DLFix: Context-based Code Transformation Learning for Automated Program Repair

Yi Li
New Jersey Inst. of Technology, USA
yl622@njit.edu

Shaohua Wang*
New Jersey Inst. of Technology, USA
davidsw@njit.edu

Tien N. Nguyen
University of Texas at Dallas, USA
tien.n.nguyen@utdallas.edu

ABSTRACT

Automated Program Repair (APR) is very useful in helping developers in the process of software development and maintenance. Despite recent advances in deep learning (DL), the DL-based APR approaches still have limitations in learning bug-fixing code changes and the context of the surrounding source code of the bug-fixing code changes. These limitations lead to incorrect fixing locations or fixes. In this paper, we introduce DLFix, a two-tier DL model that treats APR as code transformation learning from the prior bug fixes and the surrounding code contexts of the fixes. The first layer is a tree-based RNN model that learns the contexts of bug fixes and its result is used as an additional weighting input for the second layer designed to learn the bug-fixing code transformations.

We conducted several experiments to evaluate DLFix in two benchmarks: *Defect4J* and *Bugs.jar*, and a newly built bug datasets with a total of +20K real-world bugs in eight projects. We compared DLFix against a total of 13 state-of-the-art pattern-based APR tools. Our results show that DLFix can auto-fix more bugs than 11 of them, and is comparable and complementary to the top two pattern-based APR tools in which there are 7 and 11 unique bugs that they cannot detect, respectively, but we can. Importantly, DLFix is fully automated and data-driven, and does not require hard-coding of bug-fixing patterns as in those tools. We compared DLFix against 4 state-of-the-art deep learning based APR models. DLFix is able to fix 2.5 times more bugs than the best performing baseline.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

Deep Learning; Automated Program Repair; Context-based Code Transformation Learning

ACM Reference Format:
Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: Context-based Code Transformation Learning for Automated Program Repair. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3377811.3380345

*Corresponding Author

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380345

1 INTRODUCTION

Program repairing is a vital activity aiming to fix defects during software development to ensure software quality. It requires much effort, time, and budgets for a software development team. Recognizing the importance of program repairing, several automated techniques/tools have been proposed to help developers in automatically identifying and fixing software defects in programs.

Recently, some approaches aim to *mine and learn fixing patterns* from prior bug fixes [12, 16, 20, 27]. The fixing patterns, also called *fixing templates*, could be automatically or semi-automatically mined [16, 20, 21, 27]. Some approaches are integrated with static and dynamic analysis, and constraint solving to synthesize a patch [20, 27]. Instead of mining buggy and fixed code, other approaches focus on *mining code changes* to generate a similar patch (e.g., CapGen [42], SimFix [13], FixMiner [15]). *Machine learning* has been used to implicitly mine the fixing patterns and/or further rank the candidate fixes based on existing patches (e.g., Prophet [24], Genesis [22]). Other approaches [33] explore *information retrieval* for better selecting/ranking candidate fixes.

With recent advances in deep learning (DL), several researchers have applied DL to automated program repair (APR). The first group of approaches (e.g., DeepFix [10], DeepRepair [43, 44]) leverages the capability of DL models in *learning similar source code for similar fixes*. For example, DeepRepair explores learned code similarities, captured with recursive auto-encoders [44], to select the repair ingredients from code fragments similar to the buggy code. The second group of approaches treats APR as a *statistical machine translation* that translates the buggy code to the fixed code. Rachet [11] and Tufano *et al.* [40] use sequence-to-sequence translation. They use neural network machine translation (NMT) with attention-based Encoder-Decoder, and different code abstractions to generate patches, while SequenceR [6] uses sequence-to-sequence NMT with a copy mechanism [35]. CODIT [5] learns code edits with encoding code structures in an NMT model to generate fixes. Tufano *et al.* [39] learn code changes using sequence-to-sequence NMT with code abstractions and keyword replacing.

Despite their successes, the above neural network machine translation (NMT)-based models still have the following three key limitations in applying to APR. The first issue is that those approaches completely treat APR as a machine translation from buggy to fixed code. Specifically, during training, the *knowledge on what parts of the buggy code have changed and what have not for fixing a defect is not encoded in those NMT-based models for APR.* Instead, those NMT-based models are trained with a parallel corpus of the pairs of the source code of a buggy method and that of the corresponding fixed one. Without such knowledge, the models must learn to implicitly align the source code before and after the fix, and at the same time, to statistically derive the fixing patterns. *The learned*

alignment might be imprecise, making those NMT-based models *incorrectly identify the fixing locations* in the new buggy code.

The second issue of NMT-based APR models is with the sequence-based representation for source code. Source code has well-defined syntax and semantics. The sequence representations and sequence-based DL models are not suitable for capturing code structures. Thus, the process of learning the mappings between the program elements/tokens in the buggy and fixed code must implicitly recover the code structure and map two structures to learn the code transformations corresponding to the bug fix. This could lead to imprecise mappings and eventually incorrect fixes.

The third issue of NMT-based APR models is with how they handle *the context of the code surrounding the fixing locations* where the fixes occur. In general, the fix at a specific location depends on the surrounding code. For example, assuming that a bug fix of adding an operation of closing a file with FileOutputStream.close occurs in the context of the file object being already open and written to. A file closing operation is needed after that. An addition of FileOutputStream.open cannot be a candidate fix because the file was already open. The state-of-the-art approaches [10, 39, 40] using NMT to translate buggy to good code currently have limitations in considering such context. Specifically, they often take the entire methods before and after the fixes as the pairs to train a sequence-to-sequence NMT model. The issue is that the context of entire method contains too much noise that makes the model difficult to correctly align the changed and unchanged code between the buggy and the fixed code. At the meantime, other approaches [5, 6, 11] limit the scope of the input code only to the buggy statement and the corresponding fixed one. With this, the models avoid the noise caused by too much context, however, facing the opposite issue that they lack the contexts to derive the correct fix. Thus, they often pick and rank higher the popular yet incorrect fixes in the corpus, with limited consideration of surrounding code.

To address those challenges, we introduce DLFix, a two-layer tree-based deep learning model to learn code transformations from prior bug fixes to apply to fix a given buggy code. We treat the APR problem as *code transformation learning* (rather than a machine translation problem), in which transformations corresponding to bug fixes including (un)-changed parts are encoded as the input for model training. This helps DLFix to avoid the mapping task. Instead of using sequence-to-sequence NMT, we use a tree-based RNN in which the source code of the method is represented by the corresponding abstract syntax tree (AST), where the changed and un-changed sub-trees are encoded and fed to the model. To address the issue on context, we *separate the learning of the context of surrounding code* of bug fixes from *the learning of the code transformations* for bug fixes with two layers in our model. The buggy sub-tree in the AST of a buggy method is identified and replaced with a summarized node, which encodes the detailed structures of the buggy sub-tree. The non-buggy AST sub-trees together with the summarized node constitute the context and are learned with the RNN model at the context learning layer. The output of the context learning layer is a vector representing the surrounding context.

For the code transformation learning, the changed sub-tree before and after the fix is used for training another tree-based RNN model to learn the bug-fixing code transformations. The context of the transformation computed as the vector in the context learning layer is used as an additional input in this step.

The separation of context learning and transformation learning, and using the result from context learning in the latter process helps DLFix sufficiently consider the surrounding code. This strategy also enables DLFix to learn the context better due to the relative position of the summarized node (the changed sub-tree for a fix) with the other nodes (un-changed sub-trees) of the AST. Moreover, during training, the separation between two learning phases helps DLFix avoid the incorrect alignments between changed code and surrounding context code. Only the changed sub-trees before and after fixing are aligned and DLFix learns the transformations.

We conducted several experiments to evaluate DLFix in two standard bug datasets *Defects4J*, and *Bugs.jar*, and in a newly built bug datasets with a total of +20K real-world bugs in 8 large Java projects. We have compared against a total of 13 state-of-the-art pattern-based APR tools. Our results show that DLFix can auto-fix more bugs than eleven of them, and is comparable and complementary to the top two pattern-based APR tools. However, we can fix 7 and 11 new unique bugs compared with the above top two APR tools. Importantly, DLFix is fully automated and data-driven, and does not require hard-coding of bug-fixing patterns as in those tools. We also compared DLFix against four state-of-the-art deep learning (DL)-based APR models. DLFix is able to detect 2.5 times and 19.8 times more bugs than the best and worst performing baselines, respectively. In this paper, we contribute the following:

A. DL for APR: DLFix is the first DL APR that generates comparable and complementary results with powerful pattern-based tools, as recently published DL-based APR can only fix very few bugs on Defects4J. DLFix helps confirm that further research on building advanced DL to improve APR is promising and valuable.

