A Syntax-Guided Edit Decoder for Neural Program Repair

Anonymous Author(s)

ABSTRACT

Automated Program Repair (APR) helps improve the efficiency of software development and maintenance. Recent APR techniques use deep learning, particularly the encoder-decoder architecture, to generate patches. Though existing DL-based APR approaches have proposed different encoder architectures, the decoder remains to be the standard one, which generates a sequence of tokens one by one to replace the faulty statement. This decoder has multiple limitations: 1) allowing to generate syntactically incorrect programs, 2) inefficiently representing small edits, and 3) not being able to generate project-specific identifiers.

In this paper, we propose Recoder, a syntax-guided edit decoder with placeholder generation. Recoder is novel in multiple aspects: 1) Recoder generates edits rather than modified code, allowing efficient representation of small edits; 2) Recoder is syntax-guided, with the novel provider/decider architecture to ensure the syntactic correctness of the patched program and accurate generation; 3) Recoder generates placeholders that could be instantiated as project-specific identifiers later.

We conduct experiments to evaluate Recoder on 395 bugs from *Defects4J v1.2* and 420 additional bugs from *Defects4J v2.0*. Our results show that Recoder repairs 53 bugs on Defects4J v1.2, which achieves 26.2% improvement over the previous state-of-the-art approach for single-hunk bugs (TBar). Importantly, to our knowledge, *Recoder is the first DL-based APR approach that has outperformed the traditional APR approaches on this dataset*. Furthermore, Recoder also repairs 19 bugs on the additional bugs from Defects4J v2.0, which is 137.5% more than TBar (8 bugs) and 850% more than SimFix (2 bugs). This result suggests that Recoder has better generalizability than existing APR approaches.

KEYWORDS

Automated program repair, Neural networks

ACM Reference Format:

Anonymous Author(s). 2018. A Syntax-Guided Edit Decoder for Neural Program Repair. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Automated program repair (APR) aims to reduce bug-fixing effort by generating patches to aid the developers. Due to the well-known problem of weak test suites [48], even if a patch passes all the tests, the patch still has a high probability of being incorrect. To overcome this problem, existing approaches have used different means to guide the patch generation. A typical way is to learn from existing software repositories, such as learning patterns from existing patches [3, 20, 21, 23, 33, 35, 51], and using program code to guide the patch generation [21, 36, 53, 64, 65].

Deep learning is known as a powerful machine learning approach. Recently, a series of research efforts have attempted to use deep learning (DL) techniques to learn from existing patches for program repair [8, 17, 30, 59]. A typical DL-based approach generates a new statement to replace the faulty statement located by a fault localization approach. Existing DL-based approaches are based on the encoder-decoder architecture [4]: the encoder encodes the faulty statement as well as any necessary code context into a fixed-length internal representation, and the decoder generates a new statement from it. For example, Hata et al. [17] and Tufano et al. [59] adopt an existing neural machine translation architecture, NMT, to generate the bug fix; SequenceR [8] uses a sequence-to-sequence neural model with a copy mechanism; DLFix [30] further treats the faulty statement as an AST rather than a sequence of tokens, and encodes the context of the statement.

However, despite multiple existing efforts, DL-based APR approaches have not yet outperformed traditional APR approaches. Since deep learning has outperformed traditional approaches in many domains, in this paper we aim to further improve the performance of DL-based APR to understand whether we could outperform traditional APR using a DL-based approach. We observe that, though existing DL-based APR approaches have proposed different encoder architectures for APR, the decoder architecture remains to be the standard one, generating a sequence of tokens one by one to replace the original faulty program fragment. The use of this standard decoder significantly limits the performance of DL-based APR. Here we highlight three main limitations.

Limitation 1: Including syntactically incorrect programs in the patch space. The goal of the decoder is to locate a patch from a patch space. The smaller the patch space is, the easier the task is. However, viewing a patch as a sequence of tokens unnecessarily enlarges the patch space, making the decoding task difficult. In particular, this space representation does not consider the syntax of the target programming language and includes many syntactically incorrect statements, which can never form a correct patch.

Limitation 2: Inefficient representation of small edits. Many patches only modify a small portion of a statement, and re-generating the whole statement leads to an unnecessarily large patch space. For example, let us consider the patch of defect Closure-14 in the Defects4J benchmark [1], as shown in Figure 1. This patch only changes one token in the statement, but under existing representation, it is encoded as a sequence of length 13. The program space containing this patch would roughly contain n^{13} elements, where n is the total number of tokens. On the other hand, let us consider a patch space including only one-token change edits. To generate that patch, only selecting a token in the faulty statement and a new

ESEC/FSE 2021, 23 - 27 August, 2021, Athens, Greece

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

```
- cfa.createEdge(fromNode, Branch.UNCOND, finallyNode);
+ cfa.createEdge(fromNode, Branch.ON_EX, finallyNode);
```

Figure 1: The Patch for Closure-14 in Defects4J

```
-   return cAvailableLocaleSet.contains(locale);
+   return availableLocaleSet().contains(locale);
```

Figure 2: The Patch for Lang-57 in Defects4J

token for replacement is needed. This patch space contains only mn elements, where m is the number of tokens in the faulty statement. Therefore, the size of the patch space is significantly reduced.

Limitation 3: Not being able to generate project-specific identifiers. Source code of programs often contains project-specific identifiers like variable names. Since it is impractical to include all possible identifiers in the patch space, existing DL-based APR approaches only generate identifiers that have frequently appeared in the training set. However, different projects have different sets of project-specific identifiers, and therefore only considering identifiers in the training set may exclude possible patches from the patch space. For example, Figure 2 shows the patch for defect Lang-57 in Defects4J. To generate this patch, we need to generate the identifier "`availableLocaleSet`", which is a method name of the faulty class, and is unlikely to be included in the training set. As a result, existing DL-based approaches cannot generate patches like this.

In this paper, we propose a novel DL-based APR approach, Recoder, standing for **re**pair de**coder**. Similar to existing approaches, Recoder is based on the encoder-decoder architecture. To address the limitations above, the decoder of Recoder has following two novel techniques.

Novelty 1: Syntax-Guided Edit Decoding with Provider/Decider Architecture (concerning limitation 1 & 2). To address limitation 2, the decoder component of Recoder produces a sequence of edits rather than a new statement. Our edit decoder is based on the idea of the syntax-guided decoder in existing neural program generation approaches [49, 56, 57, 67]. For an unexpanded non-terminal node in a partial AST, the decoder estimates the probability of each grammar rule to be used to expand the node. Based on this, the decoder selects the most probable sequence of rules to expand the start symbol into a full program using a search algorithm such as beam search. We observe that edits could also be described by a grammar. For example, the previous patch for defect Closure-14 could be described by the following grammar:

$$\begin{aligned} \mathit{Edit} &\rightarrow \mathit{Insert} \mid \mathit{Modify} \mid \dots \\ \mathit{Modify} &\rightarrow \texttt{modify}(\mathit{NodeID}, \mathit{NTS}) \end{aligned}$$

Here `modify` represents replacing an AST subtree denoted by its root node ID (*NodeID*) in the faulty statement with a newly generated subtree (*NTS*[1]). However, directly applying the existing syntax-guided decoder to the grammar above would not form an effective program repair approach, because the choice of expanding

[1] "NTS" stands for "non-terminal symbol".

