Practical Program Repair *via* Preference-based Ensemble Strategy

Wenkang Zhong
National Key Laboratory for Novel Software Technology, Nanjing University, China
Nanjing, China
zhongwenkang97@foxmail.com

Chuanyi Li*
National Key Laboratory for Novel Software Technology, Nanjing University, China
Nanjing, China
lcy@nju.edu.cn

Kui Liu
Huawei Software Engineering Application Technology Lab
Hangzhou, China
brucekuiliu@gmail.com

Tongtong Xu
Huawei Software Engineering Application Technology Lab
Hangzhou, China
xutongtong9@huawei.com

Jidong Ge*
National Key Laboratory for Novel Software Technology, Nanjing University, China
Nanjing, China
gjd@nju.edu.cn

Tegawendé F. Bissyandé
University of Luxembourg
Luxembourg
tegawende.bissyande@uni.lu

Bin Luo
National Key Laboratory for Novel Software Technology, Nanjing University, China
Nanjing, China
luobin@nju.edu.cn

Vincent Ng
Human Language Technology Research Institute, University of Texas at Dallas
Richardson, Texas, USA
vince@hlt.utdallas.edu

arXiv:2309.08211v1 [cs.SE] 15 Sep 2023

ABSTRACT

To date, over 40 Automated Program Repair (APR) tools have been designed with varying bug-fixing strategies, which have been demonstrated to have complementary performance in terms of being effective for different bug classes. Intuitively, it should be feasible to improve the overall bug-fixing performance of APR via assembling existing tools. Unfortunately, simply invoking all available APR tools for a given bug can result in unacceptable costs on APR execution as well as on patch validation (via expensive testing). Therefore, while assembling existing tools is appealing, it requires an efficient strategy to reconcile the need to fix more bugs and the requirements for practicality. In light of this problem, we propose a **P**reference-based **E**nsemble **P**rogram **R**epair framework (**P-EPR**), which seeks to effectively rank APR tools for repairing different bugs. **P-EPR** is the first non-learning-based APR ensemble method that is novel in its exploitation of repair patterns as a major source of knowledge for ranking APR tools and its reliance on a dynamic update strategy that enables it to immediately exploit and benefit from newly derived repair results. Experimental results show that **P-EPR** outperforms existing strategies significantly both in flexibility and effectiveness.

*Corresponding authors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3623310

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

program repair, ensemble strategy

ACM Reference Format:
Wenkang Zhong, Chuanyi Li, Kui Liu, Tongtong Xu, Jidong Ge, Tegawendé F. Bissyandé, Bin Luo, and Vincent Ng. 2024. Practical Program Repair *via* Preference-based Ensemble Strategy. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3623310

1 INTRODUCTION

Bug fixing is a challenging, time-consuming, and labor-intensive task, often consuming a significant portion of developers' efforts [41]. To address this challenge, Automated Program Repair (APR) [31] has been dedicated to automatically fixing bugs without human intervention, and has become a hot field in the software engineering community. To date, more than 40 APR tools have been proposed as the momentum for program repair is growing.

Practitioners have been exploring advanced techniques that could overwhelmingly outperform all the other APR techniques in all lines of bug-fixing performance. Nevertheless, various experimental results in the literature suggest that there is at least one APR tool whose bug-fixing merit cannot be achieved by other APR tools [1, 9, 23, 24]. Different APR tools present complementary repairability to each other. For example, as the recent state-of-the-art APR tool, AlphaRepair [46] can correctly fix 50 Defects4J bugs under normal fault localization setting, while there are 108 Defects4J

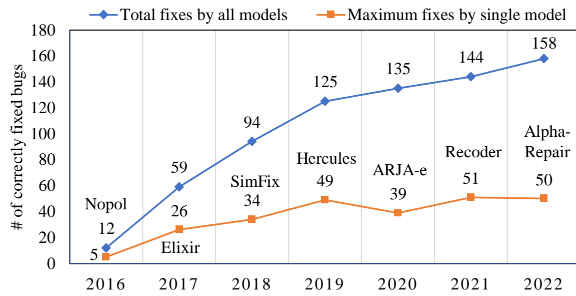


Figure 1: Fixing capabilities of single top-performing APR system vs. their integration over the years (2016-2022) on Defects4J v1.2 bugs under normal fault localization setting.

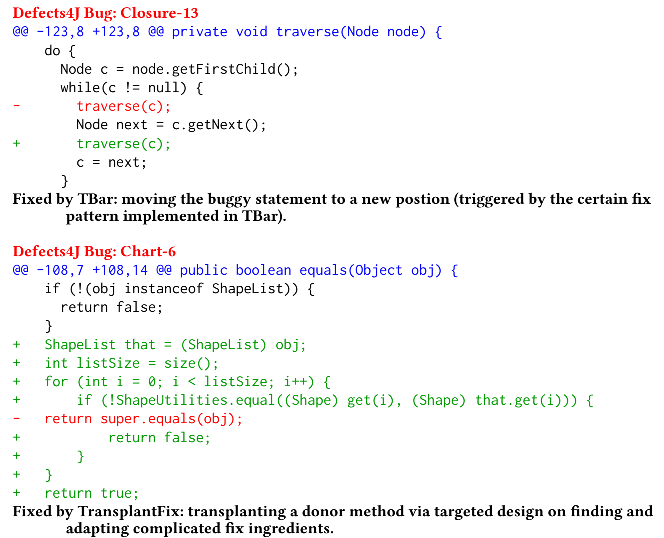


Figure 2: Examples of two bugs fixed by APR tools with different bug-fixing strategies.

bugs that cannot be fixed by it but can be correctly fixed by other APR tools, as shown in Figure 1.

Given the results in Figure 1, it is tempting to try all APR tools so that more bugs can be fixed, but trying all APR tools to fix a given bug is simply impractical due to the unacceptable costs of tool invocation and patch validation. For instance, Durieux et al. [9] spent 314 days executing and validating 11 APR tools even with Grid'5000 [3]. Additionally, a single APR tool could generate some plausible patches for a given bug [36], and more APR tools would generate more plausible patches [24], which will considerably increase the difficulty of distinguishing the correct patch from plausible ones. For this problem, the current state-of-the-art strategy, E-APR [1], frames the selection as a supervised machine learning task, specifically a multi-label classification task, that involves identifying the set of tools that should be used to fix a given bug from an ensemble of tools. However, training E-APR requires a large amount of labeled data, which is costly to obtain in the program repair field (manually validated patches); moreover, model retraining is necessary whenever a new tool is to be added, which limits its flexibility and practicability.

In this paper, we propose a novel ensemble strategy for APR that is motivated by the following hypothesis: different APR tools achieve differing performance in bug fixing because they have different repair *preferences* (i.e., a feature set of bugs that an APR tool can fix). Our hypothesis is formulated based on our examination of a number of bug-fixing examples, two of which are shown in Figure 2. For example, Closure-13 is fixed by moving the buggy statement to a new position implemented in TBar [22] with a certain fixing pattern, whereas Chart-6, a bug that requires multi-line patches, is fixed by TransplantFix [50] via target design on finding and adapting complicated fix ingredients. The preference that an APR tool has in fixing bugs is driven in part by the *repair pattern(s)* that the tool explicitly or implicitly employs as well as its *repair history*.

Given the above discussion, we propose **P-EPR** (Preference-based Ensemble Program Repair), a new ensemble strategy that leverages the repair preferences of each tool in the ensemble to improve repairability in practice. **P-EPR** is novel in the following respects:

No model training. **P-EPR** is the first *non-learning-based* ensemble method for APR. While E-APR casts the task as a multi-label *classification* task that involves identifying the subset of tools in the ensemble of tools for fixing a given bug, **P-EPR** casts the task as a *ranking* task that ranks the tools in the ensemble based on how likely a tool can correctly fix a given bug. Moreover, unlike E-APR, which requires training a classification model in a supervised manner, **P-EPR** does not require any model training. Specifically, **P-EPR** ranks the tools in the ensemble by independently scoring each tool based on how likely it can fix the bug using several sources of information in a heuristic manner.

New knowledge sources. As mentioned before, **P-EPR** employs two sources of information that encodes a tool's repair preferences, namely repair patterns and repair history. To our knowledge, the use of repair patterns has not been explored in existing ensemble methods for APR. Note that repair patterns encode a significant amount of human knowledge of the types of bugs a tool is adept at fixing and therefore they are likely to be more useful for scoring and ranking tools than any other program-independent and dependent features that one can possibly come up with. In fact, we believe that augmenting the feature set currently employed by E-APR with our repair patterns will likely boost its performance.