B. Model: A novel DL-based APR approach with a novel two-layer tree-based model, effective program analysis techniques, and CNN-based re-ranking approach to better identify correct patches.

C. Empirical Results: (Code and data are published [2])

1) **Comparable and Complementary to Pattern-based APR**. We show DLFix can auto-fix more bugs than 11/13 state-of-the-art pattern-based APR tools, and its result is comparable and complementary to the ones from the two best pattern-based tools. DLFix do not require hard-coding of bug-fixing patterns as in those tools. DLFix is fully automatic and data-driven.

2) **Improving over all the DL-based APR**. DLFix is able to detect 2.5 times more bugs than the best performing baseline. DLFix can fix 253 new bugs (out of 1158 in Bugs.jar) than all the other DL-based APR techniques combined.

2 MOTIVATION

2.1 Motivating Example

In this section, we present a real-world example and our observations to motivate our approach.

Figure 1 shows an example of a real-world bug fix in the project *PIG* in the *Bugs.jar* dataset [32]. In the method *deleteDir*, as part of the task of deleting a directory, the string for the command is first built (lines 5–8). A bug occurs when the API call *runCommand* needs to have the third argument as $false$ (an option to indicate no connection to a socket), and it does not need to return an object

```
1       private void deleteDir(String server, String dir) {
2         if (server.equals(LOCAL)){ ...
3         }
4         else {
5           String[] cmdarray = new String[3];
6           cmdarray[0] = "rm";
7           cmdarray[1] = "-rf";
8           cmdarray[2] = dir;
9           try {
10  -         Process p = runCommand(server, cmdarray);
11  +         runCommand(server, cmdarray, false);
12          } catch(Exception e){
13             log.warn("Failed to remove ..." + dir);
14          }
15      ...
16      }
```

Figure 1: A Bug-Fixing Example from Project *PIG* in *Bugs.jar*

Process. The fix is shown at line 11 where the variable p of *Process* was deleted and the new argument $false$ was added.

From this example, we have drawn the following observations:

Observation 1 (Code Structure). The change to fix a bug could range from a simple change to a program entity, to a complex transformation of the code structure in the buggy code. For example, in Figure 1, the fix involves the removal of the variable p and the declaration type *Process*, and the addition of the third argument $false$. In other words, the structure of the code statement was changed. In other cases, the changes might be more complex. Generally, *a bug-fixing change can be viewed as a code transformation applying to the buggy code to transform it to the correct code.*

Observation 2 (Context). A bug fix often depends on the context of its surrounding code. That is, bug fixes might vary for different contexts of surrounding code. For example, the fix in Figure 1 makes sense in the context of the surrounding code when the string command was built (lines 5-8), and the API runCommand was called (line 10), and an exception catching occurs afterward at lines 12–14. The fix might not be the same in a different usage of those objects. Generally, *the decision to choose a specific bug-fixing change needs to consider the context of surrounding code where the fix occurs.*

Table 1: Results from Different NMT-based Approaches on Figure 1. NMT: Neural Machine Translation. NMT_M: NMT at method-level. NMT_S: NMT at statement-level.

Models	Fixed Results
NMT_M [39, 40]	$log.warn("Failed\ to\ remove..."); (line 13)$
NMT_S [5, 11]	$Process\ cmdarray{=}runCommand(server, dir);$

The state-of-the-art approaches for automated program repair, particularly the ones that rely on statistical machine translation (SMT) have dealt with code structure and bug-fixing context in different ways with different results. *NMT_M* [39, 40] uses a neural network-based machine translation model (NMT) that considers an entire method as a sentence consisting of words and learns to translate buggy code into correct code.

While using the entire method body as the context, *NMT* and generally, SMT-based APR approaches [39, 40] have an important limitation in the way that *they treat automated bug fixing as a machine translation problem: during training, a NMT-based model needs to implicit learn the alignments of code elements in a pair of buggy code and its fixed code in order to learn the fixing changes to apply to a new buggy code.* During training, the bug-fixing changes between the buggy and fixed code are known. Instead of equipping a model with the knowledge on such transformations, SMT-based APR approaches [39, 40] do not use the knowledge on the changes and let a model learn the alignments between the buggy and fixed code. Thus, the model might incorrectly align the similar code in different positions in the same method to one another. For example, the lines containing the variable *cmdarray* could be incorrectly aligned to one another in the two versions before and after the fix. Incorrect alignment could lead to incorrectly identifying the fixing locations. As seen in Table 1, *NMT_M* fixes the statement at line 13.

Another issue is that *SMT-based model treats source code as sequences of words* for translation. Source code has well-defined structure and semantics. Bug fixes involve the transformations in code structure, which are disregarded during learning the alignments between the buggy and fixed code in the existing SMT-based approaches. Such mappings between the code structures of buggy and fixed code might not be correctly learned by such models, leading to imprecise alignments and inaccuracy in translation.

To address the issue of incorrect fixing locations, the state-of-the-art NMT-based APR approach [5, 11] restricts the fixing scope to a statement. For example, a fault localization method [3] could be applied first to localize the fix to the statement at line 10. Then, an NMT model is used to translate the buggy statement into a fixed one. However, by limiting the fixing scope, the model cannot leverage the context of the surrounding code of the bug fixes, often leading to incorrect fixes. Despite code abstractions, those models cannot compensate for the lack of context of a bug fix. For example, for the example, as seen in Table 1, keyword NMT_S model made an incorrect fix as in $Process\ cmdarray{=}runCommand(server, dir);$.

2.2 Key Ideas

From the observations, we draw the following key ideas for DLFix:

[Transformation Learning] First, we aim to develop DLFix, a novel deep learning model to learn the code transformations from the previous bug-fixing changes, and apply to fix a given buggy code. The key departure point from the state-of-the-art neural network machine translation (NMT)-based models is that we consider the buggy and fixed code in the same space and during training, we equip our model with the knowledge on code transformations corresponding to prior bug fixes, rather than letting a sequence-to-sequence NMT-based model learn the mappings between the buggy and fixed code. That eliminates the incorrect alignments that could lead to imprecise fixing locations. For example, in Figure 1, for training our model, we encode the transformations from the abstract syntax tree (AST) of the fragment $Process\ p = runCommand(server, cmdarray);$ to that of the fragment $runCommand(server, cmdarray, false);$.

[Explicit Context Learning] Second, the context of the code surrounding a fix remains the same after bug fixing. We encode such code structures surrounding the changes into DLFix as contextual information. For example, the code structures surrounding the fix at line 10 are encoded as the context, and such context is

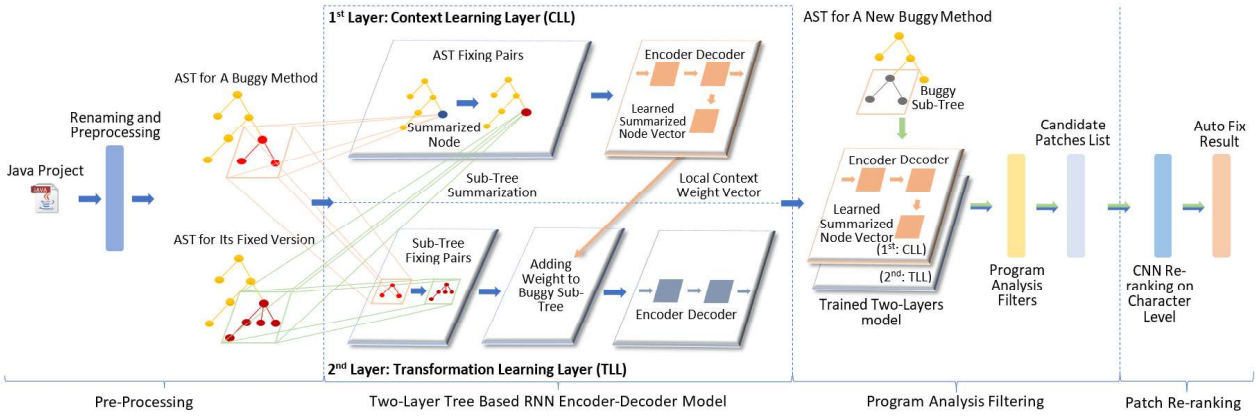


Figure 2: DLFix: Context-based Transformation Learning for Automated Program Repair

treated separately and used as a weight in emphasizing the code transformations from buggy to fixed code. That is our second departure point from existing models that helps DLFix avoid the issues with context. As an NMT_M-based model [39, 40] uses the source code of an entire method as the context for training a translation model, the incorrect alignments could occur. In contrast, an NMT_S model [5, 11] uses only the fixing statement for training faces the issue of lacking the context to correctly learn bug fixes. In comparison with the existing code change learning approach [39], in DLFix, we explicitly represent the context as a vector capturing the surrounding code, while the changes and the contexts are mixed as the input of their model.