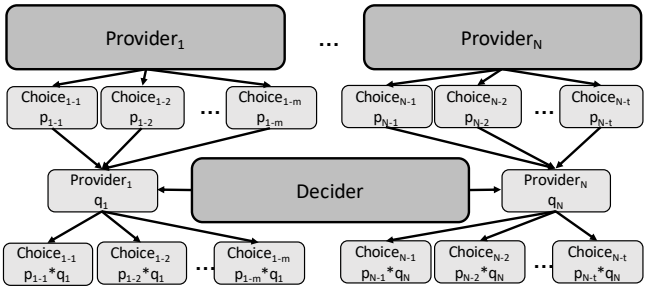


Figure 3: Provider/Decider Architecture

different non-terminal nodes may need to be deduced along with different types of dependencies. First, the expansion of some non-terminals depends on the local context, e.g., the choice of *NodeID* depends on the faulty statement, and the neural network needs to be aware of the local context to make a suitable choice. Second, to guarantee syntax correctness (limitation 1), dependency exists among the choices for expanding different non-terminal nodes, e.g., when *NodeID* expands to an ID pointing to a node with non-terminal *JavaExpr*, *NTS* should also expand to *JavaExpr* to ensure syntactic correctness. These choices cannot be effectively pre-defined, and thus the existing syntax-guided decoders, which only select among a set of pre-defined grammar rules, do not work here.

To overcome these problems, Recoder introduces a provider/decider architecture, as shown in Figure 3. A provider is a neural component that provides a set of choices for expanding a non-terminal and estimates the probability p_i of each choice. A basic provider is the rule predictor, which, similar to existing syntax-guided decoders, estimates the probability of each grammar rule to expand the node. Fixing the Closure-14 example needs another provider, namely the subtree locator, which estimates the probability of each subtree in the faulty statement to be replaced. On the other hand, the decider is a component that estimates the probability of q_j using each provider. In this example, when expanding *Edit*, the probability of using the rule predictor is 1, and the probability of using the subtree locator is 0; when expanding *Modify*, the probability of using the rule predictor is 0 and the probability of using the subtree locator is 1 (the located subtree decides both the content of *NodeID* and the root symbol of *NTS*). Finally, the choices provided by all providers form the final list of choices, while the probability of each choice is the product of the probability predicted by its provider and the probability of the provider itself, i.e., $p_i * q_j$.

In this example, for each non-terminal, we use the choices from only one provider, and thus the probabilities of providers are either 0 or 1. Later we will see that expanding some non-terminals requires comparing the choices of multiple providers, and the probabilities of providers could be a real number between 0 and 1.

Novelty 2: Placeholder Generation (concerning limitation 3). To generate project-specific identifiers, a direct idea is to add another provider that selects an identifier from the local context. However, to implement such a provider, the neural component needs to access all of the name declarations within the current project. This is a difficult job, as the neural component could hardly encode all source code from the whole project.

Instead of relying on the neural network to generate project-specific identifiers, in Recoder the neural network generates placeholders for such identifiers, and these placeholders are instantiated with all feasible identifiers when applying the edits. A feasible identifier is an identifier compatible with constraints in the programming language, such as the type system. As for defect Lang-57 shown in Figure 2, Recoder first generates a placeholder for "`availableLocaleSet`", and it will be replaced with all methods accessible in the local context that takes no arguments and returns an object with a member method "`contains`". Each replacement forms a new patch. The key insight is that, when considering constraints in the programming language, the number of choices for replacing a placeholder with an identifier is small, and thus instantiating the placeholders with all possible choices is feasible.

To train the neural network to generate placeholders, we replace infrequent user-defined identifiers in the training set with placeholders. In this way, the neural network learns to generate placeholders for these identifiers.

Our experiment is conducted on two benchmarks: (1) 395 bugs from Defects4J v1.2 for comparison with existing approaches, and (2) 420 additional bugs from Defects4J v2.0 to evaluate the generalizability of Recoder. The results show that Recoder correctly repairs 53 bugs on the first benchmark, which are 26.2% more than TBar [33] and 55.9% more than SimFix [21], two best-performing single-hunk APR approaches on Defects4J v1.2; Recoder also correctly repairs 19 bugs on the second benchmark, which are 137.5% more than the TBar (8 bugs) and 850.0% more than SimFix (2 bugs). The results suggest that Recoder has better performance and better generalizability than existing approaches. To our knowledge, *this is the first DL-based APR approach that has outperformed traditional APR approaches.*

To summarize, this paper makes the following contributions:

- We propose a syntax-guided edit decoder for APR with a provider/decider architecture to accurately predict the edits and ensure that the edited program is syntactically correct and uses placeholders to generate patches with project-specific identifiers.
- We design Recoder, a neural APR approach based on the decoder architecture described above.
- We evaluate Recoder on 395 bugs from Defects4J v1.2 and 420 additional bugs from Defects4J v2.0. The results show that Recoder significantly outperforms state-of-the-art approaches for single-hunk bugs in terms of both repair performance and generalizability.

2 EDITS

We introduce the syntax and semantics of edits and their relations to providers in this section. The neural architecture to generate edits and implement providers will be discussed in the next section.

2.1 Syntax and Semantics of Edits

Figure 4 shows the syntax of edits. Note that our approach is not specific to a particular programming language and can be applied to any programming language (called the *host language*) that has a concept similar to the statement. In particular, it is required that when a statement is present in a program, a sequence of statements

1. *Edits* → *Edit*; *Edits* | `end`
2. *Edit* → *Insert* | *Modify*
3. *Insert* → `insert(`⟨*HLStatement*⟩`)`
4. *Modify* → `modify(`
   ⟨*ID of an AST Node with a NTS*⟩,
   ⟨*the same NTS as the above NTS*⟩`)`
5. ⟨*Any NTS in HL*⟩ →
   `copy(`⟨*ID of an AST Node with the same NTS*⟩`)`
   | ⟨*The original production rules in HL*⟩
6. ⟨*HLIdentifier*⟩ → `placeholder`
   | ⟨*Identifiers in the training set*⟩

"HL" stands for "host language". "NTS" stands for "non-terminal symbol". "⟨*HLStatement*⟩" is the non-terminal in the grammar of the host language representing a statement. "⟨*HLIdentifier*⟩" is the non-terminal in the grammar of the host language representing an identifier.

Figure 4: The Syntax of Edits

can also be present at the same location. In other words, inserting a statement before any existing statement would still result in a syntactically correct program. To ensure syntactic correctness of the edited program, the syntax of edits depends on the syntax of the host language. In Figure 4, "HL" refers to the host programming language our approach applies to.

As defined by Rule 1 and Rule 2, an *Edits* is a sequence of *Edit* ended by a special symbol `end`. An *Edit* can be one of two edit operations, `insert` and `modify`.

Rule 3 defines the syntax of `insert` operation. The `insert` operation inserts a newly generated statement before the faulty statement. As shown in rule 3, the `insert` operation has one parameter, which is the statement to insert. Here ⟨*HLStatement*⟩ refers to the non-terminal in the grammar of the host language that represents a statement. This non-terminal could be expanded into a full statement, or a copy operation that copies a statement from the original program, or a mixture of both. This behavior will be explained later in Rule 5.

Rule 4 defines the syntax of `modify` operation. The `modify` operation replaces an AST subtree in the faulty statement with a new AST subtree. The `modify` operation has two parameters. The first parameter is the ID of the root node from the AST subtree to be replaced. The ID of a node is defined as the order of a node in the pre-order traversal sequence, e.g., the 6th visited node has the ID of 6. The second parameter is an AST subtree whose root node has the same symbol, i.e., the root node cannot be changed. In this way, the replacement ensures syntactic correctness. To ensure that there is an actual change, the subtree to be replaced should have more than one node, i.e., the root node should have a non-terminal symbol.

For both `insert` and `modify`, we need to generate a new AST subtree. It is noticeable that in many patches, the AST subtree being inserted or modified is not completely original; some of its subtrees may be copied from other parts of the program. Taking advantage of this property, `copy` operation is introduced to further reduce the patch space. Rule 5 defines the syntax of this operation. It is a meta-rule applied to any non-terminal symbol of the host language. For any non-terminal symbol in the host language, we add a production

rule that expands it into a copy operation. The original production rules for this non-terminal are also kept, so that when generating the edits, the neural network could choose to directly generate a new subtree or to copy one.

