2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) | 978-1-6654-0296-5/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICSE43902.2021.00107

CURE: Code-Aware Neural Machine Translation for Automatic Program Repair

Nan Jiang
Purdue University
West Lafayette, USA
jiang719@purdue.edu

Thibaud Lutellier
University of Waterloo
Waterloo, Canada
tlutelli@uwaterloo.ca

Lin Tan
Purdue University
West Lafayette, USA
lintan@purdue.edu

***Abstract*—Automatic program repair (APR) is crucial to improve software reliability. Recently, neural machine translation (NMT) techniques have been used to fix software bugs automatically. While promising, these approaches have two major limitations. Their search space often does not contain the correct fix, and their search strategy ignores software knowledge such as strict code syntax. Due to these limitations, existing NMT-based techniques underperform the best template-based approaches.**

We propose CURE, a new NMT-based APR technique with three major novelties. First, CURE pre-trains a programming language (PL) model on a large software codebase to learn developer-like source code before the APR task. Second, CURE designs a new code-aware search strategy that finds more correct fixes by focusing on compilable patches and patches that are close in length to the buggy code. Finally, CURE uses a subword tokenization technique to generate a smaller search space that contains more correct fixes.

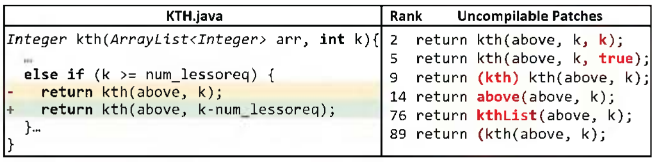
Our evaluation on two widely-used benchmarks shows that CURE correctly fixes 57 Defects4J bugs and 26 QuixBugs bugs, outperforming all existing APR techniques on both benchmarks.

***Index Terms*—automatic program repair, software reliability**

I. Introduction

Automatic program repair is crucial to reduce manual software debugging efforts [1]–[24]. There has been recent adoption of neural machine translation, a widely used technique for natural language processing (NLP) tasks, to generate correct code automatically given buggy source code [18]–[25]. Thanks to the strong learning capabilities of NMT models, NMT-based APR techniques have outperformed most existing rule-based approaches [18]–[20]. NMT models use deep-learning techniques to encode buggy source code as intermediate representation in the latent space automatically, and then decode the encoded representation into target correct code. By minimizing the loss function and updating the weight parameters, NMT models learn to capture the hidden relationship between buggy code and correct code without any manual design of fix patterns or feature templates.

For a search-based APR approach (including NMT-based techniques) to generate a correct fix, it needs to satisfy two conditions: (1) the correct fix must be in the *search space*, which is the set of all patches that the APR approach can generate, and (2) the *search strategy* must be effective to find the correct fix in a reasonable amount of time. Given that a correct patch is in the search space, it is desirable that the search space is small, so that it is easier to find the correct patch [26]. Despite being among the most effective APR approaches, NMT-based approaches still fail to fix many bugs [18]–[20].



KTH.java

```
Integer kth(ArrayList<Integer> arr, int k){
  ...
  else if (k >= num_lessoreq) {
-   return kth(above, k);
+   return kth(above, k-num_lessoreq);
  }...
}
```

Rank	Uncompilable Patches
2	`return kth(above, k, k);`
5	`return kth(above, k, true);`
9	`return (kth) kth(above, k);`
14	`return above(above, k);`
76	`return kthList(above, k);`
89	`return (kth(above, k);`

Fig. 1. Uncompilable patches generated by NMT-based models, and their ranks, for a bug in QuixBugs. The line in yellow background (starting with '−') is the buggy line. The line in green background (starting with '+') is the correct fix. The red code in generated patches disobeys Java syntax.

Compared to natural language text, source code has its own characteristics such as a strict syntax, code semantics, and an infinite number of possible identifiers. These characteristics impose unique challenges for NMT models to fix bugs automatically.

First, the strict syntax of source code is hard for NMT models to learn. A major reason is that existing techniques [18]–[20], [27] learn from buggy code snippets and the corresponding fixed correct code snippets (typically a few lines to tens of lines per bug), and do not use the entire source code repositories (typically millions of lines of code per project). Thus, existing NMT-based APR approaches have limited knowledge about the rigorous syntax of programming languages and the big picture of how developers write code. The missed opportunities are twofold: (1) existing techniques fail to take advantage of the large amount of available source code, and (2) they see partial code snippets only (which alone are often syntactically incorrect), and miss the big picture of complete methods, classes, and projects. For example, for the fix of replacing "`while (x) {`" with "`while (y) {`", the open bracket "{" is syntactically incorrect in this code snippet, i.e., missing the closing bracket "}".

Such ineffectiveness is evident as demonstrated by data. For example, up to 67%–97% of patches generated by the state-of-the-art NMT-based APR models [19], [20] are uncompilable, wasting valuable resources on incorrect patches. Figure 1 shows a bug in QuixBugs and some of the top-ranked patches generated by CoCoNuT [19]. All of these patches are uncompilable, because they call methods with wrong parameters, invoke undeclared variables, or contain mismatched parenthe-

978-1-6654-0296-5/21/$31.00 ©2021 IEEE
DOI 10.1109/ICSE43902.2021.00107

sis. One important reason that CoCoNuT fails to generate a correct patch for this bug despite generating 20,000 patches, is the large number of uncompilable patches. The code-aware NMT-based approach we propose automatically generates a correct patch (identical to the + line highlighted in green) for this bug. The ranks of these uncompilable patches are high because existing NMT-based APR techniques focus on translating buggy code snippets to correct code snippets, which are partial code segments instead of full methods or programs. Since they fail to see the whole picture of the entire program or programming languages, they generate many patches with syntax errors.

Failing to learn how developers write code, existing NMT-based APR techniques also generate compilable but obviously-incorrect patches, as they do not look like developer-written code. These uncompilable and compilable-but-incorrect patches decrease the accuracy and efficiency of APR models, preventing APR models from generating more correct patches faster.

Second, the infinite number of possible identifiers causes NMT techniques for code to handle an enormous vocabulary if using word-level tokenization, where a *vocabulary* contains all the unique tokens that an NMT model recognizes. Considering the complexity of NMT architectures, it is computationally too expensive for NMT-based APR models to use an enormous vocabulary. Yet with a limited vocabulary size, their search spaces do not contain all correct fixes. SequenceR [20] uses a small vocabulary and shirks this complexity to a later reconstruction stage, while CoCoNuT [19] uses a vocabulary of more than 130,000 tokens but still suffers from the *out-of-vocabulary* (*OOV*, i.e., an NMT model cannot recognize or generate a token) problem, resulting in its search space that still misses correct fixes.

A. Our approach

Thus, we propose an NMT-based approach that is specially designed to parse, analyze, model, and search source code (as opposed to natural language text) to fix bugs automatically. Our approach, CURE, not only improves the search space (a smaller search space containing more correct patches) but also uses a more effective search strategy to find and rank correct patches higher, which are achieved through the following three main techniques that we design and use:

(1) Programming language models: To help NMT models learn developer-like source code (i.e., not only compilable but also similar to those written by programmers), we apply the pre-training and fine-tuning workflow to the APR task. Specifically, pre-trained language models have brought great improvement to many NLP tasks [28], [29]. They learn the probability distribution over sequences of words from a large amount of natural language text. Then one fine-tunes the pre-trained language model for a specific task by adding an extra model to it (e.g., adding a classifier for classification tasks). The language model provides vectorized representations of input sequences to the model added to it. Since a pre-trained language model is typically trained on a larger dataset (since it is unsupervised learning and does not require ground truth), it offers the added model more information regarding sentence structures (e.g., syntax) and about what human-like text are (e.g., readability), which improves the quality of the generated text of the fine-tuned model for the specific task significantly.

Given the effectiveness of language models in the NLP domain, we propose to add a language model pre-trained on software code, referred to as *programming language (PL) model*, to an NMT architecture to create a new APR architecture. The PL model is trained to predict the next tokens in code sequences and learns to generate developer-like code. Then, we combine the PL model and the NMT model to form the full *APR model* and fine-tune it for APR tasks.

Our PL-enhanced NMT approach ranks correct patches higher in the search space to fix more bugs (Section V-B1).

(2) Code-aware search strategy: When using an NMT model to generate a sequence of tokens to form a patch, ideally, one prefers the sequence with the highest score, e.g., average log probability of every token in sequence. Since this is prohibitively expensive [30], in practice, one uses a search strategy to choose proper tokens at each step. *Beam search* is a common search strategy for NMT that keeps the most n probable sequences at each step, where n is the *beam size*.

The beam size of NLP tasks is typically 5 to 20 [30], [31]. Since source code has more possible identifiers and a bigger search space than natural languages [19], [20], the NMT models for APR usually require larger beam sizes (100 [20] to 1,000 [19]) to generate enough candidate patches. However, with large beam sizes, the vanilla beam search chooses many bad patches, either uncompilable or far from correct in length.

To address this challenge, we propose two solutions: valid-identifier-check strategy and length-control strategy. First, since source code is a formal language, only valid tokens are allowed, including keywords and variables in scope. Invalid tokens make a patched program uncompilable, let alone capable of passing test cases. Therefore, we propose and design a **valid-identifier-check strategy** to improve the vanilla beam search, which performs static analysis to identify all valid identifiers and then forces beam search to generate only sequences with valid tokens.

