FusionRepair: Iterative Multi-Line APR via Fusion

Jayanath Senevirathna
University of Moratuwa, Sri Lanka
chathuranga.19@cse.mrt.ac.lk

Ayesh Vininda
University of Moratuwa, Sri Lanka
ayesh.19@cse.mrt.ac.lk

Prasad Sandaruwan
University of Moratuwa, Sri Lanka
prasad.19@cse.mrt.ac.lk

Ridwan Shariffdeen
National University of Singapore, Singapore
ridwan@comp.nus.edu.sg

Sandareka Wickramanayake
University of Moratuwa, Sri Lanka
sandarekaw@cse.mrt.ac.lk

Nisansa de Silva
University of Moratuwa, Sri Lanka
nisansadds@cse.mrt.ac.lk

***Abstract*—Learning-based APR techniques continue to face challenges in generating multi-line patches. We identified two fundamental limitations in existing learning-based APR tools. First, the length of the input sequence in existing APR tools is limited, restricting them from gathering information from compacted code contexts. Second, they fail to capture semantic dependencies among generated patches. We introduce FUSIONREPAIR, a transformer-based approach designed to capture additional context information from broader contexts and fix bugs by knowledge transfer-based patch generation. For this purpose, we have adapted the Fusion-in-Decoder(FiD) architecture to provide an expanded context. We utilize an iterative program repair paradigm to generate patches based on the knowledge of previously generated patches.**

Our experiment with Defects4J v2.0, shows FUSIONREPAIR can produce 55 single-line fixes and 28 multi-line fixes, identical to the developer patch. Comparison with state-of-the-art tools such as ITER and DEAR shows 35% and 18% improvements respectively. Our results show that FUSIONREPAIR has significantly outperformed current state-of-the-art tools in addressing bugs that require multi-line patches.

***Index Terms*—automated program repair, deep learning, fusion-in-decoder**

I. INTRODUCTION

While different classes of Automated Program Repair (APR) techniques can be identified in literature, learning-based techniques [1], [2], [3], [4], [5], [6] have recently gained more attraction, specially with the emergence of Large Language Models (LLMs). Learning-based techniques focus on mining bug fixes to generate abstract templates that can be used to fix new bugs. These techniques can address the limitations found in other techniques, such as large search space [7], or being limited to defined templates [8].

Single-line patches entail modifying one line of a buggy program to fix the bug. Majority of state-of-the-art APR tools such as DLFix [4], CoCoNuT [5], and CURE [6] have demonstrated success in generating single-line patches. Multi-line patches require the buggy program to be modified at multiple different non-contiguous locations. Current state-of-the-art techniques cannot successfully fix multi-line bugs. However, bugs that need multi-line patches are commonly present in software. Generating a multi-line patch is challenging due to the interdependence between the multi-line modifications.

Few APR tools focus on multi-line patch generation. Notable examples include: ITER [1], DEAR [2], and HERCULES [9]. ITER has achieved the best results compared to existing APR multi-line patch generation tools, with 61 single-line patches and 15 multi-line patches generated in DEFECTS4J [10] benchmark. Despite their successes, the following limitations can be observed in these techniques:

- length of the input sequence is limited, hindering its ability to gather information from extensive code contexts
- does not transfer knowledge of previously generated patches to subsequent multi-line edits.

We introduce FUSIONREPAIR, a transformer-based technique that overcomes these limitations by offering more context and using an iterative repair approach to capture semantic dependencies in multi-line edits. First, we curate a new training dataset using the perturbation tool used in SELFAPR [11]. Using 8 open-source Java projects and 16 perturbation rules we curate a dataset with more than 17M data samples. Second, we adapt the Fusion-in-decoder (FiD) architecture [12], utilizing multiple encoders to expand the APR tool's context, marking the first use of FiD for enhancing program context. Third, we employ an iterative repair approach to generate patches informed by previous iterations. In each iteration, FUSIONREPAIR concentrates on one buggy location, integrating candidate fixes from previous iterations to apply knowledge from past patches to other buggy locations.

Our experiment results show that FUSIONREPAIR surpasses existing state-of-the-art multi-line bug-fixing techniques by using a **small LLM with only 60 million parameters**, thereby consuming fewer resources. FUSIONREPAIR achieves significant results on Defects4J v2.0.0, successfully fixing 51 single-line bugs and 20 multi-line bugs in the initial iteration. Using three iterations, it further improves its efficacy by resolving 55 single-line bugs and 28 multi-line bugs for an exact match.

In summary, we make the following key contributions:

- A new training dataset for learning-based APR techniques for multi-line edits
- Use of FiD architecture to overcome LLM input sequence length limits. To the best of our knowledge, the first use of FiD in APR to increase the program context.
- A novel repair technique that transfers knowledge from previously generated patches to subsequent edits.

II. Motivational Example

To illustrate the limitations of the state of the art for program repair and motivate the need for better repair capabilities, let us consider two examples extracted from the Defects4j [10] dataset. Figure 1 depicts the developer fix for the bug reported in `Closure 123`. The bug is caused due to the use of incorrect variable reference, which leads a to test failure. The developer fix the bug by updating the right hand side assignment of `rhsCOntext` in line 285 to `getContextForNoInOperator(context)`. The developer fix requires project-specific tokens such as the method name `getContextForNoInOperator` and the variable `context`. None of the traditional search-based repair tools were able to fix the bug [13] and existing neural program repair tools cannot fix such bugs since they require project-specific tokens, which are not part of the training dataset [11]. Recent LLM-based repair tools can provide additional context within the prompt to the model, however, the amount of information that can be provided as context is also limited. Without enough context, although LLMs can generate reasonable code, they fail to generate the correct fix [14]. Most learning-based APR tools generally provide 5 lines above and 5 lines below the buggy line as the code context, hence such techniques will not be able to capture the method call `getContextForNoInOperator` into the context as it is 40 lines apart from the buggy line as shown in Figure 1.