The copy operation has one parameter, which identifies the root node of the AST subtree to be copied. The AST subtree can be selected from the faulty statement or its context. In our current implementation, we allow copying from the method surrounding the faulty statement. Also, to ensure syntactic correctness, the root node of the subtree to be copied should have the same non-terminal symbol as the symbol being extended.

Finally, Rule 6 introduces `placeholder` into the grammar. Normally, the grammar of a programming language uses a terminal symbol to represent an identifier. To enable the neural network to generate concrete identifiers as well as the `placeholder`, we change identifier nodes into non-terminals, which expand to either `placeholder` or one of the frequent identifiers in the training set. In our current implementation, an identifier is considered frequent if it appears more than 100 times in the training set.

When applying the edits, the `placeholder` tokens are replaced with feasible identifiers within the context. We first collect all identifiers in the current projects by performing a lexical analysis and collect the tokens whose lexical type is ⟨`HLIdentifier`⟩, the symbol representing an identifier in the host language. Then we filter identifiers based on the following criteria: (1) the identifier is accessible from the local context, and (2) replacing the placeholder with the identifier would not lead to type errors. The remaining identifiers are feasible identifiers.

Figure 5 and Figure 6 show two example patches represented by edits. The patch in Figure 5 inserts an `if` statement, and the conditional expression contains a method invocation that is copied from the faulty statement. The patch in Figure 6 replaces the qualifier of a method invocation with another invocation, where the name of the method is a placeholder to be instantiated later.

Theorem 2.1. *The edited programs are syntactically correct.*

Proof. It is easy to see that the theorem holds by structural induction on the grammar of the edits. First, the requirement on the host programming language ensures that inserting a statement before another statement ensures syntactic correctness. Second, when replacing a subtree with `modify`, the root symbol of the subtree remains unchanged. Third, the new subtree in `insert` and `modify` is generated by either using the grammar rules of the host language, or copying a subtree with the same root symbol. Finally, instantiating a `placeholder` ensures syntactic correctness because we only replace a `placeholder` with a token whose lexical type is ⟨`HLIdentifier`⟩. □

2.2 Generation of Edits

Since the choice of expanding a non-terminal may depend on the local context or a previous choice, we use providers to provide choices and estimate their probabilities. Our current implementation has three types of providers. Table 1 shows these providers and their associated non-terminals.

For non-terminals *Edits*, *Edit*, *Insert* and ⟨*HLIdentifier*⟩, the rule predictor is responsible for providing choices and estimates the probability of each production rule. The rule predictor consists of a neural component and a logic component. After the neural component assigns the probability for each production rule, the logic component resets the probability of rules whose left-hand side is not the corresponding non-terminal to zero and normalizes the remaining probabilities.

For *Modify*, the subtree locator is responsible for providing the choices. The subtree locator estimates the probability of each AST subtree with a size larger than 1 in the faulty statement. The choice of a subtree t means that we should expand *Modify* into `modify`(*ID*, *NTS*) where *ID* is the root ID of t and *NTS* is the root symbol of t.

For any non-terminal in the grammar of the host language (note that ⟨*HLIdentifier*⟩ is a terminal symbol in the host language), both the rule predictor and the tree copier are responsible to provide

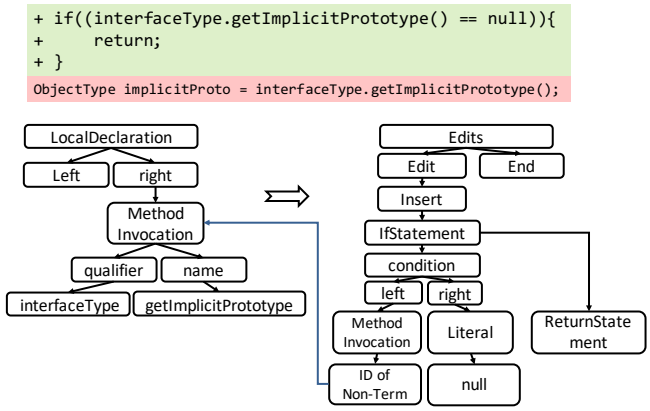


Figure 5: Example of Insert Operation (Closure-2)

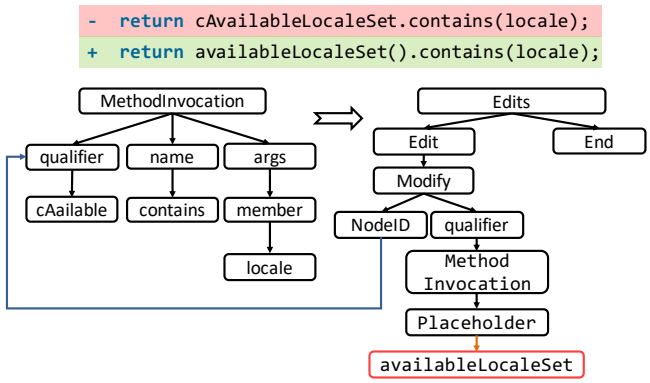


Figure 6: Example of Modify Operation (Lang-57)

Table 1: Providers for non-terminals

Component	Associated Non-terminals
Rule Predictor	*Edits*, *Edit*, *Insert*, ⟨*HLIdentifier*⟩, ⟨*Any NTS in HL*⟩
Subtree Locator	*Modify*
Tree Copier	⟨*Any NTS in HL*⟩

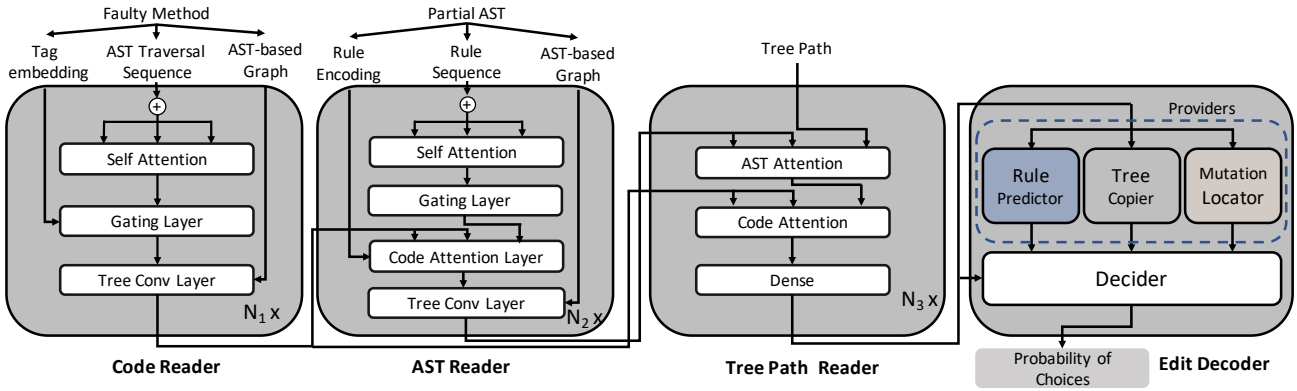


Figure 7: Overview of Model

the choices. The tree copier estimates the probabilities of each AST subtree with a size larger than 1 in the method surrounding the faulty statement. The choice of a subtree t means that we should expand the non-terminal into copy(ID), where ID is the root ID of t. Similar to the rule predictor, the tree copier employs a logic component after the neural component to reset the probabilities of subtrees whose root symbols are different from the non-terminal symbol being expanded.

Finally, the decider assigns a probability to each provider. The decider also includes a similar logic component, which resets the probability of a provider to zero if that provider is not responsible for the current non-terminal symbol. For example, if the symbol being expanded is *Modify*, the decider resets the probability of rule predictor and tree copier to zero.

