DEAR: A Novel Deep Learning-based Approach for Automated Program Repair

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

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

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

ABSTRACT

The existing deep learning (DL)-based automated program repair (APR) models are limited in fixing general software defects. We present DEAR, a DL-based approach that supports fixing for the general bugs that require dependent changes at once to one or multiple consecutive statements in one or multiple hunks of code. We first design a novel fault localization (FL) technique for multi-hunk, multi-statement fixes that combines traditional spectrum-based (SB) FL with deep learning and data-flow analysis. It takes the buggy statements returned by the SBFL model, detects the buggy hunks to be fixed at once, and expands a buggy statement *s* in a hunk to include other suspicious statements around *s*. We design a two-tier, tree-based LSTM model that incorporates cycle training and uses a divide-and-conquer strategy to learn proper code transformations for fixing multiple statements in the suitable fixing context consisting of surrounding subtrees. We conducted several experiments to evaluate DEAR on three datasets: Defects4J (395 bugs), BigFix (+26k bugs), and CPatMiner (+44k bugs). On Defects4J dataset, DEAR outperforms the baselines from 42%–683% in terms of the number of auto-fixed bugs with only the top-1 patches. On BigFix dataset, it fixes 31–145 more bugs than existing DL-based APR models with the top-1 patches. On CPatMiner dataset, among 667 fixed bugs, there are 169 (25.3%) multi-hunk/multi-statement bugs. DEAR fixes 71 and 164 more bugs, including 52 and 61 more multi-hunk/multi-statement bugs, than the state-of-the-art, DL-based APR models.

CCS CONCEPTS

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

KEYWORDS

Automated Program Repair; Deep Learning; Fault Localization;

ACM Reference Format:
Yi Li, Shaohua Wang, and Tien N. Nguyen. 2022. DEAR: A Novel Deep Learning-based Approach for Automated Program Repair. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510177

*Corresponding Author

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510177

1 INTRODUCTION

Researchers have proposed several approaches to help developers in automatically identifying and fixing the defects in software. Such approaches are referred to as *automated program repair* (APR). The APR approaches have been leveraging various techniques in the areas of *search-based software engineering*, *software mining*, *machine learning (ML)*, and *deep learning (DL)*.

For *search-based approaches* [9, 10, 24, 30], a search strategy is performed in the space of potential solutions produced by mutating the buggy code via operators. Other approaches use software mining to *mine and learn fixing patterns* from prior bug fixes [15, 17, 19, 20, 27] or similar code [28, 32]. Fixing patterns are at the source code level [19, 20] or at the change level [13, 16, 40]. *Machine learning* has been used to mine fixing patterns and the candidate fixes are ranked according to their likelihoods [21, 22, 33]. While some DL-based APR approaches learn similar fixes [11, 41, 42], other ones use machine translation or neural network models with various code abstractions to generate patches [5, 6, 12, 18, 35, 38, 39].

Despite their successes, the state-of-the-art DL-based APR approaches are still limited in fixing the *general defects, which involve the fixing changes to multiple statements in the same or different parts of a file or different files* (which are referred to as *hunks*). None of existing DL-based approaches can automatically fix the bug(s) with dependent changes to multiple statements in multiple hunks at once. They supports fixing only individual statements. If we use such a tool on the current statement, the tool treats that statement as incorrect and treats the other statements as correct. This does not hold since to fix the current statement, the remaining unfixed statements must not be treated as correct code. Thus, it might be inaccurate when using existing DL-based APR tools to fix individual statements for multi-hunk/multi-statement bugs. While DL provides benefits for fix learning, this limitation makes the DL-based APR approaches less capable than the other directions (search-based and pattern-based APR), which support multiple-statement fixes.

In this paper, we aim to advance deep learning-based APR by introducing DEAR, a DL-based model that supports *fixing for the general bugs with dependent changes at once to one or multiple buggy statements belonging to one or multiple buggy hunks of code*. To do that, we make the following key technical contributions.

First, we develop a *fault localization (FL) technique for multi-hunk, multi-statement bugs that combines traditional spectrum-based FL (SBFL) with DL and data-flow analysis*. DEAR uses a SBFL method to identify the ranked list of suspicious buggy statements. Then, it uses that list of buggy statements to *derive the buggy hunks that need to be fixed together* by fine-tuning the pre-trained BERT model [8], to learn the fixing-together relationships among statements. We also design an expansion algorithm that takes a buggy statement

s in a hunk as a seed, and expands to include other suspicious consecutive statements around *s*. To achieve that, we use an RNN model to classify the statements as buggy or not, and use data-flow analysis for adjustment and then form the buggy hunks.

Second, after the expansion step, we have identified all the buggy hunk(s) with buggy statement(s). We develop a *compositional approach to learning and then generating multi-hunk, multi-statement fixes*. In our approach, from the buggy statements, we use a *divide-and-conquer strategy* to learn each subtree transformation in Abstract Syntax Tree (AST). Specifically, we use an AST-based differencing technique to derive the fine-grained, AST-based changes and the mappings between buggy and fixed code in the training data. Those fine-grained subtree mappings help our model avoid incorrect alignments of buggy and fixed code, thus, is more accurate in learning multiple AST subtree transformations of a fix.

Third, we have enhanced and orchestrated a tree-based, two-layer Long Short-Term Memory (LSTM) model [18] with *an attention layer and a cycle training* to help DEAR to learn the proper code fixing changes in the suitable context of surrounding code. For each buggy AST subtree identified by our fault localization, we encode it as a vector representation and apply that LSTM model to derive the fixed code. In the first layer, it learns the fixing context, i.e., the code structures surrounding a buggy AST subtree. In the second layer, it learns the code transformations to fix that buggy subtree using the context as an additional weight.

Finally, there might be likely multiple buggy subtrees. To build the surrounding context for each buggy subtree *B*, in training, we include the AST subtrees *after* the fixes of the other buggy subtrees (rather than those buggy subtrees themselves). The rationale is that the subtrees after fixes actually represent the correct surrounding code for *B*. (Note: in training, the fixed subtrees are known).

We conducted experiments to evaluate DEAR on three large datasets: *Defects4J* [1] (395 bugs), *BigFix* [18] (+26k bugs), and *CPatMiner dataset* [26] (+44k bugs). The baseline DL-based approaches include DLFix [18], CoCoNuT [23], SequenceR [6], Tufano19 [38], CODIT [5], and CURE [14]. DEAR fixes 31% (i.e., +11), 5.6% (i.e., +41), and 9.3% (i.e., +31) more bugs than the best-performing baseline CURE on all three datasets, respectively, using only Top-1 patches and with seven times fewer training parameters on average. On Defects4J, it outperforms those baselines from 42%–683% in terms of the number of fixed bugs. On BigFix, it fixes 31–145 more bugs than those baselines with the top-1 patches. On CPatMiner, among 667 fixed bugs from DEAR, there are 169 (25.3%) multi-hunk/multi-statement ones. DEAR fixes 71, 164, and 41 more bugs, including 52, 61, and 40 more multi-hunk/multi-statement bugs, than existing DL-based APR tools CoCoNuT, DLFix, and CURE. We also compared DEAR against 8 state-of-the-art pattern-based APR tools. Our results show that DEAR generates comparable and complementary results to the top pattern-based APR tools. On Defects4J, DEAR fixes 12 bugs (out of 47) including 7 multi-hunk/multi-statement bugs that the top pattern-based APR tool could not fix.

In brief, the key contributions of this paper include

A. Advancing DL-based APR for general bugs with multi-hunk/multi-statement fixes: DEAR advances DL-based APR for general bugs. We show that DL-based APR can achieve the comparable and complementary results as other APR directions.

B. Advanced DL-based APR Techniques:

```
1  public boolean verifyUserInfo(String UID, String password, String SSN) {
2    String retrieved_password = "";
3    String retrieved_SSN = "";
4    if (UID != null) {
5  -    retrieved_password = getPassword(UID);
6  +     retrieved_password = getPassword(toUpperCase(UID));
7  + } else {
8  +     return false;
9  + }
10 - boolean password_check= compare(password,retrieved_password);
11 + boolean password_check= compare(passwordHash(password),retrieved_password);
12     if (password_check) {
13       retrieved_SSN = getSSN(UID);
14       boolean SSN_check = compare(SSN, retrieved_SSN);
15       if (SSN_check) {
16         return true;
17       }
18     }
19     return false;
20 }
```

Figure 1: A General Fix with Multiple Dependent Changes

1) A *novel FL technique* for multi-hunk, multi-statement fixes that combines spectrum-based FL with DL and data-flow analysis;

2) A *compositional approach* with a *divide-and-conquer strategy* to learn and generate multi-hunk, multi-statement fixes; and

3) The design and orchestration of the two-layer LSTM model with the enhancements via the attention layer and cycle training.

C. Extensive Empirical Evaluation: 1) DEAR outperforms the existing DL-based APR tools; 2) DEAR is the first DL-based APR model performing at the same level in terms of the number of fixed bugs as the state-of-the-art, pattern-based tools and generate complementary results; 3) Our data and tool are publicly available [2].

