GrasP: Graph-to-Sequence Learning for Automated Program Repair

Ben Tang
School of Information Engineering
Yangzhou University
Yangzhou, China
853082540tb@gmail.com

Bin Li
School of Information Engineering
Yangzhou University
Yangzhou, China
lb@yzu.edu.cn

Lili Bo
School of Information Engineering
Yangzhou University
Yangzhou, China
lilibo@yzu.edu.cn

Xiaoxue Wu
School of Information Engineering
Yangzhou University
Yangzhou, China
xiaoxuewu@yzu.edu.cn

Sicong Cao
School of Information Engineering
Yangzhou University
Yangzhou, China
MX120190439@yzu.edu.cn

Xiaobing Sun
School of Information Engineering
Yangzhou University
Yangzhou, China
xbsun@yzu.edu.cn

2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS) | 978-1-6654-5813-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/QRS54544.2021.00091

***Abstract*—Many deep learning models, for example, neural machine translation (NMT) models, have been developed for Automated Program Repair (APR). Due to the advantages of NMT model's strong generalization ability and less manual intervention, NMT-based methods perform well in APR. However, previous NMT-based APR approaches regard a code snippet as a sequence of tokens, which ignores the inherent structure of code.**

In this paper, we propose a novel end-to-end approach with Graph-to-Sequence learning, *GrasP*, to generate patches for buggy methods. To better represent the buggy method, we use a graph based on abstract syntax tree (AST) to represent the source code. In order to learn complex graph representation, we introduce the attention-based encoder-decoder model for graph-to-sequence learning. The empirical evaluation on the popular benchmark Defects4J shows that *GrasP* can generate compilable patches for 75 bugs, of which 34 patches are correct.

***Index Terms*—Automated program repair, Graph-to-Sequence learning, abstract syntax tree**

I. Introduction

Software bugs are inevitable in the process of software development, and developers need to spend considerable effort to fix them [1]–[3]. To improve software reliability and reduce development cost, many automated program repair (APR) techniques are proposed to repair buggy programs automatically [4]–[7].

Since APR task can be treated as translating buggy code into correct code, the NMT model, a general model in Natural Language Processing (NLP) and mainly used for translation tasks, can be applied to APR. NMT-based approaches [8]–[10] automatically learn abstract fix patterns of programs from historical bug fixing data to capture relations between buggy statements and fixed statements. Moreover, they have good generalization because these models do not rely on programming languages but only related to the historical data used for training.

Although NMT-based APR approaches have shown their advantages over traditional approaches, code representations employed by some NMT-based approaches [8]–[10] still cannot retain the rich syntax and semantics information [11] since they tend to represent source code as sequences and apply the sequence-to-sequence model to generate patches. The sequence representation of these approaches ignores the implicit semantics in source code [11]. In addition, it is difficult to apply sequence-to-sequence models in APR since these models perform not well when the input sequence is too long. Some approaches try to limit the length of the input sequence [8] or the range of context [9] to optimize the sequence representation. Such attempts are still inadequate because they cannot ensure that the sequence is short enough while retaining enough information. For example, CoCoNut [10] separates a buggy line and its context, and then input them to different encoders to get the intermediate representation. These approaches [9], [10] based on optimized sequence representation work well for single-line bugs but may have trouble when it comes to bugs that caused by multiple buggy lines in one method because it can only treat these discontinuous buggy lines as multiple single-line bugs. This may lose relations between buggy lines.

Furthermore, as sequence-to-sequence models can only handle sequence, existing approaches convert the graph into a sequence and then use the sequence-to-sequence model for training [12]. This may ignore the structural information in the source code. Therefore, how to enable the NMT model to learn more complex information from code representations is a challenging problem yet to be solved.

To overcome the above discussed challenges, we propose a novel end-to-end NMT-based APR approach with Graph-to-Sequence learning, called *GrasP* (Graph-to-Sequence Learning for Automated Program Repair). First, we use a graph representation based on Abstract Syntax Tree (AST) to represent source code. Since a method retains as much syntax and semantic information as possible with a certain code length, we choose the *method-level* as the granularity of translation.

Then, we introduce a general attention-based neural network model to learn the graph representation of source code and generate patches. Unlike the traditional sequence-to-sequence model, Graph-to-Sequence model uses a graph encoder instead of the sequence encoder to generate intermediate representations so that it can help learn complex information from graphs and generate patches.

We conduct an empirical study on *GrasP*. First, we extracted thousands of buggy methods from real-world open-source projects and train *GrasP* to predict correct versions of these bugs. Second, we evaluated our approach in Defects4J [13], a dataset of reproducible bugs seen as a standard evaluation benchmark for automated program repair.

The main contributions of our work are as follows:

- We represented the buggy method as a graph to retain more structural information, and present a novel Graph-to-Sequence learning to deal with the input graph, overcoming the limitation of information missing.
- We evaluated our approach on Defects4J dataset and 1100 buggy methods extracted from real-world open-source projects, respectively. For the evaluation benchmark dataset Defects4J, *GrasP* generates compilable patches for 75 bugs, of them 34 are correct. For 1100 methods from open-source projects, *GrasP*'s prediction can achieve 16.1% accuracy on average.