3 MODEL ARCHITECTURE

The design of our model is based on the state-of-the-art syntax-guided code generation model, TreeGen [57]. It is a tree-based Transformer [60] that takes a natural language description as input and produces a program as output. Since our approach takes a faulty statement and its context as input and produces edits as output, we replace the components in TreeGen for encoding natural language description and decoding the program.

Figure 7 shows an overview of our model. The model performs one step in the edit generation process, which is to predict probabilities of choices for expanding a non-terminal node. Beam search is used to find the best combination of choices for generating the complete edits. The model consists of four main components:

- The **code reader** that encodes the faulty statement and its context.
- The **AST reader** that encodes the partial AST of the edits that have been generated.
- The **tree path reader** that encodes a path from the root node to a non-terminal node which should be expanded.
- The **edit decoder** that takes the encoded information from the previous three components and produces a probability of each choice for expanding the non-terminal node.

Among them, the AST reader and the tree path reader are derived from TreeGen, where the code reader and the edit decoder are newly introduced in this paper. In this section, we only describe the latter two components in detail. Description of the AST reader and the tree path reader can be found in the paper of TreeGen [57].

3.1 Code Reader

The code reader component encodes the faulty statement and the method surrounding the faulty statement as its context. It uses the following three inputs. **(1) AST traversal sequence.** This is a sequence of tokens following the pre-order traversal of the AST, $\boldsymbol{c}_1, \boldsymbol{c}_2, \cdots, \boldsymbol{c}_L$, where $\boldsymbol{c}_i$ is the token encoding vector of the ith node embedded via word embedding [42]. **(2) Tag embedding.** This is a sequence of tags following the same pre-order traversal of the AST, where each tag denotes which of the following cases the corresponding node belongs to: 1. in the faulty statement, 2. in the statement before the faulty statement, 3. in the statement after the faulty statement, or 4. in other statements. Each tag is embedded with an embedding-lookup table. We denote the tag embedding as $\boldsymbol{t}_1, \boldsymbol{t}_2, \cdots, \boldsymbol{t}_L$. **(3) AST-based Graph.** Considering that the former two inputs do not capture the neighbor relations between AST nodes, in order to capture such information, we treat an AST as a directional graph where the nodes are AST nodes and the edges link a node to each of its children and its left sibling, as shown in Figure 8(b). This graph is embedded as an adjacent matrix.

The code reader uses three sub-layers to encode the three inputs above, as discussed in the following sections.

3.1.1 Self-Attention. The self-attention sub-layer encodes the AST traversal sequence, following the Transformer [60] architecture to capture the long dependency information in the AST.

Given the embedding of the input AST traversal sequence, we use position embedding to represent positional information of the AST token. The input vectors are denoted as $\boldsymbol{c}_1, \boldsymbol{c}_2, \cdots, \boldsymbol{c}_L$, and the position embedding of ith token is computed as

$$p_{(i,2j)} = \sin(pos/(10000^{2j/d})) \quad (1)$$

$$p_{(i,2j+1)} = \cos(pos/(10000^{2j/d})) \quad (2)$$

where $pos = i + step$, j denotes the element of the input vector and $step$ denotes the embedding size. After we get the vector of

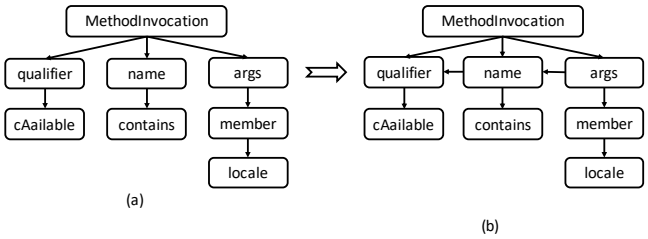


Figure 8: Example of AST-based Graph

each position, it is directly added to the corresponding input vector, where $\boldsymbol{e}_i = \boldsymbol{c}_i + \boldsymbol{p}_i$.

Then, we adopt multi-head attention layer to capture non-linear features. Following the definition of Vaswani et al. [60], we divide the attention mechanism into H heads. Each head represents an individual attention layer to extract unique information. The single attention layer maps the query Q, the key K, and the value V into a weighted-sum output. The computation of the jth head layer can be represented as

$$head_j = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{3}$$

where $d_k = d/H$ denotes the length of each extracted feature vector, and Q, K and V are computed by a fully-connected layer from Q, K, V. In the encoder, vectors Q, K and V are all the outputs of the position embedding layer $\boldsymbol{e}_1, \boldsymbol{e}_2, \cdots, \boldsymbol{e}_L$. The outputs of these heads are further joint together with a fully-connected layer, which is computed by

$$Out = [head_1; \cdots; head_H] \cdot W_h \tag{4}$$

where W_h denotes the weight of the fully-connected layer and Out denotes the outputs $\boldsymbol{a}_1, \boldsymbol{a}_2, \cdots, \boldsymbol{a}_L$ of the self-attention sub-layer.

3.1.2 Gating Layer. This sub-layer takes the outputs of the previous layer and the tag embedding as input. Gating mechanism, as defined in TreeGen [57], is used in this layer. It takes three vectors named $\boldsymbol{q}, \boldsymbol{c}_1, \boldsymbol{c}_2$ as input and aims to corporate c_1 with c_2 based on $\boldsymbol{q}$. The computation of gating mechanism can be represented as

$$\alpha_i^{c_1} = \exp(\boldsymbol{q}_i^T \boldsymbol{k}_i^{c_1})/\sqrt{d_k} \tag{5}$$

$$\alpha_i^{c_2} = \exp(\boldsymbol{q}_i^T \boldsymbol{k}_i^{c_2})/\sqrt{d_k} \tag{6}$$

$$\boldsymbol{h}_i = (\alpha_i^{c_1} \boldsymbol{v}_i^{c_1} + \alpha_i^{c_2} \boldsymbol{v}_i^{c_2})/(\alpha_i^{c_1} + \alpha_i^{c_2}) \tag{7}$$

where $d_k = d/H$ is a normalization factor, H denotes the number of heads, and d denotes the hidden size; $\boldsymbol{q}_i$ is computed by a fully-connected layer over the control vector $\boldsymbol{q}_i$; $\boldsymbol{k}_i^{c_1}, \boldsymbol{v}_i^{c_1}$ is computed by another fully-connected layer over vector $\boldsymbol{c}_1$; $\boldsymbol{k}_i^{c_2}$ and $\boldsymbol{v}_i^{c_2}$ are also computed by the same layer with different parameters over the vector $\boldsymbol{c}_2$.

In our model, we treat the outputs of the self-attention sub-layer $\boldsymbol{a}_1, \boldsymbol{a}_2, \cdots, \boldsymbol{a}_L$ as $\boldsymbol{q}$ and $\boldsymbol{c}_1$, and the tag embedding $\boldsymbol{t}_1, \boldsymbol{t}_2, \cdots, \boldsymbol{t}_L$ as $\boldsymbol{c}_2$. Thus, embedding of the ith AST node of the gating-layer can be represented as $\boldsymbol{u}_i = \text{Gating}(\boldsymbol{a}_i, \boldsymbol{a}_i, \boldsymbol{t}_i)$.

3.1.3 Tree Conv Layer. This sub-layer takes the output $\boldsymbol{u}_i$ of the previous layer and the AST-based graph G (represented as an adjacency matrix) as input. We adopt a GNN [55] layer to process the inputs, and the encoding of the neighbors r_i is computed as

$$\boldsymbol{g}_i = W_g \sum_{r^j \in G} A^n_{r^i r^j} \boldsymbol{u}^j \tag{8}$$

where W_g is the weight of a fully-connected layer and $\hat{A}$ is a normalized adjacency matrix of G. The computation of the normal operation proposed by Kipf and Welling [25] is represented as

$$\hat{A} = S_1^{-1/2} A S_2^{-1/2} \tag{9}$$

where A is the adjacency matrix of G, and S_1, S_2 are the diagonal matrices with a summation of A in columns and rows. Then, the encoding of the neighbors is directly added to the input vector.