Observation 1: Learning-based repair tools need the necessary contextual information to generate the correct patch, which can be 50 lines apart from the fix location. Thus, the model cannot capture the necessary context due to limitations of the length of the context that can be provided to it.

```
37    class CodeGenerator {
...       ...
241   // code block 1
242     case Token.COMMA:
243       Preconditions.checkState(childCount==2);
244       unrollBinaryOperator(n,Token.COMMA,",",context,
              getContextForOperator(context) ,0,0);
245       break;
...       ...
281   // buggy code block 2
282    case Token.HOOK: {
283     Preconditions.checkState(childCount == 3);
284     int p = NodeUtil.precedence(type);
285      Context rhs = Context.OTHER;
285      Context rhs = getContextForOperator(context);
```

Fig. 1: Code snippet of the developer patch for Closure 123. Note that the code is modified to include only the relevant context for brevity.

Figure 2 depicts the developer fix for the bug `Math 79`. The lines highlighted in green refer to new lines added and lines highlighted with red indicate lines that are removed. The fix for buggy location 1 requires a change in the datatype from *int* to *double* for the returning value *sum* as the expected return value type for this function is *double*.

The value for the sum is computed at Line 1627 as shown in Figure 2 using the value for variable *dp* which is also defined as a type of *int*. Hence, due to the dependency of *dp* on *sum*, the same datatype conversion from *int* to *double* needs to be applied on the variable *dp* as well. This semantic dependency between *sum* and *dp* is not captured in existing state-of-the-art tools. Prior work [1] reports that this bug `Math 79` is not fixed by any of the existing state-of-the-art tools.

Observation 2: Existing learning-based APR tools do not capture knowledge from previously generated fixes and fail to capture any semantic dependencies between generated patches. Hence, they are incapable of generating fixes for this bug.

```
1623  public static double distance(int[] p1, int[] p2) {
1624    // buggy code block 1
1625     int sum = 0;
1624     double sum = 0;
1625    for (int i = 0; i < p1.length; i++) {
1626      // buggy code block 2
1626       final int dp = p1[i] - p2[i];
1626       final double dp = p1[i] - p2[i];
1627      sum += dp * dp;
1628    }
1629    return Math.sqrt(sum);
1630  }
```

Fig. 2: Developer patch for Math 79

Addressing these two limitations, our proposed solution FusionRepair can generate a correct patch for both examples `Closure 123` and `Math 79`. Instead of supplying five lines above and below the buggy line, we incorporate multiple code blocks into the model using the FiD architecture [12]. Hence, FusionRepair can capture information from an extended context to generate such fixes.

Using the increased context, FusionRepair can utilize previously generated patches for a specific location to capture semantic dependencies from other locations. For the example `Math 79`, when a patch is generated at the first buggy location, it is provided as a context to generate a correct patch for the second buggy location.

III. Training Dataset Curation

We focus on fixing functional errors for Java programs that may require multiple fix locations within a single source file. Hence, our dataset collection targets Java programs that exhibit functional errors paired with the correct fix.

A training dataset for APR contains pairs of buggy and fixed code samples [15]. The datasets used by the existing learning-based APR work, e.g., CodeBERT [16], and CoCoNuT [5],contain only a limited context for each bug (e.g., five lines above or below the buggy location). Hence, they are unsuitable for our approach, which examines a larger context to extract information to fix bugs. Hence, we curate a new dataset that includes a larger surrounding context.

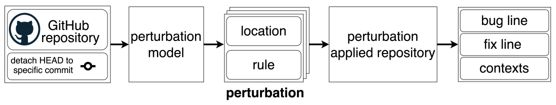

Fig. 3: Process of dataset creation.

Following prior work [11], [17] we employed a perturbation model that generates buggy code from the correct code.

Given a project repository, the perturbation model applies a perturbation rule to all possible source lines. From the mutated repository, we extract the perturbation rule, line number, perturbed line, and the surrounding context for each perturbed source line. Those paired with the corresponding source line make a training instance. Figure 3 shows the process overview.

TABLE I: Distribution of the training dataset among the considered GitHub repositories

Project	**Stars**	**Forks**	**#Samples**
finmath-lib	467	165	475843
Time4J	418	59	263159
htmlunit	815	162	178614
maven-doxia	26	41	36399
wro4j	471	106	39487
guava	49.4k	10.7k	560711
super-csv	518	139	10410
rhino	4k	818	206956
Total			1771579

We selected eight popular open-source Java projects as the source for our dataset creation. We explicitly removed projects listed in DEFECTS4J [10] to ensure no data leakage on the evaluation. Table I lists the selected repositories and the count of generated data samples in column "#Samples". We create two datasets, a single context, and multiple contexts. In the single-context dataset, each buggy line is accompanied by a single context with ten lines surrounding the buggy line. Suppose the buggy line is L. Then, in the single-context dataset, the context comprises source lines $[L-5-L-1]$ and $[L+1-L+5]$. Let's call this context C_0. For the dataset with multiple contexts, we collect five more consecutive contexts before and five more after the initial context C_0, each with ten lines. For example, context C_{-1} contains source lines $[L-10-L-5]$ and C_{-2} contains source lines $[L-15-L-10]$, whereas context C_{+1} contains source lines $[L+5-L+10]$ and C_{+2} contains source lines $[L+10-L+15]$. Hence, a buggy line in the multiple-contexts dataset accompanies 11 contexts altogether, $\{C_{-5}, C_{-4}, ..., C_0, ..., C_{+4}, C_{+5}\}$.