II. Motivation

In this section, we use an example of real-world bug in Defects4J [13] to illustrate our motivation. Fig. 1 shows a buggy method `appendTo()` fixed in project *Closure*. Three discontinuous parts of buggy lines are found in `appendTo()`. For this method, the fix version modifies four lines. There are two main problems in this buggy method, i.e., conditional expression error and method invocation expression error. To fix this buggy method, three method invocation expressions (`out.append()`) should be modified and one conditional expression should be replaced.

```
-       out.append(lineValue);
+       out.append(String.valueOf(
+               originalPosition.getLineNumber()));
        out.append(",");
-       out.append(String.valueOf(
-           m.originalPosition.getCharacterIndex()));
+       out.append(String.valueOf(
+           originalPosition.getCharacterIndex()))
-       if (m.originalName != null) {
+       if (originalName != null) {
            out.append(",");
-       out.append(escapeString(m.originalName));
+       out.append(originalName);
        }
```

Fig. 1. A real patch for method `appendTo` in Defects4J

From this example, we can draw the following observations:

Observation 1. Not all buggy methods involve only one buggy line modification. In our motivating example in Fig.1, there are three discontinuous lines in this method that should be revised. In some approaches [9], [10] which aimed at fixing single-line bugs, these buggy lines are processed independently and each buggy line is input into the NMT model with its context (the entire method). As a result, when the first line is treated as a buggy line by the model, the other three buggy lines are learned as its context. In fact, context should be a collection of statements that do not have bugs but have an impact on buggy lines [9]. The NMT model may falsely consider these buggy lines as lines with no need to change. Therefore, a model that can only fix single-line bugs have trouble in handling bugs caused by multiple buggy lines in one method.

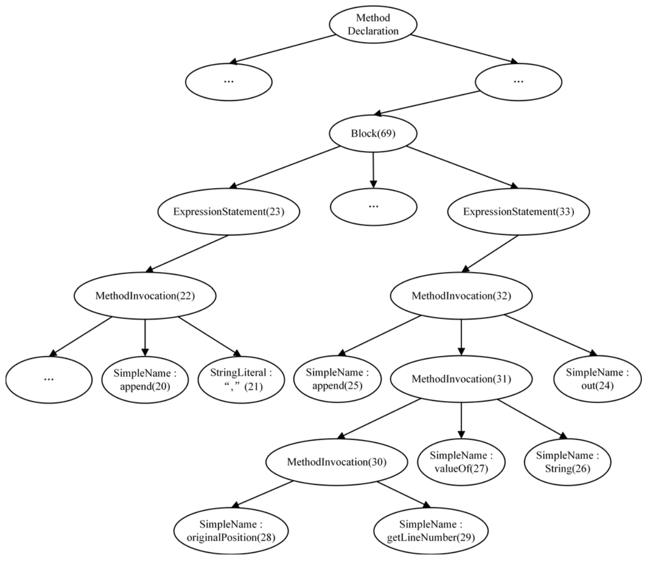


Fig. 2. Partial AST parsed by *Closure_148*

Observation 2. In buggy methods with multiple buggy lines, semantically related buggy lines may be discontinuous. In our motivating example in Fig.1, the buggy statement in `if` block is influenced by the conditional expression above, but there is a certain distance between two buggy lines. Traditional sequence-to-sequence model can not perform well in capturing the long-term dependence of the sequence. Therefore, semantically related statements should be connected when the method is transformed into suitable code representation forms.

Based on the above two observations, we propose our approach with the following key ideas. First, instead of constructing the representation around one buggy line, we use a graph based on Abstract Syntax Tree (AST) to represent the entire method to retain the complete semantics and syntax information. Compared with the sequence representation method, using AST to represent the code can make semantically related lines more closer. That is, graph representation can better learn the relations between possibly related lines. Fig. 2 shows a part of AST parsed by the method `appendTo()` in Fig. 1. As shown in Fig. 2, there are only a few hops between the two buggy method invocation statements. However, some AST nodes (e.g., *Block, MethodInvocation*) are abstract and

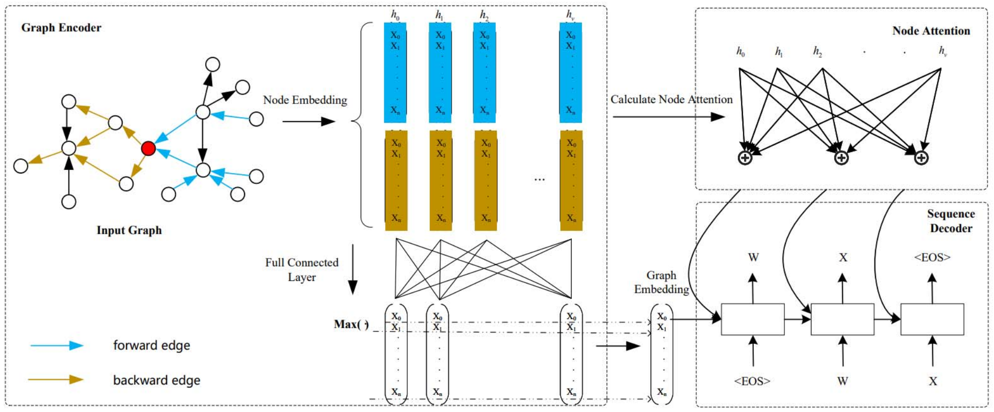


Fig. 3. Framework of Graph2Seq model

represent properties of the subtrees. They do not belong to the original code. We take some measures to deal with these virtual nodes, such as *ExpressionStatement*. Second, in order to retain structural information of AST, we use a graphic architecture design to deal with the input graph. Traditional approaches [14] directly convert complex structured graph data into sequences. The conversion process may cause the problem of structure information loss. We introduce Graph-to-Sequence learning by replacing the sequence encoder with a graph encoder.

III. Background and Related work

Graph-to-Sequence learning. Graph-to-Sequence learning is proposed to deal with input represented as graphs and learn the conversion of graphs to sequences. The Graph2Seq [12] model is one of the neural network models for graph-to-sequence learning. Fig. 3 shows the details of the Graph2Seq model. The Graph2Seq model includes a graph encoder, a sequence decoder and node attention mechanisms. The Graph Encoder first generates the embedding representation of the node, and then constructs the embedding of the entire graph based on the learned node embeddings. Then, the sequence decoder takes both graph embedding and node embedding as input, and focuses attention on the node embedding while generating the sequence. Graph2Seq replaces the sequence encoder with a graph encoder. The graph encoder aims to learn node embeddings, then reconstitutes them into a graph embedding and input the graph embedding into the decoder.