In summary, the code reader has N_1 blocks of these three sub-layers, and yields the features of the input AST, $\boldsymbol{t}_1, \boldsymbol{t}_2, \cdots, \boldsymbol{t}_L$, which would be used for the AST reader and the tree path reader.

3.2 Edit Decoder

The edit decoder takes vectors $\boldsymbol{d}_1, \boldsymbol{d}_2, \cdots, \boldsymbol{d}_T$ with length T as input. These vectors are produced by the tree path reader and contain the encoded information from all the inputs: the faulty statement with its surrounding method, the partial AST generated so far, and the tree path denoting the node to be expanded.

3.2.1 Provider. As mentioned before, there are currently three types of providers: rule predictor, tree copier, and subtree locator. These providers take the vector $\boldsymbol{d}_1, \boldsymbol{d}_2, \cdots, \boldsymbol{d}_T$ as input and output the probability of choices for different non-terminals.

Rule Predictor. The rule predictor estimates the probability of each production rule. The neural component of this decider consists of a fully-connected layer. The output of the fully-connected layer is denoted as $\boldsymbol{s}_1, \boldsymbol{s}_2, \cdots, \boldsymbol{s}_T$. Then, these vectors are normalized via softmax, which computes the normalized vectors $\boldsymbol{p}_1^r, \boldsymbol{p}_2^r, \cdots, \boldsymbol{p}_T^r$ by

$$\boldsymbol{p}_k^r(m) = \frac{\exp\{\boldsymbol{s}_k^m\}}{\sum_{j=1}^{N_r} \exp\{\boldsymbol{s}_k^j\}} \tag{10}$$

where N_r denotes the number of production rules in the grammar, and m denotes the mth dimension of the vector $\boldsymbol{p}_k^r$ (i.e., the production rule with ID m). In particular, invalid rules whose left-hand side is not the corresponding non-terminal are not allowed in our approach. For these rules, the logic component resets the output of the fully-connected layer to $-\infty$. Thus, the probability of invalid rules will be zero after softmax normalization.

Tree Copier. This provider is designed for any non-terminal to choose a subtree in the local context. The neural component is based on a pointer network [61]. The computation can be represented as

$$\boldsymbol{\theta}_i = \boldsymbol{v}^T \tanh(W_1 \boldsymbol{d}_i + W_2 \boldsymbol{t}) \tag{11}$$

where $\boldsymbol{t}$ denotes the output of the code reader, and $\boldsymbol{v}, W_1, W_2$ denote the trainable parameters. The logic component also resets $\boldsymbol{\theta}$ to $-\infty$ if the root symbol of the corresponding subtree is different from the symbol being expanded. These vectors are then normalized via softmax as Equation 11. We denote the normalized vector as $\boldsymbol{p}_1^t, \boldsymbol{p}_2^t, \cdots, \boldsymbol{p}_T^t$.

Subtree Locator. This component outputs an ID of the subtree in the faulty statement for *Modify*. The computation of this component is the same as the neural component in the tree copier. We denote the output vector of this provider as $\boldsymbol{p}_1^s, \boldsymbol{p}_2^s, \cdots, \boldsymbol{p}_T^s$

3.2.2 Decider. For these three providers, the decider estimates the probability of using each provider. The neural component also takes the output of the tree path reader, $\boldsymbol{d}_1, \boldsymbol{d}_2, \cdots, \boldsymbol{d}_T$, as input, and produces the probability of using each provider as output. The computation can be represented as

$$\boldsymbol{\lambda}_i = W\boldsymbol{d}_i + \boldsymbol{b} \tag{12}$$

where W and $\boldsymbol{b}$ denote the parameters of a fully-connected layer. The logic component resets $\boldsymbol{\lambda}$ to $-\infty$ if the corresponding provider is not responsible for the symbol being expanded. Then, the vectors are normalized via softmax as Equation 11. We denote the normalized vectors as $\boldsymbol{\lambda}_1, \boldsymbol{\lambda}_2, \cdots, \boldsymbol{\lambda}_T$. The final probability of each choice can be computed as

$$\boldsymbol{o}_i = [\boldsymbol{\lambda}_i^r \boldsymbol{p}_i^r; \boldsymbol{\lambda}_i^t \boldsymbol{p}_i^t; \boldsymbol{\lambda}_i^s \boldsymbol{p}_i^s] \tag{13}$$

where $\boldsymbol{o}_i$ will be the probability vector of the next production rule at ith step during patch generation.

3.3 Training and Inference

During training, the model is optimized by maximizing the negative log-likelihood of the oracle edit sequence. For better training, we do not use the logic component in the providers and decider.

When generating edits, inference starts with the rule *start* : *start* $\longrightarrow$ *Edits*, expanding a special symbol `start` to `Edits`. The recursive prediction terminates if every leaf node in the predicted AST is a terminal. We use beam search with a size of 100 to generate multiple edits.

Generated edits may contain placeholders. Though the number of choices for a single placeholder is small, the combination of multiple placeholders may be large. Therefore, we discard patches containing more than one `placeholder` symbol during beam search.

3.4 Patch Generation and Validation

Patches are generated according to the result of the fault localization technique. In our approach, the model described above is invoked for each suspicious faulty statement according to the result of fault localization. For each statement, we generate 100 valid patch candidates via beam search: when beam search generates a valid patch, we remove it from the search set and continue to search for the next patch until 100 candidates are generated in total for that statement. After patches are generated, the final step is to validate them via the test suite written by developers. The validation step filters out patches that do not compile or fail a test case. All generated patches are validated until a plausible patch (a patch that passes all test cases) is found.

4 EXPERIMENT SETUP

4.1 Research Questions

Our evaluation aims to answer the following research questions:

- **RQ1: What is the performance of Recoder?**
  To answer this question, we evaluated our approach on the widely used APR benchmark, *Defects4J v1.2*, and compared it with traditional and DL-based APR tools.
- **RQ2: What is the contribution of each component in Recoder?**
  To answer this question, we started from the full model of Recoder, and removed each component in turn to understand its contribution to performance.
- **RQ3: What is the generalizability of Recoder?**
  To answer this question, we conducted an experiment on 420 additional bugs from *Defects4J v2.0*. To our best knowledge, this is the first APR approach that has been applied to this benchmark. We compared Recoder with the previous two best-performing APR approaches for single-hunk bugs on Defects4J v1.2, namely TBar [33] and SimFix [21].

4.2 Dataset

The neural network model in our approach needs to be trained with a large number of history patches. To create this training set, we crawled Java projects created on GitHub [14] between March 2011 and March 2018, and downloaded 1,083,185 commits where the commit message contains at least one word from the following two groups, respectively: (1) *fix, solve*; (2) *bug, issue, problem, error*. Commits were filtered to include only patches that modify one single statement or insert one new statement, corresponding to two types of edits that our approach currently supports. To avoid data leak, we further discarded patches where (1) the project is a clone to Defects4J project or a program repair project using Defects4J, or (2) the method modified by the patch is the same as the method modified by any patch in Defects4J v1.2 or v2.0, based on AST comparison. There are 103,585 valid patches left after filtering, which are further split into two parts: 80% for training and 20% for validation.[2]

We used two benchmarks to measure the performance of Recoder. The first one contains 395 bugs from *Defects4J* v1.2 [1], which is a commonly used benchmark for automatic program repair research. The second one contains 420 additional bugs from *Defects4J v2.0* [1]. Defects4J v2.0 introduces 438 new bugs compared with Defects4J v1.2. However, GZoltar [50], the fault localization approach used by our implementation as well as two baselines (TBar and SimFix), failed to finish on the project *Gson*, so we excluded 18 bugs in *Gson* from our benchmark.

