the original GPT architecture and trains a generative model from scratch using Python and Java functions. Codex [13] is a GPT-based code model created by fine-tuning a larger GPT-3 model [9] for generating Python functions based on natural language descriptions. CodeBERT [23] and GraphCodeBERT [27] are BERT-based models for programming tasks, and are trained using the MLM training objective. GraphCodeBERT additionally encodes simple data flow information to aid in code representation. CodeTrans [22], CodeT5 [72] and PLBART [4] are unified encoder-decoder models [63] which uses denoising sequence-to-sequence training objectives to pre-train both the encoder and decoder for various coding tasks.

In this paper, we directly use large pre-trained models for cloze-style APR via zero-shot learning. While our cloze-style APR idea is general and can be achieved using all above pre-trained models, we demonstrate its potential by using the simple CodeBERT model. CodeBERT is trained using the MLM objective which can be used to generate replacement code for buggy code snippets. Also, CodeBERT is bidirectional in nature, allowing it to capture both contexts before and after for patch generation.

3 APPROACH

In this section, we introduce *cloze-style* APR, a new direction for learning-based APR that directly learns from the large number of code snippets available in the wild to generate patches. Different from all prior learning-based APR techniques, instead of viewing APR as a task of turning a buggy code snippet into a fixed code snippet, we can directly learn what the *correct code* should look like given its surrounding context. We view the problem as a *cloze task* [1, 67] where we replace the buggy snippet with blanks/masks and query pre-trained models [6] to fill the blanks with the correct code snippet. *This problem setup does not require access to any dataset containing pairs of buggy and patched code versions and our approach can directly make use of the existing large pre-trained models to automatically generate code repairs.*

Although our approach is general, in this work, we re-purpose the recent CodeBERT [23] model for program repair. The training objective for CodeBERT uses Masked Language Model (MLM) [18] where given an input sequence of tokens $X = \{x_1, x_2, ...x_n\}$, a random set of tokens in X is replaced with *mask tokens* to generate a new mask sequence X_{masked}. The training goal is to output the original tokens in X_{masked} which have been masked out, i.e. recover X given X_{masked}. Given predictor P which outputs the probability of a token, the MLM loss function can be formulated as:

$$\mathcal{L}_{MLM} = \sum_{i \in masked} -\log(P(x_i | X_{masked})) \quad (1)$$

The MLM training objective allows us to directly use CodeBERT for program repair where instead of randomly masking out tokens, we mask out all tokens which are part of the buggy code snippet. We then use CodeBERT under a zero-shot learning setting for program repair where we recover the correct tokens in place of the mask buggy tokens. As a result, AlphaRepair does not require any additional retraining or fine-tuning stage on bug fixing datasets since the MLM training is done as part of the pre-training tasks in CodeBERT. While our basic idea is applicable for APR at different levels, following state-of-the-art learning-based APR tools [32, 51, 80], we focus on single line patches in this work. Figure 2 provides an overview of our approach:

- **Step 1 (Section 3.1)**: We first take in a buggy project and separate the surrounding context and the buggy line according to fault localization information. We encode both the context before and after into token representations. Additionally, we also encode the buggy line as a comment in the natural language input for CodeBERT.
- **Step 2 (Section 3.2)**: Using the buggy line, we generate multiple **mask lines** using templates (replace entire line, replace starting/ending part of line, etc). Each mask line replaces the buggy line and is tokenized together with the surrounding context as inputs to CodeBERT.
- **Step 3 (Section 3.3)**: We iteratively query CodeBERT to generate candidate patches using mask lines. Each patch replaces the mask line with a generated code line.
- **Step 4 (Section 3.4)**: We use CodeBERT again to provide patch ranking by computing the score of the generated patch using the joint probability of the generated tokens.
- **Step 5 (Section 3.5)**: We compile each candidate patch and validate it against the test suite. Finally, we output a list of plausible patches for developers to examine.

3.1 Input Processing

To generate the inputs for AlphaRepair, first we extract the buggy line and surrounding context from the buggy project. We use the CodeBERT tokenizer which is built using byte-level byte pair encoding (BBPE) – a technique to reduce the size of the vocabulary by breaking uncommon long words into subwords that are found commonly in the corpus [71]. BBPE has been used in various models and shown to mitigate Out of Vocabulary issues [47, 62].

Figure 3 provides an example input of a buggy program. We first define our tokenization structure, a list of tokens as inputs for CodeBERT. We tokenize both the context before and after and sandwich the mask line (*placeholders* for what CodeBERT will predict, described in Section 3.2) between them. For program repair, the buggy line itself is also important to generate a patch. However, it does not make sense to include it as a part of the context since we aim to generate code to replace it.

To capture the encoding for the buggy line, we make use of the bimodal nature of CodeBERT where it can take in both programming language and also natural language (comments). We transform the original buggy line into a comment by surrounding it with the block comment characters (`/* comment */`). Recall Equation 1 which describes the basic MLM loss function, where X_{masked} is a mask sequence. X_{masked} in CodeBERT concatenates both natural language and code tokens such that $X_{masked} = W_{masked}, C_{masked}$, where W_{masked} and C_{masked} are mask sequences of natural language and code token sequences. The original MLM loss function is now:

$$\mathcal{L}_{bimodal\_MLM} = \sum_{i \in masked} -\log(P(x_i | W_{masked}, C_{masked})) \quad (2)$$

This way CodeBERT learns both modalities of function representation (natural language and function code). AlphaRepair makes use of this additional understanding and transforms the buggy line into a comment. Together the comment buggy line, context before,