Deep Learning for APR. Tufano et al. [8] first applied Neural Machine Translation (NMT) technology in natural language processing(NLP) to predict and repair bugs. They regarded APR as a translation task to translate buggy programs into fixed programs. SequenceR [9] was another NMT-based approach proposed to repair single-line bugs written in Java Programming Language. It introduced copy mechanism [15], which is simply described as the ability to directly copy the input data of the model to the output. CoCoNut [10] adopted a novel code representation to represent buggy source code and its surrounding context separately. In this architecture, two separate decoders were used to process buggy lines and context lines respectively so that long term relations between tokens can be extracted. CURE [16] pre-trained a programming language model to learn developer-like source code before the APR task and then used a new search strategy to find more correct fixes. In addition to representing source code as a sequence, DLFix [17] treated source code as a parse tree. DLFix is a two-layer model that the first layer learns the context of the buggy line and its result is an extra weighting input for the second layer designed for this buggy line. The most related NMT-based work to *GrasP* is SequenceR [9]. The main differences are that *GrasP* represents methods as graphs and takes Graph-to-Sequence learning instead of sequence-to-sequence learning.

IV. Our Approach

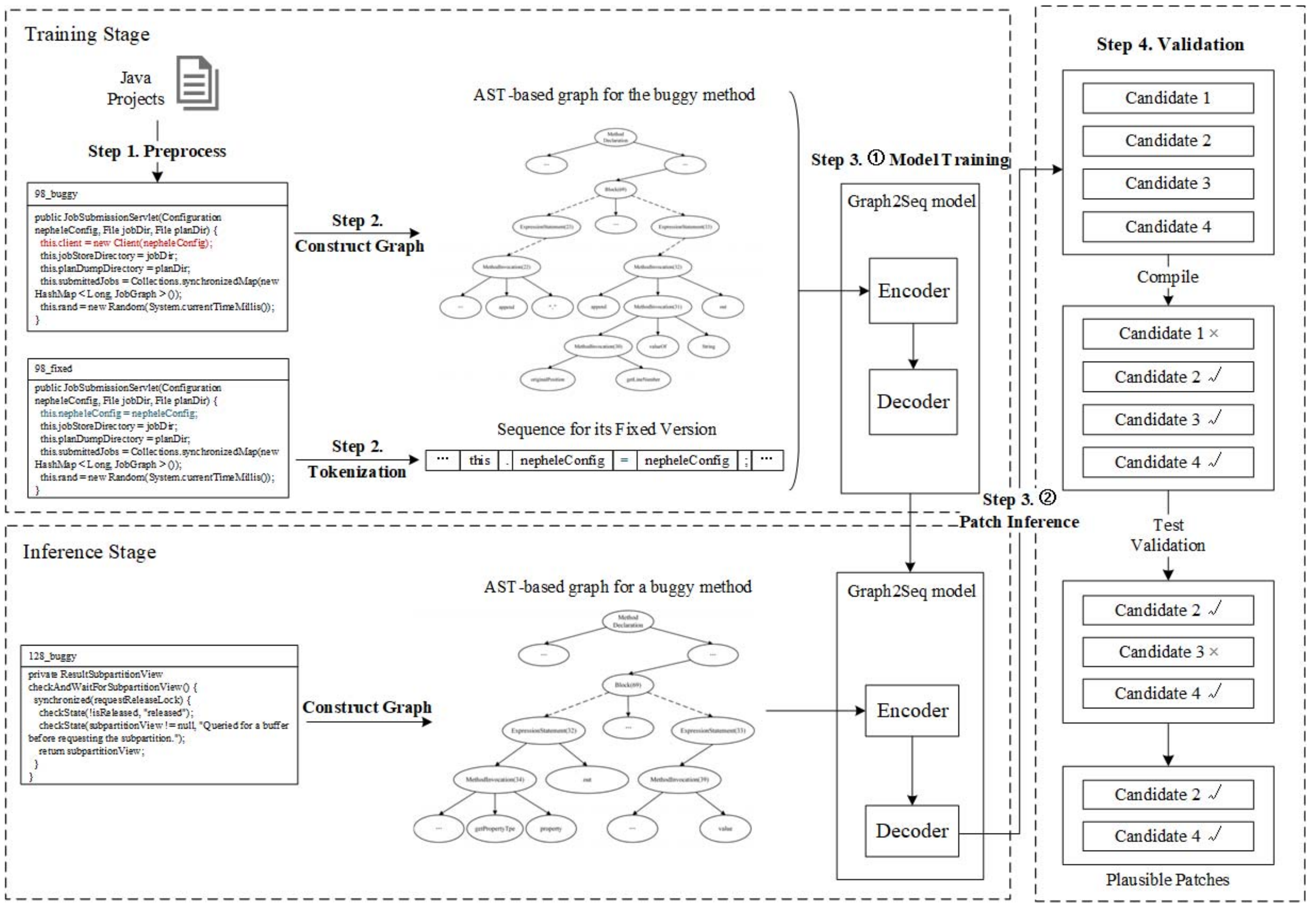
A. Overview

Fig. 4 shows the overview of our end-to-end program program repair approach, *GrasP*. *GrasP* consists of four steps: 1) preprocessing, 2) graph construction and method tokenization, 3) model training and patch inference, and 4) validation.

In the first step, we search for all changed methods and extract buggy-fixed method pairs to construct Bug-Fixed Pairs (BFP). In the second step, for each BFP, we construct an AST-based graph to represent the buggy method and tokenize its corresponding fixed version. After that, we transfer the processed BFP to the Graph2Seq model for training and then use the well-trained model to generate candidate patches for new buggy methods. Finally, we validate the correctness of candidate patches and select plausible patches for the buggy method.