Dynamic updating. As soon as a tool is used to fix a bug, **P-EPR** receives *immediate* feedback on whether it successfully fixes the bug by having its repair history updated. In other words, the repair history of a tool is updated in a dynamic fashion based on all of the bugs that it has been applied to so far. Hence, **P-EPR** has the ability to exploit information that it acquires in real time. This dynamic updating mechanism is one of the key strengths of **P-EPR** that distinguishes it from existing ensemble methods for APR.

Another key strength of **P-EPR** is its flexibility. One may argue that the need to manually identify repair patterns whenever a new non-learning-based tool is to be added to **P-EPR** makes our approach undesirable or even impractical. It turns out that **P-EPR** is flexible enough that one can add a new non-learning-based tool to it *without* identifying any repair patterns. In other words, the manual identification step is a *recommended* rather than *compulsory* step. To see the reason, recall that **P-EPR** operates by scoring each tool w.r.t. a given bug using two sources of knowledge, repair patterns and

repair history. If one source of information is absent (in this case the repair patterns), **P-EPR** can simply rely on the other source of information for scoring. In other words, the use of repair patterns can only improve ranking results, but **P-EPR** can operate even without the patterns.

In fact, **P-EPR** is even more flexible than what we just described. In the extreme case, a new tool can be added to **P-EPR** even when both sources of information (i.e., the patterns and the history) are absent. In this scenario, the new tool will receive a score of 0 at the beginning, but over time, the dynamic updating procedure will update its repair history[1] In other words, over time, **P-EPR** will be able to accumulate enough knowledge about the repair preferences of the new tool via updating its repair history even if we know nothing about it at the time of incorporation.

In sum, our work makes the following contributions. First, we propose the first non-learning-based ensemble strategy **P-EPR** for assembling APR tools that is highly flexible. Second, we manually collect 13 repair patterns of APR tools, retrieve 4 types of bug features, and construct a mapping between repair patterns and bug features. Besides, we generate a categorized performance history between 21 concrete APR tools and the feature set of bugs that they can fix, which can be reused in other ensemble program repair frameworks. Finally, we design specific evaluation metrics to measure the effectiveness of ensemble program repair strategies and conduct comprehensive experiments to evaluate **P-EPR**.

Experimental results show that **P-EPR** achieves better results than existing strategies. Two of the most significant empirical findings are that (1) when given the same amount of labeled data (which **P-EPR** uses to initialize the repair histories of the tools and E-APR uses for model training), **P-EPR** demonstrates that it is more effective at exploiting the labeled data by achieving considerably better results than E-APR; and (2) even when **P-EPR** operates *witihout* using any repair patterns, it still outperforms E-APR, suggesting the robustness of **P-EPR**.

2 BACKGROUND AND RELATED WORK

This section introduces the research background of this work.

2.1 Different Types of APR Tools

Existing APR tools can be categorized into four types:

Heuristic-based approaches rely on manually defined heuristic rules to generate patches by iterating over a search space of syntactic program modifications, of which experimental results reviewed that they normally target on fixing general bugs [11, 14, 28, 35, 40, 50, 52]. However, they suffer from low efficiency due to the large search space and the limited effectiveness caused by the large number of plausible patches [12].

Template-based approaches generate patches based on a batch of pre-defined fix patterns, acting at explicit and direct modes [8, 13, 17, 18, 20, 20–22, 25, 28, 37, 42] : (1) checking whether the buggy statement satisfies the prepositive conditions of fix patterns, and (2) continuously generating code changes based on patterns until a valid patch is generated or the fixing behavior is terminated. Obviously, the repairability of template-based tools relies on the diversity of repair templates.

Constraint-based approaches use semantic constraints to limit the search space of patches [10, 19, 29, 49] . Generally, such approaches first infer repair constraints from the buggy program or the test suite and use an SMT solver (e.g., Z3 [6]) or other strategies to solve the constraints. However, the symbolic execution and constraint solver can explode the space of generating constraints and patch candidates when fixing complex bugs.

Learning-based approaches aim to train APR systems using historical bug-fixing data that can be sourced from code repositories. For example, DeepRepair [43] relies on deep learning to sort repair ingredients via code similarities. Latest learning-based approaches [4, 5, 7, 15, 26, 39, 51] employ neural machine translation (NMT) [2] models to perform bug-fixing framework as a sequence-to-sequence translation task. Such methods rely on a large amount of bug-fix data and need to address the overfitting issue in the training process.

2.2 Empirical Studies on APR Tools

Various empirical studies on APR tools have been conducted from different aspects to boost the development of automated program repair. Qi et al. [36] analyzed the correctness and plausibility of patches generated by APR tools. Smith et al. [38] looked into the overfitting problem of patches generated by APR tools. Motwani et al. [33] investigated to what extent hard and important bugs can be fixed by APR tools. Durieux et al. [9] empirically studied the generalizability of 11 APR tools with five benchmarks and all possible repair attempts. Liu et al. [24] explored the bug-fixing efficiency of 16 APR tools. These studies demonstrated that different APR tools present varying repairability on different bugs and their specific characteristics of fixing bugs.

2.3 Related Work

As various APR tools are proposed, researchers have begun exploring advanced assembling methodologies to boost automated program repair, which can summarized into two categories:

Exploiting multiple models. CoCoNut [26] and CURE [15] are two learning-based techniques that train a neural APR model multiple times, each time with a different set of parameters. This results in multiple APR models. Each of these results is then used to independently generate patches. For example, if 10 models are trained and each one generates 300 patches, all 3000 patches will be validated. Hence, while these techniques generate an ensemble of models, strictly speaking they are *not* ensemble strategies.

Ensemble methods. E-APR [1] makes an early exploration of reusing existing tools via an ensemble strategy. It first identifies significant program-independent and dependent features by analyzing footprints of repair results of existing APR tools. Then, it predicts the effectiveness of APR tools via machine learning algorithms according to the metrics of nine features identified from 146 features. However, a major limitation is that every time a new APR tool needs to be added to the ensemble, the model must be re-trained. In contrast, **P-APR** does not require any model training.

[1] Given a bug, even a tool with *repair patterns* and a *repair history* may get a score of 0 if its *repair patterns* and *repair history* do not match the bug. Besides, a tool may even get a score lower than 0 if its *repair history* has many records of failing in fixing the current type of bug (since we will use the history of failures as a penalty). Therefore, a tool that has neither *repair patterns* nor *repair history* may be ranked higher than those with repair patterns and/or repair history.

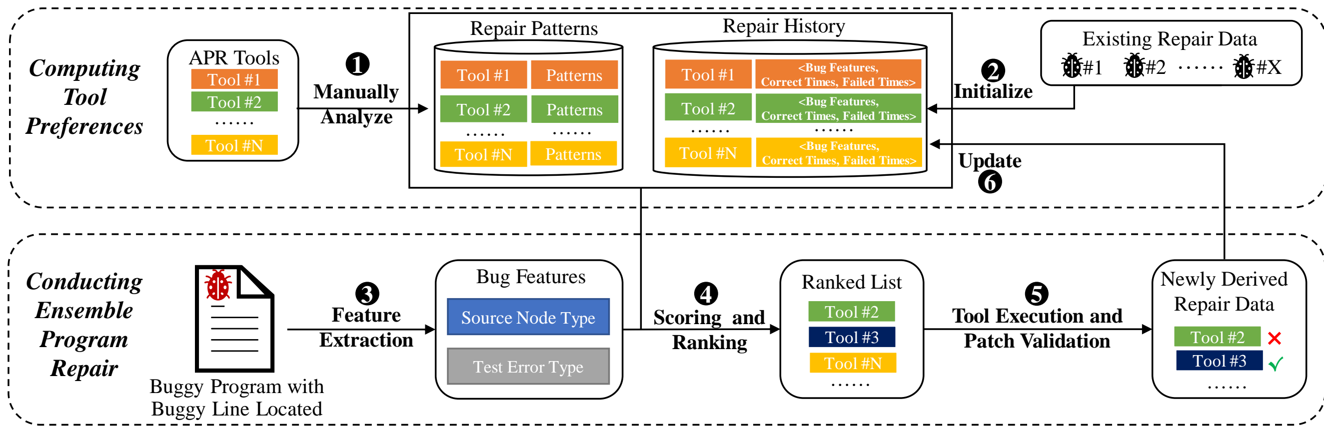


Figure 3: The overall procedure of P-EPR.