2 MOTIVATION

2.1 Motivating Example

Let us present a bug-fixing example and our observations for motivation. Figure 1 shows an example of a bug in `verifyUserInfo`, which verifies the given user ID, password and Social Security Number against users' records in the database. This bug manifests in three folds. First, the developer forgot to handle the case when `UID` is `null`. Thus, for fixing, (s)he added an `else` branch at the lines 7–9. Second, the developer forgot to perform the uppercase conversion for the `UID`, causing an error because the records for user IDs in the database all have capital letters. The corresponding bug-fixing change is the addition of the call to `toUpperCase()` on `UID` at line 6. Third, because the passwords stored in the database are encoded via hashing, the input `password` from a user needs to be hashed before it is compared against the one in the database. Thus, the developer added the call to `passwordHash()` on `password` before calling the method `compare()` at line 11. From this example, we have the following observations:

Observation 1 [A Fix with Dependent Changes to Multiple Statements]: This bug requires the *dependent fixing changes to multiple statements at once in the same fix*: 1) adding the `else` branch with the `return` statement (lines 7–9), 2) adding `toUpperCase` at line 6, and 3) adding `passwordHash` at line 11. Making changes to the individual statements one at a time would not fix the bug since both the given arguments `UID` and `password` need to be properly processed. `UID` needs to be null checked and capitalized, and `password` needs to be hashed. Those dependent changes to multiple statements must *occur at once in the same fix* for the program to pass the test cases.

The state-of-the-art DL-based APR approaches [6, 18] *fix one individual statement at a time*. In Figure 1, the fault localization tool returns two buggy lines: line 5 and line 10. Assume that such a DL-based APR tool is used to fix the statement at line 5. It will make the fixing change to the statement at line 5 (e.g., modify line 5 and add lines 7–9), however, with the assumption that the statement at line 10 and other lines are correct. With this incorrect assumption, such a fix will not make the code pass the test cases since both changes must be made. Thus, the *individual-statement, DL-based APR tools cannot fix this bug by fixing one buggy statement at a time*. In general, *a bug might require dependent changes to multiple statements (in possibly multiple hunks) in the same fix*.

Moreover, the pattern-based APR tools might not be able to fix this defect because the code in this example is project-specific and might not match with any bug-fixing patterns.

Observation 2 [Many-to-Many AST Subtree Transformations]: A fix can involve the changes to *multiple* subtrees. For example, the `if` statement has a new `else` branch. The argument of the call to `getPassword()` was modified into the call to `toUpperCase()`. This fix also involves *many-to-many subtree transformations*. In this example, a fix transforms the two buggy statements (line 5 and line 10), into four statements (the `if` statement having new `else` branch, the `return` statement at line 8, the modified statement with `toUpperCase` at line 6, and the modified statement with `passwordHash` at line 11). Thus, *a fix can be broken into multiple subtree transformations*, and if using a *composition approach with a divide-and-conquer strategy*, we can learn the individual transformations.

Observation 3 [Correct Fixing Context]: *A bug fix often depends on the context of surrounding code*. For example, to get the password from a given `UID`, one needs to capitalize the ID, thus, in correct code, the method call to `toUpperCase` is likely to appear when the method call to `getPassword` is made. Therefore, *building correct fixing context is important*. In Figure 1, a model needs to learn the fix (line 5 → line 6) w.r.t. surrounding code, which needs to include the *fixed code* at line 11 (rather line 10 because line 10 is buggy). To fix line 5, the correct context must include `passwordHash` at line 11. Thus, the correct context for a fix to a buggy statement must include the fixed code of another buggy statement *s'*, rather than *s*' itself.

2.2 Key Ideas

From the observations, we draw the following key ideas:

Key Idea 1. A Fault Localization Method for Multi-hunk, Multi-statement Patches: From Observation 1, we design a novel FL method that *combines traditional spectrum-based FL (SBFL) with DL and data-flow analysis*. We use a SBFL to obtain a ranked list of candidate statements to be fixed with their suspiciousness scores. We extend the result from SBFL in two tasks. First, we design a ***hunk-detection algorithm*** to use DL to detect the hunks that need *to be dependently changed together in the same patch*, because SBFL tool returns the suspicious candidates for the fault, but not necessarily to be fixed together. Second, we design ***an expansion algorithm*** that takes each of those detected fixing-together hunks and expands it to include consecutive suspicious statements in the hunk. In Figure 1, the SBFL tool returns line 5 as suspicious. After hunk detection, DEAR uses data dependencies via variable `retrieve_password` to include the statement at line 10 as to be fixed as well.

Key Idea 2. A Compositional Approach to Learning and Generating Multi-hunk, Multi-statement Fixes:

Divide-and-Conquer Strategy in Learning Multi-hunk/Multi-stmt Fixes. To auto-fix a bug with multiple statements, a tool needs to make *m*-to-*n* statement changes, i.e., *m* statements might generally become *n* statements after the fix. A naive approach would let a model learn the code structure changes and make the alignment between the code before and after the fix. Because a fix involves multiple subtree transformations (Observation 2), during training, a model might incorrectly align the code before and after the fix, thus, leading to incorrect learning of the fix. For example, without this step, the model might map `retrieved_password` at line 10 to the same variable at line 6 (the correct map is line 11). Thus, to facilitate learning bug-fixing code transformations, during training, we use *a divide-and-conquer strategy*. We integrate into DEAR a fine-grained AST-based change detection model to map the ASTs before and after the fix. Such mappings enable DEAR to learn the *more local fixing changes* to subtrees. For example, the fine-grained AST change detection can derive that the statements at lines 4–5, and 7 become the statements at lines 4, and 6–9; and the statement at line 10 becomes the one at line 11. We can break them into two groups and align the respective AST subtrees for DEAR to learn.

Compositional Approach in Fixing Multiple Subtrees. We support the fixes having multiple statements in one or multiple hunks by enhancing the design and orchestration of a tree-based LSTM model [18] to *add an attention layer and cycle training* (Section 3.3). While that model fixes one subtree at a time, we need to enhance it to fix multiple AST subtrees at once.

Specifically, we modify its operations in the two-layers to consider multiple buggy subtrees at once. For example, during training, we mark each of the AST subtrees of the statements at line 5 and line 10 before the fix as buggy. At the first layer, for each subtree for a buggy statement, we *replace* it with a pseudo-node, and consider the new AST with its (pseudo-)nodes as the fixing context for the buggy statement. The pseudo-node is computed via an embedding technique to capture the structure of the buggy statement (Section 3.2). At the second layer, DEAR learns the transformation from the subtree for the statement at line 5 into the subtree for the fixed statements at the lines 6–9. The vector for the fixing context learned from the first layer is used as a weight in the code transformation learning in the second layer. We repeat the same process for every buggy statement. For fixing, we perform the composition of the fixing transformations for all buggy statements at once.

Key Idea 3. Transformation Learning with Correct Surrounding Fixing Context: To learn the correct context for a fix to a statement, we need to *train the model* with *the fixed versions of the other buggy statements* (Observation 3). For example, for training, to learn the fix to the statement at line 5 with `toUpperCase`, a model needs to integrate the fixed version of the other buggy line, i.e., the code at line 11 with `passwordHash` as the fixing context (instead of the buggy line 10). If the surrounding code before the fix is used (i.e., line 10), the model will learn the incorrect context to fix the line 5.

2.3 Approach Overview

2.3.1 **Training Process**. The input for training includes the source code before and after a fix (Figure 2), which is parsed into ASTs.

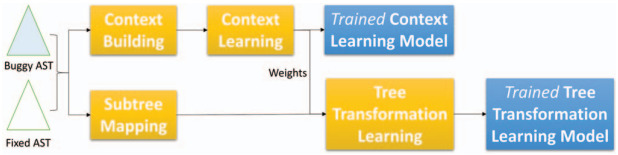

Figure 2: Training Process Overview

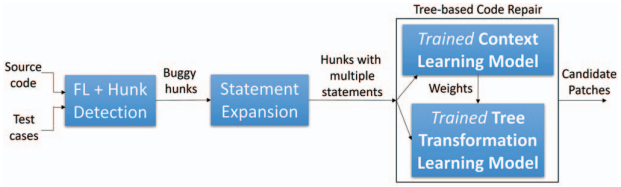

Figure 3: Fixing Process Overview

The output includes the two *trained models for context learning* and *for tree transformation learning* (fixing). The context learning model (CTL) aims to learn the weights (representing the impact of the context) to make an adjustment to the tree transformation learning result. The tree transformation learning model (TTL) aims to learn code transformation for the fix to a buggy AST subtree.

Context Learning (Sections 3.2–3.3). The first step is to build the before-/after-fixing contexts for training. With divide-and-conquer strategy, we use CPatMiner [26] to derive the *changed, inserted*, and *removed* subtrees (key idea 2). As a result, the AST subtrees for the buggy statements are mapped to the respective fixed subtrees. For each buggy subtree and respective fixed subtree, we build two ASTs of the entire method as contexts, one before and one after the fix, and use both of them for training at the input layer and the output layer of the tree-based LSTM context learning model (Section 3.3). To build the correct context for each buggy subtree, we leverage key idea 3: we train our model with the fixed versions of the other buggy subtrees. Finally, the vectors computed from this learning are used as the weights in tree transformation learning.