B. Preprocessing

The goal of preprocessing is to extract pairs of buggy methods and their corresponding fixed versions to build the dataset. We built our Java dataset from open-source communities and published work [17]. Different from approaches that represent code as sequences, our approach has stricter requirements on the format of the data. Since we need to parse source code



```
98_buggy
public JobSubmissionServlet(Configuration
nepheleConfig, File jobDir, File planDir) {
  this.client = new Client(nepheleConfig);
  this.jobStoreDirectory = jobDir;
  this.planDumpDirectory = planDir;
  this.submittedJobs = Collections.synchronizedMap(new
HashMap < Long, JobGraph > ());
  this.rand = new Random(System.currentTimeMillis());
}
```

```
98_fixed
public JobSubmissionServlet(Configuration
nepheleConfig, File jobDir, File planDir) {
  this.nepheleConfig = nepheleConfig;
  this.jobStoreDirectory = jobDir;
  this.planDumpDirectory = planDir;
  this.submittedJobs = Collections.synchronizedMap(new
HashMap < Long, JobGraph > ());
  this.rand = new Random(System.currentTimeMillis());
}
```

```
128_buggy
private ResultSubpartitionView
checkAndWaitForSubpartitionView() {
  synchronized(requestReleaseLock) {
    checkState(!isReleased, "released");
    checkState(subpartitionView != null, "Queried for a buffer
before requesting the subpartition.");
    return subpartitionView;
  }
}
```

Fig. 4. Overview of *GrasP*

into a complete graph, we should ensure the integrity and compilability of the method (compilation unit that meets the requirements of the compiler). Some measures are used to filter the dataset. First, we removed methods that are duplicate and do not conform to the grammatical specification. Second, we removed extra long methods with more than 400 tokens because it is hard to generate correct patches for long methods. 80 percents of the methods in our dataset are less than 400 in length. Third, we removed all the comments and replaced the spaces and newline marks between tokens with a single space.

After preprocessing, we got 12,000 buggy-fixed method pairs for training. These pairs conform to grammatical specifications and can be transformed into trees with *Gumtree* [18].

C. Graph Construction and Tokenization

In this step, we construct an AST-based graph for buggy methods and tokenize the corresponding fixed method.

The abstract syntax tree (AST) is a tree representation of the abstract syntax structure of the source code. We use *Gumtree* to generate AST for each method. *Gumtree* is an open source framework which can handle source code as AST. To use *Gumtree*, we add a class declaration for each method to pass compilation and delete the declaration node after constructing AST. Since our task is a text generation task, each node should be represented by text. For syntax nodes that do not exist in source code, we use TypeLabel (an attribute of node in AST generated by *Gumtree*, e.g., ExpressionStatement). For other nodes, we directly use the text in source code to represent them.

Then, we make a few modifications to the generated AST. In our generation task, some virtual nodes (e.g., *ExpressionStatement*) are not involved in the translation process. As shown in Fig. 3, each *ExpressionStatement* node has only one child. *ExpressionStatement* has no practical meaning, and it shows that the subtree is an expression statement. These nodes are redundant nodes for our task because our model does not generate these virtual tokens during the inference process. We reconstruct the directivity of some edges and remove virtual nodes to make the connection between actual nodes closer. As an example in Fig. 3, we removed *ExpressionStatement* nodes and connected *Block* node with *MethodInvocation* nodes. For similar nodes that do not affect the AST structure, we will take the same action referring to the control flow rules. We mainly process *Conditions and If Statements*, *For Loop and While Loop*, and *Switch*.

After constructing modified AST graphs for buggy methods, we separate its fixed version into tokens according to grammatical rules. Then, the constructed graph and the tokenized sequence are inputted into the Graph2Seq model for training.

D. Model Training and Patch Inference

In this step, we need to input the graph and corresponding sequence generated in the previous step and then generate patches based on the trained Graph-to-Sequence model. In order to deal with the graph representation, we chose to use the open source model Graph2Seq [12].

In the training stage, the Graph2Seq model can learn the best conversion from buggy graphs to fixed sequences by continuously adjusting the combination of weights. After continuous adjustments, the Graph2Seq model can predict sequences based on the best combination of weights.

Graph Encoder. The Graph Encoder is a graph-based neural network to convert the input graph to a sequence of vectors. For the input AST-based graph, the Graph Encoder generates node embedding for each node based on the text attribute, and then constructs graph embeddings based on the learned node embeddings. These vectors are later used to generate patches in the form of sequence.

Sequence Decoder. The Sequence Decoder is to decode the target sequence from vectors generated by the Graph Encoder. The Decoder is based on Recurrent Neural Network (RNN) and the main difference between this decoder and the traditional decoder is the *node attention mechanism*. The node attention mechanism is an extension of attention mechanism in sequence-to-sequence model. It calculates the attention of each node in graph to the token at the current position to obtain the attention score. In this model, the manifestation of attention is a context vector computed as a weighted sum of related node representations. The context vector can indicate the relationship between the current element and other elements and participate in generation process along with graph embeddings.

Inference. For time step i, the decoder predicts token y_i according to the previously predicted tokens $y_{<i} = y_1, \dots, y_{i-1}$, RNN hidden state s_i and the context vector c_i. For each token position, there is a list of tokens ranked by the likelihood of being the next token. The beam search algorithm is used to select which token should be the next. The decoder will predict the next token until the $\langle EOS \rangle$ tag is read.

Beam search. Inspired by existing works [10], [19], [20], we use beam search instead of greedy search. Beam search [21] is an optimized algorithm for greedy search. For each time step, Beam search selects k candidates with largest conditional probability among all combinations as the candidate output sequence based on the output sequence of the previous step, where k means beam size. Compared to greedy search, beam search expands the search space. Different from SequenceR [9] and CoCoNut [10], we chose 20 as beam size to generate patches. We observed that plausible patches are usually at the top of the list. In our test dataset, there is no plausible patches located after 20. It may be caused by the long sequence we predicted. When the prediction sequence is long, the patches at the bottom of the list are rarely compilable. Beam search greatly improves our prediction accuracy. We implement Graph2Seq model with *Beam Search* algorithm on the basis of public code[1]

E. Patch Validation

Compilation. Each method is only a small part of the project and our model has no access to the entire project. So we should replace the buggy method with predicted sequence and then compile the project. This step aims to filter out patches that can not be compiled.