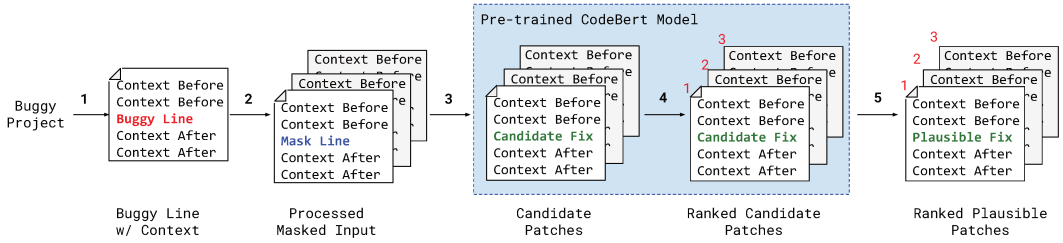


Figure 2: AlphaRepair overview

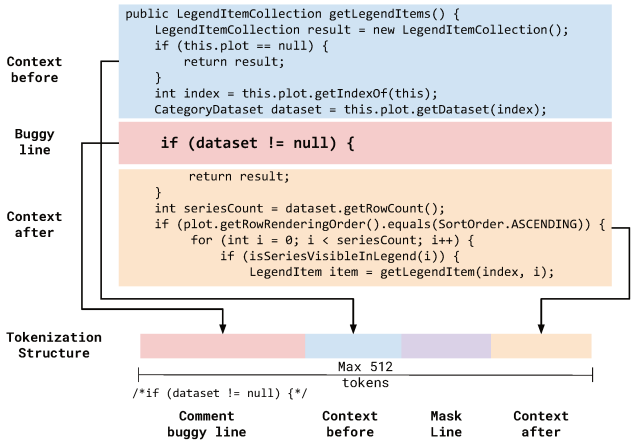


Figure 3: Example input for AlphaRepair

Complete mask	`if (fnType != null) {`
line replace	`<mask><mask> .... <mask><mask>`
line before	`<mask><mask> .... <mask><mask>` `if (fnType != null) {`
line after	`<mask><mask> .... <mask><mask>`
Partial mask	`return foundDigit && !hasExp;`
partial after	`<mask><mask> ... <mask> !hasExp;`
partial before	`return <mask><mask> ... <mask>`
Template mask	`primitiveValues.put(double.class, 0);`
method replace	`<mask><mask>...<mask>(double.class, 0);`
parameter replace	`primitiveValues.put(<mask><mask>..<mask>);`
single replace	`primitiveValues.put(double.class, <mask>);`
add parameter	`primitiveValues.put(double.class, 0, <mask>);`
Template mask	`if (endIndex < 0) {`
expression replace	`if (<mask><mask>...<mask>) {`
more ||/&& cond	`if (endIndex < 0 || <mask>..<mask>) {`
replace operator	`if (endIndex <mask> 0 ) {`

Figure 4: Different strategies to generate mask lines

mask line and context after are tokenized as input for CodeBERT. To maximize the context we can encode, we start from the buggy line and increase the context size (lines away from the buggy code) until we reach the maximum CodeBERT input token size of 512.

3.2 Mask Generation

In order to generate patches, we replace the buggy line with a **mask line**. Mask line is defined as a line with one or more mask tokens - `<mask>`. We use CodeBERT to fill each mask token with a replacement code token and together the filled mask line becomes a generated patch line. Figure 4 shows the 3 strategies we use to generate a mask line: complete, partial and template mask.

Complete mask. The simplest strategy is to replace the entire buggy line with a line containing only mask tokens. We refer to this as line replacement since we ask CodeBERT to generate a new line to replace the buggy line. We also generate mask lines where we add mask tokens before/after the buggy line. These represent bug fixes where a new line is inserted before/after the buggy location.

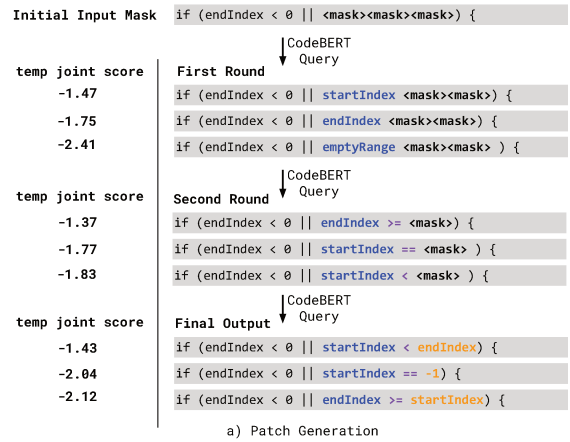
Partial mask. Another strategy of generating mask lines is by reusing partial code from the buggy line. We first separate the buggy line into its individual tokens and then keep the last/first token and replace all other tokens before/after with the mask tokens. We then repeat this process but append more tokens from the original buggy line to generate all the possible versions of partial mask lines.

For example, in the partial before strategy in Figure 4, we start with generating a mask line of `return <mask><mask>...<mask>` by keeping the first token of `return`. Then we generate another mask line by keeping the first two tokens (`return foundDigit`) to generate `return foundDigit <mask><mask>...<mask>`. In total, we generate $(L-1)$ number of mask lines, where L is the number of tokens in the buggy line, for both partial after and before generation method.

This approach is motivated by patches where the fix will reuse parts of the buggy line. By prepending and appending the mask line with a part of the buggy line, we can reduce the number of tokens CodeBERT needs to generate for a correct fix. Furthermore, the partial buggy code acts like initial starting point for CodeBERT to start generating tokens by providing important context.

Template mask. We implemented several template-based mask line generation strategies targeting conditional and method invocation statements as they are two of the most common bug patterns [25, 38, 57]. Additionally, several traditional APR tools [17, 21, 48, 76] focus solely on fixing conditional statement related bugs, showing the importance of targeting common bug patterns. Unlike the previous 2 strategies, template mask can only be generated for specific buggy lines. The first set of templates are designed to target buggy method invocations. Method replacement will replace the method call name with mask tokens. This represents asking CodeBERT to generate a replacement method call using the same parameters as before. We also use several parameter based changes: replacing the entire inputs with mask tokens, replacing one parameter with mask tokens, and adding additional parameter(s) (more than one parameters can be added since we vary the number of mask tokens, therefore CodeBERT can add multiple parameters).

We also designed template mask lines for conditional statements in the form of a Boolean expression. We generate mask lines that



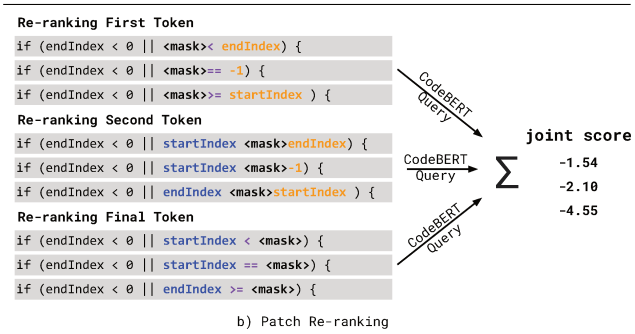


Figure 5: Patch generation and re-ranking example

replace the entire Boolean expression or add additional and/or expressions by appending the statement with mask tokens. Additionally, we also identify common operators (`<`, `>`, `>=`, `==`, `&&`, `||`, etc) and replace them directly in the buggy line with mask tokens.

These template-based mask line generations are inspired by common fixes for many bugs [25, 38, 57] and also previous APR tools that utilize preset templates to fix bugs [29, 35, 44, 45, 53]. These simple generated templates serve a similar functionality to the partial masks in providing more starting code for CodeBERT to generate potential patches. For a fix that only needs to modify a small part of the buggy code, CodeBERT only needs to generate a small number of tokens using mask lines from the template masks. By including a larger portion of the buggy code, we reduce the search space CodeBERT has to consider.

