TBar: Revisiting Template-Based Automated Program Repair

Kui Liu, Anil Koyuncu, Dongsun Kim, Tegawendé F. Bissyandé
University of Luxembourg, Luxembourg
{kui.liu,anil.koyuncu,dongsun.kim,tegawende.bissyande}@uni.lu

ABSTRACT

We revisit the performance of template-based APR to build comprehensive knowledge about the effectiveness of fix patterns, and to highlight the importance of complementary steps such as fault localization or donor code retrieval. To that end, we first investigate the literature to collect, summarize and label recurrently-used fix patterns. Based on the investigation, we build TBar, a straightforward APR tool that systematically attempts to apply these fix patterns to program bugs. We thoroughly evaluate TBar on the Defects4J benchmark. In particular, we assess the actual qualitative and quantitative diversity of fix patterns, as well as their effectiveness in yielding plausible or correct patches. Eventually, we find that, assuming a perfect fault localization, TBar correctly/plausibly fixes 74/101 bugs. Replicating a standard and practical pipeline of APR assessment, we demonstrate that TBar correctly fixes 43 bugs from Defects4J, an unprecedented performance in the literature (including all approaches, i.e., template-based, stochastic mutation-based or synthesis-based APR).

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**; *Software defect analysis*; Software testing and debugging.

KEYWORDS

Automated program repair, fix pattern, empirical assessment.

ACM Reference Format:
Kui Liu, Anil Koyuncu, Dongsun Kim, 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 '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3293882.3330577

1 INTRODUCTION

Automated Program Repair (APR) has progressively become an essential research field. APR research is indeed promising to improve modern software development by reducing the time and costs associated with program debugging tasks. In particular, given that faults in software cause substantial financial losses to the software industry [8, 54], there is a momentum in minimizing the time-to-fix intervals by APR. Recently, various APR approaches [10, 11, 17, 18, 21, 23, 25, 26, 29, 31, 36, 38, 39, 41, 42, 44, 51, 53, 67, 69, 76, 77] have been proposed, aiming at reducing manual debugging efforts through automatically generating patches.

An early strategy of APR is to generate concrete patches based on fix patterns [23] (also referred to as fix templates [40] or program transformation schemas [17]). This strategy is now common in the literature and has been implemented in several APR systems [13, 17, 23, 24, 38–40, 50, 62]. Kim et al. [23] showed the usefulness of fix patterns with *PAR*. Saha et al. [62] later proposed *ELIXIR* by adding three new patterns on top of *PAR* [23]. Durieux et al. [13] proposed *NPEfix* to repair null pointer exception bugs, using nine pre-defined fix patterns. Long et al. designed *Genesis* [41] to infer fix patterns for specific three classes of defects. Liu and Zhong [40] explored posts from Stack Overflow to mine fix patterns for APR. Hua et al. proposed *SketchFix* [17], a runtime on-demand APR tool with six pre-defined fix patterns. Recently, Liu et al. [39] used the fix patterns of FindBugs static violations [35] to fix semantic bugs. Concurrently, Ghanbari and Zhang [15] showed that straightforward application of fix patterns (i.e., mutators) on Java bytecode is effective for repair. They do not, however, provide a comprehensive assessment of the repair performance yielded by each implemented mutator.

Although the literature has reported promising results with fix patterns-based APR, to the best of our knowledge, no extensive assessment on the effectiveness of various patterns is performed. A few most recent approaches [17, 39, 40] reported which benchmark bugs are fixed by each of their patterns. Nevertheless, many relevant questions on the effectiveness of fix patterns remain unanswered.

This paper. Our work thoroughly investigates to what extent fix patterns are effective for program repair. In particular, emphasizing on the recurrence of some patterns in APR, we dissect their actual contribution to repair performance. Eventually, we explore three aspects of fix patterns:

- *Diversity*: How diverse are the fix patterns used by the state-of-the-art? We survey the literature to identify and summarize the available patterns with a clear taxonomy.
- *Repair performance*: How effective are the different patterns? In particular, we investigate the variety of real-world bugs that can be fixed, the dissection of repair results, and their tendency to yield plausible or correct patches.
- *Sensitivity to fault localization noise*: Are all fix patterns similarly sensitive to the false positives yielded by fault localization tools? We investigate sensitivity by assessing plausible patches as well as the suspiciousness rank of correctly-fixed bug locations.

Towards realizing this study, we implement an automated patch generation system, TBar (Template-Based automated program repair), with a super-set of fix patterns that are collected, summarized, curated and labeled from the literature data. We evaluate TBar on the Defects4J [20] benchmark, and provide the replication package in a public repository: https://github.com/SerVal-DTF/TBar.

