SelfAPR: Self-supervised Program Repair with Test Execution Diagnostics

He Ye
heye@kth.se
KTH Royal Institute of Technology
Sweden

Matias Martinez
matias.martinez@uphf.fr
Université Polytechnique Hauts-de-France
France

Xiapu Luo
csxluo@comp.polyu.edu.hk
The Hong Kong Polytechnic University
China

Tao Zhang
tazhang@must.edu.mo
Macau University of Science and Technology
China

Martin Monperrus
monperrus@kth.se
KTH Royal Institute of Technology
Sweden

ABSTRACT

Learning-based program repair has achieved good results in a recent series of papers. Yet, we observe that the related work fails to repair some bugs because of a lack of knowledge about 1) the application domain of the program being repaired, and 2) the fault type being repaired. In this paper, we solve both problems by changing the learning paradigm from supervised training to self-supervised training in an approach called SelfAPR. First, SelfAPR generates training samples on disk by perturbing a previous version of the program being repaired, enforcing the neural model to capture project-specific knowledge. This is different from the previous work based on mined past commits. Second, SelfAPR executes all training samples and extracts and encodes test execution diagnostics into the input representation, steering the neural model to fix the kind of fault. This is different from the existing studies that only consider static source code as input. We implement SelfAPR and evaluate it in a systematic manner. We generate 1 039 873 training samples obtained by perturbing 17 open-source projects. We evaluate SelfAPR on 818 bugs from Defects4J, SelfAPR correctly repairs 110 of them, outperforming all the supervised learning repair approaches.

ACM Reference Format:
He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2022. SelfAPR: Self-supervised Program Repair with Test Execution Diagnostics. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3551349.3556926

1 INTRODUCTION

Automated program repair (APR) aims to reduce the manual and costly work related to the bug localization and bug fixing tasks of software maintenance [22, 49, 51]. While early works in the field mainly used search-based [36, 37] or semantics-based [47, 60] techniques, recently, a different line APR research has proven successful: neural machine translation for program repair, or simply "neural program repair" [10, 11, 13, 42, 68, 83, 85].

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10.
https://doi.org/10.1145/3551349.3556926

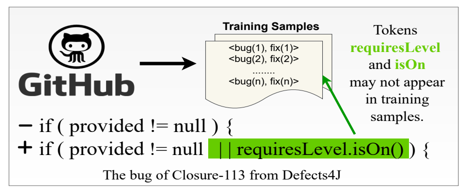

Figure 1: Motivating example of SelfAPR: supervised neural program repair is fundamentally limited by the absence of project-specific tokens at training time.

Neural program repair is based on a common encoder-decoder architecture to transform the buggy code to the correct code, yet the proposed approaches differ as follows: 1) In the input or output representations, for example, the decoders may output code edits [85]; 2) In the pre-processing or post-processing phases of the data, for example filtering the patches with invalid identifiers [31]. When those past works are compared one against the others on the same benchmark [83], those variations explain the performance differences.

Despite those differences, the previous work on neural program repair is dominantly founded on the same machine learning paradigm: supervised learning [26]. In that context, the supervised training samples come from mining real-world commits made by human developers. For example, Recoder's [85] training samples were downloaded from GitHub, totaling 1 083 185 commits done between March 2011 and March 2018. In this paper, we claim and provide evidence that this supervised paradigm for neural program repair has two fundamental limitations as follows.

Problem 1: Lack of project-specific knowledge. Past commits used for training are typically collected from open-source projects. Those projects may have little or nothing to do with the application domain of the program under repair. This is known as the exposure bias problem [5]. In other terms, the training samples do not contain project-specific knowledge. By project-specific knowledge, we refer to specific fixing tokens of variables, expressions and statements, and their semantic relationships that the neural network can use to

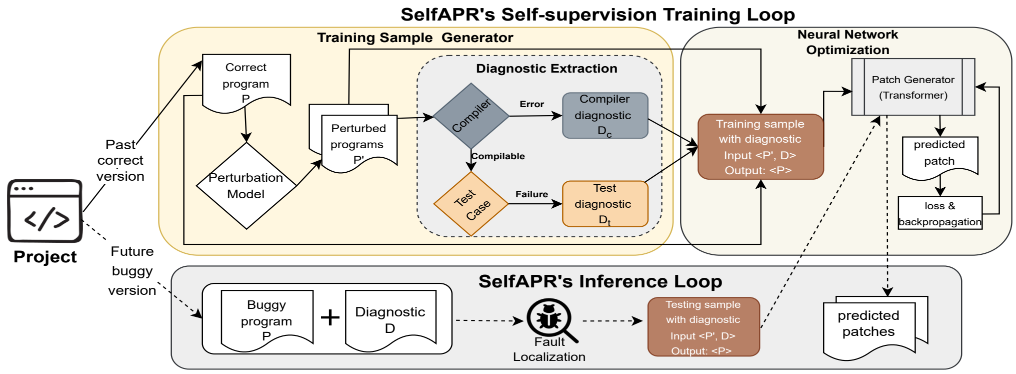

Figure 2: Overview of SelfAPR: the key novel features are the training sample generator and the diagnostic in the input representation.

synthesize a patch. Figure 1 illustrates this point. Bug `Closure-113` from Defects4J benchmark [32] is fixed by expanding a boolean expression with `requiresLevel.isOn()`. There are few projects in the wild that use such tokens. Indeed, none of the existing work on neural program repair can fix bug `Closure-113`, because the project-specific tokens `requiresLevel` and `isOn` have not been seen in training samples. Note that this is the core conceptual limitation, not a quantitative one: crawling more training samples from Github would not solve this problem.

Problem 2: Lack of execution information. Recently, Chakraborty and Ray [11] have shown that neural repair models fail because of a lack of guiding information about the bug type. Our key insight is that execution information is rich in such signals. For instance, it may be useful for the network to know that the bug to be repaired is a `NullPointerException`. For supervised training with past commits, it is virtually impossible to obtain this information at scale because: 1) it is very hard to compile past commits due to the version and dependency hell [59]; 2) many past commits do not come with a way to execute the bug, such as a test suite.

To address the above two problems, we devise, implement and evaluate SelfAPR. The radical novelty of SelfAPR is to be based on self-supervised learning instead of supervised learning. SelfAPR consists of an original training loop on a past version of the project under repair. The core intuition is that the past version contains a wealth of useful project-specific knowledge. The self-supervised learning loop means generating training samples ourselves (instead of mining commits), by perturbing an old version of the project under repair in a careful and systematic manner. This is a potent solution to solve our two problems.

Self-supervised training enables us to learn project-specific knowledge. SelfAPR leans from a past version of the program to be repaired. From a single past version, SelfAPR generates thousands of synthetic training samples, which all contain project-specific tokens by construction. This allows SelfAPR to learn project-specific knowledge, including rare tokens, idiomatic expressions, and semantic relationships between domain identifiers (types, variables, and methods). As we shall see later, the bug `Closure-113` shown in Figure 1 is correctly fixed by SelfAPR, because SelfAPR learns to use project-specific tokens from the self-supervised training samples (recall that no previous work has succeeded in repairing this bug).

Self-supervised learning enables us to add the execution information into training samples. By generating thousands of samples from a single, working, executable version of the project, it becomes doable to compile and test all training samples. In other words, the powerful concept of self-supervision opens the door to embedding execution information in the training loop. Consequently, SelfAPR proposes a novel input representation that includes the execution information, called "diagnostic" for short in the rest of this paper. Put simply, SelfAPR's self-supervised paradigm augments the training samples from `<bug, fix>` to `<bug, execution diagnostic, fix>`.

Overall, SelfAPR works as follows. It first generates thousands of training samples based on perturbation. Next, each perturbed program is executed to see whether it is buggy (does not compile or does not pass the test). For each buggy program, we extract the diagnostic and incorporate it into the input representation. Next, we train SelfAPR with a state-of-the-art transformed-based neural model [70] with this input, using the correct program before perturbation as the expected output. At inference time, we use the error of the failing test case of the program under repair as input diagnostic.

We evaluate our work on 818 bugs from 17 open-source projects from Defects4J [32]. In total, we generate 1 039 873 training samples in a self-supervised manner, each of which contains an error diagnostic. Our experimental results show that SelfAPR succeeds in repairing 65/388 bugs from Defects4J version 1.2 and 45/430 bugs from Defects4J version 2.0, which is a clear improvement over the state-of-the-art. SelfAPR is able to repair 10 bugs that have never been reported to be repaired by the related supervised learning repair approaches. More importantly, generating perturbation-based training samples based on a past version of the project under repair contributes 30% effectiveness of SelfAPR, thanks to SelfAPR's unique capability to learn and reuse project-specific tokens in the synthesized patch. In other terms, this performance breakthrough is due to the paradigm shift from supervised neural networks to self-supervised neural networks.