Second, with a large beam size, beam search finds many very short sequences such as "`{`" and "`try {`", which are incorrect code snippets to fix bugs. Since correct fixes in our training data are typically of similar length to the buggy lines, we use a **length-control strategy** to punish too-short and too-long sequences to prompt CURE to generate patches of a similar length to the buggy line.

Our code-aware beam-search strategy finds more correct fixes by generating more compilable patches and patches of similar length to the buggy lines. (Section V-B2).

(3) Subword tokenization: The enhanced word-level tokenization proposed by CoCoNuT [19] reduces the vocabulary size of code, by using camel letters, underscores, and numbers to split long identifiers. However, many compound words (such as "`binsearch`" for binary search) do not contain these special characters. The previous parsing approach keeps "`binsearch`" as one word, which is OOV, instead of breaking it into "`bin`" and "`search`", both of which are in the vocabulary. Thus, we use *byte-pair encoding* (BPE), a type of subword tokenization techniques, to tokenize compound words and rare words to further address the OOV problem.

BPE improves the search space by both including more correct patches and reducing its size (Section V-B3).

B. Contributions

We design and implement a code-aware NMT-based technique, *CURE*, to fix bugs automatically. Our contributions include:

- An approach to pre-train a PL model for APR on a very large software codebase—4.04 million methods from 1,700 open-source projects—to capture code syntax and developer-like source code,
- A new APR architecture that combines a pre-trained PL model and NMT architectures to learn both code syntax and fix patterns,
- A new code-aware beam-search strategy, which uses valid-identifier-check and length-control strategies to find more correct fixes,
- A new application of subword tokenization to the APR task, which addresses the OOV problem effectively, and
- A new APR approach, CURE, that combines the techniques above, and its evaluation on two widely-used benchmarks—Defects4J and QuixBugs, where CURE fixes the most number of bugs, 57 and 26 bugs respectively, outperforming all existing APR tools. CURE is the first NMT-based approach that outperforms all state-of-the-art APR approaches on Defects4J.

Availability: Data is available in a GitHub repository[1].

II. Background

Candidate, Plausible and Correct Patches: Following previous work [19], we call patches generated by the models candidate patches. Patches that pass the validation stage are plausible, and patches identical or semantically equivalent to developers' patches are called correct patches.

Parameters and Hyperparameters: Parameters are the weights between the connections of the network. These parameters are optimized during the training phase. Hyperparameters are arguments of the network defined before the training process. They generally include layer dimensions, number of layers, and optimization parameters.

Pre-Training and Fine-Tuning: Pre-training is the process of training a model for a general task (e.g., next word prediction) with a very large dataset. After pre-training, one gets a pre-trained model with updated parameters. A pre-trained model can be fine-tuned for a similar but specific task (e.g., text generation) with few training data. During fine-tuning, usually one needs to add extra models to the pre-trained model to fit the task, and the parameters of both the pre-trained model and added models are updated.

Context-aware neural machine translation (CoNuT) architecture: We use CoNuT as our NMT architecture in this paper. CoNuT consists of a buggy lines encoder, a context encoder, a merger, a decoder, an attention module, and a token generation module, where the encoders and decoder are implemented with convolutional sequence-to-sequence architecture [32]. The details of CoNuT is described in [19]. CoNuT has shown good results for APR, and convolutional architecture can be stacked to capture hierarchical features and long dependencies for larger contexts [19].

III. Approach

To address the challenges described in the Introduction, we design and apply three novel techniques, i.e., *subword tokenization* to improve the search space (Section III-C), a *programming language model* to learn developer-like source code and improve patch ranking (Section III-D and Section III-E), and a new *code-aware beam-search strategy* (Section III-G) to improve patch ranking and generate more correct patches.

A. Overview

Figure 2 presents an overview of our approach. CURE consists of three stages: training, inference, and validation. During the training stage, CURE extracts methods from open-source projects, referred to as *PL training data*, and tokenizes them (step 1a in Figure 2). Different from previous work [18]–[24], we use **subword tokenization**, which produces a smaller but more accurate search space that contains more correct patches. CURE uses these tokenized methods to train a **new programming language model** that learns developer-like source code with correct syntax (step ②). CURE also tokenizes the buggy lines, context, and correct fixes extracted from the commit history of open-source projects, referred to as *patch training data*, into sequences of tokens (step 1b). We use these sequences to fine-tune an ensemble of k APR models (step ③). Each APR model combines the PL model with a context-aware neural machine translation (CoNuT) model [19].

During the inference stage, a user provides a buggy project along with the location of buggy lines to CURE. These are standard input that existing APR tools require [1], [5], [19], [20], [33], [34]. CURE tokenizes the buggy and the context lines (step 1c), then analyzes the source code to extract a list of *valid identifiers* that are in scope of the buggy lines (step ④). The patch generation module generates a list of candidate patches using a **new code-aware beam-search strategy** (step ⑤). This new algorithm discards many irrelevant patches on the fly (i.e., as soon as an invalid token is generated) and penalizes patches that are unlikely to be correct (e.g., fixes that are very different from the buggy line in length), which saves a lot of resources and allows CURE to search deeper for correct patches.

In the validation stage, CURE validates candidate patches by compiling and executing the test suites of the patched

[1] https://github.com/lin-tan/CURE

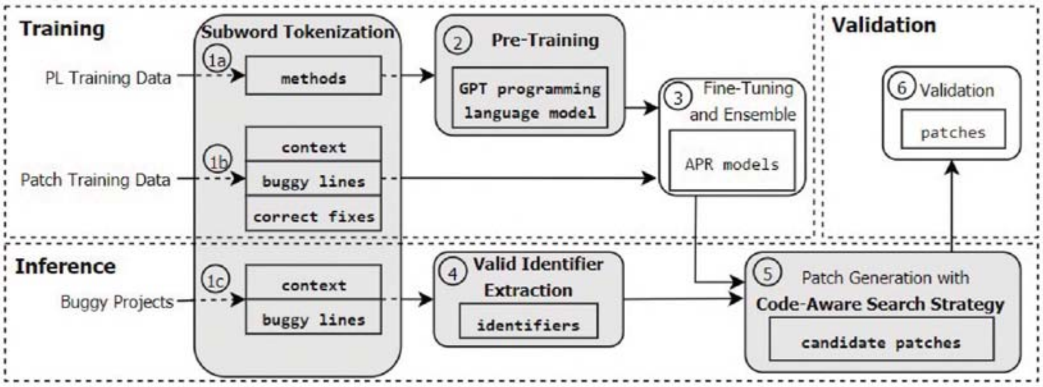


Fig. 2. Overview of CURE. Grey boxes represent major novelties of CURE. Circled numbers indicate the steps of generating patches with CURE.

projects. CURE outputs a list of plausible patches (step ⑥) for developers to examine.

B. Data Extraction

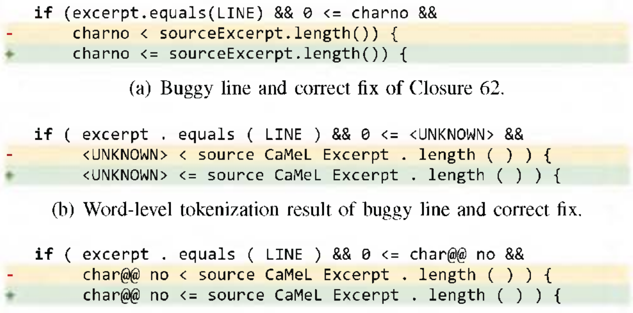
CURE uses two different types of training data. First, the GPT PL model is trained on millions of methods extracted from open-source Java projects. Second, CURE fine-tunes the PL model for the APR task. This step requires APR specific training data (i.e., buggy lines, context, and correct fixes). We use CoCoNuT's training data shared on GitHub [19]. CoCoNuT's authors extracted this dataset from open-source repositories and identified buggy commits based on keywords in commit messages ("fix," "bug," and "patch"). They also cleaned the dataset using commit message anti-patterns ("rename," "clean up," "refactor," "merge," "misspelling," and "compiler warning"). Similar to CoCoNuT, we use the method surrounding the buggy lines as context.

C. Code Representation and Tokenization

Word-level tokenization: To tokenize buggy-, context-, and fixed-lines to token sequences, CURE first uses enhanced word-level tokenization [19] to separate code lines by spaces, camel letters, underscores, strings, and numbers (except 0 and 1).

Out-of-vocabulary Issue: The vocabulary size after the word-level tokenization is larger than what is commonly used in NLP and the test set still contains 2% of OOV tokens. Excluding rare tokens is problematic for source code because OOV tokens are likely to be important project-specific tokens. Excluding such tokens makes it difficult for NMT models to fix bugs in these new projects.