IV. FUSIONREPAIR FRAMEWORK

One major limitation of current APR tools in creating multi-line patches is the restriction on input length. This limits the amount of code context from which the necessary bug-fixing information can be extracted. To address this issue, we propose adapting the Fusion-in-Decoder (FiD) sequence-to-sequence (Seq-to-Seq) model for APR. The FiD model, introduced by [12], is designed for open-domain question answering and is capable of providing accurate answers by gathering information from multiple passages related to the given question. Unlike traditional Seq-to-Seq models, FiD independently processes each passage using its encoder, aggregates the resulting representations, and then feeds them to the decoder to generate the answer. Our tool, FUSIONREPAIR, adapts FiD as CODE-FID, enabling it to process multiple code

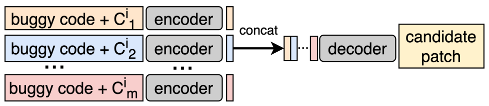

Fig. 4: Code-FiD

contexts to extract information for fixing multi-line bugs. The Figure 4 shows the CODE-FID model.

As *input*, FUSIONREPAIR requires the buggy program $\mathcal{P}_\mathcal{B}$, set of buggy locations L, set of bugs B, the CODE-FID model $\mathcal{G}$, number of attempts R and the test suite $\mathcal{T}$. $\mathcal{P}_\mathcal{B}$ has n buggy lines. As *output*, our tool produces the set of plausible patches P for the n bugs in B.

Algorithm 1: Patch Generation & Refinement.

```
Input  : $\mathcal{P}_\mathcal{B}$ - buggy program, $B$ - list of bugs, $L$ -
         list of buggy locations, $\mathcal{G}$ - CODE-FID model
         , $R$ - No.of attempts, $\mathcal{T}$ - test suite
Output: $P^*$ - Plausible patch combination.
Config : $m$ - Number of contexts
1  $P^* \leftarrow null$; $n \leftarrow length(B)$;
   // Step 1 - Initial patch generation
2  for $i = 1$ to $n$ do
3      $C^i \leftarrow$ createContexts($B[i]$);
4      $I^i \leftarrow$ createEncoderInputs($B[i], C^i$);
5      $P[i] \leftarrow \mathcal{G}(I^i, k)$;
6  $D \leftarrow$ createPatchCombinations($P$);
7  for $d \in D$ do
8      if $is\_plausible(d, \mathcal{P}_\mathcal{B}, \mathcal{T})$ then
9          $P^* \leftarrow d$; return $P^*$;
   // Step 2 - Iterative Improvement
10 for $r = 1$ to $R$ do
11     for $i = 1$ to $n$ do
12         $\bar{B}^i \leftarrow$ createNewBuggyCode($P[i]$);
13         for $j = 1$ to $k+1$ do
14             $C^i_j \leftarrow$ createContexts($\bar{B}^i[j]$);
15             $\bar{I}^i_j \leftarrow$ createEncoderInputs($\bar{B}^i[i], C^i_j$);
16             $\bar{P} \leftarrow \mathcal{G}(\bar{I}^i_j, r)$; $P[i] \leftarrow P[i] + \bar{P}$;
17 $D \leftarrow$ createPatchCombinations($P$);
18 for $d \in D$ do
19     if $is\_plausible(d, \mathcal{P}_\mathcal{B}, \mathcal{T})$ then
20         $P^* \leftarrow d$; return $P^*$;
```

Algorithm 1 shows the general workflow of the proposed framework. Let's consider the bug $B(i)$ at location i where $i \in 1, 2, ..., n$. For $B(i)$, we consider m number of contexts $C^i = \{c^i_1, ..., c^i_m\}$ derived as described in Section III (See line number 3 in Algorithm1). Using these contexts and $B[i]$ as inputs, *create_encoder_inputs* creates a set of inputs $I^i = \{I^i_1, ..., I^i_m\}$ to be processed by the CODE-FID encoder. Each input $I^i_t, t \in \{1, 2, ..., m\}$ starts with a special token `[BUG]`, followed by the buggy line and its surrounding code lines, and then the `[CONTEXT]` token, followed by one of the m contexts. Figure 5 shows a sample input for Closure 123.

```
[BUG]
break;
      }
      case Token.HOOK: {
        Preconditions.checkState(childCount == 3);
        int p = NodeUtil.precedence(type);

Context rhsContext = Context.OTHER; /* buggy line */
addExpr(first, p + 1, context);
        cc.addOp("?", true);
        addExpr(first.getNext(), 1, rhsContext);
        cc.addOp(":", true);
        addExpr(last, 1, rhsContext);
[CONTEXT]
break;
      }
      case Token.REGEXP:
        if (!first.isString() ||
            !last.isString()) {
          throw new Error("Expected children to be strings");
        }
        String regexp = regexpEscape(first.getString(), outputCharsetEncoder);
        if (childCount == 2) {
          add(regexp + last.getString());
```

Fig. 5: Sample Input for Closure 123.