To sum up, we make the following contributions:

- We devise, SelfAPR, an original self-supervised neural model for program repair based on execution. To our knowledge, this is the first learning architecture that encodes and leverages test execution diagnostics for repair. Notably, SelfAPR is able to learn project-specific repair knowledge in an effective way.

- We perform an original series of experiments and show that SelfAPR repairs 110 bugs from 17 open-source projects. Our experimental results demonstrate that self-supervised learning on a past version of the program under repair significantly increases the repair effectiveness. Our experiments show that including project-specific training samples directly contributes to repairing 20 more bugs (+30%) on Defects4J version 1.2.
- We consolidate and share our training dataset of 1 039 873 buggy training samples, including 408 858 functional errors with at least one failing test case and 631 015 compilation errors. This dataset is valuable for future researchers to understand the syntactic and semantic relationships between errors and code changes responsible for them. We make all our code and data publicly available at https://github.com/SophieHYe/SelfAPR.

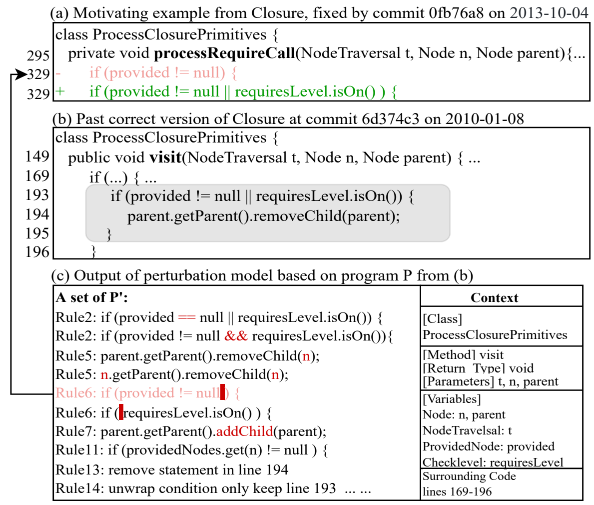

Figure 3: A running example of the perturbation model.

2 BACKGROUND

2.1 Automated Program Repair (APR)

Search-based Repair. Search-based approaches such as GenProg [36], SPR [40] and others [37, 43, 58, 73, 81, 84], typically generate a large number of tentative patches according to different edit patterns, such as inserting null checks [16], mutating operators [66] or copying statement [36]. In the patch search space, they then employ heuristic search algorithms (e.g., genetic programming) to quickly find patches that can pass the given test specification. Search-based approaches suffer from both the immense scale of search space and the effectiveness of search algorithms [55].

Semantics-based Repair. Semantics-based approaches, such as Angelix [47], S3 [35] and others [21, 45, 46, 50, 60, 61, 75, 76], first construct a constraint problem that should be satisfied to fix a bug, and then they use some kind of program synthesis to synthesize patches that satisfy the repair constraints. Semantics-based approaches effectively narrow down search spaces, yet they mostly limit the edit patterns to small-scale expressions in order to make program synthesis tractable [73].

Learning-based Repair. Learning-based approaches based on supervised learning [6, 10, 11, 13, 19, 31, 39, 42, 48, 68, 69, 83, 85] are data-driven, employing pairs of the buggy and fixed code crawled from open-source projects and learning the edit patterns from a large scale training dataset. Learning-based approaches typically rank the patches based on the maximum likelihood estimation of tokens, which potentially narrows down the search space to likely patches. Nevertheless, learning-based approaches treat source code as natural language translation, suffering a lack of knowledge in programming languages.

2.2 Self-supervised Learning

Self-supervised learning is the idea to transform unlabeled data into labeled data, without any human labeling. Self-supervised learning addresses the key limitations of supervised learning when it comes to collecting, handling, cleaning, and labeling training data [14]. In NLP, self-supervised learning has been used with great success for learning word representations [54]. In machine learning on code, it is powerful to capture contextual representations [2, 15, 20, 27] and the typical perturbation strategies are masking out or replacing tokens.

To our knowledge, the idea of self-supervised learning in program repair is largely unexplored. Only Yasunaga and Liang [79] and Allamanis et al. [3] have done preliminary investigations, which will be discussed in Section 7.

3 SELFAPR

3.1 Overview

Figure 2 gives an overview of SelfAPR, where the upper part shows the training phase and the bottom part presents the inference phase. The core idea of SelfAPR is to generate training samples in a fully project-specific and self-supervised manner. In order to be capable of doing project-specific training, we take one project's past version to generate training samples. The learned model is used to repair future bugs from the same project. For self-supervised training, the training buggy samples are generated by perturbing correct programs given as input. This is contrary to the related work on neural program repair which is based on supervised learning with mined past commits [13, 31, 42, 68, 69, 83, 85]. Notably, the perturbation-based programs can be compiled and executed with project-specific test suites while past commits do not.

Training Phase. SelfAPR's training phase consists of two main components: a training sample generator based on perturbing the source code (left part) and a neural transformer architecture fed with an input representation embedding test execution information (right part). Given a correct program $\mathcal{P}$ which can be compiled and executed (i.e., pass test suite specification), SelfAPR generates variants of them, called here "perturbation-based programs" (denoted as $\mathcal{P}'$), according to a perturbation model (Section 3.2). For one single program $\mathcal{P}$, a number of $\mathcal{P}'$ can be generated in a configurable way, depending on the required size of the training dataset. All the generated $\mathcal{P}'$ are compiled and executed by invoking the test suite included in the correct version. Compilation and test execution determine whether the perturbation model introduces a bug, and yields the error type and the error diagnostic $\mathcal{D}$.

Finally, SelfAPR's learning model takes input as $\mathcal{P}'$ and its accompanying diagnostic $\mathcal{D}$. The goal of the machine learning model is to predict the expected output $\mathcal{P}$, the original source code before perturbation. They are denoted as follows:

$$training\_samples = \{input :< \mathcal{P}', \mathcal{D} > output :< \mathcal{P} >\}$$

Perturbation Model. The goal of the perturbation model is to generate training samples with bugs. Perturbation-based programs are valuable for training a neural repair network if they can be considered as buggy in some sense. To determine this, SelfAPR employs a compiler and the available test suite to identify their correctness. If a perturbation-based program $\mathcal{P}'$ produces either a compiler or a test execution error, it is deemed as buggy, and consequently, used as a training sample later. The perturbation model will be described in Section 3.2.

Diagnostic Extraction. Not only SelfAPR uses the compiler and test execution to select buggy training samples, it also uses them to extract diagnostics $\mathcal{D}$ about the bug. The $\mathcal{D}$ is then a first-class part of the neural network input. Our intuition is that error diagnostics could be useful to guide patch generation, for example, the fact that a `NullPointerException` is thrown provides explicit information regarding the repair action to be used. We note that this radically departs from the related work which only uses source code as input [31, 68, 69, 83, 85].

Input Representation. A training sample is represented as a sequence of tokens. We follow the existing neural program repair [11, 31, 42, 83] to include the context code of $\mathcal{P}'$ in the input representation. The context code is enriched with a summary of the following information: 1) class and method name, 2) variables in the buggy method scope and 3) the surrounding source code. For diagnostic $\mathcal{D}$, we concatenate the input representation of diagnostics as a sequence of tokens, which is coming from the compiler or test suite execution failures.

Subtokenization. Once a training sample is represented as a sequence of tokens, we use subtokenization before entering it into the neural network. This is essential in order to reduce vocabulary size [31]. SelfAPR follows [11, 83] to use a sentence-piece tokenizer [34]. Sentence-piece tokenization divides every token into a sequence of subtokens.

Inference Phase. In the inference phase, we use the trained model to repair future bugs from the same project. Specifically, SelfAPR takes an input of a buggy program ($\mathcal{P}$) and the failing test diagnostic $\mathcal{D}$. We employ fault localization (FL) (e.g., Ochiai [1] or Gzoltar [57]) to generate a ranked list of suspicious buggy lines. For a given suspicious statement found by FL, SelfAPR constructs an inference input per our representation with: 1) the suspicious statement and its context, and 2) the test execution diagnostic. This input is given to the patch generator (transformer-based neural model), which outputs the most likely patch, or may enumerate the K best patches for that suspicious statement with beam search, where K is fully configurable, a.k.a, the beam search size.

To our knowledge, our work is novel in two aspects. First, it is the first to propose project-specific training using a different version of the same project under repair. This enables the neural model to learn project-specific knowledge and mitigate the training discrepancy when the training and testing datasets come from different projects. Second, SelfAPR is the first to add compiler and test suite execution information as part of the input representation for neural program repair. As an opposite, the previous supervised-learning studies all consider as input a pair of the buggy and fixed source code. The related two self-supervised learning approaches [3, 79] neither include test execution diagnostics, nor are founded in project-specific training with a past version of the project under repair.