Some existing NMT-based APR models [19] do not generate OOV tokens, missing the opportunity to fix more bugs. SequenceR uses a special token as a placeholder for OOV tokens, and then uses a copy mechanism to reconstruct them. The copy mechanism replaces the placeholder tokens with the most likely token from the input buggy lines. However, this solution would fail to generate some patches, since it can only copy tokens appearing in the buggy lines.



```
  if (excerpt.equals(LINE) && 0 <= charno &&
-     charno < sourceExcerpt.length()) {
+     charno <= sourceExcerpt.length()) {
```

(a) Buggy line and correct fix of Closure 62.

```
  if ( excerpt . equals ( LINE ) && 0 <= <UNKNOWN> &&
-      <UNKNOWN> < source CaMeL Excerpt . length ( ) ) {
+      <UNKNOWN> <= source CaMeL Excerpt . length ( ) ) {
```

(b) Word-level tokenization result of buggy line and correct fix.

```
  if ( excerpt . equals ( LINE ) && 0 <= char@@ no &&
-      char@@ no < source CaMeL Excerpt . length ( ) ) {
+      char@@ no <= source CaMeL Excerpt . length ( ) ) {
```

(c) Subword-level tokenization result of buggy line and correct fix.

Fig. 3. Tokenized results that use word-level tokenization and subword tokenization of Closure 62 in Defects4J.

Subword tokenization: To address the OOV problem and reduce the vocabulary size, we use *byte pair encoding (BPE)*, which is an unsupervised learning algorithm to find the most frequent subwords in a corpus by merging the most frequent byte pair iteratively [35]. BPE has been used in many NLP tasks and is useful to reduce vocabulary size and mitigate the OOV problem efficiently [35]–[37].

Figure 3 shows an example from the inference stage demonstrating the effectiveness of the subword tokenization. Lines starting with '−' are the buggy lines (input) and those starting with '+' are the correct fixes. Figure 3(a) shows the source code of a real bug in Defects4J [38], while Figure 3(b) shows the code after using the enhanced word-level tokenization. Figure 3(c) shows the same code tokenized by our subword tokenization. In Figure 3, each consequence separated by space is a token excluding the '−' and '+' signs.

When using only the enhanced word-level tokenization, the variable "charno" is an OOV token. Thus, CoCoNuT and SequenceR fail to fix this bug since CoCoNuT cannot generate OOV tokens and SequenceR does not fix it correctly with the copy mechanism. With our subword tokenization, "charno" is split into two tokens, both of which appear in the vocabulary—"char@@" ("@@" indicates that the token needs to be concatenated with the following token) and "no", enabling CURE to generate a correct patch for this bug.

By applying subword tokenization, we use a smaller vocabulary to form a smaller but better search space that contains more correct patches. Section V-B3 evaluates the impact of our subword tokenization approach.

D. Programming Language Model

To address the challenges of learning developer-like source code, we train a language model on open-source programs, referred to as a *programming language model (PL model).* A PL model optimizes the probability of a sequence of tokens being a real-world code snippet. We use Generative Pre-trained Transformer (GPT) [37] for PL modeling because GPT has been shown to improve the performance of many different NLP tasks [37], [39].

Pre-training a PL model allows for separating programming language learning from patch learning. The advantages are twofold. First, GPT learning is unsupervised and only requires complete methods; therefore one can extract a large amount of data automatically and accurately to train it. Second, during fine-tuning, the APR model already knows the PL syntax (thanks to the PL model), making the fine-tuning more efficient.

Given a sequence of tokens representing a method, $\mathbf{x} = (x_1, ..., x_n)$, where x_i is a token in the method sequence $\mathbf{x}$, the PL modeling objective is to maximize the average likelihood:

$$L_{GPT}(\mathbf{x}) = \frac{1}{n}\Sigma_{i=1}^{n}\log P(x_i|x_1, ..., x_{i-1}; \Theta) \tag{1}$$

where Θ represents matrices of trainable weights of the PL model. $P(x_i|x_1, ..., x_{i-1}; \Theta)$ is the conditional probability of token x_i being the next token, given a sequence of $x_1, ..., x_{i-1}$, which is calculated by the PL model with weights Θ. At a high-level, the objective of the PL model training is to find the best weights (Θ) so that sequences of tokens $x_1, ..., x_n$ representing real methods in projects obtain a higher L_{GPT} score than other sequences. Since methods in popular open-source projects are dominantly well-formed correct blocks of code, we use them to train our PL model to learn if a given sequence of tokens is likely to form real-world code (compilable and looks like written by programmers).

E. Fine-Tuning for APR with a PL Model

After pre-training the PL model, CURE fine-tunes the GPT PL model for the APR task by combining it with an NMT model as the APR model. We use the CoNuT (Section II) as CURE's NMT architecture.

The APR model takes buggy lines and their context as input and aims to generate a correct patch. During the fine-tuning process, the APR model is trained to learn the transformation from the buggy lines and context (e.g., the buggy method) to the correct fix. We use $\mathbf{x_b} = (x_{b_1}, ..., x_{b_n})$ to denote the buggy lines, $\mathbf{x} = (x_1, ...x_{c_n})$ to denote the context, and $\mathbf{y} = (y_1, ..., y_{f_n})$ to denote the correct fixes, where $b_1, ..., b_n$ are the indices of the buggy lines in the context, while c_n and f_n are the lengths of the context and correct fixes respectively.

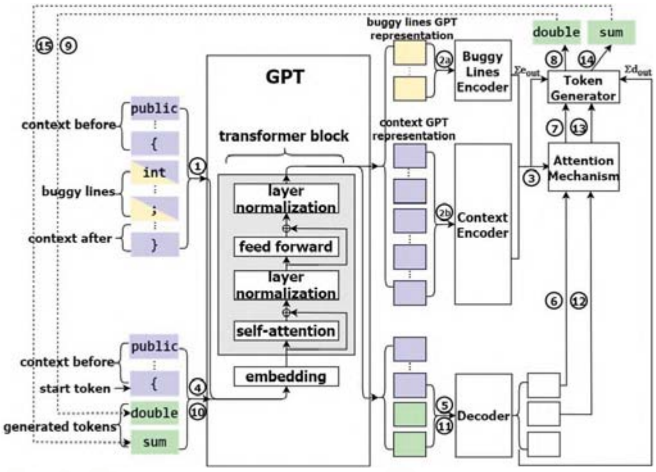


Fig. 4. Architecture of the APR models used in CURE. Yellow, purple and green boxes refer to the buggy lines, context and the generated patch.

We denote the weights of the PL model as Θ and weights of CoNuT as Φ. The APR model is fine-tuned by updating Θ and Φ to maximize the following average log-likelihood:

$$\begin{aligned} L_{NMT}(\mathbf{x}, \mathbf{x_b}, \mathbf{y}) &= \frac{1}{f_n}\Sigma_{i=1}^{f_n}\log P(y_i|\mathbf{x}, \mathbf{x_b}, y_0, \ldots, y_{i-1}; \Theta, \Phi) \\ y_0 &= x_{b_1-1} \end{aligned} \tag{2}$$

$P(y_i|\mathbf{x}, \mathbf{x_b}, y_0, \ldots, y_{i-1}; \Theta, \Phi)$ is the conditional probability calculated by the APR model with weights Θ and Φ, where y_i is the token following the sequence $(y_0, y_1, \ldots, y_{i-1})$ in the correct fix, given the buggy lines $\mathbf{x_b}$ and context $\mathbf{x}$. For the first token in the correct fix, the probability is the conditional probability of token y_1 given y_0, where y_0 is the token right before the correct fix, i.e., x_{b_1-1}. For example, the entire method "`kth()`" is the context in Figure 1, while the buggy lines and the correct fixes start at the same index in the context, and y_0 is the token { right before the "`return`" statement.

To prevent the PL model from losing the information it learned during pre-training, we include the language modeling (i.e., L_{GPT}) as an auxiliary objective to the fine-tuning process. It also improves the generalization of the fine-tuned model [37]. Therefore, the APR model is fine-tuned by maximizing the combined average log-likelihood:

$$\begin{aligned} L_{APR}(\mathbf{x}, \mathbf{x_b}, \mathbf{y}) = L_{NMT}(\mathbf{x}, \mathbf{x_b}, \mathbf{y}) + \lambda L_{GPT}(\mathbf{y}') \\ \mathbf{y}' = (x_1, x_2, ..., x_{b_1-1}, \mathbf{y}) \end{aligned} \tag{3}$$

where $\mathbf{y}'$ is the token sequence from the beginning of the buggy method to the last token in the correct fix ($x_1, x_2, ..., x_{b_1-1}$ is the prefix of $\mathbf{x}$ before $\mathbf{x_b}$). Probability $L_{GPT}(\mathbf{y}')$ is the likelihood of $\mathbf{y}'$ being a real source code snippet, while λ is a hyperparameter referring to the coefficient of L_{GPT} in the combined log-likelihood L_{APR}.

The fine-tuning stage aims to find the best set of parameters Θ and Φ to maximize L_{APR} for all buggy lines, context, and correct fixes in the training data.

In the training mode, the APR model takes the pre-trained GPT module (the PL model) and the patch training data as input. The patch training data consists of the buggy lines, the context, and the correct fixes. We train the APR model for multiple epochs (i.e., multiple passes on the training data) to obtain the best combination of weights Θ and Φ.

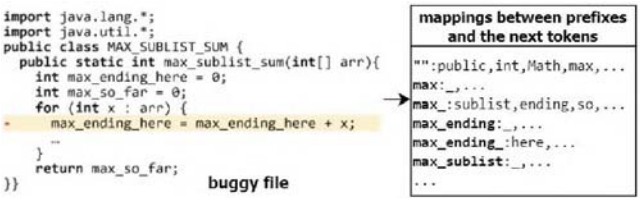
In the inference mode, the APR model has access to only the buggy lines and their context and outputs a sequence of tokens representing the patch. Figure 4 shows a simplified view of the architecture of our combined APR model and how the model is used in inference mode. Our APR model consists of two components: a PL model (GPT) and an NMT model (CoNuT).