The encoder in CODE-FID processes I_t^i independently to produce a vector representation containing information related to $B[i]$ from the corresponding context c_t^i. Next, all these m number of vector representations are concatenated and fed to the decoder to generate k candidate patches for $B[i]$. In addition to these candidate patches, line deletion is considered as an additional patch as it can be very helpful in solving bugs that need multi-line patches. Suppose P denotes the set of patches generated by FUSIONREPAIR for the n bugs in B where $|P| = n \times (k+1)$. Then using the combinations of these candidate patches, *create_patch_combinations* function create a set of $(k+1)^n$ patches. Let this list be D such that $D = \{d_1, \ldots, d_{(k+1)^n}\}$. We apply each of these patch combinations, d_u to $\mathcal{P}_{\mathcal{B}}$ and identify the plausibility of each d_u by executing the test suite $\mathcal{T}$ (See line 8). If any one of the patch combinations is plausible, then we have found a multi-line patch and we terminate the process. In Figure 6, the process discussed in this paragraph is shown as **Step -1**.

Suppose we cannot find a plausible patch combination in the previous step. Then for each bug $B[i]$, we attempt to improve the generated patches $P[i]$ in an iterative manner as these patches may be partially correct [1] and iterative improvement might lead us to the correct complete patch. We create a new set of derivative buggy codes $\bar{B}^i$ corresponding to $B[i]$ by replacing the buggy line at $L[i]$ with generated patches $P[i]$ (See line 13 of Algorithm 1). Thus the set $\bar{B}^i$ has elements $\bar{b}_j^i$ such that $j \in 1, 2, ..., k+1$. Next, we create a new set of inputs as described before and apply CODE-FID to generate k number of patches. Suppose P_j^i denotes the set of improved patches for $P[i,j]$ patch. After generating improved patches for $k+1$ initial patches in $P[i]$, we add these new improved candidate patches to $P[i]$, where $P[i] = P[i] + P_0^i + P_1^i + ... + P_{k+1}^i$. Then, similar to as described in **Step -1**, we create patch combinations considering patches for each bug in P. Finally, we evaluate the plausibility of each patch combination by executing the test suite $\mathcal{T}$. If we find a plausible patch combination, we terminate the process. If not, we carry out this for all the bugs in $\bar{B}^i$ R times. In Figure 6, the process discussed in this paragraph is shown as **Step -2**.

Algorithm 2: KT based Patch Generation

```
Input  : $\mathcal{P}_{\mathcal{B}}$ - buggy program, $B$ - list of bugs, $P$ -
         list of patches from Algo-1, $\mathcal{G}$ - CODE-FID
         model , $R$ - No.of attempts, $\mathcal{T}$ - test suite
Output: $P^*$ - Plausible patch combination.
Config : $m$ - Number of contexts
1  $P^* \leftarrow null$;
2  for $i = 1$ to $n$ do
3      for $q = 1$ to $n$ do
4          if $q == i$ then
5              continue;
6          for $j = 1$ to $k+1$ do
7              $C_{P[q,j]} \leftarrow$ createSingleContext($P[q,j]$);
               $\hat{I}^i_{P[q,j]} \leftarrow$createEncoderInputs($B[i], C_{P[q,j]}$);
               $\hat{P} \leftarrow P_q^i + \mathcal{G}(\hat{I}^i_{P[q,j]}, k)$;
8              $D \leftarrow$ createPatchCombinations($[P[q,j], \hat{P}$);
9              for $d \in D$ do
10                 if isPlausible($d, \mathcal{P}_{\mathcal{B}}, \mathcal{T}$) then
11                     $P^* \leftarrow d$; return $P^*$;
```

If the previous steps fail to generate a plausible patch combination for B, FUSIONREPAIR attempts to generate candidate patches by transferring knowledge between generated patches, a process which we call *knowledge transfer based patch generation*. Let's consider $B[i]$. We first generate a new set of inputs, $\hat{I}^i$, for the CODE-FID encoder, where each of these inputs starts with the special token `[BUG]`, followed by the $B[i]$ and its surrounding code lines, and then the `[CONTEXT ]` token, followed by a context $C_{P[q,j]}$ which comprises of $P[q,j]$ and its surrounding code lines where, $q \in 1, 2, ..., n$ and $q \neq i$ and $j \in 1, 2, ..., k$. We feed each of these to the CODE-FID encoder and its output is fed to the decoder to generate a set of candidate patches $\hat{P}$ for $B[i]$ that captures knowledge transferring from $P[q,j]$. Then we create patch combinations considering $P[q,j]$ and $\hat{P}$ and evaluate the plausibility of each patch combination by executing the test suite $\mathcal{T}$. In Figure 6, the process discussed in this paragraph is shown as **Step -3**.

V. EVALUATION METHODOLOGY

A. Evaluation Setup

We adopt a CodeT5-small 60M parameter model hosted by Salesforce on the HuggingFace platform. First, we fine-tune the model using our single-context data set over 3 epochs with a batch size of 30 using the Adam optimizer and a learning rate of 2×10^{-5}. Next, the fine-tuned model is applied to the FiD architecture and trained using a multi-context dataset with a context size of 11 over 1000 steps with a single batch size. We conduct our experiments using the NVIDIA Tesla T4.