3.2 Perturbation Model

The goal of the perturbation model is to generate training samples with bugs. Recall that the perturbation model generates bugs from a correct program. We train the neural model with the buggy programs as input and learn to generate the correct version as output. In SelfAPR, we design the perturbation rules (*Rules*) driven by learning objectives that we expect the neural model to learn. For example, if the learning objective is to learn how to insert statements, a perturbation model may generate training samples by deleting statements: reversed, the training sample teaches the neural model to learn inserting code that has been deleted.

3.2.1 *Running Example.*

Figure 3 gives a running example to demonstrate how the perturbation model creates the perturbation-based programs ($\mathcal{P}'$) and the corresponding context information. Recall that the motivating example (Figure 1) from project Closure was fixed by the developer on 2013-10-04, shown in Figure 3(a), by adding a clause to an if condition. To create training samples that are useful for this bug, SelfAPR perturbs a past version of Closure and removes a clause in some selected if conditions. We note that although the buggy method `processRequireCall` in (a) does not exist in the past version, there are similar methods that could be perturbed in order to create training samples.

Figure 3(c) shows the output of SelfAPR's perturbation model with a set of $\mathcal{P}'$ at locations 193 and 194 of (b), as well as the context information including class, method, variables and surrounding code. The set of $\mathcal{P}'$ are generated based on different perturbation rules that are described later. One sample generated using $Rule_6$ is a training sample that is useful to learn to fix the bug in Figure 1.

From this running example, we see that first, for one single AST statement, multiple perturbation-based programs are generated. Second, by learning on a past correct version of the project under repair, SelfAPR creates valuable training samples encoded project-specific knowledge to fix a new and unseen bug appearing three years later.

3.2.2 *Perturbation Rules.*

In SelfAPR, we systematically design perturbation rules (called as *Rule* in the following) according to learning objectives. Listing 1 lists the perturbation rules and we now explain them in detail.

Learning objective: learning to only use valid identifiers. The first learning objective is enabling the neural network to understand how to use valid identifiers according to scoping and typing rules. The following rules are designed accordingly.

$Rule_1$ perturbs a correct declaring type with a wrong one. SelfAPR takes specific care of the interchangeability between the types according to the type hierarchy (e.g., replacing Set with List). $Rule_2$

```
Rule1: modify declaring type ...
    - double a;
    + float a;
Rule2: modify operator ==, !=, &&, ||, +, -,*,%, ...
    - if( a > b )
    + if( a >= b )
Rule3: modify literal, "STRING", true, false, 1, 0,...
    - return a.length-2;
    + return a.length-1;
Rule4: modify constructor
    - new ClassA(a,b)
    + new ClassB(b,c)
Rule5-1: modify argument
    - invoc1(a, b)
    + invoc1(a, x)
Rule5-2: swap argumens
    - invoc1(a, b)
    + invoc1(b, a)
Rule6-1: reduce boolean expression
    - if (exp1 && exp2 )
    + if (exp1)
Rule6-2: expand boolean expression
    - if (exp1 && exp2 )
    + if (exp1 && exp2 || exp3)
Rule7-1: modify invocation
    - a.invoc1(a, b)
    + a.invoc1(a, b, c)
Rule7-2: replace invocation
    - a.invoc1(a, b)
    + a.invoc2(a)
Rule8: compound of Rule1 - Rule7
    -  if (exp1 > 1 && exp2 )
    +  if (exp1 >= 0 || exp2 == null )
Rule9: replace by transplanting a similar donor statement
    - target statement
    + donor statement
Rule10: move a later statement before the target statement
    +  later statement;
        target statement;
    -  later statement;
Rule11: transplanting a donor statement
        target statement;
    +  donor statement;
Rule12: wrap target statement with an existing conditional block
    + if (exp1) {
            statement1;...
    +}
Rule13: insert an existing block (if, loop, etc)
    + if(exp1) {
    +   statement1;...
    + }
Rule14: delete statement
        if (exp1!=null && exp1>exp2) {
    -   statement 1; ...
        }
Rule15: unwrap block
    - if (exp1!=null && exp1>exp2) {
            statement1; statement2;
    - }
Rule16: remove block
    - for (exp1) {
    -     statement; ...
    - }
```

Listing 1: Perturbation rules for self-supervised training.

perturbs the correct operator with the wrong one. If an AST statement has more than one operator, SelfAPR iterates over each of them. $Rule_3$ perturbs the correct literal with the wrong one. The literal replacement follows typing constraints. $Rule_4$ perturbs the correct constructor with the wrong or an overloading one, where the added constructor has already been used in the same class file. SelfAPR chooses the wrong constructor with the required variables following typing and scoping constraints. $Rule_5$ perturbs variables: $Rule_{5-1}$ modifies a correct variable with a wrong one, following typing and scoping constraints. Moreover, $Rule_{5-2}$ swaps two arguments if they share the same type. $Rule_7$ perturbs an invocation: $Rule_{7-1}$ modifies the correct invocation with an overloading one (if there exists one). $Rule_{7-2}$ replaces the correct invocation with a new one that appears in the class file. For both rules, SelfAPR synthesizes the arguments following the typing and scoping constraints of available variables. $Rule_8$ generates a compound statement by stacking $Rule_1$ - $Rule_7$ from a correct statement in order to increase the complexity of the learning task for some training samples.

Learning objective: learning to reuse existing code from the same program. Ever since GenProg [36], it is known that reusing code from the program repair is valuable [4, 44]. Hence, we want perturbations that encourage the neural network to reuse code from in the close vicinity of the buggy location.

$Rule_6$ are dedicated to boolean expressions. $Rule_{6-1}$ perturbs boolean expressions by removing a clause. $Rule_{6-2}$ expands a correct boolean expression by transplanting an existing binary expression in the method scope, encouraging clause reuse. $Rule_9$ modifies the correct statement by transplanting a similar statement taken from the class scope, called the donor statement. In SelfAPR, the donor statements are selected based on edit distance with the target statement, and statements with a higher similarity score than the default threshold are selected. Notably, code transplantation is a more generic strategy than the others, it augments the diversity of the generated training samples behind the transformations by the previous *Rules*.

Learning objective: learning to synthesize code according to the context. It is known that code has low entropy, because of its high contextual redundancy [25]. We design the following perturbations in order to for the neural network to learn to synthesize a patch according to the closest surrounding code.

$Rule_{10}$ modifies the order of correct statements. It shuffles the target statement with one from the surrounding 3 context lines of the target statement. $Rule_{14}$ deletes the target statement. The learned repair actions are diverse depending on the characteristics of the deleted statement. $Rule_{15}$ unwraps the condition and only keeps the `then` branch. This enables the neural model to learn to generate conditions that are semantically related to the target statement to be wrapped. $Rule_{16}$ deletes a complete AST block. This enables the neural model to learn to generate complex and multi-line code.

Learning objective: learning to delete. Deleting code is an option for repairing bugs [23, 55]. To train the neural network to delete code, we design perturbations that add superfluous code. Recall that the perturbed code is then given as input, meaning that the expected output is indeed a code removal.

$Rule_{11}$ inserts donor statements randomly before or after the statement under perturbation, where the donor comes from the surrounding code. $Rule_{12}$ transplants a donor conditional expression and wrap a target statement. The conditional expression is taken from the class scope and with a textual similarity with the target statement. $Rule_{13}$ transplants an entire code block before or after the target statement under perturbation. The block must exist in the class scope and with a default textual similarity with the target statement.

Sanity check of perturbation-based training samples. All training samples are validated with the following sanity check: 1) they are different from the correct program; 2) they are unique, i.e., we deduplicate the training samples even if different *Rules* generate the same training samples; 3) they are buggy. Then, we guarantee that the perturbed code indeed triggers a bug by executing them against the compiler and test suite.

Diversity of perturbation-based training samples. Our perturbation rules mix fixed transformations and generic ones. As a result, the generated training samples are diverse, we will give quantitative evidence in Figure 4. This diversity enables the network to learn a variety of repair actions that can go beyond the fixed transformations, as demonstrated in Figure 6.

3.3 Diagnostic Extraction

SelfAPR generates training samples that are guaranteed to be buggy (recall Section 3.2). For each perturbed program $\mathcal{P}'$, SelfAPR executes it against the compiler and available test suite to determine whether it is a valid buggy training sample. Next, SelfAPR extracts a diagnostic $\mathcal{D}$ of the error. The diagnostic may be a compiler error diagnostic (D_c) or a test failure diagnostic (D_t).