First, CURE generates the GPT representation of the context lines (step ① in Figure 4). As explained in Section III-D, the GPT model was trained on complete methods, therefore the input of the GPT model needs to be a method. If we directly feed the first token of the buggy line to the GPT model (“`int`” in Figure 4), the GPT model will generate a bad embedding for it since it expects the first token of a sequence to be the first token of a method (e.g., “`public`”).

Hence, the GPT model generates an embedding for all tokens in the buggy method. The CoNuT model contains two encoders. The buggy lines encoder only takes the representation of the buggy line as input. Therefore, CURE extracts the subsequence that corresponds to the buggy line embedding from the buggy method embedding (yellow boxes in Figure 4) and forwards it to the buggy lines encoder (step ②a). The second encoder is for the context and takes the embedding of the entire buggy method (purple boxes in Figure 4) as input (step ②b). CURE merges the output of the two encoders (step ③) before sending it to the attention mechanism and the token generator.

To start generating tokens, the attention mechanism and the token generator need the encoder's and the decoder's output. At the start of the inference, none of the fixed tokens have been generated yet. CoCoNuT started the decoding sequence with an initial “`<START>`” token. However, it is better to initialize the sequence with the last token of the context before the buggy line to provide additional contextual information. To obtain the embedding of this token, we pass the context before the buggy line to the GPT model (step ④) and then feed the embedding of the last token (“{”) to the decoder (step ⑤). The decoder generates a representation of the token, which is forwarded to the attention mechanism (step ⑥).

The attention mechanism combines the output of the two encoders and the output of the decoder to form the attention map between the last token (“{” in the example) and the buggy method. Then, the token generation outputs the first token of the fixed sequence (“`double`” in step ⑧). This token is then appended to the decoder input (step ⑨). Then, the decoder starts the next iteration (steps ⑩ to ⑮) with the input “`{ double''` and generates the token “`sum`”. This iterative process continues until the end-of-sequence token “`<EOS>`” is generated.



```
import java.lang.*;
import java.util.*;
public class MAX_SUBLIST_SUM {
  public static int max_sublist_sum(int[] arr){
    int max_ending_here = 0;
    int max_so_far = 0;
    for (int x : arr) {
-     max_ending_here = max_ending_here + x;
      ...
    }
    return max_so_far;
}}
```

```
"":public,int,Math,max,...
max:_,...
max_:sublist,ending,so,...
max_ending:_,...
max_ending_:here,...
max_sublist:_,...
...
```

Fig. 5. An example of extracting mappings between prefixes and valid next tokens from buggy projects. Line with yellow background is buggy line.

F. Ensemble Learning

Prior work [19] shows that ensemble learning, i.e., combining multiple models, enables NMT-based APR approaches to fix more bugs: the number of bugs correctly fixed rises from 22 to 44 when the number of models increases from 1 to 20. Therefore, we combine (1) models with different hyperparameters and (2) models with two different architectures (CoNuT and FConv [32]) for our ensemble learning. The GPT PL model is general as it represents the entire PL. Thus, each APR model starts with the same PL model, fine-tunes it, and combines it with CoNuT or FConv architectures that have different hyperparameters (step ③ of Figure 2).

Balancing the computation cost and tuning effectiveness, we use random search to pick different hyperparameter values (e.g., number of convolution layers, convolution dimensions, and dropout) in a reasonable range and tune each model for one epoch. Based on each model's perplexity (i.e., a measurement of how well a model predicts an instance) on our validation data, we choose the top k models for ensemble learning and keep training them until convergence.

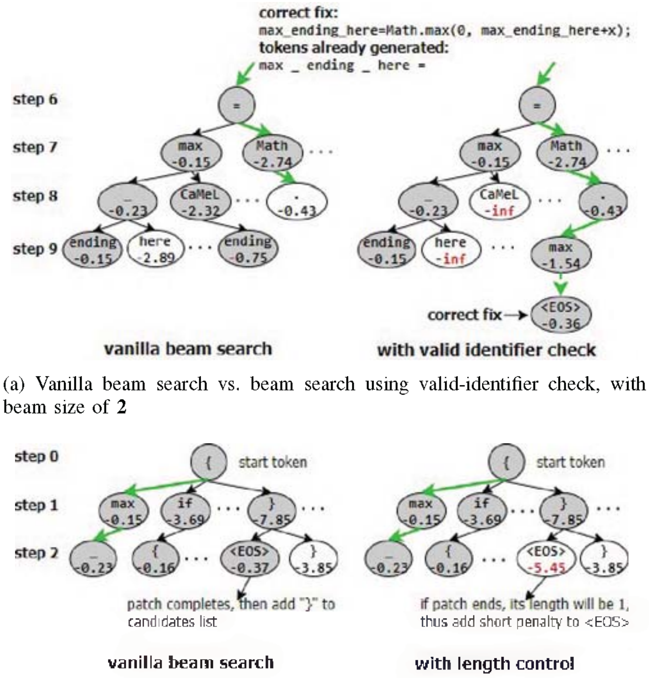
G. Code-Aware Beam-Search Strategy and Patch Generation

The goal of patch generation is to generate the sequence with the highest probability given the buggy line and its context. The APR model generates one token with its probability at a time. Searching for the sequence with the highest probability is exponential in the length of the output sequence. Thus, we need an effective search strategy to find a sequence with a high probability.

Beam search is an optimized greedy strategy and the most common search strategy used for NMT. Beam search keeps only the n (n is beam size, a hyperparameter of beam search) optimal nodes, instead of all nodes, in the search tree to expand at every step and remove the rest. A major issue of the vanilla beam search is that it considers only the log-probability provided by the model to generate the next token. Since other information about code (e.g., variables in scope) is unavailable to the APR model, it often generates a high score for an out-of-scope variable, producing an uncompilable candidate patch.

Therefore, we design two techniques—*valid-identifier check* and *length control*—to make the beam search code-aware.

Valid-Identifier Check: Only a few tokens are valid in a certain Java code snippet since correct code must follow Java syntax and compilation rules. To generate valid identifiers only, CURE first uses static analysis to analyze and extract valid identifiers. Then CURE tokenizes these identifiers



(a) Vanilla beam search vs. beam search using valid-identifier check, with beam size of **2**

(b) Vanilla beam search vs. beam search using length control in the same bug, but with beam size of **1,000**

Fig. 6. Examples using the vanilla beam search and beam search with valid-identifier-check and length-control strategies. ***Green arrows*** are the paths to the correct fixes. ***Grey circles*** are the nodes kept by the search strategies in the search tree at every level, and *white circles* are nodes discarded. ***Red numbers*** are the log probability changed by search strategies.

(e.g., "`max_ending_here`" becomes [`max`, `_`, `ending`, `_`, `here`]), and builds the mappings between all prefixes and their valid succeeding tokens (as showns in Figure 5). These mappings are necessary for the beam-search algorithm to know that after generating the sequence "`max_ending_`", "`here`" is a valid next token because "`max_ending_here`" is a valid identifier.

At every decoding step, the NMT model outputs a probability distribution of all the tokens in vocabulary. CURE's new valid-identifier-check strategy first analyzes the token sequence already generated to get the "prefix" of the next token needed to be generated. If the generated token sequence does not contain any prefix, (i.e., the next token will be the beginning of a new identifier), the "prefix" will be set to the empty string. Then, based on the mappings between all possible prefixes and their valid succeeding tokens, the valid-identifier-check strategy modifies the probability distribution and sets the probability of invalid tokens to $-inf$. By doing this, the valid-identifier-check strategy discards many impossible nodes, increasing the possibility of finding the correct patch.

Figure 6(a) illustrates how our code-aware beam search outperforms the vanilla beam search. The correct fix is "`max_ending_here=Math.max(0,max_ending_here)`", and the start of the output sequence ("`max_ending_here=`") has already been generated in steps 1 to 6. We use a beam size of 2 to simplify the illustration. The path to the correct fix is marked with green arrows and the nodes considered by the beam-search strategies are in light grey.

During step 7, the two most likely nodes according to the APR model are "`... max`" and "`... Math`", where "`...`" refers to "`max_ending_here=`" which has already been generated. However, at step 8, the average log-likelihood of "`... Math .`", which is the sequence denoted by the green path in the left subfigure of Figure 6(a), is less than that of "`... max _`" and "`... max CaMeL`", so the vanilla beam search drops it. Thus, the entire subtree containing the correct fix is excluded.

In contrast, with our valid-identifier-check strategy, the average log-likelihood of "`... max CaMeL`" is set to `-inf` because there is no valid identifier starting with "`max CaMeL`". Therefore, our code-aware beam search keeps searching the subtree of node "`... Math`", which leads to the correct fix.

Length Control: In our training data, most correct fixes have a similar length as the buggy lines. We find the length difference of 75% of the bugs in our 2.7 million patch training data is less or equal to 5 tokens. This means that most of the time, the correct fixes are small modifications to the buggy lines, and more complex changes are less common.