Tree Transformation Learning (Section 3.4). We first use CPatMiner [26] to derive the subtree mappings. To learn bug-fixing tree transformations, each buggy subtree T itself and its fixed subtree T' after the fix are used at the input layer and the output layer of the second tree-based LSTM for training. Moreover, the weight representing the context computed as the vector in the context learning model is used an additional input in this step.

2.3.2 **Fixing Process**. Figure 3 illustrates the fixing process. The input includes the buggy source code and the set of test cases.

Fault Localization and Buggy-Hunk Detection (Section 4.1). From key idea 1, we first use a SBFL tool to locate buggy statements with suspiciousness scores. Hunk detection algorithm uses those statements to derive the buggy hunks that need to be fixed together.

Multi-Statement Expansion (Section 4.2). Because SBFL might return one statement for a hunk, we aim to expand to potentially include more consecutive buggy statements. To do so, we combine RNN [7] and data-flow analysis to detect more buggy statements.

Tree-based Code Repair (Section 4.3). For the detected buggy statements from multi-statement expansion, we use key idea 2 to derive fixes to multiple buggy subtrees at once. For a buggy subtree T, we build the AST of the method as the context, and use it as the input of the trained context learning model (CTL) to produce the weight representing the impact of the context. The buggy subtree T is used as the input of the trained tree transformation model (TTL) to produce the context-free fixed subtree T'. Finally, that weight is used to adjust T' into the fixed subtree T'' for a candidate patch. We apply grammatical rules and program analysis on the current candidate code to produce the fixed code. We re-rank and validate the fixed code using test cases in the same manner as in DLFix [18].

3 TRAINING PROCESS

3.1 Pairing Buggy and Fixed Subtrees

The training data contains the pairs of the source code of the methods before and after the fixes. Note that a fix might involve multiple methods. Instead of pairing the entire buggy method with the fixed one, we use a divide-and-conquer strategy to help the model to better learn the fixing transformations in the proper contexts. First, we use the CPatMiner tool [26] to derive the fixing changes.

If a subtree corresponds to a statement, we call it *statement subtree*. From the result of CPatMiner, we use the following rules to pair the buggy subtrees with the corresponding fixed subtrees:

1. A buggy subtree (S-subtree) is a subtree with `update` or `delete`.
2. If a S-subtree is `deleted`, we pair it with an empty tree.
3. If a buggy S-subtree is marked as `update`, (i.e, it is *updated* or its children node(s) could be *inserted, deleted* or *updated*), we paired this buggy S-subtree with its corresponding fixed S-subtree.
4. If a S-subtree is `inserted` and its parent node is another S-subtree, we pair it with that parent S-subtree. If the parent node is not an S-subtree, we pair an empty tree to the corresponding inserted S-subtree.

3.2 Context Building

Figure 4 illustrates our context building process. For each pair of the buggy AST I_1 and fixed AST O_1 (Section 3.1), we perform alpha-renaming on the variables. In Step 1, we encode each AST node with the vector using the word embedding model GloVe [29] (which captures well code structure) by considering a statement node as a sentence and each code token as a word. We use those vectors to label the AST nodes in I_1 and O_1. The ASTs after this step are the vectorized ASTs I_2 and O_2, before and after the fix.

In Step 2, we process each pair of the buggy S-subtree T_b in I_2 and the corresponding fixed S-subtree T_f in O_2. First, we perform node summarization on T_b and T_f by using TreeCaps [4] to capture the tree structures of T_b and T_f into V_s and V_s', respectively. Second, for each of the other buggy S-subtrees, e.g., T_b', and their corresponding fixed S-subtrees, e.g., T_f', we process as follows. Because T_f' is the fixed version of T_b', we replace T_b' with T_f' in the building of the resulting context I_3 before fixing (key idea 3). That is, we replace each of the other buggy subtrees with its fixed version. However, to build the resulting context O_3 after fixing, we keep T_f' because it is the fixed subtree, thus, providing the correct context.

The resulting AST, I_3, is used as the before-the-fix context for the buggy S-subtree T_b and used at *the input layer* of the encoder in the context learning model (CTL). The resulting AST, O_3, is used as the after-the-fix context for T_f and used at *the output layer* of the decoder in CTL (Figure 4). Finally, the vectors V_s and V_s' will be used as the weighting inputs for tree transformation learning later.

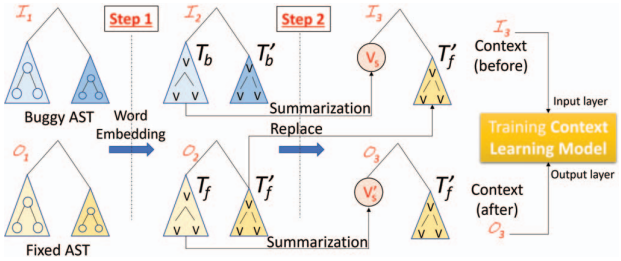

Figure 4: Context Building to Train Context Learning Model

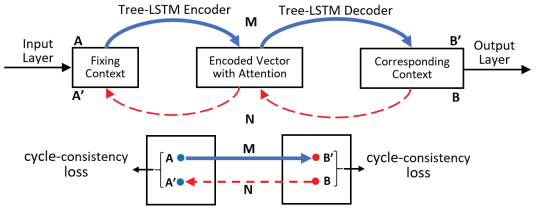

Figure 5: Cycle Training in Attention-based Tree-based LSTM

3.3 Context Learning via Tree-based LSTM with Attention Layer and Cycle Training

For context learning and tree transformation learning, we enhance the two-layer, tree-based, LSTM models in DLFix [18] with attention layer and cycle training. We have added an attention layer into that model, which now has 3 layers: encoder layer, decoder layer and attention layer (Figure 5). For the encoder and decoder, to learn the fixing context expressed in ASTs, we use Child-Sum Tree-based LSTM [36]. Unlike the regular LSTM that loops for each time step, this model loops for each subtree to capture structures.

We also use *cycle training* [45] for further improvement. Cycle training aims to help a model learn better the mapping between the input and output by continuing to train and re-train to emphasize on the mapping between them. This is helpful in the situations in which a buggy code can be fixed in multiple ways into different fixed code, or multiple buggy code can be fixed into one fixed code. This makes the regular tree-based LSTM less accurate. With cycle training, the pair of an input and the most likely output is emphasized to reduce the noise of such one-to-many or many-to-one relations.

Cycle training occurs between encoder and decoder. We use the forward mapping M: $A \rightarrow B$ to denote the process of *encoder* $\rightarrow$ *attention* $\rightarrow$ *decoder*, and the backward mapping N: $B \rightarrow A$ to denote the process of *decoder* $\rightarrow$ *attention* $\rightarrow$ *encoder* (Figure 5). We apply the adversarial losses for both M and N to get the two loss functions $L_{run}(M, D_B, A, B)$ and $L_{run}(N, D_A, B, A)$. The difference between $N(M(A))$ and A, and that between $M(N(B))$ and B are used to generate cycle-consistency loss $L_{cyc}(M, N)$ for M and N to ensure the learned mapping functions are cycle-consistent. Mathematically, we have two loss functions $L_{run}(M, D_B, A, B)$ and $L_{run}(N, D_A, B, A)$. With the incentive cycle consistency loss $L_{cyc}(M, N)$, the overall loss function is computed as follows:

$$L_{cyc}(M,N) = E_{b\sim p_{data}(b)}[||N(M(a)) - a||_1] + E_{a\sim p_{data}(a)}[||M(N(b)) - b||_1] \quad (1)$$

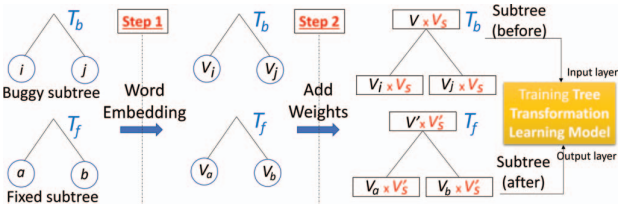

Figure 6: Tree Transformation Learning (V_S, V_S' in Figure 4)

$$L(M,N,D_a,D_b) = L_{run}(M,D_B,A,B) + L_{run}(N,D_A,B,A) + \alpha L_{cyc}(M,N) \quad (2)$$

Where $L(M, N, D_a, D_b)$ is the loss function for the entire cycle training; M and N are the mapping functions to map A to B and B to A; D_A is aimed to distinguish between the predicted result $N(M(A))$ and the real result A; D_B is aimed to distinguish between the predicted result $M(N(B))$ and the real result B; L_{run} is the cycle consistency loss function for the running function M, N; L_{cyc} is the incentivized cycle consistency loss; and α is the parameter to control the relative importance of the two objectives.