4.3 Fault Localization

In our experiment, two settings for fault localization are used. In the first setting, the faulty location of a bug is unknown to APR tools, and they rely on existing fault localization approaches to localize the bug. Recoder uses Ochiai [2] (implemented in GZoltar [50]), which is widely used in existing APR tools [21, 33]. In the second setting, the actual faulty location is given to APR tools. This is to measure the capability of patch generation without the influence of a specific fault localization tool, as suggested and adopted in previous studies [6, 37, 59].

[2]We will publish the dataset if this paper is accepted.

4.4 Baselines

We selected existing APR approaches as the baselines for comparison. Since Recoder generates only single-hunk patches (patches that only change a consecutive code fragment), we chose 10 traditional single-hunk APR approaches that are often used as baselines in existing studies: jGenProg [29], HDRepair [27], Nopol [66], CapGen [62], SketchFix [19], TBar [33], FixMiner [26], SimFix [21], PraPR [13], AVATAR [32]. In particular, TBar correctly repairs the highest number of bugs on Defects4J v1.2 as far as we know. We also selected DL-based APR approaches that adopt the encoder-decoder architecture to generate patches and have been evaluated on Defects4J as baselines. Four approaches have been chosen based on this criteria, namely, SequenceR [59], CODIT [6], DLFix [30], and CoCoNuT [37].

For Defects4J v1.2, the performance data of the baselines are collected from existing papers [33, 34]. For additional bugs from Defects4J v2.0, two best-performing single-hunk APR approaches on Defects4J v1.2, TBar and SimFix, are adapted and executed for comparison.

4.5 Correctness of Patches

To check the correctness of the patches, we manually examined every patch if it is the same with or semantically equivalent to the patch provided by Defects4J, as in previous works [13, 21, 30, 33, 37]. To reduce possible errors made in this process, every patch is examined by two of the authors individually and is considered correct only if both authors consider it correct. Furthermore, we also publish all the patches generated by Recoder for public judgment[3].

4.6 Implementation Details

Our approach is implemented based on PyTorch [46], with parameters set to $N_1 = 5, N_2 = 9, N_3 = 2$, i.e., the code reader contains a stack of 5 blocks, the AST reader contains a stack of 9 blocks, and the decoder contains a stack of 2 blocks, respectively. Embedding sizes for all embedding vectors are set to 256, and all hidden sizes are set following the configuration of TreeGen [57]. During training, dropout [18] is used to prevent overfitting, with the drop rate of 0.1. The model is optimized by Adam [24] with learning rate 0.0001. These hyper-parameters and parameters for our model are chosen based on the performance on validation set.

We set a 5-hour running-time limit for Recoder, following existing studies [21, 30, 37, 54].

5 EXPERIMENTAL RESULTS

5.1 Performance of Recoder (RQ1)

5.1.1 Results without Perfect Fault Localization. We first compare Recoder with the baselines in the setting where no faulty location is given. Results as Table 2 shown only include baselines that have been evaluated under this setting. As shown, Recoder correctly repairs 53 bugs and outperforms all of the previous single-hunk APR techniques on Defects4J v1.2. In particular, Recoder repairs 26.2% more bugs than the previous state-of-the-art APR tool for single-hunk bugs, TBar. Within our knowledge, Recoder is the first DL-based APR approach that has outperformed the traditional APR approaches.

[3]The source code of Recoder, generated patches, and an online demo are available at https://github.com/FSE2021anonymous/Recoder

```
- this(time, RegularTimePeriod.DEFAULT_TIME_ZONE,Locale.getDefault());
+ this(time, zone, Locale.getDefault());
```

Figure 9: Chart-8 - A bug fixed by Recoder with Modify operation

```
-  if (result != null) {
+ if(((result != null) && !result.isNoType())){
```

Figure 10: Closure-104 - A bug fixed by Recoder with *placeholder* generation

We show a few example patches that are possibly generated with the help of the novel techniques in Recoder. As shown in Figure 9, Chart-8 is a bug that DLFix fails to fix. The correct patch only changes a parameter of the method invocation while DLFix needs to generate the whole expression. By contrast, Recoder generates a *modify* operation that changes only one parameter. Figure 10 shows a bug only repaired by Recoder. This patch relies on a project-specific method, "`isNoType`", and thus cannot be generated by many of the existing approaches. However, Recoder fixes it correctly by generating a placeholder and then instantiating it with "`isNoType`".

5.1.2 Results with Perfect Fault Localization. Table 3 shows the result where the actual faulty location is provided. As before, only baselines that have been evaluated under this setting are listed. Recoder still outperforms all of the existing APR approaches, including traditional ones. Also, compared with Recoder using Ochiai for fault localization, this model achieves a 36.5% improvement. The result implies that Recoder can achieve better performance with better fault localization techniques.

5.1.3 Degree of Complementary. We further investigate to what extent Recoder complements the three best-performing existing approaches for fixing single-hunk bugs, TBar, SimFix, and DLFix. Figure 11 reveals the overlaps of the bugs fixed by different approaches. As shown, Recoder fixes 19 unique bugs when compared with three baselines. Moreover, Recoder fixes 34, 28, 27 unique bugs compared with SimFix, TBar, and DLFix, respectively. This result shows that Recoder is complementary to these best-performing existing approaches for single-hunk bugs.

5.2 Contribution of Each Component (RQ2)

To answer RQ2, we conducted an ablation test on Defects4J v1.2 to figure out the contribution of each component. Since the ablation test requires much time, we only conducted the experiment based on Ochiai Fault Localization scenario.

Table 4 shows the results of the ablation test. We respectively removed three edit operations, `modify`, `copy`, and `insert`, as well as the generation of placeholders. As shown in the table, removing any of the components leads to a significant drop in performance. This result suggests that the two novel techniques proposed in Recoder are the key to its performance.

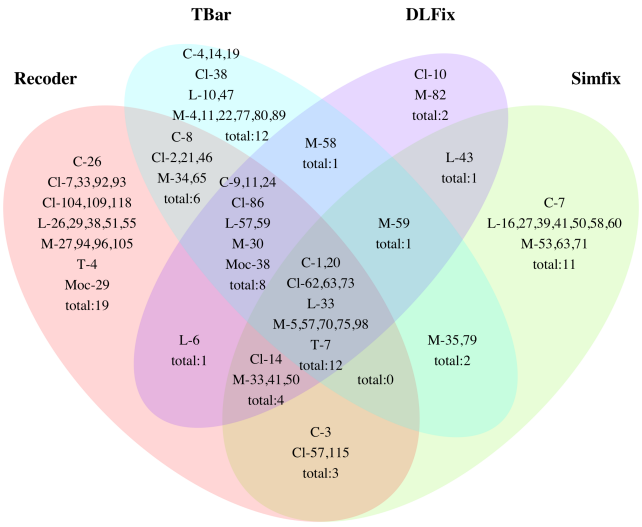
Table 2: Comparison without Perfect Fault Localization

Project	jGenProg	HDRepair	Nopol	CapGen	SketchFix	FixMiner	SimFix	TBar	DLFix	PraPR	AVATAR	Recoder
Chart	0/7	0/2	1/6	4/4	6/8	5/8	4/8	**9/14**	5/12	4/14	5/12	8/14
Closure	0/0	0/7	0/0	0/0	3/5	5/5	6/8	8/12	6/10	12/62	8/12	**17/31**
Lang	0/0	2/6	3/7	5/5	3/4	2/3	**9/13**	5/14	5/12	3/19	5/11	**9/15**
Math	5/18	4/7	1/21	12/16	7/8	12/14	14/26	**18/36**	12/28	6/40	6/13	15/30
Time	0/2	0/1	0/1	0/0	0/1	1/1	1/1	1/3	1/2	0/7	1/3	**2/2**
Mockito	0/0	0/0	0/0	0/0	0/0	0/0	0/0	1/2	1/1	1/6	**2/2**	**2/2**
Total	5/27	6/23	5/35	21/25	19/26	25/31	34/56	42/81	30/65	26/148	27/53	**53/94**
P(%)	18.5	26.1	14.3	**84.0**	73.1	80.6	60.7	51.9	46.2	17.6	50.9	56.4

In the cells, x/y:x denotes the number of correct patches, and y denotes the number of patches that can pass all the test cases.