Therefore, we use length-control strategy to generate patches of a similar length of the buggy lines, by punishing short and long patches. At every decoding step, the length-control strategy calculates the length of the sequence already generated. If the current length is much smaller than the buggy lines, it decreases the log-likelihood of "`<EOS>`" to prevent this patch from reaching the end. And if the current length is already much larger than the buggy lines, it increases the log-likelihood of "`<EOS>`" to prompt this patch to end.

To determine the penalty value, we leverage the length difference distribution in the patch training data to calculate the log-probability of each length difference, denoted as function F_{len}. Our length-control strategy modifies the log-likelihood of token "`<EOS>`" by adding the following penalty to it:

$$penalty = \begin{cases} 0 & -5 <= l_b - l_p <= 5 \\ F_{len}(l_b - l_p) & \text{otherwise} \end{cases}$$

where the lengths of the buggy lines and the patch sequence already generated are l_b and l_p respectively. We empirically set a tolerance threshold of 5 to increase flexibility.

Figure 6(b) illustrates this issue. The bug is the same as in Figure 6(a) but with a larger beam size of 1,000. In step 2, the sequence "{ }" reaches the "`<EOS>`" token. Using the vanilla beam search, patch "{ }" has a low average log-probability but is still in the top 1,000, this is added to candidate patches because the beam size is large (1,000). Such low score patches take up the valuable slots and prevent correct patches from being selected. In our code-aware beam-search strategy, since the complete sequence, "{ }", is much shorter than the buggy sequence, the "`<EOS>`" token receives a large penalty and is not selected as a candidate node (not included in the 1,000 highest average log-probabilities), allowing the search strategy to search deeper along other paths.

While CURE focuses on fixing bugs whose fixes are similar in length to the buggy lines, our length-control strategy is general and can be adapted to generate longer patches by modifying the penalty weights. Given the complexity of APR, fixing similar-length bugs and other bugs separately may be an effective way to decompose this complex task.

H. Patch Validation

After APR models generate candidate patches, we reconstruct the token sequences to code statements. We first concatenate tokens end with "@@" to their successors, which is the reconstruction from subwords to words. Then we extract donor code from the buggy code file to reconstruct the abstracted tokens (numbers and strings).

Reconstructed statements are ranked by the average log-probability of their tokens and then inserted into the buggy code file to replace the buggy lines. Every patched project is compiled to filter the uncompilable patches and we run the test suites until we find a patch that satisfies two conditions: (1) passing all the test cases that the buggy project passes and (2) passing at least one test case failed on the buggy project, which are the same criteria for validation used in previous work [19], [40].

IV. Experimental Setup

Realistic Evaluation: To make the evaluation realistic, we need to avoid using future data [19], [41]. We address this issue by using data committed before the first bug in our benchmark (i.e., 2006) for pre-training, fine-tuning, and validation.

PL Training Data: We download all (1,700) open-source Java projects from GitHub that have at least one commit before the first bug in Defects4J according to GHTorrent [42] and roll them back to a version before 2006 to avoid using future data. Then, we use JavaParser [43] to extract all methods except abstract methods and those longer than 1,024 tokens. The PL training data contains 4.04 million methods, with 30,000 instances for validation.

Patch Training Data: We use CoCoNuT's training data shared on GitHub [19] as our patch training data, which is extracted from 45,180 Java projects. These Java projects are a superset of the projects used for PL training data since we need more projects to extract enough patch data and it is too expensive to use all these projects for PL training. Then we discard the instances whose context or correct fixes are longer than 1,024 tokens after subword tokenization. Removing instances from the training set is a common practice for machine learning, and since the test set (bugs in Defects4j and QuixBugs are untouched), this setup is valid. Our patch training data contains 2.72 million training instances and 16,920 validation instances.

Subword Tokenization, Training, Fine-Tuning, and Inference: We set the target vocabulary size to be 50,000 for BPE. For the GPT model, considering previous work's recommendation [37] and our hardware limits, we use an embedding size of 384, eight layers of transformer blocks, and six attention heads. We train GPT for five epochs, using a batch size of 12. We use Adam optimizer [44], and the learning rate increases from 0 to $2.5e^{-4}$ at the first 2,000 training steps and then decreases using a cosine schedule.

To fine-tune the hyperparameters of an APR model, we use random search with the following ranges: convolution dimension (128–512), kernel size (2–10), number of convolutional layers (1–5), and dropout (0–0.5). λ is empirically set to 0.3. We train 100 APR models on a smaller subset of patch training data for one epoch and keep the top five models combining GPT with CoNuT model and top five APR models combining GPT with FConv model. We use Adam optimizer with a learning rate of $6.25e^{-5}$ to keep tuning the top models on our patch training data for one epoch, with a batch size of 12.

In inference mode, we use beam search with a beam size of 1,000, and CURE generates 10,000 candidate patches for every bug. During the validation stage, considering the time cost and that most correct fixes have high ranks, we validate the top 5,000 candidate patches per bug.

Infrastructure: We use GPT implemented by Hugging Face [45], CoNuT and FConv implemented using fairseq [19], [46]. We train and evaluate our models on one 56-core server with one NVIDIA TITAN V and three Xp GPUs.

V. Evaluation and Results

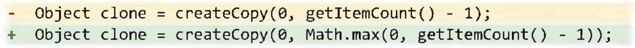
We use two widely-used benchmarks, Defects4J (v1.4.0) [38] and QuixBugs [47] for evaluation. Following [19], we remove two Defects4J bugs, Closure 63 and Closure 93, from our evaluation as they are duplicates of other Defects4J bugs. We compile the patched projects and run the test suites to find plausible patches, i.e., patches that pass the relevant test cases. Two co-authors independently check plausible patches and consider correct only those that are identical or semantically equivalent to developers' patches (92% of agreement, Cohen's k of 0.84), then discuss to resolve disagreements.

We compare CURE with 25 APR techniques [1], [3], [5], [8]–[15], [18]–[20], [33], [34], [48]–[56]. Table I shows the comparison results. The table lists only a few top-ranked techniques in terms of the number of bugs that they fix in each benchmark, including state-of-the-art pattern-based techniques [33], [48], three NMT-based techniques [18]–[20] and the techniques that have been evaluated on QuixBugs. None of these tools uses subword tokenization, pre-trained PL model or, code-aware search strategy. Other tools (e.g., AVATAR [54], kPAR [53], SimFix [10]) either fix fewer bugs than the listed tools or were not evaluated on these benchmarks. Results from *all* 9 techniques in Table I except Astor [6] and Hercules [48] use the perfect fault localization (FL) of bugs to report bug fixing results. As stated in previous work [53], [54], having APR techniques use the same FL techniques (e.g., perfect FL) is a fair way to compare APR techniques since different FL methods affect APR techniques

TABLE I

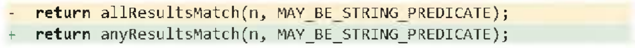
COMPARISON WITH STATE-OF-THE-ART APR TOOLS. THE NUMBERS ARE SHOWN AS X/Y, WHERE X IS THE NUMBER OF BUGS FIXED CORRECTLY AND Y IS THE NUMBER OF BUGS WITH PLAUSIBLE PATCHES. '-' MEANS THE APPROACH HAS NOT BEEN EVALUATED ON THAT BENCHMARK.

Tool	Defects4J 393 bugs	QuixBugs 40 bugs
Astor [6]	-	6/11
Nopol [5]	2/9	1/4
RsRepair [34]	10/24	2/4
Hercules [48]	49/72	-
TBar [33]	52/85	-
SequenceR [20]	14/19	-
DLFix [18]	36/65	-
CoCoNut [19]	44/85	13/20
CURE	**57/104**	**26/35**



```
- Object clone = createCopy(0, getItemCount() - 1);
+ Object clone = createCopy(0, Math.max(0, getItemCount() - 1));
```

Fig. 7. Chart 17 in Defects4J is a bug only fixed by CURE



```
- return allResultsMatch(n, MAY_BE_STRING_PREDICATE);
+ return anyResultsMatch(n, MAY_BE_STRING_PREDICATE);
```

Fig. 8. Closure 10 in Defects4J is a bug that CURE fixes but CoCoNuT does not.

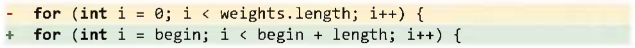
differently. CURE's correctly generated patches are available in our GitHub Repository.

A. RQ1: How does CURE perform against state-of-the-art APR techniques?

In Table I, the results are displayed as x/y, where x is the number of bugs fixed correctly and y is the number of bugs with plausible patches.

CURE fixes the most number of bugs, 57 and 26 respectively, in both Defects4J and QuixBugs. Specifically, CURE generates plausible patches for 104 Defects4J bugs, 57 of which are correctly fixed by CURE, outperforming the best existing approach TBar by five bugs. Compared to NMT-based approaches [18]–[20], CURE correctly fixes 13 more bugs than the best NMT-based approach CoCoNuT. CURE fixes 26 QuixBugs bugs (twice as many bugs as CoCoNuT), including 12 bugs that none of the four existing tools that have been evaluated on QuixBugs can fix. In Defects4J, CURE fixes one unique bug, Chart 17, that has not been fixed by any of the 25 existing approaches.