3 P-EPR

Figure 3 shows the overall procedure of **P-EPR**, which consists of two parts: (1) ***Computing Tool Preferences*** (Section 3.1), and (2) ***Conducting Ensemble Program Repair*** (Section 3.2).

3.1 Computing Tool Preferences

To capture the preference of an APR tool, we compute a tool's preferences in an offline manner (see the first part of Figure 3) through two steps, as described below.

Step 1: Repair Pattern Collection via Manual Analysis. We define a tool's repair action (e.g., Inserting Range Checker) for fixing a certain type of target bug as a Repair Pattern. Each repair pattern has corresponding pre-requirements on bugs (e.g., "If an array or collection is accessed without being checked" for Inserting Range Checker), which describes the characteristics of a bug that would trigger and be fixed by the repair pattern/action. An APR tool may have multiple repair patterns and one repair pattern may be shared by different APR tools. We summarize the repair patterns for existing APR tools by going through their methodologies and implementations. For example, for template-based tools, the repair patterns can be inferred directly from their implemented patterns. For other types of tools, a repair pattern can be searched by checking if the tool implements certain repair actions that must be triggered by conditions related to the input bugs. Note that there are no repair patterns defined in learning-based APR tools.

Specifically, we collect all repair patterns by investigating the 42 APR tools listed in [32] and [22]. In total, we derive 13 repair patterns that are implemented in 19 non-learning-based APR tools. Table 1 presents the four example patterns, describing their name, pre-requirements and implemented APR systems.

Next, we complete the judgement logic of whether an input bug satisfies the conditions of a certain repair pattern. Motivated by a previous template-based APR tool TBar [22], we manually analyze the pre-requirements of the collected patterns and design four types of bug features (BF1-4) that could cover the automatic judgment logic of all patterns, as shown in Table 2. For example, to check if a given bug satisfies *P4 Throw Exception*, **P-EPR** would examine whether BF4 (the type of the test error) of the bug is an *Exception Thrown* error.

Step 2: Repair History Initialization using Existing Repair Data. Among the four features (BF1-4) defined in the previous step, BF1 (the node type of the buggy statement) and BF4 (the type of test error) can be regarded as program-independent since they can be extracted from any buggy program. Thus, we reuse BF1 and BF4 to initialize the repair history of a tool. Concretely, we store a tuple *<tool, bug_feature, failed_times, correct_times>* for each tool, where *bug_feature* can be BF1 or BF4. When configuring an APR tool into **P-EPR**, the existing repair history of the tool can be loaded to initialize the repair history. For example, APR tools are usually empirically evaluated in some bug-fix benchmarks before publication. Thus, before deploying **P-EPR** in practice, those existing repair results can be utilized to enhance the performance of **P-EPR**. Such design in **P-EPR** ensures the generalization of **P-EPR** on integrating any kinds of APR tools with existing repair results.

3.2 Conducting Ensemble Program Repair

Given the Repair Patterns and the Repair History of the APR tools involved, **P-EPR** can be used to repair a given bug. The input of **P-EPR** is the buggy class file along with its suspicious faulty lines located by fault localization techniques [44].

Step 3: Feature Extraction. To automatically determine whether the input bug satisfies the repair patterns and repair history of APR tools, we need to first extract features of the given bug. Recall that we define 4 features for bugs in Table 2. Bug features 1, 2, and 3 are properties of code elements within the buggy statement. We use spoon [34] to parse the buggy code and extract those features. Bug feature 4 corresponds to the type of failed test error triggered by the bug, such as *java.lang.IndexOutOfBoundsException.* Trivial test error, i.e., *junit.framework.AssertionFailedError*, is disregarded since it is too general. If a bug produces multiple failed test cases, only the first non-trivial test error is considered.

Step 4: Tool Scoring and Ranking. Algorithm 1 presents the core step of **P-EPR**, where a score representing the chance that the given bug can be correctly fixed is calculated for each APR tool. Concretely, the score is derived by matching the prepared tool preferences (i.e., including Repair Pattern and Repair History) with features of the given bug. To score each APR tool, we use the faulty code file and the faulty line IDs as inputs. The algorithm is

Table 1: Example of repair patterns collected from existing APR systems.

No.	Pattern Name	Pre-requirement	Implemented APR systems
P1	Insert Cast Checker	A buggy statement that contains at least one unchecked cast expression	heuristic-based: HDRepair, SimFix, CapGen template-based: AVATAR, Genesis, kPAR, SketchFix, TBar, SOFix
P2	Insert Null Pointer Checker	A buggy statement if, in this statement, a field or an expression (of non-primitive data type) is accessed without a null pointer check	heuristic-based: HDRepair, SimFix, CapGen tempate-based: AVATAR, Genesis, Elixir, FixMiner, NPEfix, SOFix kPAR, TBar
P3	Insert Range Checker	Inserting a range checker for the access of an array or collection if it is unchecked	template-based: AVATAR, Elixir, kPAR, SketchFix, TBar, SOFix
P4	Throw Exception	The failed test type is throwing an exception	heuristic-based: ACS

* Due to the space limitations, we only list 4 patterns in this table. Concrete information of all 13 collected patterns can be found in the supplementary.

Table 2: Bug features needed for matching different patterns

No.	Bug Feature	Pattern*
BF1	Node type of the buggy statement	P6, 11, 12
BF2	Child node types within the buggy statement	P3, 5, 7-9
BF3	BF1 & BF2	P1, 10
BF4	Type of the test error	P4

* Corresponding descriptions of each pattern can be found in the supplementary. We ignore P2 since its pre-requirement is too general that almost every statement can match the pattern.

Algorithm 1: Calculate preference scores for APR tools

```
   Input: The faulty line IDs, allFaultyLineIds
   Input: The faulty class file, buggyFile
   Input: The bonus coefficient of pattern match, EM_α
   Output: The preference scores of all tools, preferScores
 1 preferScores ← ∅ ;
 2 EM_α ← 0.5 ;
 3 for lineId ∈ allFaultyLines do
 4     bugFeatures ← ExtractFeature (lineId, buggyFile);
 5     for tool ∈ availableTools do
 6         finalScore ← 0;
 7         historyScore ← 0;
 8         for feature ∈ bugFeatures do
 9             historyScore ←historyScore +
                 CalculateHistoryScore(tool, feature);
10         end
11         if PatternMatch(tool, bugFeatures) then
12             finalScore ← historyScore * (1 + EM_α);
13         else
14             finalScore ← historyScore;
15         end
16         preferScores.set(tool, finalScore);
17     end
18 end
```

compatible with bugs containing any number of hunks. For each faulty line, the bug features defined in Table 2 are first extracted (line 4). Then, for each tool in the available toolset, we calculate a *historyScore* according to the existing repair history of the tool (line 8-10). Recall that **P-EPR** stores the repair history of a tool with a tuple <*tool, bug_feature, fail_times, correct_times*>, so the *historyScore* is calculated as:

$$CalculateHistoryScore(tool, feature) = \\ correct_times_{tf}/(correct_times_{tf} + fail_times_{tf}) \quad (1)$$

where t represents the tool and f represents the feature. We index the repair history with BF1 and BF4. For example, if the node type of the buggy statement is *CtInvocationImpl*, the corresponding preference score will be the fixed rate when the tool encounters bugs of such type. Then, **P-EPR** will judge if the bug features match the preferred patterns of tools in the available set (line 14). If yes, the preference score of the tool will get a bonus (line 15). At this step, we introduce a configurable coefficient EM_α to control the bonus degree. We use multiplication to combine the pattern and history preference scores. The final preference score of each tool is the sum of the preference score of all faulty lines. The higher the score is, the more likely it is for the corresponding tool to fix the bug. All the tools are ranked in descending order of scores.

Step 5: Tool Execution and Patch Validation. After the tools are ranked in descending order of scores for a bug, a human developer can use the tools sequentially to fix the bug. However, it is not necessary to use these tools sequentially. For instance, if there are K available computing threads/human developers available to fix the bug with APR tools at the same time, the Top-K tools could be adopted simultaneously. It is worth mentioning is that generating patches and checking if a patch is plausible is achieved by the tool automatically, but checking if a plausible patch is a correct one can only be achieved by a human developer. This is why we calculate two different costs for fixing a bug when evaluating **P-EPR**.