If a perturbed program does not compile, D_c is the compiler error message. If a perturbed program $\mathcal{P}'$ is compilable, then $\mathcal{P}'$ is executed against the available test suite. If one test case fails, this results in a test execution diagnostic (D_t).

This diagnostic extraction is a key novelty of our work. The intuition is that they provide signals related to the bug type. The diagnostics enable the neural patch generator to generate patches according to tokens of the error diagnostic. No previous supervised learning-based repair approaches [13, 31, 42, 68, 69, 85] include diagnostics into the input representation.

SelfAPR represents the diagnostics by concatenating them with the context and buggy code (see the input representation in Section 3.1). This means that diagnostics is a considered token sequence, tokenized the same way as the code to leverage the reference to code elements and literals in the diagnostic. The token sequence is separated into four parts. The first part is a special token (`[CE]` for compiler errors and `[FE]` for test execution errors). The second part is the error type, which means in our case the type of the thrown exception: runtime exceptions (e.g., `NullPointerException`) or test-driven exceptions (e.g., `AssertionFailedError`). The third part is the error message and the last part is the failing test method name. An example of a test execution diagnostic in Java would be as follows:

$$\overbrace{\underbrace{[FE]}_{\text{Special token}}\ \underbrace{ComparisonFailure}_{\text{error type}}\ \underbrace{expected:1 but was:0}_{\text{error message}}\ \underbrace{testEquals}_{\text{failing test method name}}}^{\text{Example of Test Execution Diagnostic } (D_t)}$$

Finally, we note that compiler errors are not directly related to our final goal of repairing functional errors (see "Inference Phase" in Section 3.1). However, we have strong conceptual arguments and empirical evidence for doing so. The compiler errors are useful for providing training samples related to project-specific typing and related to scoping information. For example, typical compiler error diagnostics obtained from perturbation are `cannot find symbol` and `incompatible types`. They help the neural network to capture the fact that some identifiers cannot be used in a certain context.

3.4 Neural Architecture and Training

A training sample consists of an expected output $\mathcal{P}$, i.e., the correct program without being perturbed (see Section 3.1), a perturbation-based program $\mathcal{P}'$ (see subsection 3.2), and an error diagnostic $\mathcal{D}$ (see Section 3.3). The architecture provides guarantees that all training samples have been executed at least once.

Training. SelfAPR uses a transformer neural network, which is considered state-of-the-art [2, 11, 20, 31, 42, 83]. The transformer model learns a conditional probability distribution during the training process to translate from the perturbation-based program $\mathcal{P}'$ to the expected correct program $\mathcal{P}$. Given the model parameters θ, the training loop aims at updating θ to maximize the probability (Φ) of generating the correct code given $\mathcal{P}'$ and $\mathcal{D}$:

$$\max_{\theta} \Phi(\mathcal{P}|\mathcal{P}', \mathcal{D}) \tag{1}$$

3.5 Patch Ranking

We follow the typical neural program repair process and employ beam search for patch ranking. The beam search is a greedy algorithm that computes the most likely tokens and ranks the outputs by the maximum likelihood estimation (MLE) score of the overall prediction. Thus, SelfAPR outputs the ordered top K most likely sequences based on the likelihood of each sequence, where K is configured as beam width.

3.6 Implementation

We implement our perturbation model based on Spoon[53] which consists of 85 functions and more than 5K lines of code. We implement SelfAPR's patch generator with state-of-the-art Transformer based architecture [56] from HuggingFace. The encoder and decoder consist of 6 layers. We configure SelfAPR to take a maximum of 384 input tokens from buggy and context code and generate a patch with a maximum of 76 tokens. SelfAPR is trained for 10 epochs, using a batch size of 32 and a vocabulary size of 32 128.

4 EXPERIMENTAL METHODOLOGY

4.1 Research Questions

- **RQ1 (Effectiveness of Self-Supervision)**: To what extent does self-supervised training compare to the state-of-the-art in program repair?
- **RQ2 (Project-specific Training)**: To what extent does project-specific training contribute to the overall effectiveness?
- **RQ3 (Ablation Study)**: To what extent does each component in SelfAPR contribute to the final effectiveness?

4.2 Training and Testing Datasets

We construct a dataset of programs $\mathcal{P}$ to seed the self-supervision loop. Recall that SelfAPR evaluates the perturbation-based training samples with compiler and test suite. This forces the correct programs $\mathcal{P}$ to meet the following two requirements: 1) $\mathcal{P}$ needs to be buildable in order to capture the compiler error diagnostics. 2) $\mathcal{P}$ needs to have a test suite, that can be automatically executed using a test driver, in order to capture test execution failure diagnostic.

Consequently, we look for benchmarks that comply with this hard compilation and testing constraints. We chose to use the widely accepted Defects4J [32] benchmark version 2.0, which is composed of 835 real-world buggy programs from 17 open-source projects, each of which is compilable and executable.

To collect the project-specific training samples, we perturb on the correct past version of the same 17 open-source projects. We split them into training and testing datasets by time as follows: the training dataset of correct programs is made of the fixed version from the earliest commits by the project. All the remaining bugs

Projects	Self-supervised Training							Testing on Real-word Bugs	
	CommitID	Date	LOC	# Test Func	# CE	# FE	# Training Samples	Date since	# Bugs
Closure	6d374c3	2010-01-08	60875	5280	142420	91824	234244	2010-02-05	173
Chart	68e4916	2007-07-06	78566	2444	127391	98562	225953	2007-08-28	25
JacksonDatabind	88f44d8	2013-05-17	42965	2693	96775	36946	133721	2014-05-28	111
Time	e0559c5	2010-12-05	26795	5012	68910	39875	108785	2011-02-15	25
Lang	bb16716	2006-07-21	16623	2522	44912	24098	69010	2006-08-18	63
JacksonCore	b40ac81	2013-08-28	15882	354	37198	23836	61034	2013-09-21	25
JxPath	fab38ab	2007-01-10	19373	441	24741	22325	47066	2007-05-16	21
Collections	a270ff6	2015-06-04	26415	2764	23781	9266	33047	2015-09-28	3
Math	41ba9e0	2006-06-05	9479	1074	17940	14824	32764	2006-07-06	105
Compress	004124a	2009-03-26	6741	105	14004	14796	28800	2009-03-30	46
Gson	c6a4f55	2010-11-02	5418	992	8458	8887	17345	2015-10-22	17
JacksonXml	2d7683e	2016-01-06	4683	436	8012	5442	13454	2016-06-09	5
Codec	52d82d1	2008-04-27	2584	258	4283	8202	12485	2009-07-13	17
Jsoup	27a52f9	2011-07-02	2546	146	3829	3151	6980	2011-07-02	92
Mockito	c1f2c4e	2009-07-09	5506	1060	4190	2283	6473	2009-11-08	37
Cli	b0e1b80	2007-05-15	1937	152	2961	3148	6109	2007-05-22	38
Csv	de1838e	2012-03-27	806	79	1210	1393	2603	2013-04-08	15
Total	-	-	327194	25812	631015	408858	1039873	-	818

Table 1: Training and testing datasets used in our experiments.

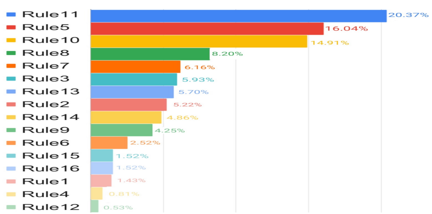

Figure 4: Distribution of training samples generated by different perturbation rules.

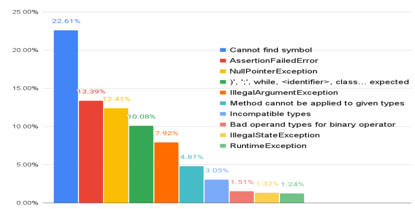

Figure 5: Distribution of errors in the generated training samples.

in the same project from a later commit are used for testing. This is different from all the previous works on neural program repair which consider Defects4J as the only testing dataset, we leverage the bugs from Defects4J for both training and testing.

Training sample generation. Table 1 shows the details of our training and testing sets. The first column gives the name of the open-source project, and the second to the eighth columns give the details of the training data including the number of perturbation-based samples generated. The last two columns give information about the testing set, including starting date of the bugs and the number of bugs. For example, the first row shows that for the Closure project, we use the source code in commit `6d374c3` from `2010-01-08` for generating perturbation-based training samples, which is composed of 60 875 lines of source code over 5 280 test cases. From this data, SelfAPR's perturbation model generates 234 244 training samples, where 142 420 are training samples trigger compiler errors (CE) and 91 824 are training samples trigger functional errors (FE). We note that the number of training samples samples has a positive correlation with both number of source code (LOC) and test functions. This is explainable, because the source code provides the statements being perturbed and test functions specifies functional bugs.