For each generated mask line, we increase the number of generated tokens from 1 until the total number of tokens in that line becomes $(L + 10)$ where L is the number of tokens in the original buggy line. For example, if we use the "partial-after" strategy on a buggy line with L of 12 and we keep the first 5 original tokens, we will vary the number of masked tokens from 1 to 17. This process is done for each masking strategy that we apply (except for the replace operator strategy where we only replace common operators with a single mask token). We apply the input processing step (Section 3.1) for each mask line to obtain a list of processed inputs for CodeBERT.

3.3 Patch Generation

In order to generate a patch that replaces the original buggy code, we use CodeBERT to generate code token for every mask token in our input. To this end, we leverage the original training objective of mask replacement used in CodeBERT. CodeBERT is trained by predicting the correct token value that has been replaced with a mask token given its surrounding context. For each mask token, CodeBERT outputs a probability for each token in its vocabulary to replace the mask token. Typically, during training, a small percentage of tokens is masked out (< 15%) and the model will attempt to predict the actual token that has been replaced.

The task we have is similar to the original training objective of CodeBERT where we also preprocessed the inputs such that a small set of tokens has been masked out. However, *a key difference between our input and the masked training data is our mask tokens are grouped together.* A distinguished feature of CodeBERT and BERT family of models [18, 23, 27, 47] is the bidirectional nature where each token representation is predicated not only on context before but also context after. In order to generate replacement tokens for the mask tokens, CodeBERT looks at the tokens before and after the mask location. This works well for the training objective since the mask tokens are spread out where each token has sufficient tokens before/after to give context. However, for our input data, the mask tokens are together. To generate an output for a mask token in the middle, the immediate context before/after are all mask tokens.

In order to facilitate token generation for grouped mask tokens, we iteratively output tokens by replacing mask tokens with previously generated tokens. Figure 5a shows an example of how this process is done. We start with the initial input mask line of `if (endIndex <0 || <mask><mask><mask>) {` and use CodeBERT to determine the top N most probable replacement tokens for the first mask token. N is the *beam width* and is an adjustable parameter for our approach. In this example, N is 3 meaning we take the top 3 most likely token along with its conditional probability. In the next iteration, we query CodeBERT again by replacing the first mask token with the top 3 replacement tokens (`1.startIndex, 2.endIndex, 3.emptyRange`) to find the top 3 token pairs with the highest joint conditional probability (`1.startIndex <, 2.startIndex ==, 3.endIndex >=`). We call this joint conditional probability value *temp joint score*. Given n as the mask token length, $T = \{t_1, ...t_p, MASK_{p+1}, ...MASK_n\}$ as the mask line with p tokens generated so far ($p <= n$), let $C^*(T, t)$ be the CodeBERT conditional probability of generating t when all tokens in T after and including t have been masked out, the temp joint score is defined as:

$$\textit{temp joint score}\ (T) = \frac{1}{p}\sum_{i=1}^{p}\log(C^*(T, t_i)) \tag{3}$$

We note here that the temp joint score does not represent the actual probability of the generated token once the complete line has been generated (all mask tokens have been replaced). This is because CodeBERT uses both context before and after when determining the likelihood of a replacement token, the probability value does not account for the mask tokens to be generated in future. When computing the temp joint score of a token sequence ($\{t_1, ...t_{p-1}, t_p, MASK_{p+1}, ...MASK_n\}$) in the mask line T, CodeBERT sees the values of the tokens before t_p ($\{t_1, ...t_{p-1}\}$), however all tokens after are masked out ($\{MASK_{p+1}, ...MASK_n\}$). That said, *the temp joint score is conditioned on the mask tokens*

whose values have not yet been decided, and the conditional probability does not account for the future concrete values of those mask tokens. Thus, we use this probability value as a proxy only temporarily and further re-assign the likelihood after (described in Section 3.4) for more precise patch ranking. In each iteration, we use the temp joint score to keep only the top 3 highest score generated token sequences. We repeat this process until we finish generating tokens for all mask tokens in our input.

By generating tokens sequentially, we guarantee that when generating any mask tokens, CodeBERT has at least one side of the immediate context. This helps with generating more syntactically correct candidate patches since CodeBERT can use the previous immediate context to inform what the best next tokens should be. This process is similar to beam search commonly used in code or natural language generation tasks [66]. One difference is that traditional code generation can accurately calculate the likelihood of a sequence as the average of the log conditional probability of the generated tokens. For our approach, the naive average is only an approximation of the likelihood since the probability outputs for tokens in the beginning of the mask line do not include future generated tokens. To address this issue, we further re-rank each candidate patch by re-querying CodeBERT to obtain an accurate likelihood value as shown in Section 3.4.

3.4 Patch Re-Ranking

The re-ranking procedure makes use of the CodeBERT model again. The key idea is to provide an accurate score (i.e. likelihood) for each patch after it is *fully* generated for more effective patch ranking. We start with the complete patch with all the generated tokens. We then mask out only one of the tokens and query CodeBERT to obtain the conditional probability of that token. We apply the same process for all other previous mask token locations and compute the joint score which is an average of the individual token probabilities. Given n generated tokens in a sequence: $T = \{t_1, t_2, ...t_n\}$, let $C(T, t)$ be the CodeBERT conditional probability when masking out only token t in the sequence T, the joint score is defined as:

$$\textit{joint score}\ (T) = \frac{1}{n}\sum_{i=1}^{n}\log(C(T, t_i)) \tag{4}$$

The joint score can now be understood as the conditional probability of the generated sequence (i.e. given both contexts before and after, what is the likelihood of the generated patch according to CodeBERT?). This is done for all patches generated across all mask generation strategies (complete, partial, and template mask). We divide it by n to account for the token length difference since different mask lines have different numbers of mask tokens.

Figure 5b shows an example of the re-ranking process. We use the 3 patches from the patch generation example and for each of them mask out the first token and obtain the probability value from CodeBERT. We repeat this process for the other two tokens and finally we end up with joint scores for all 3 patches. We use the joint scores to provide a ranking for each patch. By re-querying CodeBERT to obtain the joint score we can provide more accurate patch ranking that allows for prioritization when only a subset of generated patches can be validated.

3.5 Patch Validation

For each candidate patch we generate, we apply the corresponding changes to the buggy file. We compile each patch and filter out any patches that fail to compile. We then run the test suite against each compiled patch to find plausible patches that pass all the tests.