Table 3: Comparison with Perfect Fault Localization

Project	SequenceR	CODIT	DLFix	CoCoNuT	TBar	Recoder
Chart	3	4	5	7	**11**	**11**
Closure	3	3	11	9	17	**26**
Lang	3	3	8	7	**13**	10
Math	4	6	13	16	**22**	19
Time	0	0	2	1	2	**3**
Mockito	0	0	1	**4**	3	2
Total	13	16	40	44	68	**71**



Project Names: C:Chart, CL:Closure, L:Lang, M:Math, Moc:Mockito, T:Time

Figure 11: Degree of Complementary.

5.3 Generalizability of Recoder (RQ3)

The results are shown in Table 5. As shown from the table, all three approaches repair a smaller proportion of bugs, suggesting that the additional bugs on Defects4J v2.0 are probably more difficult to repair. Nevertheless, Recoder still repairs most bugs, 19 in total, achieving 137.5% improvement over TBar and 850.0% improvement over SimFix. We believe that the considerable performance drops of TBar and SimFix are caused by their design: TBar is based on validated patterns on Defects4J v1.2, which may not generalize beyond the projects in Defects4J v1.2; SimFix relies on similar code snippets in the same project, but new projects in Defects4J v2.0 are much smaller, and thus the chance to find similar code snippets become smaller. On the other hand, Recoder is trained from a large set of patches collected from different projects and is thus more likely to generalize to new projects.

Table 4: Ablation Test for Recoder on Defects4J v1.2

Project	`-modify`	`-subtreecopy`	`-insert`	`-placeholder`	Recoder
Chart	4	6	7	8	8
Closure	6	12	12	11	16
Lang	3	6	5	5	10
Math	7	8	9	9	15
Time	1	1	1	1	2
Mockito	2	1	1	1	2
Total	23	34	35	35	53

Table 5: Comparison on the 420 additional bugs

Project	# Used Bugs	Bug IDs	TBar	SimFix	Recoder
Cli	39	1-5,7-40	1/7	0/4	**3/3**
Clousre	43	134 - 176	0/5	**1/5**	0/7
JacksonDatabind	112	1-112	0/0	0/0	0/0
Codec	18	1-18	**2/6**	0/2	**2/2**
Collections	4	25-28	0/1	0/1	0/0
Compress	47	1-47	1/13	0/6	**3/9**
Csv	16	1-16	1/5	0/2	**4/4**
JacksonCore	26	1-26	0/6	0/0	0/4
Jsoup	94	1-94	3/7	1/5	**7/13**
JxPath	22	1-22	0/0	0/0	0/4
Total	420	-	8/50	2/25	**19/46**

6 RELATED WORK

DL-based APR Approaches. APR has been approached using different techniques, such as heuristics or random search [28, 29, 38,

47], semantic analysis [7, 10, 19, 22, 32, 39–41], manually defined or automatically mined repair patterns [3, 12, 20, 21, 26, 35, 45, 52], and learning from source code [21, 36, 62–65]. For a thorough discussion of APR, existing surveys [11, 15, 43, 44] are recommended to readers.

A closely related series of work is APR based on deep learning. The mainstream approaches treat APR as a statistical machine translation that generates the fixed code with faulty code. DeepFix [16] learns the syntax rules via a sequence-to-sequence model to fix syntax errors. Nguyen et al. [45] and Tufano et al. [59] also adopt a sequence-to-sequence translation model to generate the patch. They use sequence-to-sequence NMT with a copy mechanism. Chakraborty et al. [6] propose CODIT, which learns code edits by encoding code structures in an NMT model to generate patches. Li et al. [30] propose a Tree-LSTM to encode the abstract syntax tree of the faulty method to generate the patches. CoCoNuT, as proposed by Lutellier et al. [37], adopts CNN to encode the faulty method and generates the patch token by token. Compared to them, our paper is the first work that aims to improve the decoder and employs a syntax-guided manner to generate edits, with the provider/decider architecture and placeholder generation.

Multi-Hunk APR. Most of the above approaches generate single-hunk patches—patches that change one place of the program. Recently, Saha et al. [54] propose Hercules to repair multi-hunk bugs by discovering similar code snippets and applying similar changes. Since lifting single-hunk repair to multiple-hunk repair is a generic idea and can also be applied to Recoder, we did not directly compare Recoder with the multi-hunk repair tools in our evaluation. Nevertheless, though Recoder only repairs single-hunk bugs, we notice that it still outperforms Hercules by repairing 5 more bugs on the Defects4J v1.0 (including all projects from v1.2 except for Mockito), the dataset Hercules has been evaluated on.

DL-based Code Generation. Code generation aims to generate code from a natural language specification and has been intensively studied during recent years. With the development of deep learning, Ling et al. [31] propose a neural machine translation model to generate the program token by token. Being aware that code has the constraints of grammar rules and is different from natural language, Yin and Neubig [67] and Rabinovich et al. [49] propose to generate the AST of the program via expanding from a start node. To alleviate the long dependency problem, a CNN decoder [56] and TreeGen (a tree-based Transformer) [57] are proposed to generate the program. In this paper, we significantly extend TreeGen to generate the edit sequence for program repair.

DL-based Code Edit Generation. Several existing DL-based approaches also use the idea of generating edits on programs. Tarlow et al. [58] view a program as a sequence of tokens and generate a sequence of token-editing commands. Brody et al. [5] view a program as a tree and generate node-editing or subtree-editing commands. Dinella et al. [9] view a program as a graph and generate node-editing commands. Compared with our approach, there are three major differences. First, the existing approaches are not syntax-guided and treat an edit script as a sequence of tokens. As a result, they may generate syntactically incorrect edit scripts and do not ensure syntactic correctness of the edited program. On the other hand, our approach introduces the provider/decider architecture and successfully realizes the syntax-guided generation for edits. Second, none of the existing approaches support placeholder generation, and thus are ineffective in generating edits with project-specific identifiers. Third, the editing commands they use are atomic and are inefficient in representing large changes. For example, to insert a variable declaration, in our approach there is one `insert` operation: `insert(int var = 0;)`. However, the existing approaches have to represent this change as a sequence of 5 insertions, where each insertion inserts one token.

7 THREATS TO VALIDITY

Threats to external validity mainly lie in the evaluation dataset we used. First, though our approach applies to different programming languages, so far, we have only implemented and evaluated it on Java, so future work is needed to understand its performance on other programming languages. Second, though we have evaluated on Defects4J v2.0, which contains a diverse set of projects, it is yet unknown how our approach generalizes to different datasets. This is a future work to be explored.

Threats to internal validity mainly lie in our manual assessment of patch correctness. To reduce this threat, two authors have independently checked the correctness of the patches, and a patch is considered correct only if both authors consider it correct. The generated patches also have been released for public assessment.

8 CONCLUSION

In this paper, we propose a syntax-guided edit decoder for neural program repair. Our decoder uses a novel provider/decider architecture to ensure accurate generation and the syntactic correctness of the edited program and generates placeholders for project-specific identifiers. In our experiment, Recoder achieved 26.2% improvement over the existing state-of-the-art APR approach for single-hunk bugs. Importantly, Recoder is the first DL-based APR approach that has outperformed traditional APR techniques. Further evaluation on Defects4J v2.0 shows that Recoder has better generalizability than some state-of-the-art APR approaches.

REFERENCES

[1] 2020. Defects4J. In *https://github.com/rjust/defects4j*.