[Patch Re-ranking] Third, ideally, a correct patch for a bug should be pushed onto the top-1 position (i.e., top-1 recall) in a list of candidate patches, so that it can be the first to be picked up in the patch validation phrase. Therefore, during the training of a ranking model, the training target should be the top-1 recall, instead of the relatively ranked positions of a correct patch in a list. Our third departure point from existing approaches is that during the training, given a list of candidate patches containing the ground-truth patch for a bug, we build a CNN-based binary classification approach and train it using the ground-truth patch as one positive example and the rest candidate patches as negative ones. Given a new list of patches, we aim to push the correct patch onto the top of the list. In this motivation example, our first two key ideas can help get a group of possible candidates for auto-fix. Our patch re-ranking helps make the right answer at line 11 be the top-1 patch.

2.3 Approach Overview

For training, DLFix consists of the following key steps (Figure 2).

2.3.1 **Pre-Processing**. To prepare to train DLFix, we take the following steps: first, DLFix performs alpha-renaming on the names of variables within a method of a project. This helps it learn from a method to fix in another method since different methods might use different variables' names despite they have similar structures or functionality. Given a method pair (M_b, M_f), M_b is a buggy method and M_f is M_b's fixed version, DLFix uses Word2Vec [26] on the sequence of code tokens to obtain their vector representations for each unique code token in M_b and M_f. DLFix identifies the buggy sub-tree (namely T_b^{sub}) within the AST of M_b, and T_b^{sub}'s corresponding changed sub-tree (namely T_f^{sub}) within the AST of M_f. The root of the changed sub-tree is defined as the common ancestor node of all changed and added nodes in the buggy AST and all changed and inserted nodes in the fixed AST. The final task of this step is to obtain a vector for T_f^{sub}, and another vector for T_b^{sub}. We adopted a DL-based code summarization model [41] to summarize a tree into a vector for a node (called *a summarized node*). The obtained individual vectors for T_b^{sub} and T_f^{sub} are used in the next step of our process.

2.3.2 **Context Learning**. We designed a two-layer learning model that learns the code transformations for bug fixes and the context of the code surrounding the fixes. The first one is dedicated for local **C**ontext **L**earning **L**ayer (CLL). To train our model at this layer, we replace the changed sub-tree with a *summarized node* obtained in the previous step. All other AST nodes that were not changed by the fix are kept the same to provide the context of the code surrounding the fix. The vector for the summarized node is obtained as explained earlier and used in this step. Given pre-processed pairs of methods, we develop a tree-based encoder-decoder model using tree-based LSTMs [37] for this local context learning. We compute the vector for the AST of a method with the summarized node in a pair of methods (M_b, M_f). The obtained vectors are used as weights representing context in the next step, which learns the code transformations for bug fixes.

2.3.3 **Code Transformation Learning**. The second layer is dedicated to code **T**ransformation **L**earning **L**ayer (TLL) for bug-fixing changes. In this TTL, the changed sub-tree before and after the fix is used for training to learn the bug-fixing code transformations. Moreover, the context of the transformation computed as the vector in the context learning layer is used an additional input in this step. Specifically, we use the same tree-based encoder-decoder model in the first layer, CLL, to encode both structural and token changes.

2.3.4 **Program Analysis Filtering**. Next, we setup program analysis filters, including the filter to check the existence of variables,

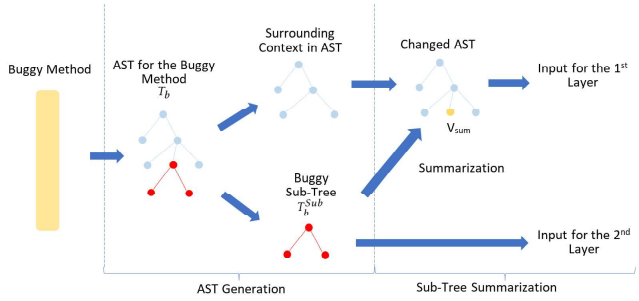


Figure 3: Key Steps of Pre-Processing

methods, and class names, the filter to convert the keywords back to the right names, and the filter to check the syntax of final result. With these filters, DLFix derives the possible candidate fixes.

2.3.5 **Patch Re-ranking**. Finally, we use a Convolutional Neural Network (CNN) [14] based binary classification model to re-rank the generated candidate patches. This module performs more analysis on the detailed code contexts for the buggy statement and aims to push the correct patch onto the top of a list. In this step, we re-rank the list of possible patches based on the detailed contextual information, which helps better selecting the results.

For fixing, as in other DL-based models, a new buggy method M_b and a specific fixing location are the inputs. Our trained model processes M_b with the same steps to produce the candidate fixes.

3 DLFIX: CONTEXT-BASED TRANSFORMATION LEARNING

3.1 Pre-Processing

The goal of pre-processing (Figure 3) is to compute the vector representations for the code that has been changed for bug-fixing as well as the code in the context surrounding the fixing changes. Given method pairs and each method pair containing, M_b (i.e., a buggy method) and M_f (M_b's fixed version), to pre-process them for training, DLFix works in four main steps: renaming, AST generation, token vectors learning, and sub-tree summarization.

3.1.1 Renaming. The goal of this step is to alpha-rename the variables in a method in order to increase the chance for the model to learn the fix in one place to apply to another in the similar/same scenarios because different methods in the same or different projects might use different variable names. In addition, we also keep the type of a variable in the new name to avoid the accidental clashing names. For example, the variable a of the type A in the method M_1 calls the method m (of A) as in $a.m()$. The variable b of the type B in the method M_2 calls the method m (of B) as in $b.m()$. If we do not keep the types A and B of a and b during alpha-renaming, $a.m()$ and $b.m()$, which mean two different operations m in two classes, might get accidentally renamed to the same one, e.g., $v.m()$. In this case, $a.m()$ becomes $v[A].m()$, and $b.m()$ becomes $v[B].m()$. We maintain a data structure of the variables and associated information similar to an entity table in a compiler to recover the actual names.

3.1.2 AST Generation. Next, DLFix generates the ASTs for a method pair, M_b and M_f (i.e., M_b is a buggy method and M_f is M_b's fixed version). Given the fault location information of a method, DLFix detects the root of the changed sub-tree (i.e., the buggy sub-tree) in the AST. It then marks all the nodes under that root. Therefore, for a given method pair, we generate four ASTs: an AST for M_b (namely T_b), an AST for M_f (namely T_f), a buggy sub-tree of T_b (namely T_b^{Sub}), and a changed sub-tree of T_f (namely T_f^{Sub}).

3.1.3 Token Vectors Learning. Next, we aim to compute the vector representations for all of the code tokens within the method pairs in a project. In this step, we use the source code after alpha-renaming. We consider each statement S in the block of statements within a method as a sentence. We collect all the sentences in all the projects in the training corpus. Then, we use Word2Vec [26] on all the sentences in the corpus to obtain the vectors for all the code tokens.

3.1.4 Sub-tree Summarization. The goal of this step is to represent the changed/buggy sub-tree as a single vector representation, which will be used in the local context learning layer. The rationale of representing the entire changed sub-tree as a single node is that in the context learning layer, we focus on the context of surrounding code of the fixing changes. If we include all the nodes in the changed sub-tree for learning the context, those nodes become noises for such learning. At the same time, we do not want to replace that changed/buggy sub-tree with a dummy node because the changes are important as well. Therefore, we decide to encode the buggy sub-tree (T_b^{Sub}) with a vector as well as the changed sub-tree (T_f^{Sub}). We will use them as additional inputs in the process of learning the transformations from the buggy sub-tree to the fixed one.

To achieve that, we adopted an existing model in [41]. The model is capable of representing a sub-tree with a vector by combining the encoding of the tree structure and that of the sequence of tokens belonging to the sub-tree. The authors use a tree-based RNN model to combine with a regular RNN model with a deep reinforcement learning mechanism. Using that combined model, we obtain the vector called a summarized vector V_{sum} to represent a sub-tree.