As summarized in the last row, we obtain 1 039 873 training samples, including 631 015 compiler error training samples, 408 858 functional error training samples. All those training samples are obtained by perturbing 17 open-source projects with 327 194 lines of source code and specified with 25 812 test cases in total. Notably, the testing set is composed of all bugs from Defects4J version 2.0 minus the bugs used for training. This gives 818 (835 - 17) testing samples.

Perturbation analysis. Figure 4 shows the distribution of the perturbation-based samples generated with SelfAPR's perturbation rules. From this figure, we make the following observations: 1) All perturbation rules contribute to generating training samples and result in a diversity of training samples. 2) Not all the rules equally generate samples. This is explained by the characteristics of the code under the perturbation. For example, the variable perturbation $Rule_{11}$ leads to the most training samples while the perturbation $Rule_{12}$ yields the least (wrap the target statements with a block).

Diagnostic analysis. We now look at the composition of the diagnostics of the perturbation-based training samples. Figure 5 demonstrates the Top-10 distribution by error type. It can be seen that our perturbation algorithm creates a diverse set of compiler diagnostics and functional diagnostics, with no error type dominating the other. The top-1 diagnostic is `cannot find symbol`, which is a semantic error by the compiler, those training samples are an important signal to the neural network to generate code with tokens complying with the available variables, types and methods according to scoping and typing constraints. The next two are the most common functional error diagnostics (`AssertionFailedError` and `NullPointerException`), which show that our perturbations relate to behavior.

Dataset filtering. Note that our testing Defects4J bugs are all caused by function errors. To ensure the close distribution between training data and testing data, we conduct training dataset filtering as follows: 1) keep the `[FE]` samples and those `[CE]` samples with semantic errors, and 2) remove `[CE]` samples with syntactic compilation errors, e.g., "; is expected ".

			Spectrum-based FL		Perfect FL	
Approaches	# Training	# Beam	D4J (v1.2)	D4J (v2.0)	D4J (v1.2)	D4J (v2.0)
Nopol [76] (no learning)	-	-	1/30	-	2/8	-
DynaMoth [18] (no learning)	-	-	1/22	-	3/13	-
GenProg-A [84] (no learning)	-	-	2/30	-	12/36	-
SimFix [30] (no learning)	-	-	25/68	2/25	29/50	
TBar [37] (no learning)	-	-	24/72	8/50	52/85	
SequenceR [13] (supervised learning)	35 578	50	-	-	14/19	-
CoCoNuT [42] (supervised learning)	24 471 491	1000	-	-	43/85	-
CURE [31] (supervised learning)	24 471 491	1000	-	-	55/102	-
RewardRepair [83] (supervised learning)	2 307 241	200	27/-	24/-	44/-	43/-
Recoder [85] (supervised learning)	103 585	100	**49/90**	19/46	64/-	-
BugLab [3] (self-supervised learning)	415 687	50	-	-	17/27	6/11
SelfAPR (self-supervised learning)	1 039 873	50	39/65	**28/42**	**65/79**	**45/51**

Table 2: SelfAPR's effectiveness w.r.t. the state-of-the-art over two testing datasets. In the cells, x/y : x denotes the number of correct patches, and y denotes the number of plausible patches that pass all human-written test-suite. A '-' indicates that the APR approach has not been evaluated on the considered benchmark.

4.3 Patch Verification

The patch verification follows the existing related work [13, 31, 42, 85]. We first execute all patches with the compiler and human-written test suite to identify plausible patches. Then, the plausible patches are executed against independent automatically generated test cases by prior work [82]. Lastly, we manually analyze the patches based on the ground truth developer's patch. All manual analysis results are confirmed by two authors, to avoid human errors and author bias. To sum up, a patch is deemed correct if 1) it is plausible according to the developer-written and the augmented test suite [82], and 2) it is identical to the developer patch or if it is considered as correct by manual analysis done by two authors.

4.4 Methodology for RQ1

In RQ1, we compare SelfAPR against the state-of-the-art of 1) supervised learning repair approaches: SequenceR [13], CoCoNuT [42], CURE [31], Recoder [85], and RewardRepair [83]. We do not include TFix [6] because it is trained on JavaScript, which cannot be used to evaluate Defects4J bugs in Java. 2) Semantics-based repair approaches: Nopol [76] and DynaMoth [18]; 3) Search-based repair approaches: GenProg-A [84] (the Java implementation of Genprog [36]), SimFix [30] and TBar [37].

Moreover, we re-implement the Java version of the self-supervised learning approach of BugLab [3] (originally designed for Python) with all four perturbation rules regarding operators, variables and literals. Recall that the goal of BugLab is not to obtain project-specific training samples, thus it is not explicitly executed on a past version of the project under repair. For a fair comparison, we run BugLab on the same considered past projects as SelfAPR, which results in 415 687 training samples generated. Notably, there is no diagnostic included in BugLab's perturbation-based training samples by its construction.

We report the quantitative results from the corresponding papers and repositories [38]. Recall that we use 17 Defects4J bugs for training. For a fair comparison, we also remove those bugs from their reported results. We follow the related work by employing spectrum-based fault localization [77, 78] (FL) Gzoltar [57] and also assuming the fault has been correctly localized [13, 31, 42], an evaluation technique known as the perfect fault localization assumption [38], so that our work could be fairly compared with theirs. For a fair comparison, we follow the related work to use a beam search size of 50, which is the common range of considered beam width [13, 69, 85]. We follow the related work [31, 42] to use ensemble training models from 10 training epochs for patch generation.

We compute the two APR performance metrics on the testing dataset: 1) the number of bugs that are correctly repaired, and 2) the ranking information of correct patches configured by beam width in the beam search algorithm per the developer acceptance perspective shown by Noller et al. [51].

4.5 Methodologies for RQ2 & RQ3

In RQ2, we evaluate the effectiveness of SelfAPR with and without training samples from the project under repair. Recall that in RQ1, our training set and testing set contain the same projects, while the testing test comes from the latter commits to guarantee the fairness and the practical usage of SelfAPR (i.e., make sure the fixes are not used during the training). Nevertheless, the training samples from a past version contain project-specific context, e.g., code tokens and similar expressions.

Consequently, in RQ2, we exclude project-specific training samples. We create one new training set per project by discarding perturbation-based samples from the project under repair. For example, to test SelfAPR's effectiveness on project `Chart`, we create a training set without the 225 953 training samples from `Chart`. We compute the metric about the number of correctly repaired bugs as in RQ1.

In RQ3, we conduct an ablation study to evaluate the contribution of each component to SelfAPR, specifically, we respectively remove three components from SelfAPR: diagnostics $\mathcal{D}$, FE training samples, and CE training samples. We train three models without each of the above components (ablation) and evaluate them on Defects4J (v1.2) per the number of correctly repaired bugs.

5 EXPERIMENTAL RESULTS

5.1 Answers to RQ1: Effectiveness of Self-supervision

In RQ1, we compare the effectiveness of SelfAPR with the state-of-the-art in APR. Table 2 shows the patch generation results of SelfAPR and related work on two versions of Defects4J benchmark [32]: D4J (v1.2) and D4J (v2.0) with both spectrum-based fault localization (FL) and the perfect fault localization assumption. The first column is the approach name and its bibliographic reference.

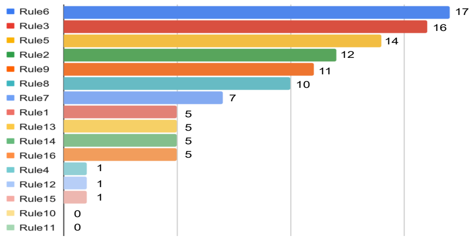

Figure 6: Correlation between the number of repaired bugs according to the corresponding perturbation rule.

The second and the third column give the experimental setup, incl. the number of training samples and the beam search configuration for learning-based approaches. The fourth to seven columns show the number of correct patches and plausible patches by each APR approach on the two considered benchmarks, denoted in the format of x/y. The results are those reported in the literature, either in the original paper or in subsequent comparative experiments [38]. As shown, approaches repair a smaller number of bugs on D4J (v2.0) than D4J (v1.2), suggesting that bugs from D4J (v2.0) are more difficult to repair.

In total, SelfAPR correctly repairs 39 bugs from D4J (v1.2) and 28 bugs from D4J (v2.0) with spectrum-based FL. This number is increased to 65 and 45 respectively when providing perfectly localized buggy locations. Overall, our experimental results validate the novel concept of generating perturbation-based training samples based on a past version of projects under repair. SelfAPR outperforms all related work but Recoder on benchmark D4J (v1.2) with spectrum-based FL. This could be explained by effectiveness from different FLs and potential benchmark overfitting to D4J (v1.2) [17]. In the following, our comparison fully focuses on the 110 bugs repaired with perfect FL, for a fair comparison with the closely related work.