3.4 Tree Transformation Learning

Figure 6 illustrates the tree transformation learning process. We use the same tree-based LSTM model with attention layer and cycle training as in Section 3.3 to learn the code transformation for each buggy S-subtree T_b. In Step 1, we build the word embeddings for all the code tokens as in Section 3.2. Each AST node in the buggy S-subtree (T_b) and the fixed one (T_f) is labeled with its vector representation (Figure 6). Next, we use the summarized vectors V_s and V_s' computed from context learning in Figure 4 as the weights and perform cross-product for each vector of the node in the buggy S-subtree T_b and for each one in the fixed S-subtree T_f, respectively. The two resulting subtrees after cross-product are used at the input and output layers of the tree-based LSTM model for tree transformation learning. We use cross-product because we aim to have a vector as the label for a node and use it as a weight representing the context to learn code transformations for bug fixes.

4 FIXING PROCESS

4.1 Fixing-together Hunk Detection Algorithm

The first step of fixing multi-hunk, multi-statement bugs is for our FL method to *detect buggy hunk(s) that are fixed together in the same patch*. To do that, we fine-tune Google's pre-trained BERT model [8] to learn the *fixing-together relationships among statements* using BERT's sentence-pair classification task. Then, we use the fine-tuned BERT model in an algorithm to detect fixing-together hunks. Let us explain our hunk detection algorithm in details.

4.1.1 **Fine-tuning BERT to Learn Fixing-together Relationships among Statements.** We first fine-tune BERT to learn if two statements are needed to be fixed together or not. Let H be a set of the hunks that are fixed together for a bug. The input for the training process is all the sets Hs for all the bugs in the training set.

Step 1. For a pair of hunks H_i and H_j in H, we take every pair of statements S_k and S_l, one from each hunk, and build the vectors

with BERT. We consider the pair of statements (S_k, S_l) as being fixed-together in the same patch to fine-tune BERT.

Step 2. Step 1 is repeated for all the pairs of the statements (S_k, S_l)s in all the pairs H_i and H_j in H. We also repeat Step 1 for all Hs. We use them to fine-tune the BERT model to learn the fixing-together relationships among any two statements in all pairs of hunks.

4.1.2 **Using Fine-tuned BERT for Hunk Detection.** After obtaining the fine-tuned BERT, we use it in determining whether the hunks of code need to be fixed together or not. The input of of this procedure is the fine-tuned BERT model, buggy code P, and test cases. The output is the groups of hunks that need to be fixed together. The process is conducted in the following steps.

Step 1. We use a spectrum-based FL tool (in our experiment, we used Ochiai [3]) to run on the given source code and test cases. It returns the list of buggy statements and suspiciousness scores.

Step 2. The consecutive statements within a method returned by the FL tool are grouped together to form the hunks H_1, H_2, ..., H_m.

Step 3. To decide if a pair of hunks (H_i,H_j) needs to be fixed together, we use the BERT model that was fine-tuned. Specifically, for every pair of statements (S_k,S_l), one from each hunk (H_i,H_j), we use the fine-tuned BERT to measure the fixing-together relationship score for (S_k,S_l). The fixing-together score between H_i and H_j is the average of the scores of all the pairs of statements within H_i and H_j, respectively. If the average score for all the statement pairs is higher than a threshold, we consider (H_i,H_j) as needed to be fixed together. From the pairs of the detected hunks, we build the groups of the fixing-together hunks. The group of hunks that has any statement with the highest Ochiai's suspiciousness score will be ranked and fixed first. The rationale is that such a group contains the most suspicious statement, thus, should be fixed first.

4.2 Multiple-Statement Expansion Algorithm

A detected buggy hunk from the algorithm in Section 4.1 might contain only one statement since each of those suspicious statements is originally derived by a SBFL tool, which does not focus on detecting consecutive buggy statements in a hunk. Thus, in this step, we take the result from the hunk detection algorithm, and expand it to include potentially more statements in a hunk.

Key Idea. Our idea is to combine deep learning with data flow analysis. We first train an RNN model with GRU cells [7] (will be explained in Section 4.2.2) to learn to decide whether a statement is buggy or not. We collect the training data for that model from the real buggy statements. We then use data-flow analysis to adjust the result. Specifically, if a statement is labeled as buggy by the RNN model, no adjustment is needed. However, even when the RNN model decides a given statement s as *non-buggy*, and if s has a data dependency with a buggy statement, we still mark s as *buggy*.

4.2.1 **Expansion Algorithm**. The input of Multi-Statement Expansion algorithm is the buggy statement buggyS, i.e., the seed statement of a hunk. The output is a buggy hunk of consecutive statements.

First, it produces a candidate list of buggy statements by including N statements before and N statements after buggyS (*Expand2NCandidatesList* at line 2). In the current implementation, N=5. Then,

Algorithm 1 Multiple-Statement Expansion Algorithm

```
1: function MultiStatementsExpansion(buggyS)
2:     candStmts = Expand2NCandidatesList(buggyS)
3:     predResult = RNNClassifier(candStmts)
4:     expandResult = DataDepAnalysis(candStmts, predResult)
5:     return expandResult
6: function DataDepAnalysis(buggyS,candStmts, predResult)
7:     buggyHunk = GetCenterBuggyHunk(predResult)
8:     DDExpandHunk(buggyS, buggyHunk, TopHalf(candStmts))
9:     DDExpandHunk(buggyS, buggyHunk, BotHalf(candStmts))
10:    return buggyBlock
11: function DDExpandHunk(buggyS, buggyHunk, candStmts)
12:    for each (stmt ∈ candStmts) & (stmt ∉ buggyHunk) do
13:        if HasDataDep(stmt, buggyS) then
14:            buggyHunk = buggyHunk ∪ stmt
15:        else break
16:    return buggyHunk
```

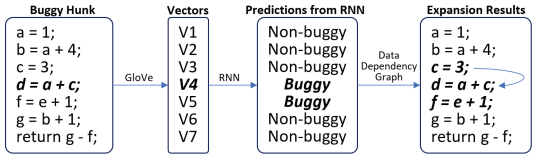

Figure 7: Multiple-Statement Expansion Example

it uses the RNN model to act as a classifier to predict whether each statement (except buggyS) in the candidate list is buggy or not (*RNNClassifier* at line 3). To train that RNN model, we use the buggy statements in the buggy hunks in the training data (see Section 4.2.2). TreeCaps [4] is used to encode the statements.

In *DataDepAnalysis* (line 4), to adjust the results from the RNN model, we obtain *buggyHunk* surrounding the buggy statement buggyS, consisting of the statements before and after buggyS, that were predicted as buggy by the RNN model (line 7). We then examine statement-by-statement in the upward direction from the center buggy statement in the candidate list (line 8, via *TopHalf*) and in the downward direction (line 9, via *BotHalf*). In *DDExpandHunk*, we continue to expand (upward or downward) the current buggy hunk *buggyHunk* to include a statement that is deemed as buggy by the RNN model or has a data dependency with the center buggy statement *buggyS* (lines 13–14). We stop the process (upward or downward), if we encounter a non-buggy statement without data dependency with *buggyS* or we exhaust the list (line 15). Finally, the buggy hunk containing consecutive buggy statements is returned.

In Figure 7, the SBFL tool returns the buggy statement at line 4. All statements are encoded into the sets of vectors via GloVe [29] and classified by the RNN model. We expand from the statement at line 4 upward to include line 3 (even though the RNN model predicted it as non-buggy), since line 3 has a data dependency with the buggy statement at line 4 via the variable c. We include line 5 because the RNN model predicts the line 5 as buggy. At this time, we stop the upward and downward directions because we encounter the non-buggy statements at lines 2 and 6 that do not have data dependency with line 4. That is, lines 1–2 and 6–7 are excluded. The final result includes the statements at lines 3–5 as the buggy hunk.

4.2.2 **Buggy Statement Prediction with RNN.** We present how to use an GRU-based RNN model [7] to predict a buggy statement.

Training. To train the RNN model, we use the buggy/non-buggy statements in all the hunks in the training dataset. We use GloVe [29]

to encode each token in a statement so that a statement is represented by a sequence of token vectors. We use the neural architecture of the GRU-based RNN model [7] to consume the GloVe vectors of statements associated with the buggy/non-buggy labels.

The RNN model operates in the time steps. At the time step k (k>=1), at the input layer, GRU consumes the GloVe vectors of the k^{th} statement S_k. At the output layer, S_k is labeled as 1 if it is a buggy statement and as 0 otherwise. In addition to the input at the time step $k+1$, we feed the output of the time step k to the GRU.

Prediction. The trained GRU-based RNN model is used in Expansion Algorithm at line 3 to predict if a statement in the hunk is buggy or not. The model takes a statement in the form of GloVe token vectors. It takes the vectors of all the statements in a hunk and labels them as buggy or non-buggy in multiple-time-step manner.