3.2 Local Context Learning

The goal of the Context Learning Layer, CLL, is to learn local context of the code surrounding the bug-fixing changes (i.e., the unchanged code surrounding the changed one). Specifically, given a pair of ASTs (T_b and T_f for buggy and fixed methods), DLFix first builds changed T_b and T_f in which the buggy and changed sub-trees (T_b^{Sub} and T_f^{Sub}) are replaced by the summarized nodes. Each summarized node is represented by a vector generated in the Step 1.4 (Section 3.1.4). The learned summarized node vector in CLL is computed as the output of the decoder in the auto-fix phase. In training, the summarized nodes for T_b^{Sub} and T_f^{Sub} are given to train the model parameters. For all other nodes in a changed T_b (or T_f) except the summarized node, each node is represented as a vector generated by using Word2Vec in the Step 1.3 (Section 3.1.3). As the buggy and changed sub-trees (T_b^{Sub} and T_f^{Sub}) have different structures and AST nodes, the summarized node vector of T_b^{Sub} should be different from the one of T_f^{Sub}. During the generation of vector embeddings, we set each Word2Vec vector and a summarized node vector to have the same length.

The changed T_b and T_f represents the context of the bug fix for a buggy method. To learn such context, we use the pair of changed T_b and T_f as an input for training an encoder-decoder model with a tree-based RNN model [37] as an encoder and another same tree-based RNN as a decoder. We keep all hidden results from the encoder-decoder model and use them as the context of a fix. The hidden results will be passed down to the Code Transformation Learning Layer (TLL).

The tree-based RNN model [37] is designed to work with tree-structured data. Unlike the regular RNN model that loops for each time-step, the tree-based RNN loops for each sub structure. In DLFix, we use a tree-based LSTM, especially the Child-Sum Tree-LSTM [37] because of the different numbers of child nodes. It is reasonable to directly use Tree-LSTM to model the code for preserving both code structures and sequential syntax, instead of applying normal RNN just on sequences of code tokens. Figure 4 illustrates how the Child-Sum Tree-LSTM models an AST. As seen in Figure 4, Child-Sum Tree-LSTM model considers a parent node and its children nodes each time. As for this, we have

$$h^j_{sum} = \sum_{k \in Child(j)} h_k \tag{1}$$

$$i_j = \sigma(W_i x_j + U_i h^j_{sum} + b_i) \tag{2}$$

$$f_{jk} = \sigma(W_f x_j + U_f h_k + b_f) \tag{3}$$

$$o_j = \sigma(W_o x_j + U_o h^j_{sum} + b_o) \tag{4}$$

$$u_j = tanh(W_u x_j + U_u h^j_{sum} + b_u) \tag{5}$$

$$c_j = i_j \circ u_j + \sum_{k \in Child(j)} f_{jk} \circ c_k \tag{6}$$

$$h_j = o_j \circ tanh(c_j) \tag{7}$$

Where $Child(j)$ is set of the children nodes of the parent node j; h^j_{sum} is the sum of all hidden results from children nodes; i_j is the input/update gate's activation vector; o_j is the output gate's activation vector; f_{jk} is the forget gate's activation vector; c_j is the cell state vector; W, U are weight matrices; b are bias vector parameters; σ is sigmoid function; and $\circ$ denotes the Hadamard product. For example, in Figure 4, the node A has no child nodes, the node A becomes the input X_1 and the A' becomes the output Y_1 after going through the model. Another example is the node F that has children nodes, the node F becomes the input X_1, the E' node becomes the input X_2, the D' become the input X_3, and F' become the output Y_1 to go through the model. In the example of Figure 4, the model first process the nodes A,B, and C in the same level, then goes up to process the nodes E and D in the same level, and lastly, processes the node F. All the nodes go through the model.

3.3 Code Transformation Learning

The goal of the Code Transformation Learning layer (TLL) is to learn the code transformations of the bug fix with the additional input information of the context computed by the Context Learning Layer (CLL). We employ a child-sum tree-based LSTM encoder-decoder model that has the same architecture as the one in CLL (Section 3.2) to learn code transformations. To train such a model, we feed the pairs of sub-trees (i.e., a pair contains a buggy sub-tree, T^{Sub}_b, and its corresponding changed sub-tree, T^{Sub}_f) along with their learned context vectors (W_{lc}) into the tree-based encoder-decoder.

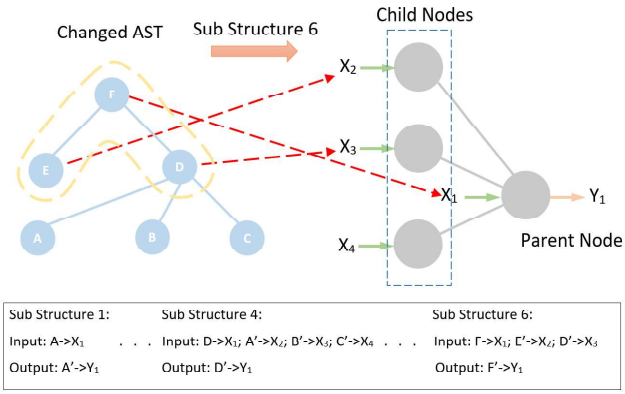


Figure 4: Tree-based LSTM (Note: ′ means the encoded vector for the node and will be used for further training, X_2, X_3, and X_4 include hidden result vector h_j and cell state vector c_j)

Specifically, to integrate the context vector W_{lc} as a weight, we use cross-product multiplication to combine the weight vector with the vector of each AST node in a sub-tree. Cross-product is better than concatenation of vectors, because the local context and detailed node information contains different kinds of information, and cross-product multiplication is more expressive and effective in combining different kinds of information. Next, after performing cross-product, the results are fed into the child-sum tree-based LSTM encoder-decoder model. Once the tree-based LSTM encoder-decoder model is trained, given a buggy sub-tree, this layer TLL automatically generates the fixed sub-tree for the buggy sub-tree.

Both CLL and TLL use the same loss function as defined in *seq2seq* [4]. Even though we used tree-based RNN, the output of each timestep is in the same vector format as *seq2seq* [4].

3.4 Program Analysis Filtering

The goal of the filtering step has two folds. First, DLFix aims to recover the candidate fixes in the source code form, and second, it aims to filter the candidate fixes that violate certain pre-defined program analysis rules. For the first task, DLFix takes the AST_{fix} of a candidate result, and the AST of the entire method as the input, and produces the fixed code. To do that, DLFix first builds the complete AST for the method $AST_{MethFix}$ after the fix by replacing the buggy sub-tree AST_{sub} with the fixed sub-tree AST_{fix}. It then uses the Word2Vec vectors computed in the token vectors learning step (Section 3.1.3) to find the most likely candidates for each token in the $AST_{MethFix}$. Next, DLFix uses the data structure similar to the entity table that it maintains to reverse the alpha-renaming process. Finally, it obtains the list of candidate tokens of the entire method after the fix.

For the second task, we use a set of filters to verify program semantics. First, the syntax checking filter is used. If there is a syntax error, we will go back to change the buggy token to the next possible token and check it again. The second filter is to validate the names of the variables, methods, and classes in the project. If a

name is not valid, DLFix goes back to use the next most likely token in the candidate list at the position and the process is repeated until the newly code passes this filter. The third filter aims to verify if a particular name is correctly referred to. We use our entity table for this task. It repeats the same process as before until finding the right candidate. Finally, as a result, DLFix has a list of fix candidates, which will be used as the input for the re-ranking process.

3.5 Patch Re-ranking

The output of the previous step is a list of possible patches. Each patch is an automatically generated code statement for the bug. The goal of this re-ranking step aims to push the correct patch onto the top of the list. To do so, we propose a Convolutional Neural Network (CNN) based binary classification model. Specifically, given a list of patches, we first process each candidate patch into characters. Next, we use Word2Vec to train vectors for each character. A candidate patch is represented as a set of character vectors. Then, we apply CNN [14] containing one Convolutional layer [7], pooling and fully connected layers, and a softmax function, on the character vectors of a candidate patch to classify the candidate patch into correct or incorrect. Last, we re-rank the given list of patches based on their possibilities of being a correct patch. During the training, given lists of patches with ground-truth correct patches, for each list of patches, we classify the ground-truth patch into one group as a positive example and the rest goes into another group as negative examples. The training target is to achieve the highest number of times a correct patch is placed onto the top-1 position in a list. We apply the CNN on the characters of a patch instead of tokens, as to classify a patch, the character-level contexts of a patch can carry more information than the tokens for classification. Empirically, we also tested the CNN on tokens of patches, and the CNN on characters outperforms the CNN on tokens for classifying patches.