Bugs that only CURE fixes: Figure 7 shows the unique bug in Defects4J and the correct fix that CURE generates, which is equivalent to developers' patch. The correct fix requires ensuring the second parameter to be non-negative. Pattern-based approaches (e.g., TBar and Hercules) fail to fix it because they have no fix patterns to ensure that a method parameter is non-negative. NMT-based approaches (e.g., SequenceR, DLFix, and CoCoNuT) fail to fix it, since such a fix is uncommon. In our patch training data (already 2.72 million training instances from 45,180 projects), there are only two similar fixes. Thus, it is hard for NMT-based models to capture this transformation due to the lack of more similar fixes. However, adding "`Math.max()`" to ensure non-negativeness is common in Java methods and is captured by our PL model, allowing CURE to fix the Chart 17 bug in Defects4J correctly.



```
- for (int i = 0; i < weights.length; i++) {
+ for (int i = begin; i < begin + length; i++) {
```

Fig. 9. Math 41 in Defects4J is a bug that CURE fixes but pattern-based tools TBar and Hercules do not.

As explained in the Introduction, Figure 1 shows the KTH bug in QuixBugs, which only CURE fixes and none of the existing techniques evaluated on QuixBugs does. CoCoNuT fails to fix this bug as it generates too many uncompilable patches. Nopol, RSRepair, and Astor cannot repair this bug as they do not implement the required fix pattern.

Comparing with the existing best-performing NMT-based approach CoCoNuT, CURE fixes 13 more bugs in Defects4J. Figure 8 shows an example bug that CURE fixes and CoCoNuT fails to fix. The correct fix of Closure 10 requires "`anyResultsMatch`", which is nonexistent in the buggy line or context. CoCoNuT prioritizes tokens in the buggy line and context, thus fails to generate the correct token to fix this bug. In contrast, CURE's code-aware beam-search strategy extracts all valid identifiers, including "`anyResultsMatch`" which is declared out of the context, and generates the correct fix.

Comparing with the best pattern-based approach, CURE fixes five more bugs in Defects4J than TBar, most of which require complex transformations to fix. Figure 9 shows an example. The correct fix of Math 41 requires changes to the initialization of "`i`" and the condition for the loop. TBar does not have such a complex fix pattern. CURE fixes Math 41 since similar transformations can be learned from the patch training data.

Compilable patch rate: In addition to the number of correctly fixed bugs, we use the average compilable rate to measure the effectiveness of CURE learning PL syntaxes and developer-like code. We compare the average compilable rates of the top-k candidate patches generated by different NMT-based models, for bugs in two benchmarks. Table II shows that CURE generates more compilable patches in top-30 candidates than SequenceR, and more compilable patches in all top-30, 100, 1,000, and 5,000 than CoCoNuT (DLFix does not offer compilable rate data). Comparing different rows shows that each component has contributed to the higher compilable patch rate. For example, comparing row "BPE+GPT+CoNuT+vanilla" with row "CURE" shows that our code-ware search has increased the average compilable patch rate by 6% (from 22% to 28%) for the top 100 patches. CURE generates more portions of compilable patches than existing NMT-based approaches, thanks to the PL model and the valid-identifier-check strategy.

These examples and compilable patch rates demonstrate that (1) *the unique capabilities of our model that combines a GPT PL model and an NMT model to learn both developer-like code and fix patterns* to fix more bugs and (2) *the effectiveness*

TABLE II

AVERAGE COMPILABLE RATES OF THE TOP-K CANDIDATE PATCHES FOR BUGS IN TWO BENCHMARKS FROM DIFFERENT MODELS. "VANILLA" AND "CODE-AWARE" DENOTE VANILLA BEAM SEARCH AND CODE-AWARE BEAM SEARCH RESPECTIVELY. CURE IS "BPE+GPT+CONUT+CODE-AWARE". '-' INDICATES DATA UNAVAILABILITY.

Model	Top 30	Top 100	Top 1000	Top 5000
SequenceR [20]	33%	-	-	-
CoCoNuT (CoNuT+vanilla) [19]	24%	15%	6%	3%
BPE+CoNuT+vanilla	28%	18%	7%	4%
GPT+CoNuT+vanilla	28%	20%	9%	5%
BPE+GPT+CoNuT+vanilla	32%	22%	10%	6%
CURE	**39%**	**28%**	**14%**	**9%**

TABLE III

RESULTS OF ABLATION STUDY ON TWO BENCHMARKS. (COCONUT USES 20 MODELS FOR ENSEMBLE WHILE THE REST USE ONLY 10 MODELS.)

Model	Defects4J	QuixBugs
CoCoNuT (CoNuT+vanilla)	44/85	13/20
BPE+CoNuT+vanilla	45/85	16/25
GPT+CoNuT+vanilla	44/84	19/27
BPE+GPT+CoNuT+vanilla	51/94	22/27
CURE (BPE+GPT+CoNuT+code-aware)	**57/104**	**26/35**

of our PL model and the context-aware search strategy in generating more compilable patches.

Type of bugs that CURE is applicable for: Similar to most state-of-the-art G&V APR techniques [1], [3], [5], [8]–[10], [12]–[15], [18]–[20], [33], [34], [49], [51]–[55], CURE is designed to fix single-hunk bugs (i.e., the buggy lines and patches are single code segments, and each buggy hunk has separate test cases).

B. RQ2: What are the contributions of CURE's components?

To study the impact of each novel technique (i.e., GPT PL model, code-aware beam-search strategy, and subword tokenization) of CURE, we compare the following four techniques with CURE: **CoCoNut ("CoNuT+vanilla")** An ensemble of ten CoNuT models and ten FConv models, using word-level tokenization and vanilla beam-search strategy. CoCoNuT uses twice as many models as CURE and the next three techniques (20 versus 10 models) and generates twice as many candidate patches. Each of the next three techniques is an ensemble of five CoNuT models and five FConv models with the vanilla beam-search strategy. The differences are that **"BPE+CoNuT+vanilla"** uses subword tokenization, **"GPT+CoNuT+vanilla"** uses a GPT PL model, and **"BPE+GPT+CoNuT+vanilla"** uses both subword tokenization and a GPT PL model.

All models use a beam size of 1,000, generate 10,000 candidate patches, validate the top 5,000 candidate patches for every bug (except CoCoNuT that generates and validates 20,000 candidate patches for each bug), and are trained on the same dataset.

1) Impact of the GPT PL model: Table III lists the result of the ablation study on two benchmarks. Rows "BPE+GPT+CoNuT+vanilla" versus "BPE+CoNuT+vanilla" show that the GPT PL model helps APR models fix six more bugs in each benchmark. Comparing "GPT+CoNuT+vanilla" with CoCoNuT shows that the GPT PL model helps fix six more QuixBugs bugs. Although they fix the same number of Defects4J bugs, CoCoNuT uses an ensemble of 20 models, while "GPT+CoNuT+vanilla" uses only 10. CoCoNuT with 10 models fixes 38 bugs only [19], which shows an improvement of six more bugs of "GPT+CoNuT+vanilla" versus CoCoNuT with 10 models.

In Table II, comparing "BPE+CoNuT+vanilla" and "BPE+GPT+CoNuT+vanilla" shows that GPT increases the average compilable rate by 2%–4%. In addition, the average rank (the highest rank one is the best) of the correct patches (before validation) generated by "BPE+GPT+CoNuT+vanilla" is 68% higher than that of "BPE+CoNuT+vanilla" (131 vs. 414), indicating that the GPT PL model not only enables APR models to fix more bugs but also improves the ranks of correct patches.

2) Impact of Code-Aware Beam-Search Strategy: Comparing "BPE+GPT+CoNuT+vanilla" and CURE in Table III shows that our code-aware beam-search strategy helps APR models find more correct patches and fix more bugs (six more in Defects4J and four more in QuixBugs). Comparison between "BPE+GPT+CoNuT+vanilla" and CURE in Table II shows that our code-aware beam search increases the average compilable rate by 3%–7%. The average rank of the correct patches (before validation) generated by CURE is 21% higher than "BPE+GPT+CoNuT+vanilla" (101 vs. 131), indicating that our new search strategy also increases the rank of correct patches.

To measure the impact of the length-control strategy, we compare the length of candidate patches generated by "BPE+GPT+CoNuT+vanilla" and CURE. For the "BPE+GPT+CoNuT+vanilla" model, the average length difference between candidate patches and correct fixes is seven tokens. In contrast, the average length difference between the CURE's candidate patches and correct fixes is five tokens. This shows the length-control strategy helps generate more candidate patches with similar length to the correct fixes. Specifically, it helps fix long bugs (e.g., it fixes the longest bug in QuixBugs that cannot be fixed without length-control strategy) since it searches deeper.

3) Impact of subword tokenization: Subword tokenization improves the search space by reducing the size of vocabulary (from 139,423 to 50,057 tokens) and covering more correct fixes. CoCoNuT versus "BPE+CoNuT+vanilla" shows that subword tokenization helps fix four more bugs (one in Defects4J and three more in QuixBugs). "GPT+CoNuT+vanilla" versus "BPE+GPT+CoNuT+vanilla" also shows that subword tokenization helps fix 10 more bugs.

We also compare the number of OOV tokens with different tokenization techniques. With word-level tokenization, 14 bugs contain OOV tokens in our benchmarks (e.g., "`binsearch`" and "`charno`"). In contrast, all these OOV tokens are separated into more common tokens when using subword tokenization. This shows that subword tokenization helps reduce the vocabulary size, improve the vocabulary, make the model easier to train, and eventually fix more bugs.