The input to an encoder in FiD is similar to Figure 5. Before the input is provided to CodeT5-small, the buggy line is replaced with the token $< extra\_id\_0 >$. This token is used because CodeT5-small is pre-trained for code generation tasks where $< extra\_id\_0 >$ is given.

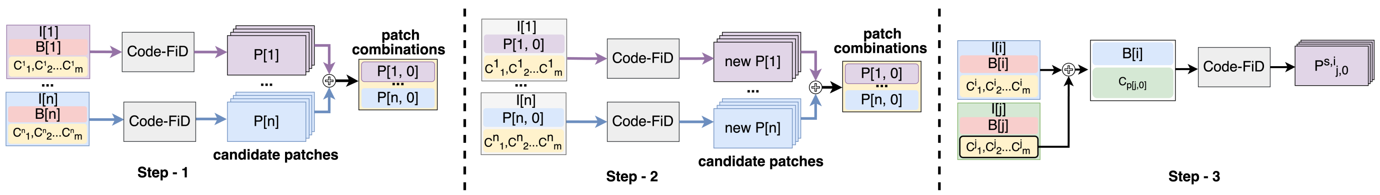

Fig. 6: Step-1:Initial patch generation.Step-2:Patch Refinement.Step-3:Knowledge transfer based patch generation.

B. Evaluation Dataset

For the evaluation, we used the DEFECTS4J [10] benchmark. Existing APR tools are evaluated on the DEFECTS4J v1.2.0 and v2.0.0. DEFECTS4J v1.2.0 has 391 bugs over 6 projects, and v2.0.0 has 835 bugs over 17 projects. For a fair comparison with other existing tools, we evaluated the FUSIONREPAIR over those two versions. Testing samples are derived from the DEFECTS4J by identifying buggy lines with their fixes and gathering contexts surrounding the buggy lines. Two evaluation datasets are created similar to the training datasets, one containing a single context for assessing the CodeT5 model, and the other containing 11 contexts to evaluate the FUSIONREPAIR tool.

C. Research Questions

- RQ1: How does FUSIONREPAIR perform against existing DL-based APR models on DEFECTS4J?
- RQ2: What is the contribution of each component to the overall performance of FUSIONREPAIR?
- RQ3: How does the prediction length impact the efficacy of FUSIONREPAIR?
- RQ4: Does patch refinement improve the efficacy of FUSIONREPAIR?

D. Experimental Methodology

1) RQ1: Comparison with SOTA: Among existing APR tools, we select those that have multi-line repair capabilities for Java programs. Namely we select DLFIX [4], COCONUT [5], CURE [6], DEAR [2], HERCULES [9], and ITER [1] to perform the comparison with FUSIONREPAIR. The results for each tool are the reported values obtained from their respective publications. In our evaluations, following prior work we assume perfect fault localization [18], where the exact fix location of the bug is provided for evaluation.

ITER was evaluated on DEFECTS4J v2.0.0 and the rest of the tools were compared using DEFECTS4J v.1.2.0. For comparison with ITER, using perfect fault localization, we executed ITER on a subset of bugs from the total pool of 835 bugs to ensure fairness. Bugs meeting the following criteria were filtered out to maintain a fair comparison between ITER and FUSIONREPAIR.

- Bugs that do not belong to the 10 projects used in ITER evaluation (i.e. Mockito, JacksonDatabind)
- Bugs that include at least one faulty location spanning more than one line of code. The input for ITER is derived from the output of GZoltar [19] after fault localization.
- Bugs where at least one faulty location wasn't accurately identified using the fault localization technique. If GZoltar fails to provide the correct faulty locations as input to the repair tool.
- Bugs which has more than two buggy locations.

After filtering, the remaining 199 bugs were used to conduct the evaluation. The filtered bugs are executed on a modified version of ITER (ITER*) allowing to generate patches for bugs using perfect fault localization. The modifications are:

- Upon fault localization, remove the incorrect suspicious lines and retain only those that match perfect localization.
- Perform fault localization once initially, then focus on fixing the most suspicious lines first.
- After generating patches, check for an exact match to the required fix. If found, mark the buggy location as fixed and proceed to the next.

A comparison between ITER* and FUSIONREPAIR is conducted with no re-attempts while generating 50 predictions per buggy location.

2) RQ2: Ablation Study: This section examines the contribution of each step of FUSIONREPAIR for overall performance on DEFECTS4J 2.0.0. The considered steps are as follows:

- Fine-tuning pre-trained CodeT5-small model
- Integrating Fusion in Decoder (FiD) architecture
- Manually introducing line deletion patch
- Two iterations of refinement on generated patches.
- Knowledge transfer-based patch generation

We selected CodeT5-small as the starting model due to its existing code-generation capabilities, first assessing its inherent ability to fix bugs. The buggy line is replaced with the `<extra_id_0>` token, aligning with CodeT5-small's pretraining to generate code at this token. We then fine-tune the model using our single-context dataset created from perturbations.

FiD extends input length by dividing it into chunks, each processed by separate encoders that capture information from these chunks. The outputs are combined and fed to the decoder, enabling FUSIONREPAIR to make predictions based on this combined input. Incorporating FiD into FUSIONREPAIR enhances its ability to handle lengthy inputs, hence we compare results before and after adding FiD to assess its impact.

3) RQ3: Best Configuration for FusionRepair Performance.: Evaluation is conducted for different prediction lengths 5, 10, 20, 30, 40, and 50. The outcomes are analyzed to identify the most suitable prediction length.