Correlations between repaired bugs and perturbation rules. We look at how the 110 repaired bugs correlate to the perturbation rules. We map the repaired bugs to the corresponding perturbation rules based on manual analysis. Figure 6 gives the result as a bar chart. The 14/16 perturbation rules contribute to repair at least one buggy program, showing their usefulness and complementarity. We note that the number of training samples for a rule is not linearly related to the corresponding bug. For example, the top-1 repaired bug type relates to $Rule_6$: restrict/relax wrong boolean expressions, however, the proportion of training samples generated by $Rule_6$ is low compared to the rest. This suggests that learning happens across perturbation rules.

Comparison to BugLab. BugLab [3] is the closest related work. As seen in Table 2, SelfAPR outperforms BugLab by a large margin. The reason comes from the perturbation model. The one of BugLab is narrow and restricted to specific bug types. On the contrary, SelfAPR's perturbation model considers more rules and they are generic in nature. This results in diversity and genericity of training samples. In particular, no code transplantations and no deletions are considered in BugLab. This not only decreases the chances of learning to insert and learning to delete, but also fails to generate more training samples with project-specific knowledge (usage of domain types and methods).

Top-1 (beam=1)	Top-5 (beam=5)	Top-10 (beam=10)	Top-20 (beam=20)	Top-30 (beam=30)	Top-40 (beam=40)
30.8%	53.8%	60.0%	75.4%	83.1%	92.3%

Table 3: The ranking information of correct patches w.r.t the beam search configuration.

Deeply integrated prioritization. SelfAPR outperforms the related work on search-based and template-based repair approaches. One key reason may relate to enumeration and prioritization. In search-based and template-based repair, one has to enumerate all possible solutions for a given transformation point or template hole. To prioritize some patches, ad hoc solutions based on heuristics are baked into the enumeration. On the contrary, SelfAPR has built-in prioritization. SelfAPR learns to prioritize repair actions in a fully data-driven manner based on the training set, with no manually defined prioritization rules or heuristics [36, 74]. This results in a natural and effective patch ranking. Table 3 shows the distribution of the 110 correct patches' ranking position output by SelfAPR. We can see that 60.0% of correct patches are ranked in the top-10 by the beam search algorithm. According to recent work [51], correct patches ranked at top-10 are important for developers to accept in practice.

Quantity versus quality of training samples. SelfAPR is trained on fewer samples than CoCoNuT, CURE and RewardRepair, yet yields better performance. This suggests the perturbation-based training samples are of higher quality and contain more information. There are two reasons for it: project-specific knowledge and no noisy commits. For supervised program repair, the past commits used for training from GitHub suffer from many limitations: they are not guaranteed to be atomic bug fix commits [62] and they may include unrelated code changes (e.g., new functions, comments and optimization, etc). On the contrary, all training samples generated by our perturbation model are controlled and guaranteed to bug-triggering samples with no unrelated changes.

Uniquely repaired bugs. Compared with all learning-based repair approaches, SelfAPR uniquely repairs 10 bugs that have never been reported as repaired by other learning-based approaches, while the other three approaches CURE, Recoder and RewardRepair also uniquely repair respectively 6, 9, and 2 different bugs. We manually look at the patches for those 10 uniquely repaired bugs, that are repaired thanks to the learned project-specific knowledge and explicit test diagnostics. The result shows the complementary between SelfAPR and other supervised learning approaches, which further suggests the usage of combination training samples from self-supervised learning and supervised learning, even pre-training models [9].

> Answer to RQ1: SelfAPR correctly fixes 65 and 45 bugs for D4J (v1.2) and D4J (v2.0) respectively. This state-of-the-art performance is explained by 1) SelfAPR's novel project-specific training loop, providing essential domain knowledge for repair (project-specific tokens and their semantic relationships); 2) SelfAPR's novel input representation based on execution diagnostics.

```
- for (int i = 0; i < weights.length; i++) {
+ for (int i = begin; i < begin + length; i++) {
```

Listing 2: SelfAPR's patch for Math-41, reusing statements in project-specific samples.

```
- if (target != null ) {
+ if (target != null && target.getType() == Token.STRING ) {
```

Listing 3: Patch for Closure-57 only repaired by SelfAPR.

```
for (int i = 0; i < array.length; i++) {
- classes[i] = array[i].getClass(); {
+ classes[i] = array[i] == null ? null : array[i].getClass();
```

(a) SelfAPR's patch for Lang-33, identical to the human-written patch.

```
Diagnostic: [FE] java.lang.NullPointerException
```

(b) Diagnostic for bug Lang-33

Listing 4: SelfAPR's patch for Lang-33 guided by diagnostics.

5.2 Answers to RQ2: Project-specific Training

In RQ2, we explore in-depth the importance of project-specific training, per the original protocol described in Section 4.5. Table 4 shows the effectiveness of SelfAPR with and without project-specific training samples. The first column gives the test project of D4J (v1.2). The second column shows the number of bugs correctly repaired without project-specific training samples. The third column shows the number of bugs repaired by including project-specific samples, summing to 65 as reported in Table 2. The improvement percentage is given in the last column.

Over all six projects, SelfAPR's model without project-specific training samples correctly repairs 45 bugs. On the contrary, SelfAPR with project-specific training samples correctly repairs 65 bugs, which represents 20 more bugs and an overall improvement of 30.8%. This shows that project-specific training with perturbation-based training samples for the project under repair is valuable. The largest improvement is for project `Closure` (+40.0%).

Project-specific training samples with reusable statements and expressions. Listing 2 gives a SelfAPR's patch for bug `Math-41`, which is identical to the human-written patch. This bug is only repaired by including training samples from the project under repair, here `Math`. Fixing this bug is non-trivial, because it requires correctly updating the initialization value of `i` from `0` to `begin`, and correctly updating `weights.length` to `begin+length`, in a single fixing attempt. By analyzing the project-specific training samples, we see that the same fixing for-loop statement (in green) appears over 300 times. Notably, the version of project `Math` used for self-supervised training was from `2006-06-05`, and the bug `Math-41` was being fixed on `2011-11-30`. Despite learning on a five years old version, SelfAPR captures valuable information from the project-specific training samples and makes a valuable contribution to a new and unseen bug appearing five years after. This clearly shows that SelfAPR succeeds in capturing project-specific knowledge in the neural network.

Project-specific training samples enable learning semantic relationship between unique tokens.A past version not only enables the neural model to learn to use unique fixing tokens, it also enables the network to capture their semantic relationships. For example, Listing 3 shows a bug from `Closure-57` only repaired by SelfAPR in the literature. In this bug, there is no identical fixing expression `target.getType()==Token.STRING` in the training samples. Yet, the two project-specific tokens `target.getType()` and `Token.STRING` separately appear in the training samples. SelfAPR learns to repair the bug with the semantic relationship (co-occurrence) of these expressions based on unique tokens. This bug is only repaired by including project-specific training samples.

Project	SelfAPR w/o Project	SelfAPR with Project	Improvement
Chart	6	7	+1 (+14.3%)
Closure	12	20	+8 (+40.0%)
Lang	7	10	+3 (+30.0%)
Math	16	22	+6 (+27.3%)
Mockito	2	3	+1 (+33.3%)
Time	2	3	+1 (+33.3%)
Total	45	65	+20 (+30.8%)

Table 4: Effectiveness of SelfAPR with and without project-specific training samples.

Project	SelfAPR only CE	SelfAPR only FE	SelfAPR w/o D	SelfAPR
Chart	4 (-42.9%)	7 (-0%)	7 (-0%)	7
Closure	10 (-50.0%)	16 (-20.0%)	17 (-15.0%)	20
Lang	6 (-40.0%)	9 (-10.0%)	8 (-20.0%)	10
Math	12 (-45.5%)	21 (-4.55%)	19 (-13.6%)	22
Mockito	1 (-66.7%)	2 (-33.3%)	2 (-33.3%)	3
Time	1 (-66.7%)	4 (+33.3%)	3 (-0.0%)	3
Total	34 (-47.7%)	59 (-9.23%)	56 (-13.8%)	65

Table 5: Ablation study for SelfAPR.

> Answer to RQ2: Project-specific training on a past version of the project under repair contributes to the 30% effectiveness improvement of SelfAPR. This novel and original training strategy is a key: 1) for helping the model identify rare, yet important project-specific tokens, that may be outside the buggy context at inference time, and 2) for encouraging the model to reuse valuable domain-specific statements and expressions from the program under repair.

5.3 Answers to RQ3: Ablation Study

In this RQ, we do an ablation study by training SelfAPR on training sample with only compiler errors (CE), with only functional errors (FE), without including diagnostics (SelfAPR w/o D). Table 5 gives the corresponding results. As shown in Table 5, the three ablated models respectively repair 34, 59, 56 bugs for Defects4J (v1.2), which are fewer than the whole SelfAPR model, which repairs 65. This demonstrates the necessity of each and every component.