4.3 Tree-based Code Repair

Figure 8 illustrates this process. After deriving the buggy hunk(s) in the method(s), DEAR performs code repairing for all buggy statements in all the hunks at once using the trained LSTM models. The tree-based code repair is conducted in the following steps:

Step 1. Identifying buggy S-subtrees. For each hunk, we parse the code into AST, and identify the buggy S-subtrees corresponding to the derived buggy statements. In Figure 8, the S-subtrees T_1 and T_2 are identified as buggy. If a buggy S-subtree is part of another larger buggy S-subtree, we just need to perform fixing on the larger S-subtree since that fix also fixes the smaller S-subtree.

Step 2. Embedding and Summarization. We perform word embedding using GloVe [29] and tree summarization using TreeCaps [4] on all the buggy subtrees to obtain the contexts. For example, in Figure 8, T_1 and T_2 are summarized into two vectors V_1 and V_2.

Step 3. Predict Context. We use the trained context learning model (CTL) to run on the context with the AST nodes including also the summarized nodes to predict the context. In the resulting AST, the structure is the same as the AST for the input context, except that the summarized nodes become the new ones. For example, V_1 and V_2 in the context becomes V_1' and V_2' after Step 3.

Step 4. Adding Weights. The weights V_1 and V_2 from Step 2 are used in a product with the vectors in the buggy subtrees T_1 and T_2. Each node in T_1 and T_2 is represented by a multiplication vector between the original vector of the node and the weight vector V_1' or V_2'.

Step 5. Predict Transformations. We use the trained tree transformation learning model to predict the subtrees T_1' and T_2' of the fix.

Step 6. Removing Weights. We remove the weight from Step 4 to obtain the candidate fixed subtree for a buggy one. For example, we remove V_1' and V_2' to obtain the candidate fixed subtrees T_{1f} and T_{2f}. However, because we know the cross product and a vector, we can get the unlimited number of solutions. Thus, to produce a single solution T_{1f} and T_{2f}, for each node in T_1' and T_2', we assume that V_1' and V_2' are vertical with the node vector V_{n1} in T_1' and V_{n2} in T_2'. Then, we can get the unweighted node vector V_{n1}' in T_{1f} as follows:

$$V_{n1}' = \frac{V_1' \times V_{n1}}{V_1' \dot{V}_1'} \quad (3)$$

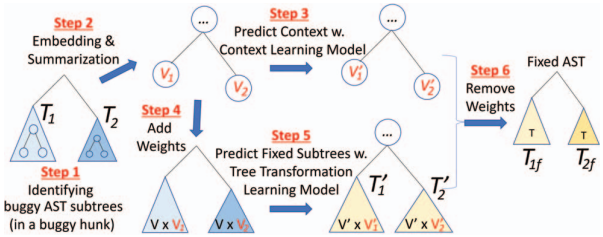

Figure 8: Tree-based Code Repair

After having the vector for each node in a fixed S-subtree T_{1f}, we generated candidate patches based on word embedding. For each node, we calculated the cosine similarity score between its vector V_{n1}' and each vector in the vector list for all tokens. To generate candidate patches, we select the token t in the list. Token t has its similarity score $Score_t$ for one node in the fixed S-subtree. By adding all $Score_t$ for all the tokens, we have the total score, $Score_{sum}$, for a candidate. We select the top-5 candidates for each node to generate the candidates and sort them based on $Score_{sum}$.

4.4 Post-processing

A naive approach would face combinatorial explosion in forming the candidate fix(es) because for each node in the sub-tree, we maintain top-5 candidates. However, when we combine the candidates for all the nodes in the fixed sub-tree, many candidates are not valid for the current method in the project. Therefore, when we form a candidate by combining all the candidates for the nodes, we apply a set of filters to verify the program semantics in the same manner as in DLFix [18]. This allows us to eliminate invalid candidates immediately. Specifically, we use the alpha-renaming filter to change the names back to the normal Java code using a dictionary containing all the valid names in the scope, the syntax-checking filter to remove the candidates with syntax errors, and the name validation filter to check the validity of the variables, methods, and classes. Moreover, for further improvement, we use beam search to maintain only the top-ranked candidate fixes. Thus, we do not exhaust all compositions in forming the statements. This helps maintain a manageable number of candidates.

After applying all the filters, we also used DLFix [18]'s re-ranking scheme on the candidate patches. We then used test cases to conduct patch validation on those candidates. We verify each patch from the top to the bottom until a correct patch is identified and the patch validation ends. If all candidates for fixing a location cannot pass all the test cases, we select the next location to repeat the process.

5 EMPIRICAL EVALUATION

5.1 Research Questions

To evaluate DEAR, we seek to answer the following questions:
RQ1. Comparative Study with Deep Learning-based APR models on Defects4J benchmark. How well does DEAR perform in comparison with existing *DL-based* APR models on *Defects4J*?
RQ2. Comparative Study with Deep Learning-based APR models on Large Bug Datasets. How well does DEAR perform in comparison with *DL-based* APR models on *large-scale bug datasets*?

RQ3. Comparative Study with Pattern-based APR approaches on Defects4J. How well does DEAR perform in comparison with the state-of-the-art, *pattern-based* APR approaches?
RQ4. Sensitivity Analysis of DEAR. How do various factors affect the overall performance of DEAR in APR?
RQ5. Time Complexity and Model's Training Parameters. What is time complexity and the numbers of training parameters?

5.2 Data Collection

We have conducted our empirical evaluation on three datasets:
1) *Defects4J* v1.2.0 [1] with 395 bugs with test cases;
2) *BigFix* [18] with +26k bugs in +1.8 million buggy methods;
3) *CPatMiner* [26] with +44k bugs from 5,832 Java projects.

All experiments were conducted on a workstation with a 8-core Intel CPU and a single GTX Titan GPU.

5.3 Experimental Methodology

5.3.1 **RQ1. Comparison with DL-based APR on Defects4J.** *Comparative Baselines.* We compare DEAR with five state-of-the-art DL-based APR models: **DLFix** [18], **CoCoNuT** [23], **SequenceR** [6], **Tufano19** [38], **CODIT** [5], and **CURE** [14].

Procedure and Settings. We replicated all DL-based APRs except CURE, which is unavailable. We re-implemented CURE following the details in their paper. We trained all DL approaches on the bugs and fixes in CPatMiner dataset and tested them on all 395 bugs in Defects4J (no overlap between the two datasets). All DL approaches were applied with the same fault localization tool, Ochiai [3], and patch validation with the test cases in Defects4J. Following prior experiments [13, 18], we set a 5-hour running-time limit for a tool for patch generation and validation.

We tuned DEAR with the following key hyper-parameters using the beam-search: (1) BERT for hunk detection: epoch size (e-size) (2, 3, 4, 5), batch size (b-size) (8, 16, 32, 64), and learning rate (l-rate) ($3e^{-4}$, $1e^{-4}$, $5e^{-5}$, $3e^{-5}$, $1e^{-5}$); (2) LSTM for Multi-Statement Expansion and code repair: e-size (100, 150, 200, 250), b-size (32, 64, 128, 256), and l-rate (0.0001, 0.0005, 0.001, 0.003, 0.005); (3) GloVe for representation vectors: vector size (v-size) (100, 150, 200, 250), l-rate (0.001, 0.003, 0.005, 0.01), b-size (32, 64, 128, 256), and e-size (100, 150, 200, 250). The other default parameters were used.

The best setting for DEAR is (1) e-size=4, b-size=32, l-rate=$1e^{-4}$ for BERT; (2) e-size=200, l-rate=0.003, b-size=128 for LSTM; (3) v-size=200, l-rate=0.001, b-size=64, e-size=200 for GloVe. For other models, we tuned with the parameters in their papers, e.g., the vector length of word2vec, learning rate, and epoch size to find the best parameters for each dataset. We tuned all approaches with the aforementioned parameters on the same *CPatMiner* dataset to obtain the best performance. Once we obtained the best parameters for each model, we used them for later experiments.

Quantitative Analysis. We report the numbers of bugs that a model can auto-fix for the following bug-location types:

Type-1. One-Hunk, One-Statement: A bug with the fix involving only one hunk with one single statement.

Type-2. One-Hunk, Multi-Statements: A bug with the fix involving only one hunk with multiple statements.

Type-3. Multi-Hunks, One-Statement: A bug with the fix involving multiple hunks; each hunk with one fixed statement.

Type-4. Multi-Hunks, Multi-Statements: A bug with the fix involving multi-hunks; each hunk has multiple statements.

Type-5. Multi-Hunks, Mix-Statements: A bug with the fix involving multiple hunks, and some hunks have one statement and other hunks have multiple statements.

Evaluation Metrics. We report the *number of bugs* that can be correctly fixed and the number of plausible patches (i.e., passing all test cases, but not the actual fixes) using the *top candidate patches*.

5.3.2 **RQ2. Comparison with DL-based APR on Large Datasets.** *Comparative Baselines.* We compare DEAR with the same baselines as in RQ1 on two large datasets: BigFix and CPatMiner.