Overall, our investigations have yielded the following findings:

ISSTA '19, July 15–19, 2019, Beijing, China

ACM ISBN 978-1-4503-6224-5/19/07…$15.00
https://doi.org/10.1145/3293882.3330577

(1) **Record performance:** TBar creates a new higher baseline of repair performance: 74/101 bugs are correctly/plausibly fixed with perfect fault localization information and 43/81 bugs are fixed with realistic fault localization output, respectively.
(2) **Fix pattern selection:** Most bugs are correctly fixed only by a single fix pattern while other patterns generate plausible patches. This implies that appropriate pattern prioritization can prevent from plausible/incorrect patches. Otherwise, APR tools might be overfitted in plausible but incorrect patches.
(3) **Fix ingredient retrieval:** It is challenging for template-based APR to select appropriate donor code, which is an ingredient of patch generation when using fix patterns. Inappropriate donor code may cause plausible but incorrect patch generation. This motivates a new research direction: *donor code prioritization.*
(4) **Fault localization noise:** It turns out that fault localization accuracy has a large impact on repair performance when using fix patterns in APR (e.g., applying a fix pattern to incorrect location yields plausible/incorrect patches).

2 FIX PATTERNS

For this study, we systematically review[1] the APR literature to identify approaches that leverage fix patterns. Concretely, we consider the program repair website [3], a bibliography survey of APR [52], proceedings of software engineering conference venues and journals as the source of relevant literature. We focus on approaches dealing with Java program bugs, and manually collect, from the paper descriptions as well as the associated artifacts, all pattern instances that are explicitly mentioned. Table 1 summarizes the identified relevant literature and the quantity of identified fix patterns targeting Java programs. Note that the techniques described in the last four papers (i.e., HDRepair, ssFix, CapGen, and SimFix papers) do not directly use fix patterns: they leverage code change operators or rules, which we consider similar to using fix patterns.

Table 1: Literature review on fix patterns for Java programs.

Authors	APR tool name	# of fix patterns	Publication Venue	Publication Year
Pan et al. [55]	-	27	EMSE	2009
Kim et al. [23]	PAR	10 (16*)	ICSE	2013
Martinez et al. [49]	jMutRepair	2	ISSTA	2016
Durieux et al. [13]	NPEfix	9	SANER	2017
Long et al. [41]	Genesis	3 (108*)	FSE	2017
D. Le et al. [25]	S3	4	FSE	2017
Saha et al. [62]	ELIXIR	8 (11*)	ASE	2017
Hua et al. [17]	SketchFix	6	ICSE	2018
Liu and Zhong [40]	SOFix	12	SANER	2018
Koyuncu et al. [24]	FixMiner	28	UL Tech Report	2018
Liu et al. [35]	-	174	TSE	2018
Rolim et al. [60]	REVISAR	9	UFERSA Tech Report	2018
Liu et al. [39]	AVATAR	13	SANER	2019
D. Le et al. [29]	HDRepair†	11	SANER	2016
Xin and Reiss [74]	ssFix†	34	ASE	2017
Wen et al. [69]	CapGen†	30	ICSE	2018
Jiang et al. [18]	SimFix†	16	ISSTA	2018

*In the PAR paper [23], 10 fix patterns are presented, but 16 fix patterns are released online [2]. In Genesis, 108 code transformation schemas are inferred for three kinds of defects. In ELIXIR, there is one fix pattern that consists of four sub-fix patterns.

2.1 Fix Patterns Inference

Fix patterns have been explored with the following four ways:

(1) **Manual Summarization**: Pan et al. [55] identified 27 fix patterns from patches of five Java projects to characterize the fix ingredients of patches. They do not however apply the identified patterns to fix actual bugs. Motivated by this work, Kim et al. [23] summarized 10 fix patterns manually extracted from 62,656 human-written patches collected from Eclipse JDT.
(2) **Mining**: Long et al. [41] proposed Genesis, to infer fix patterns for three kinds of defects from existing patches. Liu and Zhong [40] explored fix patterns from Q&A posts in Stack Overflow. Koyuncu et al. [24] mined fix patterns at the AST level from patches by using code change differentiating tool [14]. Liu et al. [35] and Rolim et al. [60] proposed to mine fix patterns for static analysis violations. In general, mining approaches yield a large number of fix patterns, which are not always about addressing deviations in program behavior. For example, many patterns are about code style [39]. Recently, with *AVATAR* [39], we proposed an APR tool that considers static analysis violation fix patterns to fix semantic bugs.
(3) **Pre-definition**: Durieux et al. [13] pre-defined 9 repair actions for null pointer exceptions by unifying the related fix patterns proposed in previous studies [12, 22, 45]. On the top of *PAR* [23], Saha et al. [62] further defined 3 new fix patterns to improve the repair performance. Hua et al. [17] proposed an APR tool with six pre-defined so-called code transformation schemas. We also consider operator mutations [49] as pre-defined fix patterns, as the number of operators and mutation possibilities is limited and pre-set. Xin and Reiss [74] proposed an approach to fixing bugs with 34 predefined code change rules at the AST level. Ten of the rules are not for transforming the buggy code but for the simple replacement of multi-statement code fragments. We discard these rules from our study to limit bias.
(4) **Statistics**: Besides formatted fix patterns, researchers [18, 69] also explored to automate program repair with code change instructions (at the abstract syntax tree level) that are statistically recurrent in existing patches [18, 37, 48, 68, 81]. The strategy is then to select the top-*n* most frequent code change instructions as fix ingredients to synthesize patches.