Specifically, we make the following observations: First, the FE training samples yield higher performance, and hence have a better value than CE training samples. This is because the neural model cannot learn enough repair actions from CE samples only, for example, an operator replacement rarely causes a compiler error. Yet, CE training samples are important, because they provide more training samples encoding project-specific knowledge, and they help to capture typing and scoping constraints that the compiler checks. Second, having the diagnostic in the input representation is essential for SelfAPR to generate 9 more correct patches. For example, the bug `Lang-33` shown in Listing 4, is only repaired by SelfAPR trained with FE training samples, clearly guided by the `NullPointerException` diagnostic.

Answer to RQ3: All components of SelfAPR are important: the compiler error training samples, the functional error training samples, and the diagnostics. The most important component to the final effectiveness of SelfAPR is the input representation with test execution diagnostics.

6 DISCUSSION

6.1 Threats to Validity

An internal threat relates to 1) our implementation of the perturbation model may contain bugs that could prevent generating more appropriate perturbation-based training samples and 2) manual patch correctness assessment. To mitigate these threats, we make the perturbation tool and generated patches publicly available for further assess with related techniques [63–65, 74, 80]. A threat to external validity relates to whether the performance of SelfAPR generalizes to arbitrary programming languages. Per the standards of the field, our approach has been tested in one language (Java) and the evaluation is carried out on well-established benchmarks. In principle, our approach can be applied to other programming languages and datasets.

6.2 Multi-location Bugs

Repairing bugs spread in different locations (e.g., multi-hunk bugs) remains challenging for the program repair community. Our work SelfAPR, as much as the related work does not succeed in repairing the 420/818 multi-location bugs in Defects4J. The complexity of repairing multi-location bugs not only comes from the fixing tokens and expressions, but also from the interaction between fixes in different locations. To our knowledge, only three prior work from semantics-based and search-based approaches target on multi-location bugs: Angelix [47], VarFix [73] and Hercules [58].

7 RELATED WORK

We have already discussed in Section 2 the recent related work on APR and in Section 4.4 about close related work on neural program repair with supervised training.

7.1 Creation of Perturbed Programs

There are other techniques and usages for perturbating programs. For example, the mutants of mutation testing tools [29, 33] can be considered as perturbed programs. However, the mutation testing operators are meant to emulate likely programmer errors. In this paper, the goal of perturbation is entirely different, it is to create valuable training data points according to specifical learning objectives. Patra and Pradel [52] create perturbed programs to complement mutation testing with a neural approach. They use learned token embeddings that encode the semantic similarities of identifiers and literals in the perturbation process. Different from above work, our perturbation approach is not neural, it is based on code transformation at the AST level with program analysis. While their model is restricted to low-level modifications of operators, identifiers and literals, SelfAPR generates samples with a larger functional impact based on code transplantations and deletions.

7.2 Self-supervised Learning on Code

Self-supervised learning based APR has been little explored. Loriot et al. [41] devise a self-supervised learning loop to repair formatting issues. Yasunaga and Liang [79] propose self-supervised learning for repairing compilation errors. In both cases, the idea is to do character level perturbations (e.g., replacing or deleting punctuation). On the contrary, we use AST level perturbations, which are a much larger scope and are more impactful. Only our AST level perturbation can trigger functional errors and learn to repair them. Allamanis et al. [3] train a bug detection and repair model called BugLab. The key differences with our work are that they do not execute the perturbation-based programs, therefore, no test diagnostics are included into the input representation of a bug. SelfAPR is the first to generate training samples in a project-specific manner with a past version of the project under repair.

A line of work considers self-supervised learning for other downstream tasks than program repair, e.g., code retrieval and code summarization [2, 7, 8, 20]. The perturbation strategies employed are typically based on token masking or single token perturbation. None of those previous works involves a perturbation model at the AST level that needs to respect strict constraints of programming languages (e.g., variable scopes) as SelfAPR does.

7.3 Training based on Execution

A series of works include test case execution as input to train a neural model. Foivos et al. [67] extract test execution traces to train a neural model to learn to distinguish runtime patterns for passing versus failing executions for a given program. Emad et al. [28] propose to represent the test execution traces to a fixed-length numerical vector for neural model training. Mesbah et al. [48] extract compiler diagnostic information as an input source for repairing compilation errors but do not use execution information. Wang and colleagues [71, 72] leverage execution trace to learn semantic aspects in program embeddings. These works are not about repair from diagnostics, as we do in this paper. Chen et al. [12] and Gupta et al. [24] propose execution-guided synthesis. Those works do not execute test case diagnostics as we do hence cannot capture assertion failures. Furthermore, they do not use self-supervision and perturbation-based programs to learn the semantics of errors.

8 CONCLUSION

We present a novel neural program repair model called SelfAPR, which introduces two major novelties wrt the related work: First, it uses self-supervision with automatically generated training samples with program perturbation. Secondly, it adds execution information into the input representation that captures the failing assertion of the program under repair. Thanks to those two key features, SelfAPR repairs 110 out of 818 bugs from Defects4J. Notably, 10 of them were never repaired before by the supervised learning repair approaches, demonstrating the value and power of project-specific training and test diagnostics embedded.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for the insightful feedback. This work was supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. Some experiments were performed on resources provided by the Swedish National Infrastructure for Computing. We thank Xuechun Xu from KTH Royal Institute of Technology for the support of computing resources.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. 2007. On the Accuracy of Spectrum-based Fault Localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*. 89–98.

[2] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Association for Computational Linguistics(ACL)*. 2655–2668.

[3] Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. 2021. Self-Supervised Bug Detection and Repair. In *Advances in Neural Information Processing Systems*.

[4] Earl T. Barr, Yuriy Brun, Premkumar T. Devanbu, Mark Harman, and Federica Sarro. 2014. The Plastic Surgery Hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 306–317.

[5] Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. 2015. Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1* (Montreal, Canada) *(NIPS'15)*. MIT Press, 1171–1179.

[6] Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. 2021. TFix: Learning to Fix Coding Errors with a Text-to-Text Transformer. In *Proceedings of the 38th International Conference on Machine Learning*, Vol. 139. 780–791.

[7] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2021. InferCode: Self-Supervised Learning of Code Representations by Predicting Subtrees. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1186–1197.

[8] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2021. Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM, 511–521.

[9] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar Devanbu, and Baishakhi Ray. 2022. NatGen: Generative pre-training by "Naturalizing" source code. https://doi.org/10.48550/ARXIV.2206.07585

[10] 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

[11] Saikat Chakraborty and Baishakhi Ray. 2021. On Multi-Modal Learning of Editing Source Code. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[12] Xinyun Chen, Chang Liu, and Dawn Song. 2019. Execution-Guided Neural Program Synthesis. In *International Conference on Learning Representations*.

[13] Z. Chen, S. J. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus. 2019. SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Transactions on Software Engineering* (2019).

[14] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. 2011. Natural Language Processing (Almost) from Scratch. 12 (2011), 2493–2537.

[15] Andrew M Dai and Quoc V Le. 2015. Semi-supervised Sequence Learning. In *Advances in Neural Information Processing Systems*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett (Eds.), Vol. 28. Curran Associates, Inc.

[16] Thomas Durieux, Benoit Cornu, Lionel Seinturier, and Martin Monperrus. 2017. Dynamic patch generation for null pointer exceptions using metaprogramming. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 349–358. https://doi.org/10.1109/SANER.2017.7884635

[17] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019. Empirical Review of Java Program Repair Tools: A Large-scale Experiment on 2,141 Bugs and 23,551 Repair Attempts. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 302–313.

[18] Thomas Durieux and Martin Monperrus. 2016. DynaMoth: Dynamic Code Synthesis for Automatic Program Repair. In *2016 IEEE/ACM 11th International Workshop in Automation of Software Test (AST)*. 85–91.

[19] Zhiyu Fan, Xiang Gao, Abhik Roychoudhury, and Shin Hwei Tan. 2022. Improving automatically generated code from Codex via Automated Program Repair. https://doi.org/10.48550/ARXIV.2205.10583

[20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Empirical Methods in Natural Language Processing(EMNLP)*.

[21] Xiang Gao, Sergey Mechtaev, and Abhik Roychoudhury. 2019. Crash-Avoiding Program Repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. ACM, 8–18.

[22] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2017. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* (2017).

[23] Davide Ginelli, Matias Martinez, Leonardo Mariani, and Martin Monperrus. 2022. A comprehensive study of code-removal patches in automated program repair. *Empirical Software Engineering* 27 (2022).