4 EXPERIMENTAL DESIGN

4.1 Research Questions

In this paper, we study the following research questions:

- **RQ1:** How does AlphaRepair compare against state-of-the-art APR tools?
- **RQ2:** How do different configurations impact the performance of AlphaRepair?
- **RQ3:** What is the generalizability of AlphaRepair for additional projects and multiple programming languages?

We demonstrate the effectiveness of AlphaRepair by comparing against both state-of-the-art traditional and learning-based APR tools with *perfect fault localization* - the exact fix location of the bug is provided and *not perfect fault localization* - use the suspicious locations generated by fault localization [75] as inputs. Note that the former is the preferred or only comparison setting for all recent learning-based techniques since it eliminates the impact of other factors (such as fault localization) and can show the pure potential of different patch generation strategies [32, 51, 68, 80]. Therefore, this paper also uses perfect fault localization by default unless specifically mentioned. We also show the contribution for each of our design components by conducting an ablation study. Finally, we evaluate the generalizability of AlphaRepair to additional projects in Defects4J 2.0 and QuixBugs. Additionally, we also evaluate multilingual repair capability of AlphaRepair by testing on the Python version of QuixBugs.

4.2 Implementation

AlphaRepair is implemented in Python with PyTorch [60] implementation of the CodeBERT model. We directly reuse the model parameters of the pre-trained CodeBERT model. For perfect fault localization patch generation, we use a beam width of 25 and generate at most 5,000 patches which is comparable to other baselines [32, 51]. For not perfect fault localization patch generation, we use a beam width of 5 and consider the top 40 most suspicious lines same as the recent Recoder tool [80]. We use Ochiai fault localization [3], same as previous approaches [44, 80]. For patch validation, we use the UniAPR tool [11]. All patches generated are validated and we evaluate AlphaRepair on an 8-core workstation with Intel i7 10700KF Comet Lake CPU @3.80GHz and 16GB RAM, running Ubuntu 20.04.3 LTS and OpenJDK Java 64-Bit Server version 1.8.0_312 with NVIDIA GeForce RTX 3080 Ti GPU. For all our experiments, we set a time-out of 5-hour end-to-end time limit for fixing one bug, consistent with previouslearning-based tools [42, 51, 65, 80].

4.3 Subject Systems

For evaluation, we use the widely used benchmark of Defects4J [34]. Defects4J is a collection of reproducible bugs from open-source

projects in Java. We first use Defects4J version 1.2 to answer research questions 1 and 2. Defects4J 1.2 contains 391 bugs (after removing 4 depreciated bugs) across 6 different Java projects. To address research question 3, we use Defects4J 2.0 which adds 438 bugs on top of Defects4J 1.2. Since AlphaRepair is designed for single line bug fixing, we evaluate only on the 82 single line bugs present in the new bugs in Defects4J 2.0, this setup is similar to previous single line APR tools [14, 32, 51, 80]. We also use QuixBugs [43] that contains 40 small classic algorithms with single line bugs used to evaluate many APR tools [19, 20, 32, 51, 78, 80]. QuixBugs contains both Python and Java versions of the same buggy programs. We evaluate AlphaRepair on both Python and Java versions to demonstrate the multilingual ability of our tool.

4.4 Compared Techniques

We compare AlphaRepair against state-of-the-art baselines containing both learning-based and also traditional APR tools. For learning-based APR, we choose 6 recently published tools evaluated on Java or Python bug datasets: Recoder (Java) [80], DeepDebug (Python) [19], CURE (Java) [32], CoCoNuT (Java and Python) [51], DLFix (Java) [42], and SequenceR (Java) [14]. These tools use NMT models to generate patches given the buggy line and surrounding context. Following the most recent Recoder work, we also compare against 12 state-of-the-art traditional single-hunk APR tools: TBar [44], PraPR [25], AVATAR [45], SimFix [31], FixMiner [35], CapGen [74], JAID [12], SketchFix [29], NOPOL [17], jGenProg [52], jMutRepair [53] and jKali [53]. In total, our baseline comparisons comprise of 18 different APR tools.

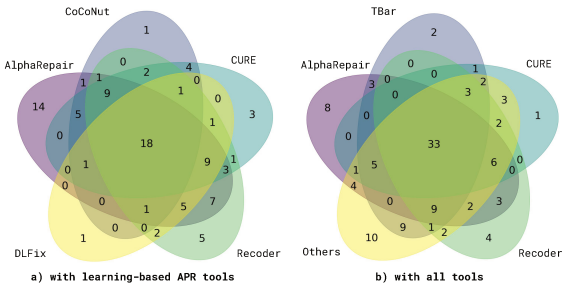
Following prior work [25, 32, 44, 51, 80], we use patch correctness results gathered from previous papers [25, 32, 80] for Defects4J 1.2 evaluation and remove depreciated bugs. We use the recently updated results of Recoder by the authors [59] instead of the outdated results in the original paper [80]. For many tools, we can only obtain either perfect fault localization or not perfect fault localization, therefore we only compare against baselines where the evaluation is under the same localization setting. For Defects4J 2.0 evaluation, we directly run the two best performing baselines of TBar (template-based) and Recoder (learning-based) with perfect fault localization under the same setting as our tool and report the results. For QuixBugs evaluation, we compare against several learning-based APR tools since they have shown to perform the best on QuixBugs for both Java and Python. All baseline results on QuixBugs are taken from previous papers/experiments [19, 32, 80].

For evaluating our technique against state-of-the-art tools, we use the standard metrics of both *plausible* patches that just pass the entire test suite of a project, and *correct* patches that are syntactically or semantically equivalent to the developer patches. Following the common practice for APR, the correct patches are determined by manually inspecting each plausible patch for semantic equivalency.