Step 6: Preference Update. As aforementioned, the history score is computed according to the repair history of the APR tools on bugs that have the same features as the given bug. Therefore, at the end of each repair procedure, **P-EPR** updates the repair history of each tool according to their performance on the bug that they just handled if they are executed. In our implementation of **P-EPR**, we maintain a repair result list of each APR tool in the available toolset, and each repair result is represented by <*FixStatus*,*BugFeatures*,*Tool*>, where *FixStatus* denotes the repair status with three enum types (correct, overfit, fail) and *BugFeatures* identifies the characteristics of the buggy program. Like in Repair History, we use BF1 and BF4 defined in Table 2 as the content of *BugFeatures* in repair results.

3.3 Integrating Improved or New APR Tools

To integrate an improved or new tool into **P-EPR**, we need to perform the two steps below:[2]:

(1) Updating Repair History. This means updating the Repair History table shown in Figure 3 for the target APR tool with buggy programs that have been repaired by the target tool either successfully or unsuccessfully. Users are only required to provide ① the buggy class file, ② the suspicious line locations, and ③ the test error type (if available). **P-EPR** first transforms the given buggy programs into bug features (shown in Table 2) and then updates the corresponding tuple *<tool, bug_feature, fail_times, correct_times>* for the tool. Any buggy program adopted by any APR tool as repair history can be integrated into our Repair History table since our bug feature BF1 is program-independent, i.e., any buggy program has a value for BF1. In other words, our **P-EPR** can be generalized to any improved or new APR tool with any kind of repair history regardless of whether it has Repair Patterns.

(2) Updating Repair Patterns. This step is needed only when the improved/new APR tool to be added is non-learning-based. Given the improved/new tool, we need to identify the set of repair patterns associated with it. If all of the repair patterns it is associated with are among the 13 patterns that currently exist in **P-EPR**, then nothing needs to be done. otherwise, for each new pattern, we need to add it to the pattern repository and update the mapping of patterns to bug features (see the current mapping in Table 2).

Table 3: Correct/overfit patches generated by the 21 APR systems on 395 bugs from Defects4J v1.2

	System	# Correct	# Overfit	Source
heuristic-based	jGenProg	6	10	[24]
	GenProg-A	8	21	
	RSRepair-A	9	25	
	ARJA	11	25	
	SimFix	29	21	
	jKali	2	6	
	Kali-A	5	37	
	jMutRepair	5	6	
	TransplantFix	36	33	[50]
constraint	Nopol	2	7	[24]
	ACS	16	5	
	Cardumen	2	14	
	DynaMoth	3	10	
template	kPAR	33	30	[24]
	AVATAR	30	20	
	FixMiner	34	29	
	TBar	54	30	
learning	SequenceR	27	24	[53]
	CodeBERT-ft	29	28	
	RewardRepair	43	22	
	Recoder	56	22	
	Total	122	121(58)	

4 EVALUATION SETUP

4.1 Research Questions

We aim at answering the following three research questions for evaluating **P-EPR**.

[RQ1. Performance] What is the overall performance of P-EPR compared with the other ensemble strategies? Concretely, we conduct different ensemble strategies to select APR tools for each bug in the Defects4J v1.2 dataset. To thoroughly evaluate **P-EPR**, we consider a maximum set of 21 tools. Note that we do not execute each tool to derive the progress due to the unaffordable costs. Instead, we rely on all published patches of different tools for each bug in Defects4J v1.2.

[RQ2. Ablation Study] To what extent does each component of P-EPR contribute to its overall performance? We seek to gain insights into **P-EPR** by understanding the impacts of its components on the performance, such as the Test error type and the coefficient EM_α, via ablation experiments.

[RQ3. Practicality] To what extent can P-EPR save computational costs in practice compared with adopting every single tool? 'in practice' means executing the selected tool by **P-EPR** to calculate the computational cost (e.g., time for generating patches, time for verifying patches, and computer memory, etc.) instead of performing simulations using existing patches. Considering that the Defects4J dataset has been used by almost all APR tools and that the performance of **P-EPR** on Defects4J is evaluated in RQ1, we use another dataset, Bears, to verify the performance of **P-EPR** in practice (where the repair history of the APR tools on Defects4J are used for initializing their Repair History in **P-EPR**).

[2]For more concrete instructions of integrating improved or new APR tools into **P-EPR**, as well as using **P-EPR**, please refer to the tool's repository: https://github.com/kwz219/P-EPR-Artefact

4.2 Tool Selection and Data Collection

Since **P-EPR** is compatible with any kind of APR tool, we select a variety of APR tools. However, empirically executing a large number of APR tools and validating generated patches is prohibitively expensive. So, we choose to evaluate the performance of **P-EPR** through a simulated experiment with published repair results of APR tools, instead of actually running APR tools. Simulation means that we directly get the repair results of APR tools on each bug, skipping the tool execution and patch validation process. To reduce the biases brought by the simulated experiment, we select APR tools for the simulated experiment according to the following criteria:

C1: The fault localization setting of each APR tool should be the same. Since **P-EPR** extracts features of faulty code lines, different fault localization results can impact the calculated score of the same bug. However, it is challenging to maintain the same fault localization setting when considering Normal Fault Localization (NFL). This is because NFL settings of existing APR tools vary significantly on the FL tool and considered fault locations. For example, SketchFix [13] considers only the top 50 most suspicious statements in the ranked list, while ELIXIR [37] considers up to the top 200 suspicious locations. Therefore, to minimize biases in our experiment, we only considered tools evaluated within the Restricted Fault Localization (RFL) scenario [24], where the accurate faulty line of the buggy program is provided.

C2: The patch generation setting of each APR tool should be the same. To satisfy this criterion, we opt to refer to empirical studies on APR tools, which typically use the same settings across the studied tools, instead of collecting repair results from individual APR publications. We first obtain the repair results of 16 test-suite-based APR tools from a relevant empirical study [24]. Additionally, we include four state-of-the-art learning-based tools in our evaluation by re-running them on Defects4J with NPR4J [53], a framework tool that supports running these tools.

Given these criteria, we collect the repair results of 21 APR systems on Defects4J v1.2. Those systems cover 4 types of APR tools: 9 are *heuristic-based* (jGenProg [28], GenProg-A [52], RSRepair-A [52], ARJA [52], SimFix [14], jKali [28], Kali-A [52], jMutRepair [28], TransplantFix [50]), 4 are *constraint-based* (Nopol [49], ACS [48], Cardumen [29], DynaMoth [10]), 4 are *template-based* (kPAR [20], AVATAR [21], FixMiner [18], TBar [22]) and 4 are *learning-based* (SequenceR [5], CodeBERT-ft [30], RewardRepair [51], Recoder [54]). Among the 21 systems, we re-run the four learning-based tools (for execution and validating settings, ref to Section 4.5) since they do not provide required data (i.e., both correct and overfit patches generated by the tool) and use the published patches of other systems. In total, the 21 o tools correctly/plausibly fix 122/180 bugs from Defects4J. 9 of them have repair patterns (SimFix, jMutRepair, Nopol, ACS, Dynamoth, kPAR, AVATAR, FixMiner, TBar).

4.3 Metrics

First, to estimate the repairability and costs when deploying **P-EPR** on a set of APR tools to fix bugs, we use the following metrics:

(1) The number of correctly/plausibly fixed bugs. A plausible patch can pass all test cases, but it may not be correct. A correct patch can pass all test cases and human validation.

(2) Tool Invocation Times (TIT). It measures the machine resource costs when invoking a set of APR tools. For simplicity, we define one tool invocation as whether the APR tool should be invoked when a bug is given.

(3) Human Validation Times (HVT). It measures the human labor costs of checking plausible patches. For a bug, a tool selection strategy may generate more than one plausible patch. We define HVT as the number of manual checks needed to find a correct patch. If no correct patches are generated, the HVT is equal to the number of generated plausible patches.