VI. Evaluation Results

A. RQ1: Comparison with SOTA

We evaluated FusionRepair on both Defects4J v1.2.0 and v2.0.0 for both plausible patch generation and correct patch generation (using exact matching). On Defects4J v1.2.0 FusionRepair generates a plausible patch for 42 single-line bugs and 25 multi-line bugs. However, considering the exact match to the developer-generated patch, the correct number of fixes are 31 single-line bugs and 19 multi-line bugs. Similarly, on Defects4J v2.0.0 FusionRepair generates a plausible patch for 81 single-line bugs and 46 multi-line bugs, totalling to 127 bugs. Using exact matching as the correctness criteria, FusionRepair generates the correct patch for 55 single-line bugs and 28 multi-line bugs.

Since FusionRepair assumes perfect fault localization, we compare its efficacy with the baseline tools using values reported in prior work [1], [2]. We analyzed the results of different baseline tools on the number of correct fixes using exact matches for multi-line bugs in Defects4J v1.2.0 dataset under perfect fault localization setting. DEAR and HERCULES report the most number correct multiline patches with 16 and 12 respectively, while the rest of the tools (i.e. CURE, CoCoNuT, and DLFix) do not generate correct fixes for more than 10 bugs. Notably, we exclude ITER in this comparison because the authors do not report their results for Defects4J v1.2.0 on perfect fault localization setting. FusionRepair outperforms our baseline tools with **19 correct fixes**.

Comparison with the recent state-of-the-art tool ITER [1] on Defects4J v2.0.0. We modified ITER under the same configurations on the bugs filtered out according to the process mentioned in Section V-D1. Our modified ITER generated the correct patch for 18 bugs with 16 single-line and 2 multi-line patches. FusionRepair generated **correct patches for 33 bugs** with 26 single-line and 7 multi-line patches.

SOTA Comparison: Our experiment results on Defects4J v1.2.0 and v2.0.0 show FusionRepair outperforms the existing state of the learning-based repair tools.

B. RQ2: Ablation Study

We evaluate the contribution of each component in FusionRepair for the overall efficacy on Defects4J v2.0.0. Table II summarizes the number of bugs for which the correct patch was generated by FusionRepair, by disabling a single component. Each row captures the number of bugs that are correctly fixed with a specific configuration. Two consecutive rows capture the contribution of a single component indicated in the column "Variations".

Pre-trained CodeT5-small model alone was only able to fix 3 number of bugs correctly. Fine-tuning using our generated data set, helps to improve the number from 31 single-line and 8 multi-line patches. Using a FiD architecture has allowed FusionRepair to capture information from a larger context and fix more bugs. Providing a larger context using the FiD architecture mentioned in Section IV, improves the number of correctly fixed single-line bugs to 41. Adding a deletion operator for the repair generates fixes for 51 single-line and 18 multi-line bugs. Refining previously generated patches by re-iterating allows FusionRepair to improve the efficacy further. Lastly, knowledge-based patch generation allows FiD architecture allows FusionRepair to correct and fix 55 single-line bugs and 28 multi-line bugs.

TABLE II: Impact of each component of FusionRepair for the overall efficacy on Defects4J v2.0.0

	Variations				Correct Fixes	
Model	**F**	**D**	**R**	**KT**	Single	Multiple
$\mathbf{M}_p$	✗	✗	✗	✗	3	0
$\mathbf{M}_f$	✗	✗	✗	✗	31	8
$\mathbf{M}_f$	✓	✗	✗	✗	41	8
$\mathbf{M}_f$	✓	✓	✗	✗	51	18
$\mathbf{M}_f$	✓	✓	✓	✗	55	20
$\mathbf{M}_f$	✓	✓	✓	✓	55	28

F: FiD architecture, **D**: deletion operator, **R**: re-attempts, **KT**: Knowledge Transfer-based patche generation, $\mathbf{M}_p$: pre-trained CodeT5-small, $\mathbf{M}_f$: fine-tuned CodeT5-small

Ablation Study: Each component in FusionRepair significantly contributes towards the overall efficacy.

C. RQ3: Impact of Prediction Length

We evaluate the impact of the maximum prediction length parameter of CodeT5-small model in FusionRepair, by varying the values and analysing the overall efficacy. Figure 7 plots the efficacy for single-line fixes and multi-line fixes, against variation of maximum prediction length. Efficacy is measured in terms of a number of bugs for which the correct patch (i.e. exact match) was found. We varied the maximum prediction length but kept the number of return sequences fixed to 50.

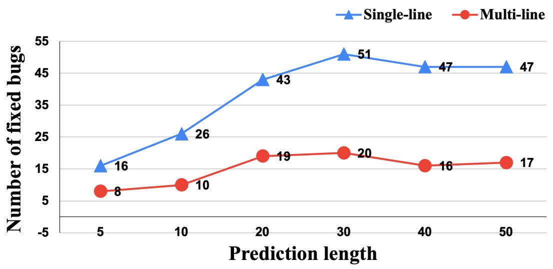

Fig. 7: Impact of prediction length for the first attempt.