5 RESULT ANALYSIS

5.1 RQ1: Comparison against state-of-the-art

5.1.1 Perfect Fault Localization. We first compare AlphaRepair with state-of-the-art learning-based and traditional APR tools under the preferred perfect fault localization setting. Table 1 shows the performance of AlphaRepair along with other baselines that also use perfect fault localization. AlphaRepair can successfully generate correct fixes for 74 bugs which outperforms all previous baselines including both traditional and learning-based APR techniques.



a) with learning-based APR tools b) with all tools

Figure 6: Correct patch Venn diagrams for Defects4J 1.2

```
int k = 0;
for (Matcher m : matchers) {
-     if (m instanceof CapturesArguments) {
+     if (m instanceof CapturesArguments && i.getArguments().length > k) {
          ((CapturesArguments) m).captureFrom(i.getArguments()[k]);
      }
```

mask type: template mask - more && condition
mask line: if (m instanceof CapturesArguments && <mask>...<mask>) {
bug-ID: Mockito 34

a)

```
char c = s.charAt(i);
switch (c) {
+     case '\0': sb.append("\\0"); break;
      case '\n': sb.append("\\n"); break;
      case '\r': sb.append("\\r"); break;
```

mask type: complete mask - line replace
mask line: <mask>...<mask> case '\n': sb.append("\\n"); break;
bug-ID: Closure 77

b)

Figure 7: Example bug fixes in Defects4J 1.2

To show the effectiveness of AlphaRepair further, we evaluate the number of unique bugs that only AlphaRepair can fix. We first compare against learning-based APR tools. Figure 6a shows the unique fixes of AlphaRepair and other learning-based tools (we exclude SequenceR since it has 0 unique bug fixes). We observe that AlphaRepair is able to fix the most number of unique bugs of 14. Figure 6b shows the unique fixes of AlphaRepair, the 3 best performing baselines and all other APR tools combined (Others in Figure 6b). We observe that AlphaRepair is able to fix the most number of unique bugs of 8. This also demonstrates that AlphaRepair can be used together with other techniques to further increase the number of correct patches that can be generated.

We provide a few examples of unique bugs that only AlphaRepair can fix. Figure 7a shows a bug with a missing length check on the array obtained from `i.getArguments()`. This is a difficult bug for both traditional and learning-based tools to fix since the array used is not a variable but is obtained from a method call. Traditional APR tools such as template-based tools can detect that the method call returns an array, however it would be infeasible to add a length check for every such case as the search space would be too huge to traverse. Learning-based tools rely on bug fixing changes for training data. While there could be many inserted length check fixes, this specific example, inserting a length check on a return value from a method call, can be rare to find in the dataset. AlphaRepair can fix this bug since the usage of the CodeBERT model does not require any bug fix code pairs and learns directly from large amount of open-source data where many of them contain similar code

Table 1: Baseline comparisons with perfect fault localization

Project	AlphaRepair	Recoder	TBar	CURE	CoCoNuT	PraPR	DLFix	SequenceR
Chart	9	10	11	10	7	7	5	3
Closure	23	21	16	14	9	12	11	3
Lang	13	11	13	9	7	6	8	2
Math	21	18	22	19	16	10	13	6
Mockito	5	2	3	4	4	3	1	0
Time	3	3	3	1	1	3	2	0
Total Correct / Plausible	**74 / 109**	65 / 112	68 / 95	57 / 104	44 / 85	41 / 146	40 / 68	14 / 19

Table 2: Baseline comparisons w/o perfect fault localization

Tool	Correct / Plaus.	Tool	Correct / Plaus.
AlphaRepair	**50 / 90**	CapGen	22 / 25
Recoder	49 / 96	JAID	25 / 31
AVATAR	27 / 53	SketchFix	19 / 26
DLFix	30 / 65	NOPOL	5 / 35
TBar	42 / 81	jGenProg	5 / 27
PraPR	41 / 146	jMutRepair	4 / 17
SimFix	34 / 56	jKali	1 / 22
FixMiner	25 / 31		

where the length checks can be placed on different expressions not just simple array variables. Furthermore, AlphaRepair also captures the context after and identifies the usage of `k` in accessing `i.getArguments()` to insert the correct length check.

Figure 7b shows another bug that only AlphaRepair can fix. The correct fix is to insert an additional case statement to handle the missing case. This is a difficult bug to fix since it does not just slightly mutate any existing code line, but a completely new line needs to be added to handle a specific case in the program execution (when `c` is `\0`). AlphaRepair can generate the correct fix for this bug by identifying its surrounding context. A case statement makes sense to insert here given the context of switch block and other case statements. CodeBERT is able to generate the appropriate case since other case statements use similar identifier formats (`\n`, `\r`). The outcome of the case also follows similarity to nearby context by adding block `sb.append(); break;`. Traditional APR tools cannot fix this bug since it requires adding a new semantic line into the program which is beyond the ability of traditional APR tools built for modifying existing lines or inserting simple statements (try catch, null pointer checker, etc). Learning-based APR tools also struggle with generating the correct patch for this bug since the added line does not fit a common edit pattern found in the training dataset. By observing the surrounding context and using previously seen examples (repeating case statements in other projects), AlphaRepair can generate the correct fix for this bug. These examples combined with the new state-of-the-art results achieved show that AlphaRepair opens up a new promising direction for APR.

5.1.2 Not Perfect Fault Localization. We also compare against state-of-the-art tools without perfect fault localization. Table 2 shows the performance of AlphaRepair with other techniques also evaluated under this setting. AlphaRepair is able to produce 50 correct patches which outperforms previous state-of-the-art tools. Additionally, AlphaRepair is able to correctly fix *7 unique bugs* (the highest among all studied techniques) that cannot be fixed by any other technique. For not perfect fault localization, since we do not have access to the ground truth location of the bug, AlphaRepair generates patches for multiple suspicious lines. To account for this, we lower the beam width of AlphaRepair for this setting in order to generate fewer patches per suspicious line. In this experiment, we show that even with the reduced number of patches generated per suspicious line, AlphaRepair can still achieve state-of-the-art results.

Table 3: Component contribution

Component	#Correct Patch	#Plausible Patch
Complete mask	+20	+29
Partial begin	+13	+24
Partial end	+15	+21
Template	+21	+30
Comment buggy line	+5	+5
Total	74	109

5.2 RQ2: Ablation Study

To study the contribution of adding different components in the design of AlphaRepair, we conduct an ablation study. Table 3 contains the result with each row representing one component and the increase in number of correct/plausible patches AlphaRepair can produce. To show how each mask generation strategy (Section 3.2) improves the number of bugs fixed, we start with the most basic strategy and iteratively add more complex masking strategies. To begin with, we only use complete mask where the entire buggy line is replaced with all mask tokens. This is the case where we give CodeBERT the entire freedom to generate any variety of edits. However, this is often not desirable as the search space grows exponentially with the number of mask tokens and it becomes hard for CodeBERT to obtain a correct fix. We observe that we only achieve 20 correct patches when solely using this mask generation strategy. We obtain increases in correct patches generated as we start to use more mask generation strategies. The highest increase in performance is the usage of template mask lines which add an additional 21 new fixes. Compared to complete mask, template mask only masks out certain parts of the buggy line (parameter, boolean expression, function calls). This allows CodeBERT to fill out only a small number of mask tokens which limits the search space and allows AlphaRepair to quickly find the correct patch. In addition, we also see an increase in performance when we add the encoding for the commented version of the buggy line as input to CodeBERT. The buggy line itself is important for patch generation since it contains important information such as specific variables used and the type of line. This demonstrates that AlphaRepair is able to make use of the buggy line to help guide the generation of valid fixes. Combining all components in AlphaRepair we are able to achieve the final number of correct patches generated.