Validation. In addition to be compilable, plausible patches must pass test suites. We recorded the number of test cases our compilable patches passed. By comparing with test copy version fails, we filtered out patches passing fewer test suites. *GrasP* is performed on the basis that all buggy methods are perfectly positioned(the input). We built our validation framework based on the Defects4J framework. We first checked out all bug versions in Defects4J. Then, we extracted all changed methods between buggy and fixed files. We labeled these changed methods in source files. We converted these buggy methods into graphs and input them into the Graph2Seq model to generate patches. When patches were generated, we would replace labeled methods with their predicted version and validate them. Since our patches are method-level, our validation criteria are different from previous works [10]. The validation criteria existing works meet are as follows. First, the fixed version must also pass the test cases that the buggy version can pass. Second, the fixed version must pass at least one test case that the buggy version failed. We found that there are bugs with buggy lines outside the method. For some bugs in Defects4J, if buggy lines outside methods are not fixed, the correct patch for method in the same file will not pass more test cases. Therefore, for patches that can not pass more test cases, we will manually check their effectiveness. After these steps, we consider passed patches as *plausible patches*. Among plausible patches, those that have been manually checked and are deemed to be consistent with the semantics of the manual patches will be regarded as *correct* patches.

V. Experiments

A. Research Questions

To evaluate the effectiveness of our approach in bug fixing and the rationality of the components, we aim to investigate the following three research questions:

***RQ1*: How well does our approach perform in comparison with the state-of-the-art end-to-end APR approach?**

This question aims to study the performance difference between *GrasP* and other state-of-the-art APR techniques. We evaluated the effectiveness of our approach on a popular baseline dataset Defects4J [13]. We checked out the buggy and fixed versions of each bug in Defects4J, and extracted all the change methods. We used a graph based on AST to represent

[1]https://github.com/IBM/Graph2Seq

each method in Defects4J, and then transformed the graph into our well-trained model. We set beam size 20 to search for possible patches. In order to evaluate our approach objectively and fairly, we selected the state-of-the-art APR approach [9] for comparison.

***RQ2*:What kind of bugs can be fixed by our approach compared with state-of-the-art NMT-based approaches?**

Regarding the question of whether it can learn effective unique patterns, we will answer this question through several specific case studies. *GrasP* is an automatic end-to-end program repair approach that can automatically learn fix patterns from fixing history and use these patterns to generate patches. Since these patterns learned by our model are abstract and hard to explain, we used fix patterns classified by TBar [20] to explain our results.

***RQ3*: How do various components affect the overall performance of our approach?**

This question is designed to answer whether the components of our approach (mainly the node attention mechanism and vocabulary) are effective for APR task. In order to migrate Graph-to-Sequence learning to APR task, we consider some localization operations including applying node attention mechanism, limiting the vocabulary. Since the Graph2Seq model is designed to learn the conversion of nodes in the graph, we used a node attention mechanism to learn the alignments between nodes and sequence elements better. To reduce the impact of the vocabulary, we limited the vocabulary by word frequency.

B. Experimental Setup

Dataset. Due to the lack of a standard training dataset for Java, we constructed our training dataset from open-source communities and public dataset [17]. We mainly extracted BFPs from public dataset (containing more than 50,000 buggy-fixed pairs and complete context) to eliminate differences in the dataset. We randomly divided the dataset into training set, validation set and test set at the ratio of 8:1:1 according to the convention for each experiment. In ablation studies, we took the average of results of five experiments as the final result.

To evaluate our approach, we chose Defects4J [13] as our test dataset. Defects4J is a Java dataset of reproducible bugs seen as a standard evaluation benchmark [9], [22]–[25] for automated program repair. It has a collection of reproducible Java bugs and a supporting infrastructure that can help software engineering research. The latest version of Defects4J contains 835 Java bugs.

Baseline. To evaluate the performance of *GrasP*, we chose SequenceR [9], the most relevant work to our work, as the baseline. SequenceR is a novel end-to-end APR approach based on sequence-to-sequence learning. The main differences between SequenceR and *GrasP* are that *GrasP* adapts a graph representation and applies Graph-to-Sequence learning to learn abstract fix patterns. In addition to the approach we selected, there are many works that produce good results, but we did not compare our approach with them in that *GrasP* is a totally end2end solution for APR and based on the perfect positioning. This means that *GrasP* knows the location of each buggy method and takes methods as input instead of files or projects. In the industrial scenario, before generating a patch, it is necessary to find the position of the bug by applying bug localization technique. At present, function-level and file-level positioning are the mainstream of bug localization technologies. This is also one of the reasons that we choose to represent the method as a graph and generate patches for the entire method. In order to eliminate the impact of bug localization technology, we only compare *GrasP* with those approaches and tools that assume perfect bug localization. On the other side, *GrasP* is an end2end solution with single translation model. Some NMT-based APR approaches [17], [26] use multiple deep learning models instead of single model to extract fix patterns from non-sequential representations. Therefore, we selected the end-to-end APR approach based on perfect bug localization assumption [9] as our baseline to understand the performance of *GrasP*.

Similar to previous work [10], we extracted experimental results from original paper [9] for comparison because of the particularity of our dataset (a dataset composed of graphs constructed by methods). It is plausible because the choice of datasets and how to extract and represent data are a key component of a technique [10].

C. Experiment Implementation

We trained and evaluated *GrasP* on the server with Nvidia Graphics Tesla T4, windows 10 and 64g RAM. *GrasP* is implemented by python and the implementation of Graph-to-sequence model is based on the previous work [12]. The conversion from methods to graphs mainly relies on Gumtree [18] framework.