Increasing the maximum prediction length from 5 to 30 drastically increases the number of bugs that can be fixed correctly for both single-line and multi-line patches. Increased prediction length allows the model to generate more patches that correctly fix the bug. This is expected as the correct patch may require a larger patch, which is not generated with a more restricted configuration. However, increasing from 30 to 50, we observe a drastic change in the effectiveness with a declination in the number of bugs that are correctly fixed. This implies that increasing the prediction length is not necessarily effective, as the prediction length is larger the prediction accuracy gets lower. The optimum results we observed is at a maximum prediction length of 30, with 51 single-line bugs and 20 multi-line bugs being correctly fixed by FUSIONREPAIR.

Impact of Prediction Length: Varying the prediction length impacts the efficacy of the model, it is a trade-off between generating a high number of patches that are small in size vs a low number of patches that are larger in size.

D. RQ4: Impact of Patch Refinement

We analyze the effect of the iterative patch refinement step in FUSIONREPAIR. We attempted to refine partially correct patches for two iterations for bugs in DEFECTS4J v1.2.0 and v2.0.0, as summarized in Table III. Column "Correct Fixes" captures the number of bugs that are correctly fixed by generating a patch identical to the developer-provided patch for each version of DEFECTS4J. Column "Configuration" captures the variation from no refinement to 2 iterations of patch refinement.

TABLE III: Improvement using Patch Refinement

Configuration	Defects4J v1.2.0			Defects4J v2.0.0		
	r-0	r-1	r-2	r-0	r-1	r-2
Single Line	29	29	31	51	52	55
Multi Line	13	15	19	20	23	28
Total	42	44	50	71	75	83

Note: r-n - n no.of re-attempts.

Iteratively refining partially correct patches helps to improve the overall efficacy from 42 bugs to 50 bugs in v1.2.0, and from 71 bugs to 83 bugs in v2.0.0. Significant improvement can be observed for multi-line fixes in both versions of DEFECTS4J.

```
1   // Option.java file
2   public class Option implements Cloneable, Serializable {
3       ...
4       /** the type of this Option */
5       private Class type = String.class;
6       private Class type;
7       ...
8    }
9
10  // OptionBuilder.java file
11  public final class OptionBuilder {
12      private static void reset(){
13          ...
14          longopt = null;
15          type = String.class;
16          type = null;
17          ...
18       }
19  }
```

Fig. 8: Patch Refinement Example.

Figure 8 shows the correct patch generated by FUSION-REPAIR for the bug `Cli 34`. The patch is generated in two iterations of refinement. The first iteration generated a patch at Line 7 with `private Class type = Object.class`. Second iteration of refinement re-corrected the patch from `Object.class` to `String.class`.

Impact of Patch Refinement: Iterative patch refinement can transform partially correct patches into correct patches. Thereby improving the overall efficacy with a significant margin.

VII. RELATED WORK

Decades of research led to the evolution of APR through different stages of APR techniques such as search-based [20], [21], template-based [22], constraint-based [23], [24], and learning-based techniques [5], [6]. Several excellent surveys summarize these techniques [25], [8].

Learning-based APR: Learning-based APR techniques have progressed significantly with the integration of Natural Language Processing (NLP) and Neural Machine Translation (NMT) methods [26]. CoCoNuT [5] introduced a CNN-based encoder architecture that also employs a context-aware NMT architecture, similar to DLFix [4]. Prior work also captured code structure features, leading to techniques that utilize Abstract Syntax Tree (AST) based representations. DLFix [4], DEAR [2], HOPPITY [27], and CODIT [28] are few examples. TFix [29] used a pre-trained natural language model called Text-to-Text Transfer Transformer (T5) [30]. TFix approaches coding error correction as a text-to-text prediction task. Following TFix's introduction of the text-based T5 model to APR, many tools have since adopted T5.

Multi-Line Repair Prior work has proposed various methods to generate multi-line patches [23], [3], [2], [1]. HERCULES [9] identifies evolutionary siblings (similar code locations) to address particular bug types. DeepFix [3] focuses on fixing common programming errors by addressing one buggy line at a time, moving to the next line only after the previous one is repaired. ITER [1] uses an iterative approach that first localizes faults, selects the most suspicious line, and then generates and applies multiple candidate patches. The process repeats with fault localization and patch application until all tests pass, indicating a plausible fix. Recent iterative techniques offer promise for multi-line fixes, yet they still face challenges in capturing dependencies across patches.

Context Awareness SequenceR [4] captures context by concatenating buggy and surrounding lines, but this increases input length and introduces noise. CoCoNuT [5] and CURE [6] use separate encoders for buggy lines and context, with CURE's context extending to the entire buggy method. Izacard et al [12] introduced the Fusion in Decoder (FiD) architecture to handle extensive input in sequence-to-sequence models, enhancing context utilization by capturing evidence across multiple passages. In FiD, each passage is encoded independently and combined in the decoder for joint processing, optimizing knowledge fusion between encoder and decoder components. This model is especially effective for tasks requiring large context, like open-domain question answering. Building on FiD, VulMaster [31] adapts this approach for Automatic Vulnerability Repair(AVR) in C/C++ codebases, integrating inputs from code segments, ASTs, and insights from the Common Weakness Enumeration(CWE) database to manage complex, multi-source context effectively.

VIII. Conclusion

In this work, we introduce FusionRepair, a transformer-based iterative program repair technique that uses a larger context window. We develop a novel fusion technique to address the context limitation in existing state-of-the-art repair models. Using the increased context information, FusionRepair was trained in knowledge transfer-based patch generation. Our experimental results on Defects4J v.1.2.0 and v2.0.0, show that FusionRepair outperforms existing state-of-the-art techniques by a significant margin.