4 EMPIRICAL EVALUATION

We conducted several experiments to evaluate DLFix against the state-of-the-art APR approaches. All experiments were conducted on a desktop with a 4-core Intel CPU and a single GTX Titan GPU.

4.1 Research Questions

To evaluate DLFix, we seek to answer the following questions:

RQ1. Pattern-based Automated Program Repair (APR) Comparative Study. How well does DLFix perform in comparison with the state-of-the-art pattern-based APR approaches?

RQ2. Deep Learning-based APR Comparative Study. How well does DLFix perform in comparison with the state-of-the-art deep learning-based APR approaches?

RQ3. Sensitivity Analysis. How do various factors affect the overall performance of DLFix in APR?

4.2 Experimental Methodology

4.2.1 ***Datasets.*** In this paper, we evaluate approaches on three different datasets: Defects4J [1], Bugs.jar [32], and BigFix (our newly built dataset). BigFix is built from a public dataset [17] for bug detection. The bug detection dataset contains +4.9 million Java methods, and among them, +1.8 million Java methods are buggy. The bug detection dataset contains corresponding bug fixes for the buggy methods. As in the previous studies [13, 33], we evaluate the approaches on fixing one-line bugs. We use the following steps to process the bug detection dataset to build BigFix. We setup several filters to select the appropriate bugs (one-statement bugs) from all the bug reports of a project. The filters include 1) method check filter, which is used to check if the bug is inside of a method, 2) comment check filter, which is to check if the bug is in code statement instead of in comments, 3) one-hunk bug filter which is to check if the buggy position is only one hunk of code and after fixing, if the fixed code is also only one. If the bug passes all these three filters, we mark it as a bug fix and include into the dataset. In total, we collected +20K method pairs with single-hunk bugs. A method pair contains a buggy method and its fixed version.

4.2.2 ***Analysis Approaches for RQs.*** To answer our research questions, we use the following settings.

RQ1. Pattern-based APR Comparative Study.

Comparative Study with Baseline Models. We compare DLFix with 13 pattern-based state-of-the-art APR approaches as listed in Table 2. We ran DLFix on the well-known benchmark dataset Defect4J. Specifically, we trained it on the real bug fixes in BigFix and tested it on Defect4J. We ran it on 101 one-statement bugs in Defect4J as same as the ones in *Tbar* [21]. Note that, each of those bugs can be fixed by at least one previous approaches. As in the previous studies [21, 34, 42], we simply take the results reported in the respective papers, since all of the above approaches have already been well evaluated on Defect4J.

For this RQ, we compare DLFix with the pattern-based APR approaches only on Defect4J due to the following main reason. Generally, search-based baseline models take the *Generate-and-Validate* approach. Therefore, they often require test cases to conduct validation on candidate patches one-by-one. However, BigFix has no test cases and the test cases in Bugs.jar are not consistent among projects. Due to the inconsistency of test cases in Bugs.jar and quite often no published code for the above studied pattern-based approaches, it is hard to apply all approaches on Bugs.jar.

Moreover, because those pattern-based APR approaches have two following additional steps, in this experiment only, we added them into DLFix for comparison:

(1) Fault localization (FL): Conceptually, DLFix can employ any fault localization techniques to produce an ordered list of suspicious statements that require fixes. We chose Ochiai algorithm [3, 29], which has been widely used in APR [13, 15, 18, 42, 45, 46]. After Ochiai localizes a buggy line, all of the AST nodes including intermediate ones that are labeled by the parser with that buggy line are collected into an AST's subtree as a replaced subtree.

(2) Patch Validation: Once DLFix generates a ranked list of candidate patches, we use a validation technique [13, 33] to validate each candidate. Once a candidate patch pases all available test cases, DLFix stops and reports the candidate patch for manual investigation. We report the patches that are exactly matched or semantically equivalent to the ground-truth fixes in Defect4J. Finally, we performed overlapping analysis among the results from the models.

Tuning DLFix. We turned DLFix with the following key hyperparameters using beam-search, such as the vector length of word2vec (100, 150, 200), learning rate (0.001, 0.005, 0.01), and Epoch size (100,

200, 300). We set a 5-hour running-time limit for DLFix to generate candidate patches and validate them as in SimFix [13].

RQ2. Deep Learning-based APR Comparative Study.

Comparative Study with Baseline Models. We compare DLFix with the following state-of-the-art DL-based APR approaches:

1) **Ratchet.** [11] using sequence-to-sequence NMT model;

2) **Tufano et al. (2018)** [40] using encoder-decoder NMT model;

3) **CODIT** [5] using sequence-to-sequence NMT model with some abstractions on tree structures;

4) **Tufano et al. (2019)** [39] using a code change learning approach adopting NMT with some code abstractions and program analysis filtering.

We used all three datasets Defects4J, Bugs.jar, and BigFix for comparison. Deep learning (DL)-based APR approaches do not contain fault localization and patch validation steps. To have a fair comparison and avoid the bias that fault localization can introduce with its false positives [19], we did not run fault localization and patch validation steps for all DL-based APR approaches. We directly provided correct localization information to all DL-based APR approaches and compared the results using the ground-truth fixes from developers. We recorded the results of the models without fault localization and patch validation.

For a comparison on Defect4J and Bugs.jar, we trained all DL-based APR models using BigFix, and tested them on Defect4J (101 bugs) and Bugs.jar, separately. To compare the models on BigFix, we split BigFix at random into 90% for training and 10% for testing.

Qualitative Analysis. For comparison, we also performed qualitative analysis by comparing results from the models on all three datasets. We verified each patch at top-1 position automatically against the ground-truth patch. We computed how many bugs each model can fix among all one-line bugs, how many bugs that can be fixed by other baseline models were covered by our model, how many bugs our model did not cover, and how many new bugs our model can fix when comparing with other baseline models.

Tuning and evaluation metrics. We performed the same tuning process as in RQ1. As in previous APR works, we use the following metrics for evaluation: *Top K is the number of times that a correct patch is in the ranked list of top K candidate patches.*

RQ3. Sensitivity Analysis of DLFix. We evaluate the impacts of the following three main factors on DLFix's performance: (1) two-layer Tree-based Encoder-Decoder model, (2) program analysis techniques, and (3) the re-ranking of candidate patches. To do so, we added each element into the model one by one. We conducted our sensitivity analysis on BigFix and use *Top1* metric.

4.3 Experimental Results

4.3.1 Results of RQ1 (Pattern-based APR Comparative Study). Table 2 shows that DLFix can correctly fix 30 bugs and outperform the most recent pattern-based APR approaches on Defect4J, except for *SimFix* and *TBar*. Comparing with *SimFix*, our fully automatic DLFix without human-crafted fix patterns can generate comparable and complementary results. *TBar* collects all possible fix patterns from the recent APR tools and applies them to fix bugs. Naturally, *TBar* can be considered as a collection of tools and it is reasonable that DLFix fixes fewer bugs than *Tbar* on Defect4J. However, DLFix is data-driven and no hand-crafted pattern is needed.

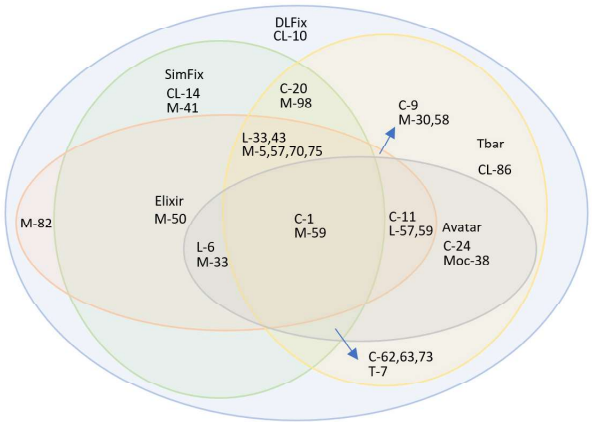


Figure 5: Overlapping Analysis for RQ1. Project names: C:Chart, CL:Closure, L:Lang, M:Math, Moc:Mockito, T:time.

Qualitative Analysis of RQ1. Figure 5 shows the overlapping analysis of the approaches on Defect4J. Due to the page limitation, we only compare with the best APR approaches including *Elixir*, *Avatar*, *SimFix*, and *Tbar*. As seen, DLFix can fix 12, 17, 11, and 7 unique bugs when comparing with *Elixir*, *Avatar*, *SimFix*, and *Tbar*, respectively (i.e., they did not detect those bugs). Specifically, our approach can fix one more new bug, *CL*-10, that cannot be fixed by *Elixir, Avatar, SimFix,* and *Tbar.*

In brief, in comparison with pattern-based approaches, DLFix can obtain comparable results in addition to complementing with them. Furthermore, DLFix is fully automatic and data-driven, and it does not require any human-crafted patterns/templates.