Second, to quantify the cost savings obtained by employing **P-EPR**, we design two novel metrics:

(4) Tool Invocation Saving Percentage (TISP). It measures how many tool invocation times can be saved when using a tool selection strategy compared with invoking all APR tools. TISP is calculated as:

$$TISP = R_{Strategy}/R_{EnsAll} - TIT_{Strategy}/TIT_{EnsAll} \quad (2)$$

where $R_{Strategy}$ and R_{EnsAll} represent the numbers of correctly fixed bugs of using a strategy and of invoking all available tools respectively, and $TIT_{Strategy}$ and TIT_{EnsAll} denote the tool invocation times of using a strategy and of invoking all available tools respectively.

(5) Human Validation Saving Percentage (HVSP). It measures how many manual checks can be saved when using a tool selection strategy compared with invoking all APR tools. HVSP is calculated as:

$$HVSP = R_{Strategy}/R_{EnsAll} - HVT_{Strategy}/HVT_{EnsAll} \quad (3)$$

where $R_{Strategy}$ and R_{EnsAll} are the same as those in Equation 2, and $HVT_{Strategy}$ and HVT_{EnsAll} denote the human validation times of using a strategy and of invoking all available tools respectively.

4.4 Baseline Systems

We compare **P-EPR** with several baselines:

(1) E-APR [1]. Since E-APR's source codes are not published, we replicate it. Specifically, we implement E-APR with Random Forest Classifier since it achieves the best performance among the four algorithms described in E-APR's paper.

(2) E-APR (enhanced). The original E-APR is trained and tested on only 10 APR tools. We consstruct an enhanced version of E-APR by re-training it using the same 21 tools that **P-EPR** uses using a Random Forest Classifier following the settings of E-APR.

(3) Random Selection. This strategy randomly selects K tools sequentially for each bug as the top-ranked tools.

(4) Invoking All Tools. This strategy simply invokes all available tools to fix each bug.

(5) Optimal Selection (Ground Truth). This is an ideal strategy that makes the optimal choice of APR tools, thus providing a rough upper bound on **P-EPR**'s performance. It prioritizes the tools as follows: ① can produce correct patches, ② can produce plausible patches, and ③ cannot produce plausible patches.

4.5 Tool Execution and Validation Settings

Our experiments involve practical execution of four APR tools (SequenceR [5], CodeBERT-ft [30], Recoder [54] and RewardRepair [51]) on Defects4J v1.2 [16] for the data collection in previous section, and Bears [27] for an empirical experiment in RQ3. We use the same tool execution and patch validation settings. For each bug, each tool generates 300 candidate patches, with a timeout of 2 hours for validating them. For patch correctness assessment, two of the authors manually validate the first test-adequate patch with a timeout of 10 minutes for every bug, adhering to the assessment criteria established by prior research [24]. A patch is deemed correct only if both reviewers agree on its accuracy. All model execution and patch validating experiments are performed on a machine equipped with an AMD Ryzen 9 5950X 16-Core Processor and two NVIDIA GeForce RTX 3090 Ti GPUs.

5 EVALUATION RESULTS

5.1 RQ1. Performance of P-EPR

Method: We design three experiments. First, we execute **P-EPR** on all of the 395 bugs from Defects4J using the 21 APR tools. We use all data in Defects4J as test data, effectively assuming that no bugs are used for initializing the repair history. Second, we compare **P-EPR** with the original E-APR [1] (which is trained and evaluated on only 10 of the 21 APR tools) by only using the 10 APR tools used by the original E-APR. The third one involves comparing **P-EPR** with the enhanced E-APR, which is re-trained on all of the 21 APR tools. In the second and third experiments, we also compare with the Random selection strategy. Besides, in these experiments, we

Table 4: P-EPR's performance on 395 bugs from Defects4J considering 21 APR tools. For each bug, the Top-K APR systems ranked by P-EPR are selected to repair it. "Opt" and "All" denote Optimal Selection and Invoking All Tools strategies respectively.

	Top-1	Top-2	Top-3	Top-4	Top-5	Top-6	Top-7	Top-8	Top-9	Opt	All
# of correctly/plausibly fixed bugs	54/89	68/100	78/113	87/129	86/133	95/138	101/146	108/157	109/160	122/180	122/180
# of plausible patches	108	159	219	278	326	389	453	478	510	180	859
Tool Invocation Times (TISP)	1010 (33%)	1461 (39%)	1925 (41%)	2330 (44%)	2701 (38%)	3102 (41%)	3488 (41%)	3906 (42%)	4282 (38%)	395 (95%)	8295
Human Valdation Times (HVSP)	98 (20%)	117 (26%)	148 (27%)	175 (27%)	191 (22%)	214 (24%)	229 (25%)	257 (24%)	271 (21%)	180 (54%)	393*

* When invoking all tools, the plausible patches are ordered in the same way as the tools were initially added to the toolset.
** Due to space limitations, we only present results from top-1 to top-9. Results of larger K values are listed in the GitHub repo.

assume that the input order of bugs is random (random seed = 1) and EM_α (see **Step 4** in Section 3.2) is 0.5 (the default setting) [3].

Results and discussion: Next, we describe and discuss the results of the aforementioned expriements.

Comparison with Invoking All Tools and the Optimal Strategy. Table 4 presents the repair results (the number of correctly/-plausibly fixed bugs), execution costs (model invocation times and human validation times), and costs saving a percentage of **P-EPR** (TISP and HVSP) when selecting Top-1 to Top-9 to repair bugs from Defects4J when all 21 tools are used, as well as the corresponding metrics of Invoking All Tools and Optimal Selection (i.e., Ground Truth) strategies. Compared with simply executing all available APR tools, **P-EPR** can significantly save costs on tool execution and human validation while reaching comparable repairability. For example, when K is 9, **P-EPR** achieves a 90% repairability compared with executing all tools (109 correctly fixed bugs for **P-EPR** vs. 122 for all tools), with a 50% reduction of execution costs in terms of the model invocation times (4282 vs. 8295) and a 30% reduction of human validation costs (271 vs. 393). As K decreases, **P-EPR** also achieves remarkable performance. When K is 1, **P-EPR** correctly fixes 54 bugs, which is very close to the best APR tool in our experiment (Recoder [54] fixes 56 bugs). When K is larger than 2, **P-EPR** outperforms any one of the 21 tools significantly. Remembering that we have a large-scale set of 21 APR tools, the results prove that **P-EPR** has the ability to correctly rank APR tools among a variety of tools. Compared with the optimal strategy, **P-EPR** achieve 35% - 44% of the optimal strategy on TISP, and 37% to 50% on HVSP. It indicates that the performance of the current **P-EPR** strategy still has a large room for optimization.

Comparison with the original E-APR. Figure 4 expresses the performance of **P-EPR**, the original E-APR, and the random selection strategy in terms of TISP and HVSP when only the 10 APR tools employed by the original E-APR are involved. As can be seen, **P-EPR** achieves significantly higher performance than E-APR and random selection. In terms of TISP, **P-EPR** always performs better than the random selection. Compared with E-APR, **P-EPR** has a significant improvement when K ranges from 1 to 5. In some cases E-APR performs worse than just invoking all APR tools. When K is 2, **P-EPR** has the highest improvement on TISP than E-APR (40% vs -15%). In terms of HVSP, **P-EPR** performs better in most cases than other strategies. Besides, when K is between 2 and 5, **P-EPR** achieves the highest TISP and HVSP, while E-APR's performance

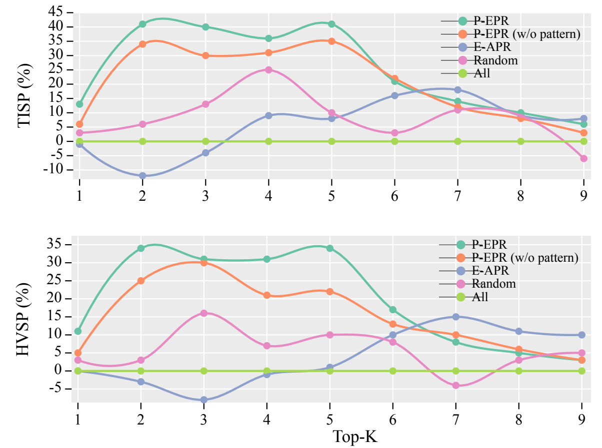


Figure 4: Comparison of P-EPR and other strategies in terms of TISP on Defects4J employing only the 10 APR tools originally used by E-APR. "All" denotes invoking all APR tools. The HVSP of the optimal strategy is 57%, and the TISP of the optimal strategy is 90%.