Artifacts: All our artifacts are released to the community. https://anonymous.4open.science/r/FusionRepair-F028

References

[1] H. Ye and M. Monperrus, "Iter: Iterative neural repair for multi-location patches," in *Proceedings of ICSE 2024*. New York, USA: ACM, 2024.

[2] Y. Li, S. Wang, and T. N. Nguyen, "Dear: A novel deep learning-based approach for automated program repair," in *Proceedings of ICSE 2022*, 2022, p. 511–523.

[3] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, 2017, p. 1345–1351.

[4] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair," in *Proceedings of ICSE 2020*, 2020, p. 602–614.

[5] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: Combining context-aware neural translation models using ensemble for program repair," 2020, p. 101–114.

[6] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *Proceedings of ICSE 2021*, 2021, p. 1161–1173.

[7] Y. Noller, R. Shariffdeen, X. Gao, and A. Roychoudhury, "Trust enhancement issues in program repair," in *Proceedings of ICSE 2022*, 2022, p. 2228–2240.

[8] K. Huang, Z. Xu, S. Yang, H. Sun, X. Li, Z. Yan, and Y. Zhang, "A survey on automated program repair techniques," 03 2023.

[9] S. Saha, R. K. Saha, and M. R. Prasad, "Harnessing evolution for multi-hunk program repair," in *Proceedings of ICSE 2019*. IEEE Press, 2019, p. 13–24.

[10] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 ISSTA*. New York, USA: ACM, 2014, p. 437–440.

[11] H. Ye, M. Martinez, X. Luo, T. Zhang, and M. Monperrus, "Selfapr: Self-supervised program repair with test execution diagnostics," in *ASE'2023*. New York, USA: ACM, 2023.

[12] G. Izacard and E. Grave, "Leveraging passage retrieval with generative models for open domain question answering," in *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, P. Merlo, J. Tiedemann, and R. Tsarfaty, Eds., 2021, pp. 874–880.

[13] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, "Empirical review of java program repair tools: a large-scale experiment on 2,141 bugs and 23,551 repair attempts," in *Proceedings of ESEC/FSE 2019*, 2019, p. 302–313.

[14] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," in *Proceedings of ICSE 2023*, 2023, p. 1430–1442.

[15] K. Pramod, W. De Silva, W. Thabrew, R. Shariffdeen, and S. Wickramanayake, "Bugsphp: A dataset for automated program repair in php," in *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*, 2024, pp. 128–132.

[16] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *EMNLP*, 2020, pp. 1536–1547.

[17] Y. He, Z. Chen, and C. L. Goues, "Precisebugcollector: Extensible, executable and precise bug-fix collection: Solution for challenge 8: Automating precise data collection for code snippets with bugs, fixes, locations, and types," in *2023 38th IEEE/ACM International Conference on ASE*, 2023, pp. 1899–1910.

[18] C. S. Xia and L. Zhang, "Less training, more repairing please: Revisiting automated program repair via zero-shot learning," in *ESEC/FSE 2022*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 959–971. [Online]. Available: https://doi.org/10.1145/3540250.3549101

[19] A. Riboira and R. Abreu, "The gzoltar project: A graphical debugger interface," in *Testing – Practice and Research Techniques*, L. Bottaci and G. Fraser, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 215–218.

[20] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[21] C.-P. Wong, P. Santiesteban, C. Kästner, and C. Le Goues, "Varfix: Balancing edit expressiveness and search effectiveness in automated program repair," in *Proceedings of ESEC/FSE 2021*, 2021, p. 354–366.

[22] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyande, "Avatar: Fixing semantic bugs with fix patterns of static analysis violations," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Los Alamitos, CA, USA: IEEE Computer Society, feb 2019, pp. 1–12. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SANER.2019.8667970

[23] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of ICSE 2016*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 691–701. [Online]. Available: https://doi.org/10.1145/2884781.2884807

[24] R. Shariffdeen, Y. Noller, L. Grunske, and A. Roychoudhury, "Concolic Program Repair," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, pp. 390–405. [Online]. Available: https://doi.org/10.1145/3453483.3454051

[25] M. Monperrus, "Automatic software repair: A bibliography," *ACM Comput. Surv.*, vol. 51, no. 1, jan 2018. [Online]. Available: https://doi.org/10.1145/3105906

[26] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, sep 2019. [Online]. Available: https://doi.org/10.1145/3340544

[27] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, "Hoppity: Learning graph transformations to detect and fix bugs in programs," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:213089769

[28] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray, "Codit: Code editing with tree-based neural models," *IEEE Transactions on Software Engineering*, vol. 48, no. 4, p. 1385–1399, Apr. 2022. [Online]. Available: http://dx.doi.org/10.1109/TSE.2020.3020502

[29] B. Berabi, J. He, V. Raychev, and M. Vechev, "Tfix: Learning to fix coding errors with a text-to-text transformer," in *Proceedings of the 38th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. Meila and T. Zhang, Eds., vol. 139. PMLR, 18–24 Jul 2021, pp. 780–791. [Online]. Available: https://proceedings.mlr.press/v139/berabi21a.html

[30] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *CoRR*, vol. abs/1910.10683, 2019. [Online]. Available: http://arxiv.org/abs/1910.10683

[31] X. Zhou, K. Kim, B. Xu, D. Han, and D. Lo, "Out of sight, out of mind: Better automatic vulnerability repair by broadening input ranges and sources," in *Proceedings of ICSE 2024*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639222