There is a big difference between programming languages and natural languages, that is, programming languages have many new words (variable names defined by developers at will). To deal with this problem, we counted the frequency of tokens in the dataset, and limited the vocabulary by word frequency. Also, we retained some common tokens (e.g., Java key words) based on experience.

As for the hyper-parameters, we randomly searched for parameters within a given range. Based on experience and suggestions given by the authors of the Graph2Seq model [12], we limited the search space: learning rate = $\{10^{-3}, 10^{-4}, 10^{-5}\}$, epoch = $\{100, 200, 300\}$, batch size = $\{16, 32\}$, dropout = $\{0.3, 0.4, 0.5\}$. The hyper-parameters we finally adopted are shown in the Table I.

TABLE I
HYPERPARAMETERS OF OUR MODEL

Embedding size	254
Learning rate	10^{-3}
Dropout	0.3
Epoch	200
Batch size	32

TABLE II
DETAILS OF CORRECT PATCHES GENERATED BY GRASP

Project Name	Number of Bugs	Number of Correct Patches
Chart	26	1
Closure	174	8
Lang	64	2
Math	106	3
Mockito	38	0
Time	26	2
Cli	39	4
Codec	18	1
Collection	4	0
Compress	47	0
Csv	16	0
Gson	18	0
JacksonCore	26	2
JacksonDatabind	112	2
JacksonXml	6	0
Jsoup	93	8
JxPath	22	1
Total	835	34

VI. EXPERIMENTAL RESULTS

A. Answer to RQ1: Evaluation on Defects4J

To evaluate our approach, we applied *GrasP* to fix real bugs in Defects4J. Table II shows the details of correct patches generated by *GrasP*. The first column represents the name of the projects in Defects4J. The second column and the third column list the number of bugs and the correct patches for each project, respectively. After validation, we generated compilable patches for 75 bugs in Defects4J. According to the evaluation criteria mentioned in Section IV-E, patches provided for 34 bugs are correct.

```
FunctionType fnType = type.toMaybeFunctionType();
-       if (fnType != null) {
+       if (fnType != null && fnType.hasInstanceType
    ()){visitParameterList(t, n, fnType);}
```

Fig. 5. Patch for *Closure_125*

Fig. 5 shows a patch for Defects4J bug Closure_125 generated by *GrasP*. *GrasP* can fix common bugs such as misusing of variable names and conditional statement errors. According to our observations, there are several characteristics with bugs that our model cannot generate patches for. 140 bugs in Defects4J involve more than one method changed in the fixed version. 74 of them have more than one changed file. *GrasP* has trouble in dealing with these 140 bugs. Also, for most long methods (with more than 300 tokens), *GrasP* cannot generate compilable patches. In addition, there are some changed variable names never occuring in our code corpus and the buggy method. These variables tend to appear in the file where buggy method is located but out of the buggy method. Because of the granularity of the input graph, *GrasP* cannot perceive the relation between buggy methods or connection between buggy methods and its context out of the methods.

To comprehensively evaluate the performance of an APR approach, it is necessary to consider the learning ability of the model, generalization ability, accuracy and other factors. Among those NMT-based approaches mentioned before, SequenceR [9] is most relevant to our approach and chosen as our baseline. It should be noted that their results were produced on the old version of Defects4J [13]. Defects4J added several new projects (e.g., JacksonDatabind) after updating. The old version has only 6 projects (Chart, Closure, Lang, Math, Time, Mockito) containing 393 bugs, so we selected our results on these 6 projects for comparison.

The results show that *GrasP* achieves better performance than SequenceR on Defects4J. SequenceR generates plausible patches for 19 single-line bugs in Defects4J, of which 14 are correct (the results are extracted from its original paper). Compared with SequenceR on these 6 projects, *GrasP* generated plausible patches for 30 bugs and 16 patches are correct. These correct patches are manually checked and semantically equivalent to human-written patch. SequenceR only focuses on single-line bugs. For too long methods or methods with multiple lines of buggy code, SequenceR can only choose to copy most of tokens in the input sequence because of the limitation of the sequence model. *GrasP* can generate more correct patches than SequenceR in that *GrasP* cannot only deal with single-line bugs but also bugs with multiple discontinuous lines in single methods. *GrasP* is able to learn more fix patterns involved more complex grammatical structural changes. By representing the source code as optimized AST-based graph, these syntactic and semantic related tokens are more closely connected. This means that fix patterns that *GrasP* can learn is not limited to one single line but in the granularity of the method.

As a conclusion, *GrasP* provides two more correct patches and 11 more plausible patches than the baseline approach. Moreover, *GrasP* can learn abstract fix patterns at the granularity of the method-level.

B. Answer to RQ2: Qualitative Case Studies

Different from SequenceR, *GrasP* is able to learn changes in the method structure from graphs. Next, we show several fix patterns learned by our model. These patterns are not captured by SequenceR [9]. Fix patterns extracted by our model are abstract and reflected by combination of parameters. Therefore, we describe the fix patterns learned by *GrasP* through some generated correct patches.

Move Statement and remove redundant statement. As shown in Fig. 6, the statement `token.add(token)` moves out of the block. At the same time, the redundant `token.add(token)` is deleted. These three discontinuous statements are semantically related and need to be modified simultaneously.

```
            {
                currentOption = options.getOption(token
                    );
-               tokens.add(token);
            }
            else if (stopAtNonOption)
            {
                eatTheRest = true;
-               tokens.add(token);
            }
+           tokens.add(token);
```

Fig. 6. Patch for *Cli_19*

Remove entire buggy method. In Fig. 7, *GrasP* removes the overiding method `getLeastSupertype`. In our validation process, we observed that there are several buggy methods need to be deleted (e.g., buggy override method). We observe that *GrasP* probabilistically generates label $\langle EOS \rangle$ when encountered `@override`. Deleting the entire method is a plausible fixing pattern in TBar [20] (namely FP15.2).

```
-  @Override
-  public JSType getLeastSupertype(JSType that) {
-    if (!that.isRecordType()) {
-      return super.getLeastSupertype(that);
-    }
-    RecordTypeBuilder builder = new
   RecordTypeBuilder(registry);
-    for (String property : properties.keySet()) {
-      if (that.toMaybeRecordType().hasProperty(
   property) &&
-          that.toMaybeRecordType().getPropertyType(
   property).isEquivalentTo(
-              getPropertyType(property))) {
-        builder.addProperty(property,
   getPropertyType(property),
-            getPropertyNode(property));
-      }
-    }
-    return builder.build();
-  }
```