4.3.2 Results of RQ2 (Deep learning-based APR Comparative Study). Table 3 shows that DLFix outperforms the state-of-the-art DL-based APR baselines on all three datasets. Our model improves the baselines by 150.6% and up to 1,980.0% in terms of *Top1*. Specifically, in 39.6% of cases on Defect4J, 34.2% of cases on Bugs.jar, and 29.4% of cases on BigFix, the top-1 ranked candidate patch from DLFix is the ground-true patch, meaning that DLFix can directly generate the correct patches for 40 bugs in Defect4J, 396 bugs in Bugs.jar, and 639 bugs in a testing dataset of BigFix. DLFix outperforms the other DL-based models in every metric. For instance, on BigFix, within 10 best guesses, DLFix can achieve 33.4%, and improve the baselines: Ratchet, Tufano *et al.*('18), CODIT, and Tufano *et al.*('19) by 384.1%, 176.0%, 82.5%, and 56.1%, respectively.

Qualitative Analysis of RQ2. Figure 6 shows the results of overlapping analysis on three datasets using the *Top1* metric. DLFix can fix more new bugs than any other baselines. Specifically, it can fix 27, 253, and 291 more new bugs than all of the other four models combined (i.e., the results from Ratchet + Tufano *et al.*('18) + CODIT + Tufano *et al.*('19)) on Defects4J, Bugs.jar, and BigFix, respectively. Also, there are 13, 145, and 349 bugs that can be fixed by DLFix and can also be fixed by at least one baseline.

4.3.3 Results of RQ3 (Sensitivity Analysis). Table 4 shows that we build three variants of DLFix with different factors and their combinations. We analyze our results as follows:

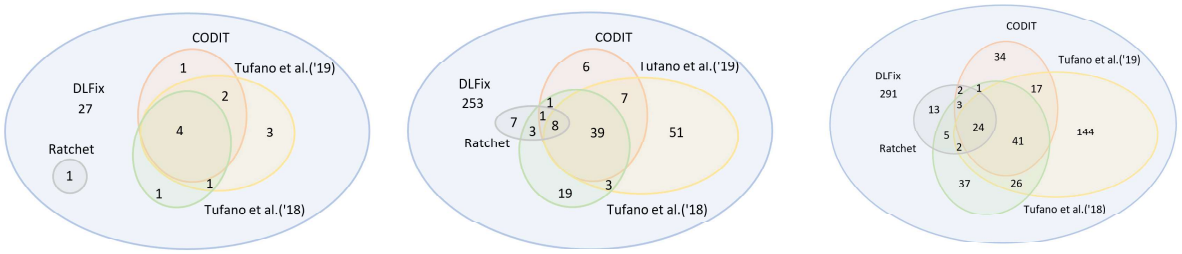
Table 2: RQ1. Comparison with the Pattern-based APR Baselines on Defect4J.

Project	jGenProg	HDRepair	Nopol	ACS	ELIXIR	ssFix	CapGen	SketchFix	FixMiner	LSRepair	AVATAR	SimFix	TBar	DLFix
Chart	0/7	0/2	1/6	2/2	4/7	3/7	4/4	6/8	5/8	3/8	5/12	4/8	**9/14**	5/12
Closure	0/0	0/7	0/0	0/0	0/0	2/11	0/0	3/5	5/5	0/0	**8/12**	6/8	**8/12**	6/10
Lang	0/0	2/6	3/7	3/4	8/12	5/12	5/5	3/4	2/3	8/14	5/11	**9**/13	5/**14**	5/12
Math	5/18	4/7	1/21	12/16	12/19	10/26	12/16	7/8	12/14	7/14	6/13	14/26	**19/36**	12/28
Mockito	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	1/1	**2/2**	0/0	1/**2**	1/1
Time	0/2	0/1	0/1	1/1	**2**/3	0/**4**	0/0	0/1	1/1	0/0	1/3	1/1	1/3	1/2
Total	5/27	6/23	5/35	18/23	26/41	20/60	21/25	19/26	25/31	19/37	27/53	34/56	**43/81**	30/65
P(%)	18.5	26.1	14.3	78.3	63.4	33.3	**84.0**	73.1	80.6	51.4	50.9	60.7	53.1	46.2

Note: P is the probability of the generated plausible patches to be correct.
In the cells, x/y: x means the number of correct fixes and y means the number of candidate patches that can pass all test cases. For example, for DLFix, 65 candidate patches can pass all test cases. However, 30 out of 65 are the correct fixes compared with the fixes in the ground truth.

Table 3: RQ2. Accuracy Comparison with DL-based APR approaches on three Datasets.

Approach	Defect4J (101 Bugs in Testing)			Bugs.jar (1,158 Bugs in Testing)			BigFix (2,176 Bugs in Testing)		
	Top1	Top5	Top10	Top1	Top5	Top10	Top1	Top5	Top10
Ratchet	2.0%	4.0%	6.9%	2.4%	4.4%	6.8%	3.0%	4.1%	6.9%
Tufano et al.('18)	6.9%	9.9%	11.9%	8.4%	11.1%	12.9%	7.9%	10.6%	12.1%
CODIT	8.9%	13.9%	15.8%	7.0%	11.8%	14.8%	6.9%	13.7%	18.3%
Tufano et al.('19)	15.8%	20.8%	23.8%	13.5%	18.7%	23.1%	15.4%	17.3%	21.4%
DLFix	**39.6%**	**43.6%**	**48.5%**	**34.2%**	**36.4%**	**37.9%**	**29.4%**	**31.1%**	**33.4%**



(a) Defects4J **(b) Bugs.jar** **(c) BigFix**

Figure 6: RQ2. Qualitative Analysis Results from DL-based APR Approaches on three Datasets.

Table 4: Sensitive Analysis – Impact of Different Factors on DLFix's Accuracy in terms of Top1 on BigFix Dataset.

Models	Top1	Improvement
Seq2Seq	1.8%	
Seq2Seq + PAT	6.4%	256%
Two-Layer-EDM	11.7%	550%
Two-Layer-EDM + PAT	24.4%	109%
Two-Layer-EDM + PAT + Re-ranking	29.4%	20.5%

Seq2seq: a simple sequence-to-sequence model; Two-Layer-EDM: Two-layer tree-based LSTM encoder-decoder model; PAT: program analysis (PA) Techniques including Renaming and PA Filters.

(1) **Impact of Two-Layer-EDM**. Our Two-Layer-EDM can improve the one-layer sequence-to-sequence model by 550% and using only *seq2seq* cannot get good results. Two-layer-EDM is designed to learn the local context of a bug fix and code transformations.

(2) **Impact of PAT**. Using program analysis (PA) techniques, PAT, including alpha-renaming and PA-filtering, is effective to improve Two-Layer-EDM by 109%. The alpha-renaming process can help improve DLFix for better training and the PA-filtering process can help elminate more the irrelevant patches.

Adding program analysis to the basic *seq2seq* model can improve it by 256%. However, the Two-Layer-EDM can improve seq2seq by 550%. Thus, the Two-Layer-EDM has more impact than PAT.

(3) **Impact of Re-ranking**. The results of *Two-Layer-EDM + PAT + Re-Ranking* show that having re-ranking can increase accuracy relatively by 20.5%. The reason is that the re-ranking process, which uses a Convolutional Layer to distinguish the best result from the others, can help increase DLFix's accuracy by pushing the right results to the top of the list of the candidate fixes.