Procedure and Settings. First, we evaluated all DL-based APR models on BigFix and CPatMiner. Following DLFix and Sequencer, we randomly split data into 80%/10%/10% for training, tuning, and testing. Second, we have cross-dataset evaluation: training DL-based approaches on CPatMiner and testing on BigFix, and vice versa. Unlike Defects4J, BigFix and CPatMiner datasets do not have test cases. Without test cases, we cannot use fault localization and patch validation for all DL approaches. Thus, we fed the actual bug locations into the DL models, including locations on buggy hunks and statements. The DL-based baselines do not distinguish hunks, instead *process each buggy statement at a time*. We use developers' actual fixes as the ground truth to evaluate the DL-based approaches.

Evaluation Metrics. We use the *top-K metric*, defined as the ratio between the number of times that a correct patch is in a ranked list of the top K candidates over the total number of bugs.

5.3.3 **RQ3. Comparison with Pattern-based APR on Defects4J.** *Comparative Baselines.* We compare DEAR with the state-of-the-art, pattern-based APR tools on Defects4J: **Elixir** [33], **ssFix** [43], **CapGen** [40], **FixMiner** [16], **Avatar** [19], **Hercules** [34], **SimFix** [13], and **Tbar** [20]. We were able to replicate the following pattern-based baselines: **Elixir, ssFix, FixMiner, SimFix, TBar** under the same computing environments. We set the time limit to 5 hours for the tools. For the other baselines, due to unavailable code, we use the results reported in their papers as they were run on the same dataset. We used the same setting and evaluation metric.

5.3.4 **RQ4. Sensitivity Analysis**. We evaluate the impacts of different factors on DEAR's performance. We consider the following: (1) hunk detection (Hunk); (2) multi-statement expansion (Expansion); (3) multi-statement tree model and cycle training; and (4) data splitting scheme. We use the left-one-out strategy for each factor. We evaluate the first three factors on Defects4J and the last one on CPatMiner since we need a larger dataset for various splitting.

5.3.5 **Time Complexity and Numbers of Parameters in Model Training**. We measure the training and fixing time for a model and its number of parameters for model training on the datasets.

6 EMPIRICAL RESULTS

6.1 RQ1. Comparison Results with DL-based APR Models on Defects4J

6.1.1 **With Fault Localization**. We first evaluate the APR models when using with the fault localization tool Ochiai [3]. Tables 1 and 2 show the comparison results among DEAR and the baseline models.

Table 1: RQ1. Comparison with DL-based APR Models on Defects4J with Fault Localization

Projects	Chart	Closure	Lang	Math	Mockito	Time	Total
Sequencer	3/3	4/5	2/2	6/9	0/0	0/0	15/19
CODIT	1/2	2/5	0/0	3/5	0/0	0/0	6/12
Tufano19	3/4	3/5	1/1	6/8	0/0	0/0	14/18
DLFix	5/12	6/10	5/12	12/18	1/1	1/2	30/55
CoCoNuT	6/11	6/9	5/13	13/21	2/2	1/1	33/57
CURE	6/13	6/10	5/14	16/23	2/2	1/2	36/71
DEAR	8/16	7/11	8/15	20/33	1/2	3/6	**47/91**

X/Y: are the numbers of correct and plausible patches, respectively.

Table 2: RQ1. Detailed Comparison with DL-based APR Models on Defects4J with Fault Localization

Bug Types	DLFix	CoCoNuT	CURE	DEAR
Type 1. One-Hunk One-Stmt	30	33	36	29
Type 2. One-Hunk Multi-Stmts	0	0	0	4
Type 3. Multi-Hunks One-Stmt	0	0	0	11
Type 4. Multi-Hunks Multi-Stmts	0	0	0	1
Type 5. Multi-Hunks Mix-Stmts	0	0	0	2
Total	30	33	**36**	**47**

As seen in Table 1, DEAR can auto-fix the most number of bugs (47) and generate the most number of plausible patches (91) that pass all test cases on Defects4J. Particularly, DEAR can auto-fix 32, 41, 33, 17, 14, and 11 more bugs than Sequencer, CODIT, Tufano19, DLFix, CoCoNuT, and CURE, respectively (i.e., 213%, 683%, 236%, 57%, 42%, and 31% relative improvements). Compared with those tools in that order on Defects4J, DEAR can auto-fix 35, 34, 41, 18, 31, and 18 bugs that those tools missed, respectively. Via the overlapping analysis between the result of DEAR and those of the baselines combined, DEAR can fix 18 unique bugs that they missed.

Table 2 shows the comparison between DEAR and the top DL-based baselines (DLFix, CoCoNuT, CURE) w.r.t. different bug types.

For single-hunk bugs (Types 1-2), DEAR fixes 33 bugs including 4 unique single hunk bugs that the other tools missed.

For multi-hunk bugs (Types 3–5), DEAR can fix 14 bugs that cannot be fixed by DLFix, CoCoNuT, and CURE. Existing DL-based APR models cannot fix those bugs since the mechanism of fixing one statement at a time does not work on the bugs that require the fixes with dependent changes to multiple statements at once. Thus, they do not produce correct patches for those cases.

For multi-hunk or multi-statement bugs (Types 2–5), DEAR fixes 18 of them (out of 47 fixed bugs, i.e., 38.3% of total fixed bugs).

6.1.2 **Without Fault Localization**. We also compared DEAR with other tools in the fixing capabilities without the impact of a third-party FL tool. All the tools under comparison (Table 3) were pointed to the correct fixing locations and performed the fixes. As seen, if the fixing locations are known, DEAR's fixing capability is also higher than those baselines (53 bugs versus 44, 40, and 48). Importantly, it can fix 20 multi-hunk/multi-statement bugs (37.7% of a total of 53 fixed bugs), while CoCoNuT, DLFix, and CURE can fix only 7, 5, and 10 such bugs.

DEAR is more general than existing DL-based models because it can support dependent fixes with multi-hunks or multi-statements. Importantly, *it significantly improves these DL-based models and raises the DL direction to the same level as the other APR directions (search-based and pattern-based)*, which can handle multi-statement bugs. Moreover, DEAR is fully data-driven and does not require the defined fixing patterns as in the pattern-based APR models.

Table 3: RQ1. Comparison with DL-based APR Models on Defects4J without Fault Localization (i.e., Correct Location)

Bug Types	DLFix	CoCoNuT	CURE	DEAR
Type 1. One-Hunk One-Stmt	35	37	38	33
Type 2. One-Hunk Multi-Stmts	1	3	3	4
Type 3. Multi-Hunks One-Stmt	4	3	6	13
Type 4. Multi-Hunks Multi-Stmts	0	0	0	1
Type 5. Multi-Hunks Mix-Stmts	0	1	1	2
Total	40	44	48	**53**

Table 4: RQ2. Comparison with DL APRs on Large Datasets

	CPatMiner (4,415 tested bugs)			BigFix (2,594 tested bugs)		
Tool/Dataset	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5
Sequencer	7.8%	8.9%	10.3%	8.5%	9.1%	10.8%
CODIT	4.5%	7.4%	9.2%	3.9%	6.3%	9.1%
Tufano19	8.6%	9.3%	11.2%	7.7%	8.8%	9.6%
DLFix	11.4%	12.3%	13.1%	11.2%	11.9%	12.5%
CoCoNuT	13.5%	14.7%	15.3%	12.2%	13.6%	14.3%
CURE	14.2%	15.1%	15.5%	12.9%	14.2%	14.1%
DEAR	**15.1%**	**15.6%**	**16.8%**	**14.1%**	**15.4%**	**16.3%**

Table 5: RQ2. Comparison with DL APRs on Cross-Datasets

Tool/Dataset	CPatMiner(Train)/BigFix			BigFix(Train)/CPatMiner		
	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5
Sequencer	5.4%	5.8%	6.2%	5.3%	6.1%	7.2%
CODIT	2.5%	4.0%	4.4%	3.2%	5.2%	6.4%
Tufano19	4.5%	5.4%	5.7%	5.9%	6.3%	7.6%
DLFix	6.3%	6.9%	7.3%	8.2%	8.7%	9.2%
CoCoNuT	6.7%	7.4%	8.1%	8.3%	9.6%	10.7%
CURE	7.1%	7.7%	8.2%	8.7%	9.9%	10.9%
DEAR	**7.5%**	**8.1%**	**8.6%**	**9.6%**	**10.2%**	**11.3%**

6.2 RQ2. Comparison Results with DL-based APR Models on Large Datasets

Table 4 shows that DEAR can fix more bugs than any DL-based APR baselines on the two large datasets. Using the top-1 patches, DEAR can fix 15.1% of the total 4,415 bugs in CPatMiner. It fixes 40–322 more bugs than the baselines with top-1 patches. On BigFix, it can fix 14.1% of the total 2,594 bugs with the top-1 patches. It can fix 31–145 more bugs than those baselines with the top-1 patches.

Table 5 shows that DEAR also outperformed the baselines in the cross-dataset setting in which we trained the models on CPatMiner and tested them on BigFix and vice versa.

Table 6 shows the detailed comparative results on CPatMiner w.r.t. different bug types. As seen, DEAR can *auto-fix more bugs on every type of bug locations on the two large datasets*. Among 667 fixed bugs, DEAR has fixed 169 multi-hunk or multi-stmt bugs of Types 2–5 (i.e., 25.33% of the total fixed bugs). DEAR fixes more bugs (71, 164, and 41 more), and fixes more bugs in each bug type than the baselines CoCoNuT, DLFix, and CURE, respectively.