After the patch generation process, AlphaRepair re-queries CodeBERT again to generate more accurate ranking of each patch. To evaluate the effectiveness of our patch ranking strategy, we compare the order of the correct patches with and without re-ranking. Figure 8 shows the patch ranking of all correct patches generated

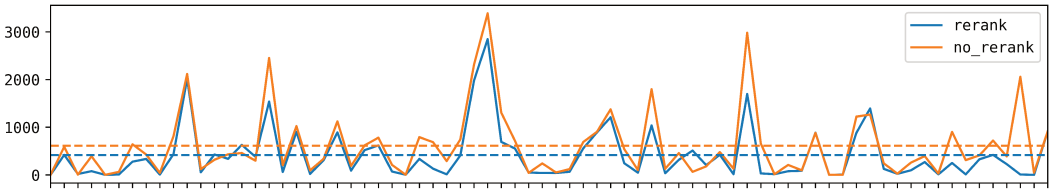


Figure 8: Patch ranking of 74 correct fixes with and without using patch re-ranking (lower is better)

Table 4: Baseline comparisons on Defects4J 2.0

Projects	AlphaRepair	Recoder	TBar
Cli	5 / 5	1 / 3	0 / 3
Codec	6 / 7	2 / 4	1 / 3
Collections	0 / 1	0 / 0	0 / 0
Compress	1 / 3	1 / 3	1 / 2
Csv	1 / 2	1 / 3	1 / 3
Gson	2 / 3	0 / 1	0 / 0
JacksonCore	3 / 3	2 / 3	1 / 2
JacksonDatabind	8 / 9	2 / 2	1 / 4
JacksonXml	0 / 0	0 / 0	0 / 0
Jsoup	9 / 16	2 / 4	3 / 8
JxPath	1 / 1	0 / 0	0 / 0
Total Correct / Plausible	**36 / 50**	11 / 23	8 / 25

```
            html.attributes().put(attribute);
        }
-    } else if (StringUtil.in(name, "base", "basefont", "bgsound", "command",
"link", "meta", "noframes", "style", "title")) {
+    } else if (StringUtil.in(name, "base", "basefont", "bgsound", "script",
"command", "link", "meta", "noframes", "style", "title")) {
            return tb.process(t, InHead);
    } else if (name.equals("body")) {
```

mask type: template mask - add parameter
mask line: } else if (StringUtil.in(name, "base", "basefont", "bgsound", <mask>...<mask>, "command", "link", "meta", "noframes", "style", "title")) {
bug-ID: Jsoup 15

Figure 9: Example bug fix in Defects4J 2.0

Table 5: Baseline comparisons on QuixBugs

Tool	AlphaRepair	CURE	DeepDebug	Recoder	CoCoNuT
Java	28 / 30	26 / 35	- / -	17 / 17	13 / 20
Python	27 / 32	- / -	21 / 22	- / -	19 / 21

with and without re-ranking (dotted line represents the average patch ranking for each strategy). We observe that on average the correct patch is ranked 612th without using the re-ranking strategy. When using re-ranking, the correct patch on average is ranked 418th (31.7% reduction). Furthermore, 61 out of 74 correct patches are ranked higher after re-ranking compared to before. As mentioned in Section 3.3, the temp joint score (no re-ranking) is not an accurate representation of the actual likelihood of the generated tokens since it is conditioned on mask tokens where the concrete values are not yet determined. By re-ranking the patches generated, we make sure that the joint score is calculated without any mask tokens, providing an accurate likelihood calculation. This demonstrates that the patch re-ranking process in AlphaRepair can effectively order the patches and prioritize patches that are ranked higher in case that only a subset of generated patches can be validated.

5.3 RQ3: Generalizability of AlphaRepair

5.3.1 Defects4J 2.0. To demonstrate the generalizability on additional projects and bugs and confirm that AlphaRepair is not simply overfitting to bugs in Defects4J 1.2, we evaluate AlphaRepair on the 82 single line bugs in Defects4J 2.0 dataset. Table 4 shows the results compared against other baselines on Defects4J 2.0. We observe AlphaRepair is able to achieve the highest number of correct patches of 36 (3.3X more than top baseline). Defects4J 2.0 contains a harder set of projects for APR with different variety of fixes compare to Defects4J 1.2. We observe that while template-based tools such as TBar was able to generate a high amount of correct patches for Defects4J 1.2, the number of correct patches it can generate for Defects4J 2.0 is limited. Learning-based tools such as Recoder will also suffer from moving to a harder evaluation dataset since the edits are learnt from training datasets which might not be present in Defects4J 2.0. In contrast, AlphaRepair does not use any fine-tuning on specific bug datasets which makes it less prone to suffer from generalizability issues of traditional template-based or learning-based tools.

Figure 9 shows an example of a bug from Defects4J 2.0 dataset that only AlphaRepair can fix. In this example, the code checks if the variable `name` is one of the string literals. The bug is caused by missing a string literal of `"script"`. This bug is particularly hard to fix for both traditional and learning-based APR tools. For traditional tools such as template-based ones, designing this specific pattern can be hard as it requires insertion of a seemingly arbitrary literal of `"script"`. For learning-based tools, it faces the similar problem in lack of example bug fix pairs where the fix is to insert this particular string literal. In order to generate a correct fix of this bug, one must *understand* the semantic meaning of the code. Upon further inspection, the string literals in this conditional statement are all HTML tags. AlphaRepair can generate the valid HTML string literal of `"script"` by understanding that the surrounding context deals with HTML documents and tags. Additionally, we observe other patches generated by AlphaRepair for this bug include other valid HTML tags such as `"head"`, `"html"`, `"font"`, etc. The specific example and improvement in repair effectiveness over the baselines demonstrate the generalizability of AlphaRepair.

5.3.2 QuixBugs. We show the multilingual repair capability of AlphaRepair by evaluating on the QuixBugs dataset, which contains both Java and Python versions of buggy programs. Table 5 shows the results against state-of-the-art Java and Python APR tools. We observe that AlphaRepair is able to achieve the highest number of correct patches in both Java and Python (28 and 27). We also observe that AlphaRepair is the only tool out of the baselines that can be directly used for multilingual repair (CoCoNuT trains 2 separate models). Traditional learning-based tools require access to bug fixing datasets which are often only in one programming language, restricting the ability for them to be used in a multilingual setting. Unlike traditional learning-based APR tools, CodeBERT is jointly

trained on Java, Python, Go, PHP, JavaScript, and Ruby code snippets, this allows AlphaRepair to be directly used for multilingual repair tasks with minimal modifications.