5 DISCUSSION AND IMPLICATIONS

Let us present in-depth case studies to show why DLFix can work.

```
1     static boolean mayBeString(Node n, boolean recurse) {
2     if (recurse) {
3     -         return allResultsMatch(n, MAY_BE_STRING_PREDICATE);
4     +         return anyResultsMatch(n, MAY_BE_STRING_PREDICATE);
5     } else {
6     ...}
```

Figure 7: Case study 1. The Bug-Fix Example from Project *Closure* with the Bug ID *Closure*-10 in Defects4J

Case Study 1. This case study shows a typical example of a bug using incorrect method call in the code. In Figure 7, the incorrect

method call *allResultsMatch*() was changed into *anyResultsMatch*(). DLFix is data-driven and automatically learns a large amount of code transformation patterns from previous fixes. In this way, DLFix can detect to use *anyResultsMatch*() to replace *allResultsMatch*(). However, the most advanced pattern-based APR approaches, including *Elixir*, *Avatar*, *SimFix*, and *Tbar*, cannot correctly fix this bug, as they rely on human-crafted rules/patterns/templates. If the defined rules/patterns/templates of a tool cannot cover the scenarios of a bug such as the one in this case study, the tool will not be able to automatically fix the bug. In this case, such replacement of a method call requires a pattern-based model to hand-craft the change. For example, *TBar* is so far the best performing APR tool that collects all of the existing patterns in the APR literature, and it still cannot fix this bug. Instead, it attempted to fix the parameters *n* and *MAY_BE_STRING_PREDICATE* for the method call *allResultsMatch*. For *Tbar* to be able to fix this bug, one needs to encode in *Tbar* the rule for the code transformation from *allResultsMatch*() to *anyResultsMatch*().

We believe that combining the well-defined crafted patterns into deep learning-based APR tools is a promising direction, as the crafted patterns can be used as a seed to automatically learn new, high-quality patterns using DL, which can lead to fixing more bugs.

```
1     public double solve(final UnivariateRealFunction f, double min,
            double max, double initial)
2         throws MaxIterationsExceededException,
                FunctionEvaluationException {
3 -       return solve(min, max);
4 +       return solve(f, min, max);
5     }
```

Figure 8: Case study 2. The Bug-Fix Example from Project *Math* with the Bug ID *Math*-70 in Defects4J

Case Study 2. Figure 8 shows a bug-fix example in which the missing parameter f was added into the method. Because this bug fix requires changes to the structure of the statement, to fix the bug, we need to know which part of the statement structure need to be changed and how to change it. DLFix can fix this bug, because it can learn both the surrounding local context (i.e., unchanged code) of a fix and code transformations of the fix. Importantly, we use Tree-based RNN to model code and learn local contexts and code transformations from tree structures, so that DLFix can identify that the method call *solve*() needs an additional parameter.

Some recent deep learning baselines, including Ratchet [11], Tufano *et al.*('18) [40], and Tufano *et al.*('19) [39], use sequence-to-sequence translation model (*seq2seq*) to deal with the bug fixing problem. *seq2seq* takes the statement as a sequence, and learns the relationships between tokens. Therefore, *seq2seq* cannot learn the structure changes during the training, even though several algorithms have been applied to transform the code. Due to the lack of their ability in learning structural changes, the existing models cannot fix this bug because they cannot learn the missing parameter of the method call *solve*().

Another DL-based APR approach, CODIT [5], uses the tree structure to learn code transformations. However, it does not directly model code using tree-based DL model. Instead, it learns rules from ASTs and then applies *seq2seq* to learn code changes. CODIT still suffers the lack of ability in learning structural changes. CODIT cannot get the right number of parameters for the method call *solve*. That is the reason that CODIT cannot fix this bug.

```
1     public Paint getPaint(double value) {
2         double v = Math.max(value, this.lowerBound);
3         v = Math.min(v, this.upperBound);
4 -       int g = (int) ((value - this.lowerBound) / (this.upperBound
5 +       int g = (int) ((v - this.lowerBound) / (this.upperBound
6             - this.lowerBound) * 255.0);
7         return new Color(g, g, g);
8     }
```

Figure 9: Case study 3. The Bug-Fix Example from Project *Chart* with the Bug ID *Chart*-24 in Defects4J

Figure 9 shows a bug-fix example from Defect4J. There are two changes in this fix: 1) the variable *value* was changed to *v*, and 2) the expression in the denominator was modified. We can see that the alpha-renaming and PA-filtering process is useful in this example because PA-filtering enables DLFix to consider the valid variable options at that fixing location and alpha-renaming helps with the renaming the variable into the correct one. Importantly, our tree-based LSTM model helps with the recognition of the code structure and helps the modification of the expression in the denominator of the right-hand side of the assignment.

For this example, the sequence-to-sequence translation models could easily produce a syntactically incorrect fix because they do not consider the code structure. Moreover, they might not be able to rename the variable *value* because they do not have the program analysis component in their solution.

Limitations of Our Approach. Through the manual analysis of the results from DLFix, especially the bugs that it cannot fix, we identify the following limitations:

(1) DLFix does not work well on very unique bugs. DLFix is the deep learning-based approach that needs a large amount of data for training. But even we have a very large dataset, very unique bugs can still exist. Therefore, if there is not sufficiently similar data in the training, DLFix cannot fix the unique bug well.

(2) DLFix does not work well on multiple bugs in one method. Our approach can only deal with one bug at a time.

(3) DLFix only works on one statement bugs and the bug and fix locations have to be the same.

6 THREATS TO VALIDITY

Programming language (PL). Our approach has been tested on Java program repair. However, the techniques used in DLFix are not tied to Java. In principle, our approach can applied to other PLs.

Generalization of the results. Our comparisons with pattern-based APR approaches were only carried out on the Defects4J dataset, which is a widely used benchmark for APR research. Further validation of the comparisons with pattern-based APR baselines on other datasets should necessarily be done in future.

Implementation of the deep learning-based baseline models. We re-implemented CODIT as its code and data is not publicly available. We tried our best to follow steps in CODIT. However, some implementation details are not mentioned in their paper, which makes that our version of CODIT could be slightly different from the one in the original paper. However, we tried our best to build and tune the CODIT on our dataset and this is the best effort

we can make when the code is not publicly available. We tuned DLFix and CODIT both on our dataset. Therefore, the comparison in our study is fair for both.

7 RELATED WORK

Let us discuss the related work on automated program repair (APR). In the earlier stage, the APR approaches aimed to automatically derive *the fixes for similar code* cloned from one place to another [28], or similar code due to porting or branching [31]. Ray and Kim [31] automatically detect similar fixes for similar code that are ported or branched out. The code in different branches is almost similar, thus, the fix can be reused. FixWizard [28] automatically derives *the fixes for similar code* that were cloned from one place to another. It can also work for the *code peers*, which are the code having similar internal and external usages.

A large group of APR approaches has explored *search-based software engineering* to tackle more general types of bugs [8, 9, 25, 30]. A search strategy is performed in the space of potential solutions produced by several operators that mutate the buggy code. Then, the test cases and/or program verification are applied to select the better candidate fixes [36]. GenProg [9] uses genetic search on repair mutations and works at the statement level by inserting, removing or replacing a statement taken from other parts of the same program. RSRepair [30] fixes buggy programs with random search to guide the patch generation process. MutRepair [25] attempts to generate patches by applying mutation operators on suspicious if-condition statements. Smith *et al.* [36] showed that these approaches tend to overfit on test cases, by generating incorrect patches that pass the test cases, mostly by deleting functionalities.

PAR [12] is an APR approach that is based on fixing templates that were manually extracted from 60,000 human-written patches. Later studies (e.g., Le *et al.* [16]) have shown that the six templates used by PAR could fix only a few bugs in Defects4J. Anti-patterns were integrated in existing search-based APR tools [38] (namely, GenProg [9] and SPR [23]) to help alleviate the problem of incorrect or incomplete fixes. A key limitation of those search-based approaches is that they rely much on the quality of mutation operations and the fixing patterns.

In contrast to the search-based approaches, other approaches have aimed to *mine and learn fixing patterns* from prior bug fixes [12, 16, 20, 27]. The fixing patterns, also called fixing templates, could be automatically or semi-automatically mined [16, 20, 21, 27]. SemFix [27] instead uses symbolic execution and constraint solving to synthesize a patch by replacing only the right-hand side of assignments or branch predicates. Long and Rinard proposed Prophet [24], that learns code correctness models from a set of successful human patches. Prophet learns a patch ranking model using machine learning algorithm based on existing patches. Genesis [22] can automatically infer patch generation transformed from developers' submitted patches for automated program repair. HDRepair [16] was proposed to repair bugs by mining closed frequent bug fix patterns from graph-based representations of real bug fixes. ELIXIR [33] uses method call related templates from PAR with local variables, fields, or constants, to construct more expressive repair-expressions that go into synthesizing patches. CapGen [42], SimFix [13], FixMiner [15] are based on the frequently occurred code change operations (e.g., Insert If- Statement) that are extracted from the patches in code change histories. Avatar [20] exploits fix patterns of static analysis violations as ingredients for patch generation.