Table 6: RQ2. Detailed Analysis. Top-1 Result Comparison with DL-based APR Models on CPatMiner Dataset

Types (#bugs)	CoCoNuT	CURE	DLFix	DEAR
	#Fixed	#Fixed	#Fixed	#Fixed
Type-1 (1,668)	28.7% (479)	29.8%(497)	23.7% (395)	29.9% (499)
Type-2 (530)	1.3% (7)	2.1% (11)	0.6% (3)	4.2% (22)
Type-3 (879)	12.4% (109)	13.2% (116)	11.8% (104)	13.7% (120)
Type-4 (1,089)	0% (0)	0% (0)	0% (0)	2.0% (22)
Type-5 (249)	0.4% (1)	0.8% (2)	0.4% (1)	2.0% (5)
Total (4,415)	13.5% (596)	14.2% (626)	11.4% (503)	**15.1% (667)**

Table 7: RQ3. Comparison with Pattern-based APR Models

Projects	Chart	Closure	Lang	Math	Mockito	Time	Total
ssFix	3/7	2/11	5/12	10/26	0/0	0/4	20/60
CapGen	4/4	0/0	5/5	12/16	0/0	0/0	21/25
FixMiner	5/8	5/5	2/3	12/14	0/0	1/1	25/31
ELIXIR	4/7	0/0	8/12	12/19	0/0	2/3	26/41
AVATAR	5/12	8/12	5/11	6/13	2/2	1/3	27/53
SimFix	4/8	6/8	9/13	14/26	0/0	1/1	34/56
Tbar	9/14	8/12	5/13	19/36	1/2	1/3	43/81
Hercules	8/10	8/13	10/15	20/29	0/0	3/5	49/72
DEAR	8/16	7/11	8/15	20/41	1/2	3/6	**47/91**

X/Y: are the numbers of correct and plausible patches; Dataset: Defects4J

DEAR fixes 52, 61, and 40 more multi-hunk/multi-stmt bugs, and 20, 104, and 2 more one-hunk/one-stmt bugs than CoCoNuT, DLFix, and CURE. For the multi-statement bugs (Types 2 and 5) that the other tools fixed, the fixed statements are independent. This result shows that fixing each individual statement at a time does not work.

6.3 RQ3. Comparison Results with Pattern-based APR Models

As seen in Table 7, DEAR performs at the same level in terms of the number of bugs as the top pattern-based tools Hercules and Tbar.

Table 8 displays the details of the comparison w.r.t. different bug types. As seen, DEAR fixes 7 Multi/Mix-Statement bugs (Types 2, 4–5) that Hercules missed. Investigating further, we found that Hercules is designed to fix *replicated bugs*, i.e., the hunks must have similar statements. Those 7 bugs are non-replicated, i.e., the buggy hunks have different buggy statements or a buggy hunk has multiple non-similar buggy statements. For Types 1 and 3, DEAR fixes 9 less one-statement bugs than Hercules due to its incorrect fixes. In total, DEAR fixes *12* bugs that Hercules misses: *Chart-7,16,20,24; Time-7; Closure-6,10,40; Lang-10; Math-41,50,91.*

Compared to Tbar, DEAR fixes 15 more multi-hunk/multi-stmt bugs. Tbar is not designed to fix multi-statements at once as DEAR. Instead, it fixes one statement at a time, thus, does not work well when those 15 bugs require dependent fixes to multiple statements. The 3 bugs of Type 2 that Tbar can fix are the ones that the fixes to individual statements are independent. The same reason is applied to SimFix. Tbar fixes 11 more correct one-hunk/one-statement bugs.

In brief, we raise DEAR, a DL-based model, to the *comparable and complementary* level with those pattern-based APR models.

6.4 RQ4. Sensitivity Analysis

6.4.1 Impact of Fixing-together Hunk Detection. As seen in Table 9, without hunk detection, DEAR can auto-fix 35 bugs. With hunk detection, DEAR can fix 14 more multi-hunk bugs (Types 3-5). It fixes two less Type-1 bugs due to the incorrect hunk detection. In brief, hunk detection is useful since the multi-hunk/multi-statement bugs require dependent fixes to multiple hunks at once.

Table 8: RQ3. Detailed Comparison with Pattern-based APRs

Bug Types	SimFix	Tbar	Hercules	DEAR
Type 1. One-Hunk One-Stmt	30	40	34	29
Type 2. One-Hunk Multi-Stmts	1	3	0	4
Type 3. Multi-Hunks One-Stmt	3	0	15	11
Type 4. Multi-Hunks Multi-Stmts	0	0	0	1
Type 5. Multi-Hunks Mix-Stmts	0	0	0	2
Total	34	43	49	47

Table 9: RQ4. Sensitivity Analysis on Defects4J

Variant	Without Hunk-Det	Without Expansion	Without Attention-cycle	DEAR
Type-1	31	30	26	29
Type-2	4	0	2	4
Type-3	0	13	9	11
Type-4	0	0	1	1
Type-5	0	0	2	2
Total	35	43	40	47

6.4.2 Impact of Multi-Statement Expansion. As seen in Table 9, without expansion, DEAR fixes *43* bugs in Defects4J. With expansion, it fixes *7* more multi-stmt bugs in Types 2,4,5 while it fixes two less Type-3 bugs and one less Type-1 bug. The reason of fixing less bugs in these two types is that the multi-statement expansion may expand the buggy hunk incorrectly by regarding a single-statement bug as a multi-statement bug. Even so, DEAR still can fix more bugs, showing the usefulness of the multi-statement expansion.

To compare the impact of Hunk Detection and Multi-Statement Expansion, let us note that the variant of DEAR without Hunk-Detection missed all 14 multi-hunk bugs (Types 3,4,5). The variant without Expansion missed all 7 multi-statement bugs (Types 2,4,5). However, let us consider how challenging it is to fix them. Among 14 multi-hunk bugs fixed with Hunk-Detection, 11 bugs are of Type-3 (multi-hunk/one-statement), in which some approaches (e.g., Hercules) can handle by fixing one statement at a time. Only 3 bugs are of Types 4–5. In contrast, all 7 bugs fixed with Expansion are multi-statement bugs (Types 2,4,5), which cannot be fixed by existing DL-based APR approaches. Thus, Expansion contributes to handling more challenging bugs than Hunk-Detection.

6.4.3 Impact of Tree-based LSTM model with Attention and Cycle Training. (Attention-cycle) To measure the impact of Attention and Cycle Training, we removed those two mechanisms from DEAR to produce a baseline. Our results show that in Defects4J, DEAR fixes 7 more bugs on all bug types than the baseline (17.5% increase). This result indicates the usefulness of the two mechanisms.

6.4.4 Impact of Training Data's Size. Table 10 shows that the size of training data has impact on DEAR's performance. As seen in Table 10, the more training data, the higher the DEAR's accuracy. This is expected as DEAR is a data-driven approach. But even with less training data (70%/30%), DEAR achieves 11.7% for top-1 result, which is still higher than DLFix (11.4% in top-1) and Sequencer (7.7% in top-1); both are with more training data (90%/10% splitting).

Table 10: Impact of the Size of Training Data

Splitting Scheme on CPatMiner dataset	90%/10%	80%/20%	70%/30%
% Total Bugs at Top-1	15.1%	13.8%	11.7%

```
1  public void excludeRoot(String path) {
2  -   String url = toUrl(path);
3  -   findOrCreateContentRoot(url).addExcludeFolder(url);
4  +   Url url = toUrl(path);
5  +   findOrCreateContentRoot(url).addExcludeFolder(url.getUrl());
6  }
7  public void useModuleOutput(String production, String test) {
8       modifiableRootModel.inheritCompilerOutputPath(false);
9  -   modifiableRootModel.setCompilerOutputPath(toUrl(production));
10 -   modifiableRootModel.setCompilerOutputPathForTests(toUrl(test));
11 +   modifiableRootModel.setCompilerOutputPath(toUrl(production).getUrl());
12 +   modifiableRootModel.setCompilerOutputPathForTests(toUrl(test).getUrl());
13 }
```

Figure 9: A Multi-hunk/Multi-statement Fix in CPatMiner

6.5 RQ5. Time Complexity and Parameters

Training time of DEAR on CPatMiner was +22 hours and predicting on CPatMiner took 2.4-3.1 seconds for each candidate patch. Training of DEAR on BigFix took 18-19 hours and predicting on BigFix took 3.6-4.2 seconds for each candidate. Predicting on Defects4J took only 2.1 seconds for a candidate due to a much smaller dataset. Test execution time was +1 second per test case. Test validation took 2–20 minutes for all the test cases for a bug fix.

The best baseline, CURE [14], fixes fewer bugs than DEAR (RQ1 and RQ2), and requires *7 and 7.3 times more* training parameters than DEAR on CPatMiner and BigFix, respectively. Specifically, DEAR and CURE require 0.39M and 3.1M training parameters on CPatMiner, and 0.42M and 3.5M parameters on BigFix. Thus, DEAR is less complex than CURE, while achieving better results.