C. Execution Time

Data extraction: Downloading and extracting 4.04 million methods from 1,700 projects as our PL training data takes one day. We use CoCoNuT's training data shared on GitHub, which takes five days to extract [19]. Both are a one-time cost.

Training PL model: It takes ten days to pre-train the GPT PL model on four GPUs for five epochs. Since the PL model is trained on large and general data, one programming language only needs one PL model and the PL model can be used to enhance tasks other than APR and does not need retraining.

Fine-tuning APR models: It takes 12 days to tune the hyperparameters by training 100 APR models for one epoch. Fine-tuning the top-10 APR models takes, on average, 10.7 hours per model. This is a one time cost as the trained APR models do not need retraining when fixing new bugs.

Cost to fix one bug: In inference, it takes 2.5 minutes on average for CURE to generate 10,000 candidate patches for one bug using four GPUs. During validation, it takes 16.5 minutes on average to validate one bug. Compared with the state-of-art NMT-based approach [19], CURE uses fewer models and validates fewer patches, thus CURE is faster and fixes more bugs.

VI. Limitations

Comparison with previous work: It is difficult to fairly compare our work with all previous work as they use different training data and FL methods. To be as fair as possible, we use the same training data as CoCoNuT, the state-of-the-art NMT technique, and demonstrate significant improvement on both benchmarks. Some previous work uses different training data, but the selection and extraction of data is also a key component of a technique. In addition, to compare with previous APR techniques, we choose to use perfect localization as it is the preferred comparison method [53] and previous work [57] evaluated most available APR techniques with perfect FL.

Generalization to other benchmarks and PL: We evaluate CURE on two Java benchmarks, but the approach is neither tied to a specific PL nor a specific benchmark. CURE is generalizable to other languages by updating the PL parser. The benchmarks we chose are very popular, Defects4J being used to evaluate 25 other APR tools. In the future, we can also evaluate all APR approaches on recent benchmarks such as Bugs.jar [58] or Bears [59].

Accuracy of the training sets: Since our training and pre-training data extraction is conducted automatically, there is a risk that such data is inaccurate. The training data was extracted in previous work [19] and showed to be reasonably accurate on a random sample. For the pre-training data, we extract all complete functions and some of them might be buggy or incorrect. However, the goal of the pre-training is to learn the syntax of the PL, therefore, we mostly care that the data follows the PL syntax, which is verified since we only keep methods successfully parsed by a Java parser.

VII. Related Work

Deep Learning for APR: Different DL-based APR techniques have been developed to fix bugs [18]–[20], [25], [27], [60], compilation issues [22]–[24], or predict the correctness of generated patches [61]. CURE is different from previous work in three ways. First, our subword-tokenization technique addresses the OOV problem encountered by all NMT-based techniques. Second, CURE integrates a new PL model to the APR models that better learns the syntax of source code. Finally, our new code-aware search strategy chooses only valid identifiers during inference, which helps filter out incorrect patches. As a result, CURE generates more reasonable and compilable patches and outperforms all existing techniques.

Automatic Program Repair: Many APR techniques have been proposed, which use genetic programming [1], condition synthesis [2], [5], [9], state abstraction [14], heuristics [8], [10], [12], [62], human-designed fix patterns [3], [6], mined fix patterns [4], [7], [13], [15], [56], [63], bytecode mutation [64], or neural program synthesis [65]. CURE uses a new code-aware NMT approach and fixes more bugs than previous state-of-the-art approaches.

Deep Learning in Software Engineering: The software engineering community had applied deep learning to performing various tasks such as source code summarization [66], [67], code clone detection [68], [69], defects prediction [17], [60], [70], [71], code completion [72], and program synthesis [73], [74]. These techniques, along with ours, demonstrate that deep learning is competitive in different software engineering tasks. Our work introduces code-awareness to DL systems to improve APR. In the future, increasing code-awareness of DL systems applied to other software tasks could also be useful.

Language Model in Software Engineering: Different programming language models have been developed [75]–[87]. None of these approaches have been evaluated on fixing software bugs and have only been used for simpler tasks such as method name generation or source code summarization. Recent work has questioned the generalizability of some of these approaches for more complex tasks [88], [89]. Compared with these models, CURE uses GPT [37], one of the most powerful language models in NLP, to capture code syntax and demonstrates its effectiveness for the more complex APR task.

VIII. Conclusion

We propose and evaluate CURE, a new NMT-based program repair technique that by design parses, models, and searches source code, as opposed to natural language text, to automatically fix bugs. CURE uses an NMT model that contains a PL model, a code-aware search strategy, and a subword-tokenization technique to create a smaller search space that contains more correct patches and find more correct patches. CURE outperforms all existing techniques on two popular benchmarks, fixing 83 bugs. We highlight this direction of code-aware NMT for automatic program repair.

REFERENCES

[1] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A Generic Method for Automatic Software Repair," *TSE*, vol. 38, no. 1, pp. 54–72, 2012.

[2] F. Long and M. Rinard, "Staged Program Repair with Condition Synthesis," in *ESEC/FSE*. ACM, 2015, pp. 166–178.

[3] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "ELIXIR: Effective Object Oriented Program Repair," in *ASE*. IEEE, 2017, pp. 648–659.

[4] F. S. Ocariza, Jr, K. Pattabiraman, and A. Mesbah, "Vejovis: Suggesting Fixes for JavaScript Faults," in *ICSE*. ACM, 2014, pp. 837–847.

[5] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs," *TSE*, vol. 43, no. 1, pp. 34–55, 2017.

[6] M. Martinez and M. Monperrus, "ASTOR: A Program Repair Library for Java (Demo)," in *ISSTA*. ACM, 2016, pp. 441–444.

[7] F. Long, P. Amidon, and M. Rinard, "Automatic Inference of Code Transforms for Patch Generation," in *ESEC/FSE*. ACM, 2017, p. 727–739.

[8] Q. Xin and S. P. Reiss, "Leveraging Syntax-Related Code for Automated Program Repair," in *ASE*. IEEE, 2017, p. 660–670.

[9] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise Condition Synthesis for Program Repair," in *ICSE*. IEEE, 2017, pp. 416–426.

[10] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping Program Repair Space with Existing Patches and Similar Code," in *ISSTA*. ACM, 2018, pp. 298–309.

[11] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "SketchFix: A Tool for Automated Program Repair Approach Using Lazy Candidate Generation," in *ESEC/FSE*. ACM, 2018, pp. 888–891.

[12] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-Aware Patch Generation for Better Automated Program Repair," in *ICSE*. ACM, 2018.

[13] X. Liu and H. Zhong, "Mining stackoverflow for program repair," in *SANER*. IEEE, 2018, pp. 118–129.

[14] L. Chen, Y. Pei, and C. A. Furia, "Contract-Based Program Repair without the Contracts," in *ASE*. IEEE, 2017, pp. 637–647.

[15] X. B. D. Le, D. Lo, and C. Le Goues, "History Driven Program Repair," in *SANER*, vol. 1. IEEE, 2016, pp. 213–224.

[16] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic Patch Generation Learned from Human-Written Patches," in *ICSE*. IEEE, 2013, pp. 802–811.

[17] S. Wang, T. Liu, and L. Tan, "Automatically Learning Semantic Features for Defect Prediction," in *ICSE*. IEEE, 2016, pp. 297–308.

[18] Y. Li, S. Wang, and T. N. Nguyen, "DLFix: Context-Based Code Transformation Learning for Automated Program Repair," in *ICSE*. ACM, 2020, p. 602–614.

[19] 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 *ISSTA*. ACM, 2020, p. 101–114.

[20] 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," *TSE*, 2019.

[21] 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," *TOSEM*, vol. 28, no. 4, 2019.

[22] E. A. Santos, J. C. Campbell, A. Hindle, and J. N. Amaral, "Finding and Correcting Syntax Errors Using Recurrent Neural Networks," *PeerJ PrePrints*, vol. 5, p. e3123v1, 2017.

[23] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing Common C Language Errors by Deep Learning," in *AAAI*, 2017, pp. 1345–1351.

[24] A. Mesbah, A. Rice, E. Johnston, N. Glorioso, and E. Aftandilian, "DeepDelta: Learning to Repair Compilation Errors," in *ESEC/FSE*. ACM, 2019, pp. 925–936.

[25] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An Empirical Investigation into Learning Bug-Fixing Patches in the Wild via Neural Machine Translation," in *ASE*. ACM, 2018, pp. 832–837.

[26] F. Long and M. Rinard, "An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems," in *ICSE*. IEEE, 2016, p. 702–713.

[27] Y. Ding, B. Ray, P. Devanbu, and V. J. Hellendoorn, "Patching as Translation: the Data and the Metaphor," in *ASE*, 2020.

[28] S. Clinchant, K. W. Jung, and V. Nikoulina, "On the use of BERT for Neural Machine Translation," in *NGT*. ACL, 2019, pp. 108–117.

[29] I. Skorokhodov, A. Rykachevskiy, D. Emelyanenko, S. Slotin, and A. Ponkratov, "Semi-Supervised Neural Machine Translation with Language Models," in *LoResMT*. AMTA, 2018, pp. 37–44.