Deep Learning-based APR approaches. Recently, deep learning (DL) has been applied to APR for directly generating patches. The first group of DL-based APR approaches leverage the capability of DL models in *learning similar source code for similar fixes*. DeepRepair leverages learned code similarities, captured with recursive auto-encoders [44], to select the repair ingredients from code fragments that are similar to the buggy code. DeepFix [10] learns the syntax rules and is evaluated on syntax errors.

The second group of approaches treat APR as a *statistical machine translation* that translates the buggy code to the fixed code. Ratchet [11] and Tufano *et al.* [40] use sequence-to-sequence translation. They use neural network machine translation (NMT) with attention-based Encoder-Decoder, and different code abstractions to generate patches, while SequenceR [6] uses sequence-to-sequence NMT with a copy mechanism [35]. CODIT [5] learns code edits with encoding code structures in an NMT model to recommend fixes. The comparison with these NMT-based APR approaches is provided in the introduction. Recently, Tufano *et al.* [39] learn code changes using sequence-to-sequence NMT with simple code abstractions and keyword replacing. Despite of treating the APR as code transformation learning problem, their approach takes entire method as the context for a bug. Thus, it has too much noise, leading to lower effectiveness than DLFix. In other words, the treatment of context from DLFix helps improve over their model.

8 CONCLUSION

We propose a new deep learning (DL) based automated program repair (APR) approach, namely DLFix to improve and complement the existing state-of-the-art APR approaches. The key ideas that enable our approach (1) using tree-based RNN to directly model code and learning tree-based structural code transformations from previous bug fixes; (2) learning the context of the code surrounding a fix, namely local context learning. In DLFix, we propose a two-layer tree-based RNN encoder-decoder model to learn local contexts and code transformations from previous fixes. In addition, we build a CNN-based classification approach to re-rank possible patches.

We have conducted extensive empirical studies to evaluate DLFix on public benchmarks. DLFix is able to outperform all of the existing state-of-the-art DL APR approaches, and obtains comparable and complementary results against pattern-based APR approaches. Our results show that DLFix can auto-fix more bugs than 11 of the state-of-the-art pattern-based approaches, and is comparable and complementary to the top two pattern-based APR tools. Importantly, DLFix is fully automated and data-driven, and does not require hard-coding of bug-fixing patterns as in those tools. We compared DLFix against four state-of-the-art deep learning-based APR models, and our results show that DLFix is able to fix 2.5 times more bugs than the best baseline in this group of APR models.

ACKNOWLEDGMENTS

This work was supported in part by the US National Science Foundation (NSF) grants CCF-1723215, CCF-1723432, TWC-1723198, CCF-1518897, and CNS-1513263.

REFERENCES

[1] 2019. *The Defects4J Data Set.* https://github.com/rjust/defects4j

[2] 2019. *The GitHub Repository for This Study.* https://github.com/ICSE-2019-AUTOFIX/ICSE-2019-AUTOFIX

[3] Rui Abreu, Peter Zoeteweij, and Arjan J.c. Van Gemund. 2006. An Evaluation of Similarity Coefficients for Software Fault Localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing (PRDC)*. 39–46. https://doi.org/10.1109/PRDC.2006.18

[4] Denny Britz, Anna Goldie, Minh-Thang Luong, and Quoc Le. 2017. Massive exploration of neural machine translation architectures. *arXiv preprint arXiv:1703.03906* (2017).

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

[6] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin 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

[7] Yann Le Cun, Conrad C. Galland, and Geoffrey E. Hinton. 1989. GEMINI: Gradient Estimation through Matrix Inversion after Noise Injection. In *Advances in Neural Information Processing Systems 1*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 141–148.

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

[9] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (Jan 2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[10] 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.

[11] Hideaki Hata, Emad Shihab, and Graham Neubig. 2018. Learning to generate corrective patches using neural machine translation. *arXiv preprint arXiv:1812.07170* (2018).

[12] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ICSE)*. IEEE Press, 802–811.

[13] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. 802–811. https://doi.org/10.1109/ICSE.2013.6606626

[14] Yoon Kim. 2014. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882* (2014).

[15] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2018. Fixminer: Mining relevant fix patterns for automated program repair. *arXiv preprint arXiv:1810.01791* (2018).

[16] Xuan Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. 213–224. https://doi.org/10.1109/SANER.2016.76

[17] Yi Li, Shaohua Wang, Tien N. Nguyen, and Son Van Nguyen. 2019. Improving Bug Detection via Context-Based Code Representation Learning and Attention-Based Neural Networks. *Proc. ACM Program. Lang.* 3, OOPSLA, Article Article 162 (Oct. 2019), 30 pages. https://doi.org/10.1145/3360588

[18] Kui Liu, Anil Koyuncu, , Kisub Kim, Dongsun Kim, and Tegawendé F. Bissyandè. 2018. LSRepair: Live Search of Fix Ingredients for Automated Program Repair. In *Proceedings of the 25th Asia-Pacific Software Engineering Conference (APSEC)*. 658–662. https://doi.org/10.1109/APSEC.2018.00085

[19] Kui Liu, Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems. In *Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 102–113. https://doi.org/10.1109/ICST.2019.00020

[20] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandè. 2019. AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 1–12. https://doi.org/10.1109/SANER.2019.8667970

[21] 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* (Beijing, China) *(ISSTA)*. Association for Computing Machinery, New York, NY, USA, 31–42. https://doi.org/10.1145/3293882.3330577

[22] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic Inference of Code Transforms for Patch Generation. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 727–739. https://doi.org/10.1145/3106237.3106253

[23] Fan Long and Martin Rinard. 2015. Staged Program Repair with Condition Synthesis. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 166–178. https://doi.org/10.1145/2786805.2786811

[24] 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* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 298–312. https://doi.org/10.1145/2837614.2837617

[25] 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

[26] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. *CoRR* abs/1310.4546 (2013). arXiv:1310.4546 http://arxiv.org/abs/1310.4546

[27] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program repair via semantic analysis. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. 772–781. https://doi.org/10.1109/ICSE.2013.6606623

[28] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar Al-Kofahi, and Tien N. Nguyen. 2010. Recurring Bug Fixes in Object-Oriented Programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE)*. Association for Computing Machinery, New York, NY, USA, 315–324. https://doi.org/10.1145/1806799.1806847

[29] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and Improving Fault Localization. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE)*. 609–620. https://doi.org/10.1109/ICSE.2017.62

[30] 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

[31] Baishakhi Ray and Miryung Kim. 2012. A Case Study of Cross-System Porting in Forked Projects. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) *(FSE)*. Association for Computing Machinery, New York, NY, USA, Article Article 53, 11 pages. https://doi.org/10.1145/2393596.2393659

[32] Ripon K. Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R. Prasad. 2018. Bugs.jar: A Large-Scale, Diverse Dataset of Real-World Java Bugs. In *Proceedings of the 15th IEEE/ACM International Conference on Mining Software Repositories (MSR)*. 10–13.

[33] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. 2017. Elixir: Effective object-oriented program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 648–659. https://doi.org/10.1109/ASE.2017.8115675

[34] Seemanta Saha, Ripon K. Saha, and Mukul R. Prasad. 2019. Harnessing Evolution for Multi-Hunk Program Repair. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE)*. 13–24. https://doi.org/10.1109/ICSE.2019.00020

[35] Abigail See, Peter J Liu, and Christopher D Manning. 2017. Get to the point: Summarization with pointer-generator networks. *arXiv preprint arXiv:1704.04368* (2017).

[36] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the Cure Worse than the Disease? Overfitting in Automated Program Repair. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 532–543. https://doi.org/10.1145/2786805.2786825

[37] Kai Sheng Tai, Richard Socher, and Christopher D Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075* (2015).

[38] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad, and Abhik Roychoudhury. 2016. Anti-Patterns in Search-Based Program Repair. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 727–738. https://doi.org/10.1145/2950290.2950295

[39] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On Learning Meaningful Code Changes Via Neural Machine Translation. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE)*. 25–36. https://doi.org/10.1109/ICSE.2019.00021

[40] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. An Empirical Investigation into Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. In *Proceedings of*

the 33rd ACM/IEEE International Conference on Automated Software Engineering (Montpellier, France) *(ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 832–837. https://doi.org/10.1145/3238147.3240732

[41] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving Automatic Source Code Summarization via Deep Reinforcement Learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 397–407. https://doi.org/10.1145/3238147.3238206

[42] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3180155.3180233

[43] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. 2019. Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 479–490. https://doi.org/10.1109/SANER.2019.8668043

[44] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 87–98.

[45] 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* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, 660–670.

[46] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise Condition Synthesis for Program Repair. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE)*. 416–426. https://doi.org/10.1109/ICSE.2017.45