6 THREATS TO VALIDITY

Internal One internal threat to validity comes from our manual analysis on the correctness of the patches. To this end, the authors carefully looked through all plausible patches and had detailed discussions in order to determine if a patch is correct. We have also released all correct patches for public evaluation along with the code to reproduce our experiments [2].

Another internal threat is the direct usage of the CodeBERT model. The evaluation benchmark of Defects4J could overlap with the training data used in CodeBERT which consists of over 6 million code functions. To address this, we calculated the number of fixed functions in Defects4J that are in the CodeBERT training dataset. Overall, there are 65 out of 391 (16.6%) Defects4J 1.2 bugs and 9 out of 82 (11.0%) Defects4J 2.0 bugs that are present in the original training data. Out of the 74 and 36 bugs that AlphaRepair can correctly fix in Defects4J 1.2 and 2.0, 10 and 5 (13.5% and 13.9%) bugs have their corresponding developer patch in the CodeBERT training data. For the 15 bugs, we manually perturb the buggy code (change variable names, add empty while, if statements, etc) and use the perturbed version for repair. We observe that AlphaRepair is still able to generate the correct fixes for all 15 bugs. We believe this adequately shows that AlphaRepair is not simply overfitting to patches that are present in the original CodeBERT training dataset. Furthermore, the overall comparison results if we were to exclude the 15 overlapping bug fixes would still improve on state-of-the-art baselines (64 vs 63 on best baseline in Defects4J 1.2 and 31 vs 10 on best baseline in Defects4J 2.0). Note QuixBugs dataset is not part of the CodeBERT training data. Future work to address this even more is to retrain the entire CodeBERT model by taking out all patched functions in original data and then re-evaluate AlphaRepair.

Additionally, another internal threat is the experimental setup causing potential differences in results. For example, a longer time-out threshold or faster machine can lead to more bug fixes. To this end, we adopt an ordinary machine configuration (detailed in Section 4.2) and follow prior learning-based APR tools [42, 51, 65, 80] by setting a 5-hour end-to-end timeout for fixing each bug. Furthermore, we follow the common practice in APR by directly taking bug fix results from previous studies instead of directly running the APR tools. To completely address this threat, one would need rerun the results from all the selected baselines APR tools on the same machine with the same time-out threshold.

External The main external threat to validity comes from the evaluation benchmarks we chose. Our claims on the performance of AlphaRepair may not translate to other datasets. To address this threat, we evaluate the generalizability of AlphaRepair on a newer dataset - Defects4J 2.0. We also evaluate our claim on the generalization to other programming languages by studying AlphaRepair on both the Python and Java versions of QuixBugs.

7 CONCLUSION

We propose and implement AlphaRepair, the first cloze-style APR technique that leverages large pre-trained code model directly for repair under a zero-shot learning setting. This opens a new dimension for multilingual learning-based APR that does not require any fine-tuning on repair datasets. We build AlphaRepair using CodeBERT and design inputs to make use of the pre-training objective of CodeBERT to directly generate fix lines from the surrounding context. We evaluate AlphaRepair on popular Java benchmarks of Defects4J and QuixBugs to show that AlphaRepair achieves new state of the art with the highest improvement being 3.3X more bugs fixed than best baseline in Defects4J 2.0. We further demonstrate the multilingual ability of AlphaRepair on the Python version of QuixBugs where we achieved similar results compared to Java.

ACKNOWLEDGMENTS

We appreciate the insightful comments from the anonymous reviewers. This work was partially supported by National Science Foundation under Grant Nos. CCF-2131943 and CCF-2141474, as well as Kwai Inc.

REFERENCES

[1] 2022. Cloze wiki page. https://en.wikipedia.org/wiki/Cloze_test.
[2] 2022. AlphaRepair Dataset. https://zenodo.org/record/6819444.
[3] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. 2007. On the Accuracy of Spectrum-based Fault Localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*. 89–98.
[4] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. arXiv:2103.06333 [cs.CL]
[5] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).
[6] Alexei Baevski, Sergey Edunov, Yinhan Liu, Luke Zettlemoyer, and Michael Auli. 2019. Cloze-driven pretraining of self-attention networks. *arXiv preprint arXiv:1903.07785* (2019).
[7] Samuel Benton, Xia Li, Yiling Lou, and Lingming Zhang. 2020. On the Effectiveness of Unified Debugging: An Extensive Study on 16 Program Repair Systems. In *ASE*. 907–918.
[8] Dalvin Brown. 2021. Hospitals turn to artificial intelligence to help with an age-old problem: Doctors' poor bedside manners. *The Washington Post* (2021). https://www.washingtonpost.com/technology/2021/02/16/virtual-ai-hospital-patients/.
[9] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. (2020). arXiv:2005.14165 [cs.CL]
[10] 2002. Software Errors Cost U.S. Economy $59.5 Billion Annually. *NIST News Release* (2002). http://www.abeacha.com/NIST_press_release_bugs_cost.html.
[11] Lingchao Chen, Yicheng Ouyang, and Lingming Zhang. 2021. Fast and Precise On-the-Fly Patch Validation for All. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1123–1134.
[12] Liushan Chen, Yu Pei, and Carlo A. Furia. 2017. Contract-based program repair without the contracts. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 637–647.
[13] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei,

Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG]
[14] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Transaction on Software Engineering* (2019).
[15] Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. 2019. BoolQ: Exploring the Surprising Difficulty of Natural Yes/No Questions. arXiv:1905.10044 [cs.CL]
[16] Valentin Dallmeier and Thomas Zimmermann. 2007. Extraction of Bug Localization Benchmarks from History. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering* (Atlanta, Georgia, USA) *(ASE '07)*. Association for Computing Machinery, New York, NY, USA, 433–436.
[17] Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. 2014. Automatic Repair of Buggy If Conditions and Missing Preconditions with SMT. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis* (Hyderabad, India) *(CSTVA 2014)*. Association for Computing Machinery, New York, NY, USA, 30–39.
[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805 [cs.CL]
[19] Dawn Drain, Colin B. Clement, Guillermo Serrato, and Neel Sundaresan. 2021. DeepDebug: Fixing Python Bugs Using Stack Traces, Backtranslation, and Code Skeletons. arXiv:2105.09352 [cs.SE]
[20] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019. 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* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 302–313.
[21] Thomas Durieux and Martin Monperrus. 2016. DynaMoth: Dynamic Code Synthesis for Automatic Program Repair. In *2016 IEEE/ACM 11th International Workshop in Automation of Software Test (AST)*. 85–91.
[22] Ahmed Elnaggar, Wei Ding, Llion Jones, Tom Gibbs, Tamas Feher, Christoph Angerer, Silvia Severini, Florian Matthes, and Burkhard Rost. 2021. CodeTrans: Towards Cracking the Language of Silicon's Code Through Self-Supervised Deep Learning and High Performance Computing. arXiv:2104.02443 [cs.SE]
[23] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv:2002.08155 [cs.CL]
[24] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 45, 1 (2019), 34–67.
[25] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical Program Repair via Bytecode Mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. ACM, New York, NY, USA, 19–30.
[26] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press, Cambridge, MA, USA. http://www.deeplearningbook.org.
[27] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. arXiv:2009.08366 [cs.SE]
[28] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-term Memory. *Neural computation* 9 (12 1997), 1735–80.
[29] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. SketchFix: A Tool for Automated Program Repair Approach Using Lazy Candidate Generation *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 888–891.
[30] Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. 2019. Inferring Program Transformations From Singular Examples via Big Code. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 255–266.
[31] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 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, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 298–309.
[32] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (May 2021).
[33] Yanjie Jiang, Hui Liu, Nan Niu, Lu Zhang, and Yamin Hu. 2021. Extracting Concise Bug-Fixing Patches from Human-Written Patches in Version Control Systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 686–698.
[34] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440.
[35] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. FixMiner: Mining relevant fix patterns for automated program repair. *Empir. Softw. Eng.* 25, 3 (2020), 1980–2024.
[36] Christoph H. Lampert, Hannes Nickisch, and Stefan Harmeling. 2009. Learning to detect unseen object classes by between-class attribute transfer. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 951–958.
[37] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: Syntax- and Semantic-Guided Repair Synthesis via Programming by Examples *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 593–604.
[38] Xuan Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 213–224.
[39] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72.
[40] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 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*. 169–180.
[41] Xia Li and Lingming Zhang. 2017. Transforming programs and tests in tandem for fault localization. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–30.
[42] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: Context-Based Code Transformation Learning for Automated Program Repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 602–614.
[43] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A Multi-Lingual Program Repair Benchmark Set Based on the Quixey Challenge *(SPLASH Companion 2017)*. Association for Computing Machinery, New York, NY, USA, 55–56.
[44] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. *TBar: Revisiting Template-Based Automated Program Repair*. Association for Computing Machinery, New York, NY, USA, 31–42.
[45] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 456–467.
[46] Yang Liu. 2019. Fine-tune BERT for Extractive Summarization. arXiv:1903.10318 [cs.CL]
[47] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. arXiv:1907.11692 [cs.CL]
[48] Fan Long and Martin Rinard. 2015. Staged Program Repair with Condition Synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 166–178.
[49] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. 2020. Can automated program repair refine fault localization? a unified debugging approach. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 75–87.
[50] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. arXiv:2102.04664 [cs.SE]
[51] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 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* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 101–114.
[52] Matias Martinez, Thomas Durieux, Jifeng Xuan, Romain Sommerard, and Martin Monperrus. 2015. Automatic Repair of Real Bugs: An Experience Report on the Defects4J Dataset. arXiv:1505.07002 [cs.SE]
[53] Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java (Demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 441–444.
[54] Ehsan Mashhadi and Hadi Hemmati. 2021. Applying codebert for automated program repair of java simple bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 505–509.
[55] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the*

38th International Conference on Software Engineering. ACM, 691–701.

[56] Devon H. O'Dell. 2017. The Debugging Mindset. *acmqueue* (2017). https://queue.acm.org/detail.cfm?id=3068754/.

[57] Kai Pan, Sunghun Kim, and E. James Whitehead. 2009. Toward an Understanding of Bug Fix Patterns. *Empirical Softw. Engg.* 14, 3 (jun 2009), 286–315.

[58] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability* 25, 5-7 (2015), 605–628.

[59] pkuzqh/Recoder 2022. Recoder correct patches update. https://github.com/pkuzqh/Recoder/commit/fae824702b8eeedc17d3394d5ff0bae325a18aed.

[60] PyTorchWebPage 2018. PyTorch. http://pytorch.org.

[61] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 24–36.

[62] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).

[63] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. arXiv:1910.10683 [cs.LG]

[64] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1985. *Learning internal representations by error propagation.* Technical Report. California Univ San Diego La Jolla Inst for Cognitive Science.

[65] Seemanta Saha, Ripon K. Saha, and Mukul R. Prasad. 2019. Harnessing Evolution for Multi-Hunk Program Repair. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 13–24.

[66] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. arXiv:1409.3215 [cs.CL]

[67] Wilson L Taylor. 1953. "Cloze procedure": A new tool for measuring readability. *Journalism quarterly* 30, 4 (1953), 415–433.

[68] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. *An Empirical Investigation into Learning Bug-Fixing Patches in the Wild via Neural Machine Translation.* Association for Computing Machinery, New York, NY, USA, 832–837.

[69] Alina Tugend. 2021. A Smarter App Is Watching Your Wallet. *The New York Times* (2021). https://www.nytimes.com/2021/03/09/business/apps-personal-finance-budget.html.

[70] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *CoRR* abs/1706.03762 (2017). arXiv:1706.03762

[71] Changhan Wang, Kyunghyun Cho, and Jiatao Gu. 2019. Neural Machine Translation with Byte-Level Subwords. arXiv:1909.03341 [cs.CL]

[72] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. arXiv:2109.00859 [cs.CL]

[73] R.W. Webster and D. Hess. 1993. A real-time software controller for a digital model railroad system. In *[1993] Proceedings of the IEEE Workshop on Real-Time Applications*. 126–130.

[74] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 1–11.

[75] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

[76] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise Condition Synthesis for Program Repair. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 416–426.

[77] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. 2020. XLNet: Generalized Autoregressive Pretraining for Language Understanding. arXiv:1906.08237 [cs.CL]

[78] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2019. A Comprehensive Study of Automatic Program Repair on the QuixBugs Benchmark. In *2019 IEEE 1st International Workshop on Intelligent Bug Fixing (IBF)*. 1–10.

[79] Lingming Zhang, Lu Zhang, and Sarfraz Khurshid. 2013. Injecting mechanical faults to localize developer faults for evolving software. *ACM SIGPLAN Notices* 48, 10 (2013), 765–784.

[80] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. *A Syntax-Guided Edit Decoder for Neural Program Repair.* Association for Computing Machinery, New York, NY, USA, 341–353.