[30] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," in *NeurIPS*. MIT Press, 2014, pp. 3104–3112.

[31] F. Stahlberg, "Neural Machine Translation: A Review," *JAIR*, 2020.

[32] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin, "Convolutional Sequence to Sequence Learning," in *ICML*. JMLR.org, 2017, pp. 1243—-1252.

[33] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "TBar: Revisiting Template-Based Automated Program Repair," in *ISSTA*. ACM, 2019.

[34] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "Does Genetic Programming Work Well on Automated Program Repair?" in *ICCIS*. IEEE, 2013, pp. 1875–1878.

[35] S. Rico, H. Barry, and B. Alexandra, "Neural Machine Translation of Rare Words with Subword Units," *Annual Meeting of the ACL*, 2016.

[36] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention Is All You Need," in *NeurIPS*, 2017, pp. 5998–6008.

[37] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving Language Understanding by Generative Pre-Training," *OpenAI Blog*, 2018.

[38] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs," in *ISSTA*, 2014, pp. 437–440.

[39] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language Models are Unsupervised Multitask Learners," *OpenAI Blog*, vol. 1, no. 8, p. 9, 2019.

[40] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, "Better Test Cases for Better Automated Program Repair," in *FSE*, ser. ESEC/FSE 2017. ACM, 2017, p. 831–841. [Online]. Available: https://doi.org/10.1145/3106237.3106274

[41] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online Defect Prediction for Imbalanced Data," in *ICSE-SEIP*, 2015, pp. 99–108.

[42] G. Gousios and D. Spinellis, "GHTorrent: GitHub's data from a firehose," in *MSR*. IEEE, 2012, pp. 12–21.

[43] N. Smith, D. Van Bruggen, and F. Tomassetti, "Javaparser: Visited," 2019. [Online]. Available: https://github.com/javaparser/javaparser

[44] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," in *ICLR*, 2015.

[45] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, and J. Brew, "HuggingFace's Transformers: State-of-the-art Natural Language Processing," in *EMNLP*, 2020.

[46] M. Ott, S. Edunov, A. Baevski, A. Fan, S. Gross, N. Ng, D. Grangier, and M. Auli, "fairseq: A Fast, Extensible Toolkit for Sequence Modeling," in *NAACL*, 2019.

[47] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "QuixBugs: A Multi-Lingual Program Repair Benchmark Set Based on the Quixey Challenge," in *SPLASH*, 2017, p. 55–56.

[48] S. Saha, R. k. Saha, and M. r. Prasad, "Harnessing Evolution for Multi-Hunk Program Repair," in *ICSE*. IEEE, 2019, pp. 13–24.

[49] Z. Qi, F. Long, S. Achour, and M. Rinard, "An Analysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems," in *ISSTA*. ACM, 2015, pp. 24–36.

[50] Y. Yuan and W. Banzhaf, "ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming," *TSE*, 2018.

[51] M. Martinez and M. Monperrus, "Ultra-Large Repair Search Space with Automatically Mined Templates: the Cardumen Mode of Astor," in *ICSBSE*. Springer, 2018.

[52] T. Durieux and M. Monperrus, "DynaMoth: Dynamic Code Synthesis for Automatic Program Repair," in *AST*, 2016, pp. 85–91.

[53] 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 *ICST*. IEEE, 2019, pp. 102–113.

[54] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations," in *SANER*. IEEE, 2019, pp. 1–12.

[55] 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," *EMSE*, pp. 1–45, 2020.

[56] K. Liu, A. Koyuncu, K. Kim, D. Kim, and T. F. Bissyandé, "LSRepair: Live Search of Fix Ingredients for Automated Program Repair," in *APSEC*, 2018, pp. 658–662.

[57] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. D. A. Bissyande, D. Kim, P. Wu, J. Klein, X. Mao, and Y. Le Traon, "On the Efficiency of Test Suite based Program Repair: A Systematic Assessment of 16 Automated Repair Systems for Java Programs," in *ICSE*, 2020.

[58] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad, "Bugs.jar: A large-scale, diverse dataset of real-world java bugs," in *MSR*, 2018.

[59] F. Madeiral, S. Urli, M. Maia, and M. Monperrus, "Bears: An Extensible Java Bug Benchmark for Automatic Program Repair Studies," in *SANER*. IEEE, 2019, pp. 468–478.

[60] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, "Hoppity: Learning Graph Transformations to Detect and Fix Bugs in Programs," in *ICLR*, 2019.

[61] H. Tian, K. Liu, A. K. Kaboreé, A. Koyuncu, L. Li, J. Klein, and T. F. Bissyandé, "Evaluating Representation Learning of Code Changes for Predicting Patch Correctness in Program Repair," in *ASE*, 2020.

[62] M. Asad, K. K. Ganguly, and K. Sakib, "Impact Analysis of Syntactic and Semantic Similarities on Patch Prioritization in Automated Program Repair," in *ICSME*. IEEE, 2019, pp. 328–332.

[63] G. Sakkas, M. Endres, B. Cosman, W. Weimer, and R. Jhala, "Type Error Feedback via Analytic Program Repair," in *PLDI*, 2020, pp. 16–30.

[64] A. Ghanbari, S. Benton, and L. Zhang, "Practical Program Repair via Bytecode Mutation," in *ISSTA*, 2019.

[65] G. Kavi, E. C. Peter, C. Xinyun, and S. Dawn, "Synthesize, Execute and Debug: Learning to Repair for Neural Program Synthesis," in *NeurIPS*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., 2020.

[66] X. Gu, H. Zhang, and S. Kim, "Deep Code Search," in *ICSE*. ACM, 2018, pp. 933–944.

[67] M. Allamanis, H. Peng, and C. Sutton, "A Convolutional Attention Network for Extreme Summarization of Source Code," in *ICML*, 2016, pp. 2091–2100.

[68] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep Learning Code Fragments for Code Clone Detection," in *ASE*. ACM, 2016, pp. 87–98.

[69] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "CCLearner: A Deep Learning-Based Clone Detection Approach," in *ICSME*. IEEE, 2017, pp. 249–260.

[70] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software Defect Prediction via Convolutional Neural Network," in *QRS*. IEEE, 2017.

[71] J. Wang and C. Zhang, "Software reliability prediction using a deep learning model based on the RNN encoder–decoder," *RESS*, 2017.

[72] L. Fang, L. Ge, Z. Yunfei, and J. Zhi, "Multi-task Learning based Pre-trained Language Model for Code Completion," in *ASE*, 2020.

[73] V. Murali, L. Qi, S. Chaudhuri, and C. Jermaine, "Neural Sketch Learning for Conditional Program Generation," in *ICLR*, 2018.

[74] W. Ling, E. Grefenstette, K. M. Hermann, T. Kočiský, A. Senior, F. Wang, and P. Blunsom, "Latent Predictor Networks for Code Generation," *Annual Meeting of the ACL*, 2016.

[75] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward Deep Learning Software Repositories," in *MSR*. IEEE, 2015, pp. 334–345.

[76] V. J. Hellendoorn and P. Devanbu, "Are Deep Neural Networks the Best Choice for Modeling Source Code?" in *ESEC/FSE*. ACM, 2017.

[77] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A Survey of Machine Learning for Big Code and Naturalness," *CSUR*, vol. 51, no. 4, p. 81, 2018.

[78] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray, "CODIT: Code Editing with Tree-Based Neural Models," *TSE*, 2020.

[79] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning Distributed Representations of Code," *ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[80] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating Sequences from Structured Representations of Code," in *ICLR*, 2019.

[81] J. Henkel, S. K. Lahiri, B. Liblit, and T. Reps, "Code Vectors: Understanding Programs through Embedded Abstracted Symbolic Traces," in *ESEC/FSE*, 2018, pp. 163–174.

[82] W. Wang, Y. Zhang, Y. Sui, Y. Wan, Z. Zhao, J. Wu, P. Yu, and G. Xu, "Reinforcement-Learning-Guided Source Code Summarization via Hierarchical Attention," *TSE*, 2020.

[83] T. Hoang, H. J. Kang, D. Lo, and J. Lawall, "CC2Vec: Distributed Representations of Code Changes," in *ICSE*. ACM, 2020, p. 518–529.

[84] H. Hu, Q. Chen, and Z. Liu, "Code Generation from Supervised Code Embeddings," in *NeurIPS*. Springer, 2019, pp. 388–396.

[85] K. Wang and Z. Su, "Blended, Precise Semantic Program Embeddings," in *PLDI*. New York, NY, USA: ACM, 2020, p. 121–134.

[86] W. Wang, Gao and Wang, "Learning Semantic Program Embeddings with Graph Interval Neural Network," in *OOPSLA*. ACM, 2020.

[87] J. Keim, A. Kaplan, A. Koziolek, and M. Mirakhorli, "Does BERT Understand Code? – An Exploratory Study on the Detection of Architectural Tactics in Code," in *Software Architecture*. Springer, 2020.

[88] H. J. Kang, T. F. Bissyandé, and D. Lo, "Assessing the Generalizability of code2vec Token Embeddings," in *ASE*. IEEE, 2019, pp. 1–12.

[89] L. Jiang, H. Liu, and H. Jiang, "Machine Learning Based Recommendation of Method Names: How Far are We," in *ASE*. IEEE, 2019, pp. 602–614.