[24] Kavi Gupta, Peter Ebert Christensen, Xinyun Chen, and Dawn Song. 2020. Synthesize, Execute and Debug: Learning to Repair for Neural Program Synthesis *(NIPS'20)*.

[25] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering*.

[26] Geoffrey E. Hinton. 2007. Learning multiple layers of representation. *Trends in Cognitive Sciences* 11 (2007), 428–434.

[27] Jeremy Howard and Sebastian Ruder. 2018. Universal Language Model Fine-tuning for Text Classification. 328–339. https://doi.org/10.18653/v1/P18-1031

[28] Emad Jabbar, Soheila Zangeneh, Hadi Hemmati, and Robert Feldt. 2022. Test2Vec: An Execution Trace Embedding for Test Case Prioritization. https://doi.org/10.48550/ARXIV.2206.15428

[29] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678. https://doi.org/10.1109/TSE.2010.62

[30] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space with Existing Patches and Similar Code *(ISSTA)*.

[31] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *Proceedings of the ACM/IEEE 43rd International Conference on Software Engineering*.

[32] Rene Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 437–440.

[33] René Just, Bob Kurtz, and Paul Ammann. 2017. Inferring Mutant Utility from Program Context. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. ACM, 284–294. https://doi.org/10.1145/3092703.3092732

[34] Taku Kudo and John Richardson. 2018. SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing. In *Empirical Methods in Natural Language Processing(EMNLP)*.

[35] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: Syntax- and Semantic-guided Repair Synthesis via Programming by Examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*.

[36] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[37] 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*. ACM, 31–42.

[38] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 2020. On the Efficiency of Test Suite Based Program Repair: A Systematic Assessment of 16 Automated Repair Systems for Java Programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. ACM, 615–627. https://doi.org/10.1145/3377811.3380338

[39] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic Inference of Code Transforms for Patch Generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. ACM, 727–739.

[40] Fan Long and Martin Rinard. 2015. Staged Program Repair with Condition Synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy). ACM, 166–178.

[41] Benjamin Loriot, Fernanda Madeiral, and Martin Monperrus. 2022. Styler: Learning Formatting Conventions to Repair Checkstyle Violations. *Empirical Software Engineering, Springer* (2022). https://doi.org/10.1007/s10664-021-10107-0

[42] 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 *(ISSTA 2020)*.

[43] Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java. In *Proceedings of ISSTA*.

[44] Matias Martinez, Westley Weimer, and Martin Monperrus. 2014. Do the Fix Ingredients Already Exist? An Empirical Inquiry into the Redundancy Assumptions of Program Repair Approaches. In *ICSE - 36th IEEE International Conference on Software Engineering*. https://doi.org/10.1145/2591062.2591114

[45] Sergey Mechtaev, Xiang Gao, Shin Hwei Tan, and Abhik Roychoudhury. 2018. Test-Equivalence Analysis for Automatic Patch Generation. *ACM Trans. Softw. Eng. Methodol.* 27, 4, Article 15 (oct 2018), 37 pages.

[46] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. DirectFix: Looking for Simple Program Repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 448–458. https://doi.org/10.1109/ICSE.2015.63

[47] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*.

[48] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. 2019. DeepDelta: Learning to Repair Compilation Errors. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. ACM, 925–936. https://doi.org/10.1145/3338906.3340455

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

[50] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. 772–781.

[51] Yannic Noller, Ridwan Shariffdeen, Xiang Gao, and Abhik Roychoudhury. 2022. Trust Enhancement Issues in Program Repair. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering*.

[52] Jibesh Patra and Michael Pradel. 2021. Semantic Bug Seeding: A Learning-Based Approach for Creating Realistic Bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. ACM, 906–918. https://doi.org/10.1145/3468264.3468623

[53] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience* 46 (2015), 1155–1179. https://doi.org/10.1002/spe.2346

[54] Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. GloVe: Global Vectors for Word Representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 1532–1543. https://doi.org/10.3115/v1/D14-1162

[55] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*.

[56] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67.

[57] André Riboira and Rui Abreu. 2010. The GZoltar Project: A Graphical Debugger Interface *(TAIC PART'10)*. Springer-Verlag, Berlin, Heidelberg.

[58] 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)*. 13–24.

[59] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. 2014. Programmers' Build Errors: A Case Study (at Google). In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. ACM, 724–734. https://doi.org/10.1145/2568225.2568255

[60] Ridwan Shariffdeen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2021. Concolic Program Repair. In *42nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[61] Ridwan Shariffdeen, Shin Hwei Tan, Mingyuan Gao, and Abhik Roychoudhury. 2021. Automated Patch Transplantation. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*. 1–36.

[62] Bo Shen, Wei Zhang, Christian Kästner, Haiyan Zhao, Zhao Wei, Guangtai Liang, and Zhi Jin. 2021. SmartCommit: A Graph-Based Interactive Assistant for Activity-Oriented Commits. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. ACM, 379–390.

[63] Haoye Tian, Yinghua Li, Weiguo Pian, Abdoul Kader Kabore, Kui Liu, Andrew Habib, Jacques Klein, and Tegawendé F Bissyandé. 2022. Predicting Patch Correctness Based on the Similarity of Failing Test Cases. *ACM Transactions on Software Engineering and Methodology* (2022). https://doi.org/10.1145/3511096

[64] Haoye Tian, Kui Liu, Yinghua Li, Abdoul Kader Kaboré, Anil Koyuncu, Andrew Habib, Li Li, Junhao Wen, Jacques Klein, and Tegawendé F Bissyandé. 2022. The Best of Both Worlds: Combining Learned Embeddings with Engineered Features for Accurate Prediction of Correct Patches. *arXiv preprint arXiv:2203.08912* (2022).

[65] Haoye Tian, Xunzhu Tang, Andrew Habib, Shangwen Wang, Kui Liu, Xin Xia, Jacques Klein, and Tegawendé F Bissyandé. 2022. Is this Change the Answer to that Problem? Correlating Descriptions of Bug and Code Changes for Evaluating Patch Correctness. *arXiv preprint arXiv:2208.04125* (2022).

[66] Christopher Steven Timperley, Susan Stepney, and Claire Le Goues. 2017. An Investigation into the Use of Mutation Analysis for Automated Program Repair. In *SSBSE*.

[67] Foivos Tsimpourlas, Ajitha Rajan, and Miltiadis Allamanis. 2021. Supervised Learning over Test Executions as a Test Oracle. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing (SAC '21)*. 1521–1531.

[68] 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 International Conference on Software Engineering* (Montreal, Quebec, Canada). 25–36.

[69] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *ACM Trans. Softw. Eng. Methodol.* 28, 4, Article 19 (Sept. 2019), 29 pages.

[70] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, undefinedukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 6000–6010.

[71] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Dynamic Neural Program Embedding for Program Repair *(ICLR)*.

[72] Ke Wang and Zhendong Su. 2020. Blended, Precise Semantic Program Embeddings. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. ACM, 121–134.

[73] Chu-Pan Wong, Priscila Santiesteban, Christian Kästner, and Claire Le Goues. 2021. VarFix: Balancing Edit Expressiveness and Search Effectiveness in Automated Program Repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 354–366.

[74] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th International Conference on Software Engineering*.

[75] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise Condition Synthesis for Program Repair. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, 416–426. https://doi.org/10.1109/ICSE.2017.45

[76] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, Sebastian Lamelas, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering* (2016).

[77] Deheng Yang, Yuhua Qi, and Xiaoguang Mao. 2017. An empirical study on the usage of fault localization in automated program repair. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 504–508.

[78] Deheng Yang, Yuhua Qi, Xiaoguang Mao, and Yan Lei. 2021. Evaluating the usage of fault localization in automated program repair: an empirical study. *Frontiers of Computer Science* 15, 1 (2021), 1–15.

[79] Michihiro Yasunaga and Percy Liang. 2020. Graph-based, Self-Supervised Program Repair from Diagnostic Feedback. In *International Conference on Machine Learning (ICML)*.

[80] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2021. Automated Classification of Overfitting Patches with Statically Extracted Code Features. *IEEE Transactions on Software Engineering* (2021).

[81] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2021. A comprehensive study of automatic program repair on the QuixBugs benchmark. *Journal of Systems and Software* 171 (2021), 110825.

[82] He Ye, Matias Martinez, and Martin Monperrus. 2021. Automated patch assessment for program repair at scale. *Empirical Software Engineering* 26, 2 (2021), 20.

[83] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural Program Repair with Execution-based Backpropagation. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering*.

[84] Yuan Yuan and Wolfgang Banzhaf. 2018. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. In *IEEE Transactions on Software Engineering*.

[85] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A Syntax-Guided Edit Decoder for Neural Program Repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. ACM, 341–353.