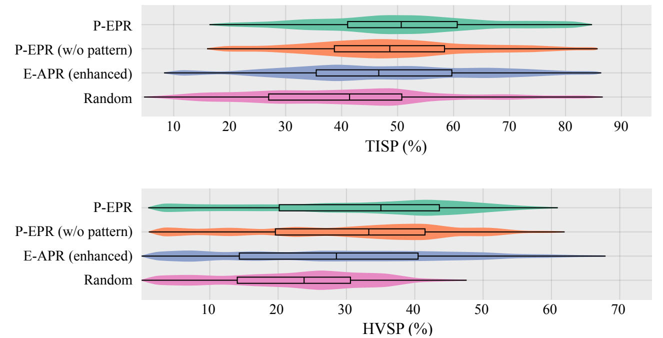


Figure 5: Distributions of TISP and HVSP of P-EPR and other strategies on the different train-test split of Defects4J projects and different K (ranges from 1 to 20) considering 21 tools.

improves as K becomes larger. This suggests that a higher level ensemble strategy of **P-EPR** and E-APR may yield even better performance.

Comparison with enhanced E-APR. In the previous comparisons, we used all of Defects4J as test data, leaving no training data that can be used to initialize a tool's repair history. In this comparison, we perform comparisons where we do partition Defect4J into a training set and a test set so that we can initialize the repair history

[3] Since the dynamic update module updates the repair history of APR tools in **P-EPR** according to the input bug and the repair history affects the performance of **P-EPR** in real time, the input order of test bugs will affect the evaluation results of **P-EPR**. Besides, *the different combinations of APR tools* may also affect the evaluation of the overall performance of **P-EPR**. So, we conduct additional experiments by setting them with different values to demonstrate the feasibility of **P-EPR**. See these experimental results and discussions in Section 2 in the supplementary file.

of each tool in **P-EPR** using the training data. More specifically, we show in Figure 5 the distributions of TISP and HVSP of **P-EPR** computed based on 400 situations (i.e., there are 20 train-test splits situations of selecting three projects from Defects4J's six projects, and for each bug, we could select Top-K tools to fix it, K ranges from 1 to 20; so, for each strategy, 400 TISPs and 400 HVSPs are calculated and plotted in Figure 5). We similarly plot the curves for two other baselines, enhanced E-APR and the random selection strategy Recall that enhanced E-APR is the version of E-APR that is being retrained on all the 21 APR tools used in **P-EPR**. For a fairer comparison with **P-EPR**, E-APR is re-trained using not only the training data used in the original E-APR but also the training portion of Defect4J in each of the 400 situations mentioned above. As shown in Figure 5, **P-EPR** still outperforms enhanced E-APR. In terms of both TISP and HVSP, **P-EPR** has a larger minimum value, Q1 (first quartile), Q2 (second quartile), Q3 (third quartile) than enhanced E-APR and random selection. Also, **P-EPR** has a shorter IRQ (Inter Quartile Range) than enhanced E-APR, which means that generally, **P-EPR** has better and more stable performance than the two strategies. However, we also observe that **P-EPR** has a lower maximum value than enhanced E-APR (especially TISP).

5.2 RQ2. Contributions of Components

Method: We investigate the impacts of each components by comparing **P-EPR** with following variants: (1) **P-EPR** without Pattern, (2) **P-EPR** without Dynamic Update, (3) **P-EPR** without Repair History Initialization, (4) **P-EPR** without Test Error Type and (5) **P-EPR** with different EM_α (0.1, 0.3, 0.5, 0.7, 0.9). For comparison, we select the Math project (contains 106 bugs) from the Defects4J project as the evaluation set and use the other five projects (contains 289 bugs) as the repair history for initialization (except variant (3)). We use the 21 APR tools and calculate their TISP and HVSP. It should be mentioned that there is no need to conduct any ablation experiment on the he Buggy element type feature (i.e., BF1) because all bugs have this feature and it is obvious that it plays an important role in ranking APR tools for bugs. Considering that while adding a new APR tool into **P-EPR**, collecting its repair patterns may be a little bit harder than collecting its repair history, we also conduct ablation experiments on Repair Patterns while comparing **P-EPR** with the other strategies to investigate whether the performance of **P-EPR** will be lower than the other strategies if we want to save the cost of collecting repair patterns of the new APR tools.

Results and discussion: Figures 6 (a) and (c) show the comparison between variants (1) - (4) in terms of TISP and HVSP, respectively. As can be seen, Repair History Initialization and Dynamic Update have significantly higher impacts than Repair Patterns and Test Error Type in **P-EPR**. Without repair history initialization, **P-EPR** has a up-to-50% performance degradation both in terms of TISP and HVSP. **P-EPR** without Dynamic Update suffers a similar performance degradation. The two components contributes to **P-EPR** via categorizing and updating history, which indicates that **P-EPR** benefits significantly from the repair data. It is recommended that a good practice for users to utilize **P-EPR** is to provide data for initialization and keep updating with newly derived repair results.

In addition to Figures 6 (a) and (c), which show the relative small impact of Repair Patterns on **P-EPR**, Figures 4 and 5 show that even without Repair Patterns, **P-EPR** still outperforms the other ensemble strategies. This implies that **P-EPR** can still help us select appropriate APR tools for bugs if we want to save the cost of collecting repair patterns of APR tools. However, these figures also show that the complete **P-EPR** outperforms the variant without Pattern in most cases when K ranges from 1 to 9. This implies that we should encourage the users to analyze patterns when integrating new tools into **P-EPR** for a even better ensemble performance.

In Figure 6 (a) and (c), we see that the variant without Test Error Type seems has relatively smaller performance difference with complete **P-EPR** comparing against other components. It suggests that **P-EPR** can be generalized in more scenarios (e.g., the bug is identified by a bug report but not a test case failure).

Figure 6 (b) and (d) compare the impacts of different EM_α values (variant 5) in terms of TISP and HVSP. As can be seen, the value of EM_alpha has little impact on the performance of **P-EPR**, which means users can pay less attention to setting this parameter.

5.3 RQ3. Practicality of P-EPR

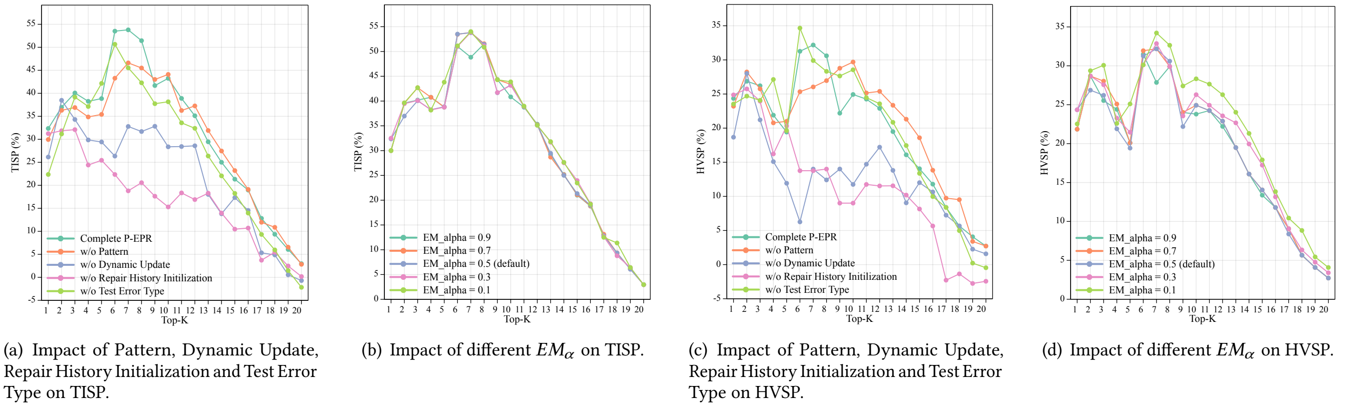
Method: We collect 83 single-hunk bugs from another dataset, i.e., Bears [27] for investigating the performance of **P-EPR** while practically executing integrated APR tools. We configure **P-EPR** with four learning-based program repair systems (i.e., SeqeunceR [5], CodeBERT-ft [30], RewardRepair [51], and Recoder [54]) [4] and initialize **P-EPR** with their existing repair results on Defects4J collected beforehand. We set EM_α to 0.5 by default. During inference, we select the top-1 system to repair every bug and record the Inference Time, Machine Validation Time, Human Validation Time, and GPU Memory Usage. In cases where multiple systems have the same top-1 score, we always choose the system with the least GPU usage (i.e., CodeBert-ft < SequenceR < RewardRepair < Recoder).