2.2 Fix Patterns Taxonomy

After manually assessing all fix patterns presented in the literature (cf. Table 1), we identified 15 categories of patterns labeled based on the code context (e.g., *a cast expression*), the code change actions (e.g., *insert an "if" statement with "instanceof" check*) as well as the targets (e.g., *ensure the program will no throw a ClassCastException.*). A given category may include one or several specialized sub-categories. Below, we present the labeled categories and provide the associated 35 **Code Change Patterns** described in simplified GNU diff pattern for easy understanding.

FP1. Insert Cast Checker. Inserting an *instanceof* check before one buggy statement if this statement contains at least one unchecked cast expression. **Implemented in:** PAR, Genesis, AVATAR, SOFix†, HDRepair†, SketchFix†, CapGen†, and SimFix†.

```
+ if (exp instanceof T) {
      var = (T) exp; ......
+ }
```

where *exp* is an expression (e.g., a variable expression) and *T* is the casting type, while "....." means the subsequent statements dependent on the variable *var*. Note that, "†" denotes that the fix pattern is not specifically illustrated in the corresponding APR tools

[1] For conferences and journals, we consider ICSE, FSE, ASE, ISSTA, ICSME, SANER, TSE, TOSEM, and EMSE. The search keywords are 'program'+'repair', 'bug' +'fix'.

since the tools have some abstract fix patterns that can cover the fix pattern. The same notation applies to the following descriptions.

FP2. Insert Null Pointer Checker. Inserting a *null* check before a buggy statement if, in this statement, a field or an expression (of non-primitive data type) is accessed without a null pointer check. **Implemented in:** PAR, ELIXIR, NPEfix, Genesis, FixMiner, AVATAR, HDRepair†, SOFix†, SketchFix†, CapGen†, and SimFix†.

```
FP2.1:  + if (exp != null) {
              ...exp...; ......
        + }
FP2.2:  + if (exp == null) return DEFAULT_VALUE;
          ...exp...;
FP2.3:  + if (exp == null) exp = exp1;
          ...exp...;
FP2.4:  + if (exp == null) continue;
          ...exp...;
FP2.5:  + if (exp == null)
        +     throw new IllegalArgumentException(...);
          ...exp...;
```

where DEFAULT_VALUE is set based on the return type (RT) of the encompassing method as below:

$$\texttt{DEFAULT\_VALUE} = \begin{cases} \text{false}, & \text{if } RT = boolean; \\ 0, & \text{if } RT = primitive\ type; \\ \text{new String()}, & \text{if } RT = String; \\ \text{“return;”}, & \text{if } RT = void; \\ \text{null}, & otherwise. \end{cases} \quad (1)$$

exp1 is a compatible expression in the buggy program (i.e., that has the same data type as *exp*). **FP2.4** is specific to the case of a buggy statement within a loop (i.e., *for* or *while*).

FP3. Insert Range Checker. Inserting a range checker for the access of an array or collection if it is unchecked. **Implemented in:** PAR, ELIXIR, Genesis, SketchFix, AVATAR, SOFix† and SimFix†.

```
      + if (index < exp.length) {
            ...exp[index]...; ......
      + }
OR
      + if (index < exp.size()) {
            ...exp.get(index)...; ......
      + }
```

where *exp* is an expression representing an array or collection.