Fig. 7. Patch for *Closure_46*

Modify and move statement. Fig. 8 shows that method invocation expression `scope=traverse(constructor ,scope)` is mutated and then moved to another position. This pattern involves simultaneous modification of multiple lines.

```
+     scope = traverseChildren(n, scope);

      Node constructor = n.getFirstChild();
-     scope = traverse(constructor, scope);
      JSType constructorType = constructor.getJSType
          ();
```

Fig. 8. Patch for *Closure_25*

By studying these three fix patterns learned by *GrasP*, we found these three patterns are involved in overall changes of multiple discontinuous lines. These fix patterns modified the structure of the entire method. SequenceR [9] cannot cope with these complex changes. SequenceR is mainly designed to deal with single-line (continuous lines) bugs. The representation it adopted can only learn the changes that occur within single continuous line, and lack cognition of the overall structure. Similar to the granularity we selected, Tufano et al. [8] presented an approach to represent the buggy method as a abstract sequence. However, Tufano et al. [8] limited the length of input sequence within 50 tokens. In fact, many real buggy methods are above 50 in length. The longest compilable patch *GrasP* generated has 292 tokens.

GrasP is able to capture complex changes in code structure. Since we adopted an AST-based graph representation, this representation shortens the distance between related tokens and can model the structural information of the method. Therefore, our model can learn these three fix patterns. However, our approach seems to have no advantage in fixing single-line bugs compared with other approaches [9], [10]. In contrast, our model performs better when it comes to complex structural transformations. By removing some virtual nodes and strengthening the connections between related nodes, our model can also learn some patterns such as **Mutate Variable**.

To summarize, *GrasP* can learn fix patterns that were not learned by SequenceR [9]. These fix patterns involved changes in the entire method structure. *GrasP* learned these patterns because our graph representation can model the structural information of the entire buggy method and the model with Graph-to-Sequence learning can learn patterns from the complex graph representation.

C. Answer to RQ3: Ablation Research

In this section, we analyze the importance of each component of our approach.

TABLE III
THE EFFECT OF CODE REPRESENTATION

Code Representation	Correct Rate	Correct Number
AST-based Graph	16.1	177
AST-based Sequence	14.5	160
Sequence	14.9	164

Code Representation. To evaluate the code semantic expression ability of the code graph representation and sequence representation, we built a new test data set and deployed a comparative experiment. The test set contains 1100 buggy methods from the same projects as buggy methods in training set. In our approach, we took an AST-based graph to represent the buggy method and used a Graph-to-Sequence model to learn abstract fix patterns. Previous NMT-based APR approaches used sequence [9], [10] or sequence generated by traversing AST as the input of the NMT model. Therefore, we considered the following three representations for comparison, namely the AST-based graph representation, the AST-based sequence representation, and the sequence representation of the code. In order to form graphs to adapt to the input of the graph encoder, we concatenated the tokens in the latter two sequence representations.

Our experiment was carried out with all default settings unchanged except for the form of code representation. The experimental results are shown in Table III. The best result is achieved by using our AST-based graph to represent the code. The model using AST-based graph correctly predicted 177 of 1100(16.1%) methods. This is because the AST-based graph representation we used can better retain the AST syntax structure and target the NMT model information. In addition, actions are taken to optimize the graph representation to shorten the distance between related tokens. The sequence generated by AST traversal contains tokens that represent syntactic structures and do not exist in the source code. Although they can grasp some of the syntactic information, they will also increase the distance between nodes of adjacent subtrees in the AST. On the other side, the source code sequence is poor in characterizing the relationship between tokens that have long-distance dependencies. For our approach, it is a better way to use graphs to represent code for preserving more of the source code syntax and semantics .

Node Attention Mechanism. Then, we deployed a comparative experiment on the same test set to evaluate the learning ability of Graph2Seq model before and after applying the attention mechanism. The node attention mechanism is an extension of attention mechanism in sequence-to-sequence model. It can calculate the attention of each node in graph to the token at the current position to obtain the attention score and then help generate the target token. The comparative experiment is deployed under the condition of consistent vocabulary and beam size. The results indicate that the node attention mechanism greatly improved the learning ability of Graph2Seq model. The model without attention mechanism correctly predicted 119 of 1100 (10.8%) methods while the model with node attention mechanism predicted 177 of 1100 (16.1%). With node attention mechanism, model can put more attention to the important node. At the process of generating, for each position, the decoder can take some related nodes into consideration to give more accurate predictions.

TABLE IV
THE EFFECT OF NODE ATTENTION MECHANISM

Model Description	Correct Rate	Correct Number
node attention=False	10.8	119
node attention=True	16.1	177

Vocabulary. We tested performances of the model without constraining the vocabulary. Although the model performs well on our own test set, it is unable to generate correct patches for bugs in Defects4J except five patches (of them four patches delete the entire methods, one patch mutates a return statement by replacing True with False). The model with only keywords in the vocabulary achieves similar results. We studied the impact of the vocabulary on our results. Due to the differences between programming languages and natural languages, we must limit the vocabulary while applying NMT model to dealing with programming languages. Developers often introduce new words (e.g., custom variable name) in code, which makes the vocabulary of code much larger than the natural language vocabulary.

```
public final class <unk> {
        if (count > <unk>)
            return new String(<unk>, start , count);
        if (count < 1)
            return <unk>;
      }
```