Results and discussion: Table 5 illustrates the performance and costs associated with executing each tool individually, as well as deploying **P-EPR** to select the top-ranked tool for execution. As depicted, **P-EPR** achieves the highest levels of repairability, successfully fixing 16 bugs, and demonstrates superior precision (57%) compared to the other four tools and the optimal selection strategy. In comparison to the strategy of executing all tools, **P-EPR** achieves a repairability of 76% (16 out of 21 bugs) while maintaining a higher precision, at a significantly reduced cost (25% of inference time and 24% of machine validation time). On human validation times, **P-EPR** also has a lower cost at 67% and 47%, compared with the optimal strategy and invoking all tools. This case serves as a concrete example of **P-EPR**'s effectiveness in bug fixing, highlighting its feasibility and practical application.

6 DISCUSSION

Practicality. One may be concerned that the design of patterns involves a large amount of manual work and is prone to human errors. While this is a valid concern, there are a few things to keep in mind. First, patterns are applicable for non-learning-based tools. Given that the majority of work on APR these days are learning-based, we

[4]Recall that the reason of conducting simulation experiments to evaluate the performance of **P-EPR** on 21 APR tools is that it is prohibitively expensive for us to empirically executing a large number of APR tools and validating generated patches. So, in the evaluation of practicality, we try our best to include APR tools into the experiments considering the computational and human resources that we can afford.



(a) Impact of Pattern, Dynamic Update, Repair History Initialization and Test Error Type on TISP.

(b) Impact of different EM_{α} on TISP.

(c) Impact of Pattern, Dynamic Update, Repair History Initialization and Test Error Type on HVSP.

(d) Impact of different EM_{α} on HVSP.

Figure 6: Ablation experiments illustrating the impact of different components of P-EPR when used with 21 APR tools.

Table 5: Performance and Computational Costs of P-EPR on the Bears benchmark. Machine Validation Time represents time costs on executing test cases. Human Validation Times represents time costs on manually checking plausible patches.

Metrics	Single System				Selection Strategy		
	Recoder	**CodeBERT-ft**	**RewardRepair**	**SequenceR**	**P-EPR (top-1)**	**Optimal***	**All***
# of correct/plausible patches	10/21	12/25	12/21	14/28	16/28	22/37	22/37
Precision	48%	48%	57%	50%	57%	59%	/
Inference Time (min)	11	2	4	4	7	6	28
Machine Validation Time (hour)	57	54	58	52	52	40	221
Human Validation Time (min)	41	54	36	62	56	84	118
TISP	20%	29%	29%	38%	48%	75%	/
HVSP	5%	7%	15%	10%	20%	29%	/
GPU Memory Usage (GB)	19.1	3.84	7.32	8.17	7.83	/	/

* We use the information produced when individually executing each program repair tool to estimate the performance and costs of invoking all tools. For example, we record the machine and manual validation time when individually executing each system and use them to estimate corresponding metrics when performing the two strategies. For the strategy that invokes all tools, we assume tools are sequentially executed.

expect that our patterns need to be updated on an occasional basis. Second, these repair patterns can be reused in other ensemble program repair frameworks. For example, it is conceivable that these patterns can be encoded as features that can be used to augment the feature sets used in existing learning-based ensemble methods for APR such as E-APR, and given the rich amount of human knowledge these patterns encode, they are likely to be useful for other ensemble program repair frameworks as well. Third, **P-EPR** is flexible enough that it can be deployed without any repair patterns (with the caveat that performance may suffer as a result). While errors could be introduced in the derivation of patterns, one could employ a second person to verify the correctness of the patterns.

Some may argue that we should instead go for a learning-based ensemble method for APR in which we train a model using program-independent and dependent features that can generalize the results to future improvements. This is exactly what is done in E-APR. While a systematic analysis of learning- and non-learning-based ensemble approaches to APR is beyond the scope of this paper, there are a few things that we should keep in mind. First, while we acknowledge the importance in developing program-independent and dependent features, these features by no means render our repair patterns useless. Specifically, these features and the repair patterns encode different kinds of knowledge. It is not even clear whether learning a model using only program-independent and dependent features will ever perform as well as one that uses patterns as features given the rich amount of human knowledge encoded in the patterns. Nevertheless, our results indicate that E-APR underperforms **P-EPR** when given the same amount of labeled data. Second, when incorporating a new tool to a learning-based ensemble method for APR, one needs to provide a possibly large amount of labeled data so that the model can learn how to classify/rank the new tool against the existing tools. In other words, if one believes that the amount of manual effort that goes into the identification of patterns makes **P-EPR** less practical, then the amount of manual effort that goes into providing labeled data may similarly make a learning-based approach impractical. Finally, while at first glance it seems that with a learning-based approach we can focus on developing program-independent and dependent features that can generalize the results to future improvements, the non-generalizable part of a learning-based approach is hidden in the manually labeled, tool-specific training data. In other words, for any ensemble approach to APR, there has to be a non-generalizable component that is specific to the new tool to be added, either in the form of labeled data (for a learning-based approach) or as explicitly stated repair patterns (as in **P-EPR**).

Performance metrics. TIT and HVT are not entirely representing the related costs, thus the newly derived TISP and HVSP can not

precisely measure the practical performance of **P-EPR**. However, the real tool execution and labor costs are also hard to be fairly and precisely measured even when executing the APR tools due to various factors such as machine performance and reviewer proficiency. Since no standard metrics are previously proposed for measuring the APR tool selection strategy, our proposed metrics contributes by providing an easy-to-compute though still imperfect quantified measurement for comparing different tool selection strategies.

Implications for practitioners. Our research has several implications. Firstly, our analysis demonstrates that the repair preference difference of current APR tools can be distinguished by simple bug features and test error types. Results of our experiments in RQ3 show practical cost savings through the deployment of **P-EPR** for selecting APR tools. It is important to note that the current version of **P-EPR** only optimizes the repair probability of different bugs. To further improve the tool selection strategy, researchers can consider incorporating additional execution information, such as GPU memory usage and test feedback during validation. This avenue holds promise for designing a better approach. Secondly, **P-EPR** provides a simple and extensive way to leverage existing APR tools for enhanced repair performance, at a lower cost on tool execution and patch validation compared with invoking all tools. For configuring and extending new APR tools to **P-EPR**, users only need to provide the APR tool's repair history and implemented pattern type (if available). This strategy can also benefit some scenarios where tools must be run locally. For instance, recent studies have explored bug repairs using large language models such as CodeX [45] and ChatGPT [47]. However, these methods may raise security concerns since the model holders only provide a remote API, which means users must post their codes to the remote host. Instead, our approach enables users to achieve comparable performance by locally executing existing tools.

7 THREATS TO VALIDITY

Threats to external validity include the evaluated dataset used in our experiment, i.e. Defects4J. We only evaluate **P-EPR** considering APR tools of Java on 395 bugs from Defects4J. However, repairing Java programs is the most popular research scene for the APR community and Defects4J is the most popular dataset. Besides, we evaluate **P-EPR** on a variety of APR tool combinations (up to 21 tools), which could alleviate the threats to some extent.

That we choose to perform a simulated experiment instead of executing APR tools could be a threat to internal validity. To reduce the threat, we collect repair results of existing APR tools following strict criteria to avoid biases brought by fault localization and patch generation settings. For learning-based tools that do not publish complete correct/plausible patches for the bugs used in our experiments, we re-run them following their experimental configurations in their papers or source code to collect the complete correct/plausible patches they generate for these bugs. Another threat is that the input orders of bugs could impact the performance of **P-EPR**. We mitigate it by conducting rich experiments under different input orders of bugs in RQ2.

8 CONCLUSION

We presented a practical approach, referred to as **P-EPR**, for selecting the most suitable automated program repair tools for a given software bug. **P-EPR** is designed as a flexible and tunable framework that can interface with any type and quantity of APR tools to align with users' preferences. We evaluated its effectiveness and generalizability using a variety of tool combinations (up to 21 APR tools) on the Defects4J dataset. Additionally, we proposed two novel metrics that measure the extent to which a model selection strategy can reduce tool invocation and human validation costs compared with invoking all tools. Our study demonstrated the potential for selecting optimal APR tools for distinct bugs, thus offering a novel and practical avenue for future research.