[2] Rui Abreu, Peter Zoeteweij, and Arjan J. C. Van Gemund. 2007. On the Accuracy of Spectrum-based Fault Localization. In *Testing: Academic & Industrial Conference Practice & Research Techniques-mutation*.

[3] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–27.

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. *CoRR* abs/1409.0473 (2015).

[5] Shaked Brody, Uri Alon, and Eran Yahav. 2020. A Structural Model for Contextual Code Changes. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 215 (Nov. 2020), 28 pages. https://doi.org/10.1145/3428283

[6] Saikat Chakraborty, Miltiadis Allamanis, and Baishakhi Ray. 2018. CODIT: Code Editing with Tree-Based Neural Machine Translation. *arXiv preprint arXiv:1810.00314* (2018).

[7] Liushan Chen, Yu Pei, and Carlo A Furia. 2017. Contract-based program repair without the contracts. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 637–647.

[8] Z. Chen, S. J. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus. 2019. SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2940179

[9] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. HOPPITY: LEARNING GRAPH TRANSFORMATIONS TO DETECT AND FIX BUGS IN PROGRAMS. In *International Conference on Learning Representations*. https://openreview.net/forum?id=SJeqs6EFvB

[10] Xiang Gao, Bo Wang, Gregory J Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2021. Beyond Tests: Program Vulnerability Repair via Crash Constraint Extraction. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–27.

[11] L. Gazzola, D. Micucci, and L. Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 45, 1 (2019), 34–67. https://doi.org/10.1109/TSE.2017.2755013

[12] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 19–30.

[13] A. Ghanbari and L. Zhang. 2019. PraPR: Practical Program Repair via Bytecode Mutation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1118–1121. https://doi.org/10.1109/ASE.2019.00116

[14] GitHub. 2020. https://github.com/. GitHub.

[15] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65.

[16] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence* (San Francisco, California, USA) *(AAAI'17)*. AAAI Press, 1345–1351.

[17] Hideaki Hata, Emad Shihab, and Graham Neubig. 2018. Learning to Generate Corrective Patches using Neural Machine Translation. *CoRR* abs/1812.07170 (2018). arXiv:1812.07170 http://arxiv.org/abs/1812.07170

[18] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2012. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR* abs/1207.0580 (2012). arXiv:1207.0580 http://arxiv.org/abs/1207.0580

[19] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Sketchfix: A tool for automated program repair approach using lazy candidate generation. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 888–891.

[20] Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. 2019. Inferring program transformations from singular examples via big code. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 255–266.

[21] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space with Existing Patches and Similar Code. In *International Symposium on Software Testing & Analysis*. 298–309.

[22] Shalini Kaleeswaran, Varun Tulsian, Aditya Kanade, and Alessandro Orso. 2014. Minthint: Automated synthesis of repair hints. In *Proceedings of the 36th International Conference on Software Engineering*. 266–276.

[23] D. Kim, J. Nam, J. Song, and S. Kim. 2013. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*. 802–811. https://doi.org/10.1109/ICSE.2013.6606626

[24] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. *CoRR* abs/1412.6980 (2015).

[25] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[26] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. Fixminer: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering* (2020), 1–45.

[27] Xuan Bach D Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 213–224.

[28] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *2012 34th International Conference on Software Engineering (ICSE)*. 3–13. https://doi.org/10.1109/ICSE.2012.6227211

[29] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[30] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: Context-Based Code Transformation Learning for Automated Program Repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 602–614. https://doi.org/10.1145/3377811.3380345

[31] Wang Ling, Phil Blunsom, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Fumin Wang, and Andrew Senior. 2016. Latent Predictor Networks for Code Generation. In *ACL*. 599–609.

[32] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 1–12.

[33] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. Tbar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 31–42.

[34] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 2020. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 615–627.

[35] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 727–739.

[36] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 298–312.

[37] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, and Lin Tan. 2020. CoCoNuT: combining context-aware neural translation models using ensemble for program repair. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*.

[38] Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java (Demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 441–444. https://doi.org/10.1145/2931037.2948705

[39] Sergey Mechtaev, Alberto Griggio, Alessandro Cimatti, and Abhik Roychoudhury. 2018. Symbolic execution with existential second-order constraints. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 389–399.

[40] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. Directfix: Looking for simple program repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 448–458.

[41] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*. 691–701.

[42] Tomas Mikolov, Kai Chen, G. S. Corrado, and J. Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *ICLR*.

[43] Martin Monperrus. 2017. Automatic Software Repair: a Bibliography. *ACM Computing Surveys* 51 (2017), 1–24. https://doi.org/10.1145/3105906

[44] Martin Monperrus. 2020. *The Living Review on Automated Program Repair*. Technical Report hal-01956501. HAL. https://www.monperrus.net/martin/repair-living-review.pdf

[45] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. 2013. SemFix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. 772–781. https://doi.org/10.1109/ICSE.2013.6606623

[46] pytorch. 2020. https://pytorch.org/. pytorch.

[47] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The Strength of Random Search on Automated Program Repair. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 254–265. https://doi.org/10.1145/2568225.2568254

[48] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems *(ISSTA)*. 24–36.

[49] Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract Syntax Networks for Code Generation and Semantic Parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, Regina Barzilay and Min-Yen Kan (Eds.). Association for Computational Linguistics, 1139–1149. https://doi.org/10.18653/v1/P17-1105

[50] André Riboira and Rui Abreu. 2010. The GZoltar Project: A Graphical Debugger Interface. 215–218. https://doi.org/10.1007/978-3-642-15585-7_25

[51] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 404–415. https://doi.org/10.1109/ICSE.2017.44

[52] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 404–415.

[53] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. 2017. ELIXIR: Effective Object Oriented Program Repair. In *ASE* (Urbana-Champaign, IL, USA). IEEE Press. http://dl.acm.org/citation.cfm?id=3155562.3155643

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

[55] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. 2009. The Graph Neural Network Model. *IEEE Transactions on Neural Networks* 20, 1 (2009),

61–80.

[56] Zeyu Sun, Qihao Zhu, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang. 2019. A Grammar-Based Structural CNN Decoder for Code Generation. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, 7055–7062. https://doi.org/10.1609/aaai.v33i01.33017055

[57] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. TreeGen: A Tree-Based Transformer Architecture for Code Generation. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 8984–8991. https://aaai.org/ojs/index.php/AAAI/article/view/6430

[58] Daniel Tarlow, Subhodeep Moitra, Andrew Rice, Zimin Chen, Pierre-Antoine Manzagol, Charles Sutton, and Edward Aftandilian. 2019. Learning to Fix Build Errors with Graph2Diff Neural Networks. *CoRR* abs/1911.01205 (2019). arXiv:1911.01205 http://arxiv.org/abs/1911.01205

[59] M. Tufano, C. Watson, G. Bavota, M. di Penta, M. White, and D. Poshyvanyk. 2018. An Empirical Investigation into Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 832–837. https://doi.org/10.1145/3238147.3240732

[60] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, undefinedukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[61] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. 2015. Pointer Networks. *ArXiv* abs/1506.03134 (2015).

[62] M. Wen, J. Chen, R. Wu, D. Hao, and S. Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 1–11. https://doi.org/10.1145/3180155.3180233

[63] Qi Xin and Steven P. Reiss. 2017. Leveraging syntax-related code for automated program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 660–670.

[64] Yingfei Xiong, Bo Wang, Guirong Fu, and Linfei Zang. 2018. Learning to synthesize. In *Proceedings of the 4th International Workshop on Genetic Improvement Workshop*. 37–44.

[65] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 416–426.

[66] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering* 43, 1 (2016), 34–55.

[67] Pengcheng Yin and Graham Neubig. 2017. A Syntactic Neural Model for General-Purpose Code Generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers*, Regina Barzilay and Min-Yen Kan (Eds.). Association for Computational Linguistics, 440–450. https://doi.org/10.18653/v1/P17-1041