Fig. 9. Abstract representation of a Java method(retain common tokens)

```
public final class <unk> {
        if (<unk> > <unk>)
            return new <unk>(<unk>, <unk>, <unk>);
        if (<unk> < <unk>)
            return <unk>;
      }
```

Fig. 10. Abstract representation of a Java method

Fig. 9 shows an abstract representation of a Java method with $\langle unk \rangle$. The label $\langle unk \rangle$ represents the new word absent in vocabulary. In this example, some variable names that rarely appear in the code corpus are replaced with $\langle unk \rangle$. The vocabulary determines the degree of abstraction and directly affects the generalization performance of the model. If we did not limit vocabulary or made too many limitations on the vocabulary (Fig. 10), the model would have trouble in learning useful fix patterns with generalization ability. Refer to the practice of SequenceR [9], we retained both Java key words and common words (filtered by word frequency in our code corpus) to limit the vocabulary. The model with limited vocabulary (about 12,000 tokens for Java) can generate 34 correct patches for bugs in Defects4J [13].

VII. Threats to Validity

In this section, we discuss the threats to our study, including internal threats and external threats.

Internal threats. Internal threats are mainly from the experimental results. Some approaches mentioned in our paper did not disclose the source code, so we could not re-implement them and had to collect their published results on Defects4J. In addition, due to the randomness of deep learning models, We conduct multiple experiments through random division of the dataset and parameter adjustment to make the experimental results converge to a stable value.

External threats. One of the main external threats to our study is the reliability of the data sources. Since there is no public benchmark training dataset, NMT-based approaches tend to obtain data from some open source communities. The quality of the training dataset may cause differences in performance. In this case, we decide to use dataset used by other works to reduce the difference caused by the data set.

However, SequenceR [9] dataset and CoCoNut [10] dataset did not meet our needs. In the end, we chose to use DLfix [17] dataset. In addition, in spite of the dataset we constructed only contains Java projects, our approach learns fix patterns from historical commits and generates patches by an end-to-end model that the artifacts are not tied to Java. So our approach can be applied to other program languages.

VIII. Conclusion and Future work

In this paper, we proposed a novel end-to-end APR approach to generate patches for buggy methods. We introduce the graph representation of the buggy method instead of sequence representation and use Graph-to-Sequence model to capture rich information of the graph. Our evaluation results demonstrate that *GrasP* can learn more fix patterns, and generate correct patches for more buggy methods.

In the future, we plan to explore more accurate code representations, which can retain more syntax and semantics information for the generation task. In addition, we will perform the evaluation on more projects and programming languages to show the generalibility of *GrasP*.

Acknowledgments

This work was supported partially by Natural Science Foundation of China under Grant No. 61972335, No. 61872312, and No. 62002309, the Yangzhou city-Yangzhou University Science and Technology Cooperation Fund Project, the Six Talent Peaks Project in Jiangsu Province (No. RJFW-053), the Jiangsu "333" Project, the Open Funds of State Key Laboratory for Novel Software Technology of Nanjing University (No. KFKT2020B15, No. KFKT2020B16), the Key Laboratory of Safety-Critical Software Ministry of Industry and Information Technology (No. NJ2020022), the Natural Science Research Project of Universities in Jiangsu Province (No. 20KJB520024), and Yangzhou University Top-level Talents Support Program (2019).

References

[1] X. Sun, T. Zhou, R. Wang, Y. Duan, L. Bo, and J. Chang, "Experience report: investigating bug fixes in machine learning frameworks/libraries," *Frontiers Comput. Sci.*, vol. 15, no. 6, p. 156212, 2021.

[2] J. Lu, X. Sun, B. Li, L. Bo, and T. Zhang, "BEAT: considering question types for bug question answering via templates," *Knowl. Based Syst.*, vol. 225, p. 107098, 2021.

[3] S. Cao, X. Sun, L. Bo, Y. Wei, and B. Li, "*BGNN4VD*: Constructing bidirectional graph neural-network for vulnerability detection," *Inf. Softw. Technol.*, vol. 136, p. 106576, 2021.

[4] X. Sun, X. Peng, B. Li, B. Li, and W. Wen, "IPSETFUL: an iterative process of selecting test cases for effective fault localization by exploring concept lattice of program spectra," *Frontiers Comput. Sci.*, vol. 10, no. 5, pp. 812–831, 2016.

[5] Z. Ni, B. Li, X. Sun, T. Chen, B. Tang, and X. Shi, "Analyzing bug fix for automatic bug cause classification," *J. Syst. Softw.*, vol. 163, p. 110538, 2020.

[6] C. L. Goues, M. Pradel, A. Roychoudhury, and S. Chandra, "Automatic program repair," *IEEE Softw.*, vol. 38, no. 4, pp. 22–27, 2021.

[7] R. S. Shariffdeen, Y. Noller, L. Grunske, and A. Roychoudhury, "Concolic program repair," in *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211*, 2021, pp. 390–405.

[8] 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 *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 832–837.

[9] Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, 2019.

[10] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: Combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 101–114.

[11] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *arXiv preprint arXiv:1711.00740*, 2017.

[12] K. Xu, L. Wu, Z. Wang, Y. Feng, M. Witbrock, and V. Sheinin, "Graph2seq: Graph to sequence learning with attention-based neural networks," *arXiv preprint arXiv:1804.00823*, 2018.

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

[14] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016, pp. 2073–2083.

[15] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," *Advances in Neural Information Processing Systems*, vol. 28, pp. 2692–2700, 2015.

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

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

[18] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, 2014, pp. 313–324. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642982

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

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

[21] C. M. Wilt, J. T. Thayer, and W. Ruml, "A comparison of greedy search algorithms," in *third annual symposium on combinatorial search*, 2010.

[22] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2016.

[23] Z. Qi, F. Long, S. Achour, and M. Rinard, "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*, 2015, pp. 24–36.

[24] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Ieee transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2011.

[25] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *Empirical Software Engineering*, pp. 1–45, 2020.

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