9 ACKNOWLEDGEMENT

This research / project is supported by the National Key Research and Development Program of China (2022YFF0711404), National Natural Science Foundation of China (62172214), Natural Science Foundation of Jiangsu Province, China (BK20201250, BK20210279), CCF-Huawei Populus Grove Fund, NSF award 2034508, and the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 949014). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s). We also thank the reviewers for their helpful comments. Chuanyi Li and Jidong Ge are the corresponding authors.

REFERENCES

[1] Aldeida Aleti and Matias Martinez. 2021. E-APR: Mapping the effectiveness of automated program repair techniques. *Empir. Softw. Eng.* 26, 5 (2021), 99. https://doi.org/10.1007/s10664-021-09989-x

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1409.0473

[3] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, et al. 2012. Adding virtualization capabilities to the Grid'5000 testbed. In *International Conference on Cloud Computing and Services Science*. Springer, 3–20.

[4] Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. 2020. Codit: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering* (2020).

[5] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1943–1959.

[6] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[7] Yangruibo Ding, Baishakhi Ray, Premkumar T. Devanbu, and Vincent J. Hellendoorn. 2020. Patching as Translation: the Data and the Metaphor. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 275–286. https://doi.org/10.1145/3324884.3416587

[8] Thomas Durieux, Benoit Cornu, Lionel Seinturier, and Martin Monperrus. 2017. Dynamic patch generation for null pointer exceptions using metaprogramming. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, Martin Pinzger, Gabriele Bavota, and Andrian Marcus (Eds.). IEEE Computer Society, 349–358. https://doi.org/10.1109/SANER.2017.7884635

[9] 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 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 302–313. https://doi.org/10.1145/3338906.3338911

[10] Thomas Durieux and Martin Monperrus. 2016. DynaMoth: dynamic code synthesis for automatic program repair. In *Proceedings of the 11th International Workshop on Automation of Software Test, AST@ICSE 2016, Austin, Texas, USA, May 14-15, 2016*, Christof J. Budnik, Gordon Fraser, and Francesca Lonetti (Eds.). ACM, 85–91. https://doi.org/10.1145/2896921.2896931

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

[12] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65. https://doi.org/10.1145/3318162

[13] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Towards practical program repair with on-demand candidate generation. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 12–23. https://doi.org/10.1145/3180155.3180245

[14] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 298–309. https://doi.org/10.1145/3213846.3213871

[15] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 1161–1173. https://doi.org/10.1109/ICSE43902.2021.00107

[16] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, Corina S. Pasareanu and Darko Marinov (Eds.). ACM, 437–440. https://doi.org/10.1145/2610384.2628055

[17] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 802–811. https://doi.org/10.1109/ICSE.2013.6606626

[18] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. FixMiner: Mining relevant fix patterns for automated program repair. *Empir. Softw. Eng.* 25, 3 (2020), 1980–2024. https://doi.org/10.1007/s10664-019-09780-z

[19] Xuan-Bach Dinh 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, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 593–604. https://doi.org/10.1145/3106237.3106309

[20] Kui Liu, Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*. IEEE, 102–113. https://doi.org/10.1109/ICST.2019.00020

[21] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, Xinyu Wang, David Lo, and Emad Shihab (Eds.). IEEE, 456–467. https://doi.org/10.1109/SANER.2019.8667970

[22] 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, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 31–42. https://doi.org/10.1145/3293882.3330577

[23] Kui Liu, Li Li, Anil Koyuncu, Dongsun Kim, Zhe Liu, Jacques Klein, and Tegawendé F. Bissyandé. 2021. A Critical Review on the Evaluation of Automated Program Repair Systems. *Journal of Systems and Software* 171 (2021), 110817. https://doi.org/10.1016/j.jss.2020.110817

[24] 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 *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 615–627. https://doi.org/10.1145/3377811.3380338

[25] Fan Long, Peter Amidon, and Martin C. Rinard. 2017. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, 727–739. https://doi.org/10.1145/3106237.3106253

[26] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: combining context-aware neural translation models using ensemble for program repair. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 101–114. https://doi.org/10.1145/3395363.3397369

[27] Fernanda Madeiral, Simon Urli, Marcelo de Almeida Maia, and Martin Monperrus. 2019. BEARS: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, Xinyu Wang, David Lo, and Emad Shihab (Eds.). IEEE, 468–478. https://doi.org/10.1109/SANER.2019.8667991

[28] Matias Martinez and Martin Monperrus. 2016. ASTOR: a program repair library for Java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 441–444. https://doi.org/10.1145/2931037.2948705

[29] Matias Martinez and Martin Monperrus. 2018. Ultra-Large Repair Search Space with Automatically Mined Templates: The Cardumen Mode of Astor. In *Search-Based Software Engineering - 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11036)*, Thelma Elita Colanzi and Phil McMinn (Eds.). Springer, 65–86. https://doi.org/10.1007/978-3-319-99241-9_3

[30] Ehsan Mashhadi and Hadi Hemmati. 2021. Applying CodeBERT for Automated Program Repair of Java Simple Bugs. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*. IEEE, 505–509. https://doi.org/10.1109/MSR52588.2021.00063

[31] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Comput. Surv.* 51, 1 (2018), 17:1–17:24. https://doi.org/10.1145/3105906

[32] Martin Monperrus. 2018. *The living review on automated program repair*. Ph. D. Dissertation. HAL Archives Ouvertes.

[33] Manish Motwani, Sandhya Sankaranarayanan, René Just, and Yuriy Brun. 2018. Do automated program repair techniques repair hard and important bugs? *Empirical Software Engineering* 23, 5 (2018), 2901–2947.

[34] 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

[35] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 254–265. https://doi.org/10.1145/2568225.2568254

[36] Zichao Qi, Fan Long, Sara Achour, and Martin C. 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*, Michal Young and Tao Xie (Eds.). ACM, 24–36. https://doi.org/10.1145/2771783.2771791

[37] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. 2017. ELIXIR: effective object oriented program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 648–659. https://doi.org/10.1109/ASE.2017.8115675

[38] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 532–543.

[39] 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 (2019), 19:1–19:29. https://doi.org/10.1145/3340544

[40] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 364–374. https://doi.org/10.1109/ICSE.2009.5070536

[41] Cathrin Weiß, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. 2007. How Long Will It Take to Fix This Bug?. In *Fourth International Workshop on Mining Software Repositories, MSR 2007 (ICSE Workshop), Minneapolis, MN, USA, May 19-20, 2007, Proceedings*. IEEE Computer Society, 1. https://doi.org/10.1109/MSR.2007.13

[42] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 1–11. https://doi.org/10.1145/3180155.3180233

[43] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. 2019. Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, Xinyu Wang, David Lo, and Emad Shihab (Eds.). IEEE, 479–490. https://doi.org/10.1109/SANER.2019.8668043

[44] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

[45] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2022. Practical Program Repair in the Era of Large Pre-trained Language Models. *CoRR* abs/2210.14179 (2022). https://doi.org/10.48550/arXiv.2210.14179 arXiv:2210.14179

[46] Chunqiu Steven Xia and Lingming Zhang. 2022. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 959–971. https://doi.org/10.1145/3540250.3549101

[47] Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the Conversation Going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. *CoRR* abs/2304.00385 (2023). https://doi.org/10.48550/arXiv.2304.00385 arXiv:2304.00385

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

[49] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian R. Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Trans. Software Eng.* 43, 1 (2017), 34–55. https://doi.org/10.1109/TSE.2016.2560811

[50] Deheng Yang, Xiaoguang Mao, Liqian Chen, Xuezheng Xu, Yan Lei, David Lo, and Jiayu He. 2022. TransplantFix: Graph Differencing-based Code Transplantation for Automated Program Repair. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[51] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural Program Repair with Execution-based Backpropagation. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1506–1518. https://doi.org/10.1145/3510003.3510222

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

[53] Wenkang Zhong, Hongliang Ge, Hongfei Ai, Chuanyi Li, Kui Liu, Jidong Ge, and Bin Luo. 2022. StandUp4NPR: Standardizing SetUp for Empirically Comparing Neural Program Repair Systems. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[54] 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 *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 341–353. https://doi.org/10.1145/3468264.3468544