FP4. Insert Missed Statement. Inserting a missing statement before, or after, or surround a buggy statement. The statement is either an expression statement with a method invocation, or a *return/try-catch/if* statement. **Implemented in:** ELIXIR, HDRepair, SOFix, SketchFix, CapGen, FixMiner, and SimFix.

```
FP4.1:  + method(exp);
FP4.2:  + return DEFAULT_VALUE;
FP4.3:  + try {
              statement; ......
        + } catch (Exception e) {   ...   }
FP4.4:  + if (conditional_exp) {
              statement; ......
        + }
```

where *exp* is an expression from a buggy *statement*. It may be empty if the method does not take any argument. **FP4.4** excludes three fix patterns (**FP1**, **FP2**, and **FP3**) that are used with specific contexts.

FP5. Mutate Class Instance Creation. Replacing a class instance creation expression with a cast *super.clone()* method invocation if the class instance creation is in an overridden clone method. **Implemented in:** AVATAR.

```
      public Object clone() {
      -    ... new T();
      +    ... (T) super.clone();
      }
```

where *T* is the class name of the current class containing the buggy statement.

FP6. Mutate Conditional Expression. Mutating a conditional expression that returns a boolean value (i.e., `true` or `false`) by either updating it, or removing a sub conditional expression, or inserting a new conditional expression into it. **Implemented in:** PAR, ssFix, S3, HDRepair, ELIXIR, SketchFix, CapGen, SimFix, and AVATAR.

```
FP6.1:  - ...condExp1...
        + ...condExp2...
FP6.2:  - ...condExp1 Op condExp2...
        + ...condExp1...
FP6.3:  - ...condExp1...
        + ...condExp1 Op condExp2...
```

where *condExp1* and *condExp2* are conditional expressions. *Op* is the logical operator '||' or '&&'. The mutation of operators in conditional expressions is not summarized in this fix pattern but in **FP11**.

FP7. Mutate Data Type. Replacing the data type in a variable declaration or a cast expression with another data type. **Implemented in:** PAR, ELIXIR, FixMiner, SOFix, CapGen, SimFix, AVATAR, and HDRepair†.

```
FP7.1:  - T1 var ...;
        + T2 var ...;
FP7.2:  - ...(T1) exp...;
        + ...(T2) exp...;
```

where both *T1* and *T2* denote two different data types. *exp* means the being casted expression (including variable).

FP8. Mutate Integer Division Operation. Mutating the integer division expressions to return a float value, by mutating its divisor or divider to make them be of type float. **Released by** Liu et al. [35], it is not implemented in any APR tool yet.

```
FP8.1:  - ...dividend / divisor...
        + ...dividend / (double or float) divisor...
FP8.2:  - ...dividend / divisor...
        + ...(double or float) dividend / divisor...
FP8.3:  - ...dividend / divisor...
        + ...(1.0 / divisor) * dividend...
```

where *dividend* and *divisor* are integer number literals or integer-returned expressions (including variables).

FP9. Mutate Literal Expression. Mutating boolean, number, or String literals in a buggy statement with other relevant literals, or correspondingly-typed expressions. **Implemented in:** HDRepair, S3, FixMiner, SketchFix, CapGen, SimFix and ssFix†.

```
FP9.1:  - ...literal1...
        + ...literal2...
FP9.2:  - ...literal1...
        + ...exp...
```

where *literal1* and *literal2* are of the same type literals, but having different values (e.g., *literal1* is `true`, *literal2* is `false`). *exp* denotes any expression value of the same type as *literal1*.

FP10. Mutate Method Invocation Expression. Mutating the buggy method invocation expression by adapting its method name or arguments. This pattern consists of four sub fix patterns:

(1) Replacing the method name with another one which has a compatible return type and same parameter type(s) as the buggy method that was invoked.
(2) Replacing at least one argument with another expression which has a compatible data type. Replacing a literal or variable is not included in this fix pattern, but rather in **FP9** and **FP13** respectively.
(3) Removing argument(s) if the method invocation has the suitable overridden methods.

(4) Inserting argument(s) if the method invocation has the suitable overridden methods.

Implemented in: PAR, HDRepair, ssFix, ELIXIR, FixMiner, SOFix, SketchFix, CapGen, and SimFix.

```
FP10.1:  - ...method1(args)...
         + ...method2(args)...
FP10.2:  - ...method1(arg1, arg2, ...)...
         + ...method1(arg1, arg3, ...)...
FP10.3:  - ...method1(arg1, arg2, ...)...
         + ... method1(arg1, ...)...
FP10.4:  - ...method1(arg1, ...)...
         + ...method1(arg1, arg2, ...)...
```

where *method1* and *method2* are the names of invoked methods. *args*, *arg1*, *arg2* and *arg3* denote the argument expressions in the method invocation. Note that, code changes on class instance creation, constructor and super constructor expressions are also included in these four fix patterns.

FP11. Mutate Operators. Mutating an operation expression by mutating its operator(s). We divide this fix pattern into three sub-fix patterns following the operator types and mutation actions.