Threats to Validity. We tested on Java code. The key modules in DEAR are language-independent, except for the third-party FL and post-processing with program analysis. Pattern-based APR tools require a dataset with test cases, thus, we compared them on Defects4J only. We tried our best to re-implement the pattern-based APR baselines and CURE for a fair comparison.

Illustrative Example. Figure 9 shows a correct fix from DEAR. It correctly detects two buggy hunks; each with multiple statements. DEAR leverages the variable names existing in the same method (`modifiableRootModel` at line 8) in composing the fixed code at lines 11–12. The DL-based baselines, Sequencer [6] and CoCoNuT [23], treat code as sequences, and does not derive well the structural changes for this fix. DLFix fixes one statement at a time, thus, does not work (the fixes at line 2 and line 3 depend on each other). For pattern-based APRs [13, 34], there is no fixing template for this bug.

Limitations. DEAR has the following limitations. First, as with ML approaches, fixes with rare or out-of-vocabulary names are challenging. With more training data, DEAR has higher chance to encounter the ingredients to generate a new name. Second, we focus only on the bugs that cause failing tests. Security, vulnerabilities, and non-failing-test bugs are still its limitations. Third, we cannot generate fixes with several new statements added or arbitrarily large sizes of dependent fixed statements. Fourth, the expansion algorithm produces incorrect hunks to be fixed, leading to fixing incorrect statements. Finally, we currently focus on Java, however, the basic representations used in DEAR, e.g., token, AST, dependency, are universal to any program language. Only third-party FL and post-processing with semantic checkers are language-dependent.

7 RELATED WORK

Deep Learning-based APR approaches. DeepRepair [42] learns code similarities to select the repair ingredients from code fragments similar to the buggy code. DeepFix [11] learns the syntax rules to fix syntax errors. Ratchet [12], Tufano *et al.* [39], and SequenceR [6] mainly use neural network machine translation (NMT) with attention-based encoder-decoder and code abstractions to generate patches. CODIT [5] encodes code structures, learns code edits, and adopt an NMT model to suggest fixes. Tufano *et al.* [38] learn code changes using a sequence-to-sequence NMT with code abstractions and keyword replacing. DLFix [18] has a tree-based translation model to learn the fixes. CoCoNuT [23] develops a context-aware NMT model. CURE [14] proposes a code-aware NMT using GPT model [31]. The existing DL-based APR models fix individual statements at a time and are ineffective for multi-hunk/multi-stmt bugs.

Pattern-based APR approaches. Those approaches have *mined and learned fix patterns* from prior fixes [15, 17, 19, 27], either automatically or semi-automatically [17, 19, 20, 27]. Prophet [22] learns code correctness models from a set of successful human patches. Droix [37] learns common root-causes for crashes using a search-based repair. Genesis [21] automatically infers patch generation from users' submitted patches. HDRepair [17] mines fix patterns with graphs. ELIXIR [33] uses templates from PAR with local variables, fields, or constants, to build fixed expressions. CapGen [40], SimFix [13], FixMiner [16] rely on frequent code changes extracted from existing patches. Avatar [19] exploits fix patterns of static analysis violations. Tbar [20] is a template-based APR tool with the collected fix patterns. Angelix [25] catches program semantics to fix methods. ARJA [44] generates lower-granularity patch representation enabling efficient searching. We did not compare with Angelix since we compared with CapGen that outperforms Angelix. We could not reproduce ARJA, however, ARJA fixes only 18 bugs, while DEAR fixes 42 bugs on the same four projects in Defects4J.

8 CONCLUSION

In this work, we make three key contributions: 1) a novel FL technique for multi-hunk, multi-statement fixes combining traditional SBFL with deep learning and data-flow analysis; 2) a compositional approach to generate multi-hunk, multi-statement fixes with divide-and-conquer strategy; and 3) enhancements and orchestration of a two-layer LSTM model with the attention layer and cycle training. On Defects4J, DEAR outperforms the DL-based APR baselines from 42%–683% in terms of the number of fixed bugs. On BigFix, it fixes 31–145 more bugs with the top-1 patches. On CPatMiner, it fixes 40–52 more multi-hunk/multi-stmt bugs than the baselines.

ACKNOWLEDGMENTS

This work was supported in part by the US National Science Foundation (NSF) grants CNS-2120386, CCF-1723215, CCF-1723432, TWC-1723198, CCF-1518897, and CNS-1513263.

REFERENCES

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

[2] 2021. *DEAR: A Novel Deep Learning-based Approach for Automated Program Repair.* https://github.com/AutomatedProgramRepair-2021/dear-auto-fix

[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] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2021. TreeCaps: Tree-Based Capsule Networks for Source Code Processing. *Proceedings of the AAAI Conference on Artificial Intelligence* 35, 1 (May 2021), 30–38. https://ojs.aaai.org/index.php/AAAI/article/view/16074

[5] Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. 2020. CODIT: Code Editing with Tree-Based Neural Models. *IEEE Transactions on Software Engineering* (2020). https://doi.org/10.1109/TSE.2020.3020502

[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). https://doi.org/10.1109/TSE.2019.2940179

[7] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Doha, Qatar, 1724–1734. https://doi.org/10.3115/v1/D14-1179

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[9] 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'12)*. 3–13. https://doi.org/10.1109/ICSE.2012.6227211

[10] 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

[11] 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.

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

[13] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space with Existing Patches and Similar Code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 298–309. https://doi.org/10.1145/3213846.3213871

[14] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE'21)*. 1161–1173. https://doi.org/10.1109/ICSE43902.2021.00107

[15] 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'13)*. 802–811. https://doi.org/10.1109/ICSE.2013.6606626

[16] 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* 25 (2020), 1980–2024. https://doi.org/10.1007/s10664-019-09780-z

[17] 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'16)*, Vol. 1. 213–224. https://doi.org/10.1109/SANER.2016.76

[18] 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

[19] 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'19)*. 1–12. https://doi.org/10.1109/SANER.2019.8667970

[20] 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'19)*. Association for Computing Machinery, New York, NY, USA, 31–42. https://doi.org/10.1145/3293882.3330577

[21] 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'17)*. Association for Computing Machinery, New York, NY, USA, 727–739. https://doi.org/10.1145/3106237.3106253

[22] 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

[23] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: Combining Context-Aware Neural Translation Models Using Ensemble for Program Repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA'20)*. Association for Computing Machinery, New York, NY, USA, 101–114. https://doi.org/10.1145/3395363.3397369

[24] 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'16)*. Association for Computing Machinery, New York, NY, USA, 441–444. https://doi.org/10.1145/2931037.2948705

[25] 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* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 691–701. https://doi.org/10.1145/2884781.2884807

[26] Hoan Anh Nguyen, Tien N. Nguyen, Danny Dig, Son Nguyen, Hieu Tran, and Michael Hilton. 2019. Graph-Based Mining of in-the-Wild, Fine-Grained, Semantic Code Change Patterns. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, 819–830. https://doi.org/10.1109/ICSE.2019.00089

[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'13)*. 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'10)*. Association for Computing Machinery, New York, NY, USA, 315–324. https://doi.org/10.1145/1806799.1806847

[29] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation. In *Empirical Methods in Natural Language Processing (EMNLP)*. 1532–1543. http://www.aclweb.org/anthology/D14-1162

[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'14)*. Association for Computing Machinery, New York, NY, USA, 254–265. https://doi.org/10.1145/2568225.2568254

[31] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. (2018).

[32] 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'12)*. Association for Computing Machinery, New York, NY, USA, Article 53, 11 pages. https://doi.org/10.1145/2393596.2393659

[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'17)*. 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 International Conference on Software Engineering (ICSE '19)*. IEEE Press, 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] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, Beijing, China, 1556–1566. https://doi.org/10.3115/v1/P15-1150

[37] Shin Hwei Tan, Zhen Dong, Xiang Gao, and Abhik Roychoudhury. 2018. Repairing Crashes in Android Apps. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE'18)*. Association for Computing Machinery, New York, NY, USA, 187–198. https://doi.org/10.1145/3180155.3180243

[38] 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'19)*. 25–36. https://doi.org/10.1109/ICSE.2019.00021

[39] 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'18)*. Association for Computing Machinery, New York, NY, USA, 832–837. https://doi.org/10.1145/3238147.3240732

[40] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE'18)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3180155.3180233

[41] 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'19)*. 479–490. https://doi.org/10.1109/SANER.2019.8668043

[42] 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.

[43] 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'17)*. IEEE Press, 660–670.

[44] Yuan Yuan and Wolfgang Banzhaf. 2020. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. *IEEE Transactions on Software Engineering (TSE)* 46, 10 (2020), 1040–1067. https://doi.org/10.1109/TSE.2018.2874648

[45] J. Zhu, T. Park, P. Isola, and A. A. Efros. 2017. Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks. In *2017 IEEE International Conference on Computer Vision (ICCV)*. 2242–2251. https://doi.org/10.1109/ICCV.2017.244