(1) Replacing one operator with another operator from the same operator class (e.g., relational or arithmetic).
(2) Changing the priority of arithmetic operators.
(3) Replacing `instanceof` operator with (in)equality operators.

Implemented in: HDRepair, ssFix, ELIXIR, S3, jMutRepair, SOFix, FixMiner, SketchFix, CapGen, SimFix, AVATAR, and PAR[†].

```
FP11.1:  - ...exp1 Op1 exp2...
         + ...exp1 Op2 exp2...
FP11.2:  - ...(exp1 Op1 exp2) Op2 exp3...
         + ...exp1 Op1 (exp2 Op2 exp3)...
FP11.3:  - ...exp instanceof T...
         + ...exp != null...
```

where *exp* denotes the expressions in the operation and *Op* is the associated operator.

FP12. Mutate Return Statement. Replacing the expression (excluding literals, variables, and conditional expressions) in a return statement with a compatible expression. **Implemented in:** ELIXIR, SketchFix, and HDRepair[†].

```
- return exp1;
+ return exp2;
```

where *exp1* and *exp2* represent the returned expressions.

FP13. Mutate Variable. Replacing a variable in a buggy statement with a compatible expression (including variables and literals). **Implemented in:** S3, SOFix, FixMiner, SketchFix, CapGen, SimFix, AVATAR, and ssFix[†].

```
FP13.1:  - ...var1...
         + ...var2...
FP13.2:  - ...var1...
         + ...exp...
```

where *var1* denotes a variable in the buggy statement. *var2* and *exp* represent respectively a compatible variable and expression of the same type as *var1*.

FP14. Move Statement. Moving a buggy statement to a new position. **Implemented in:** PAR.

```
- statement;
  ......
+ statement;
```

where *statement* represents the buggy statement.

FP15. Remove Buggy Statement. Deleting entirely the buggy statement from the program. **Implemented in:** HDRepair, SOFix, FixMiner, CapGen, and AVATAR.

```
FP15.1:    ......
         - statement;
           ......
FP15.2:  - methodDeclaration(Arguments) {
         -  ......; statement;......
         - }
```

where *statement* denotes any identified buggy statement, and *method* represents the encompassing method.

2.3 Analysis of Collected Patterns

We provide a study of the collected fix patterns following quantitative (overall set) and qualitative (per fix pattern) aspects. Table 2 assesses the fix patterns in terms of four qualitative dimensions:

(1) **Change Action**: what high-level operations are applied on a buggy code entity? On the one hand, *Update* operations replace the buggy code entity with retreived donor code, while *Delete* operations just remove the buggy code entity from the program. On the other hand, *Insert* operations insert an otherwise missing code entity into the program, and *Move* operations change the position of the buggy code entity to a more suitable location in the program.
(2) **Change Granularity**: what kinds of code entities are directly impacted by the change actions? This entity can be an entire *Method*, a whole *Statement* or specifically targeting an *Expression* within a statement.
(3) **Bug Context**: what specific AST nodes of code entities are used to match fix patterns.
(4) **Change Spread**: the number of statements impacted by each fix pattern.

Quantitatively, as summarized in Table 3, 17 fix patterns are related to *Update* change actions, 4 fix patterns implement *Delete* actions, 13 fix patterns *Insert* extra code, and only 1 fix pattern is associated to *Move* change action.

In terms of change granularity, 21 and 17 fix patterns are applied respectively at the expression and statement code entity levels [2]. Only 1 fix pattern is suitable at the method level.

Overall, we note that 30 fix patterns are applicable to a single statement, while 7 fix patterns can mutate multiple statements at the same time. Among these patterns, **FP14** and **FP15.1** can both mutate single and multiple statements.

3 SETUP FOR REPAIR EXPERIMENTS

In order to assess the effectiveness of fix patterns in the taxonomy presented in Section 2, we design program repair experiments using the fix patterns as the main ingredients. The produced APR system is then assessed on a widely-used benchmark in the repair community to allow reliable comparison against the state-of-the-art.

3.1 TBar: a Baseline APR System

Based on the investigations of recurrently-used fix patterns, we build TBar, a template-based APR tool which integrates the 35 fix patterns presented in Section 2. We expect the APR community to consider TBar as a baseline APR tool: new approaches must come up with novel techniques for solving auxiliary issues (e.g., repair precision, search space optimization, fault locations re-prioritization,

[2] Among these, four sub-fix patterns (**FP10**) can be applied to either expressions or statements, given that constructor and super-constructor code entities in Java program are grouped into statement level in terms of abstract syntax tree by Eclipse JDT.

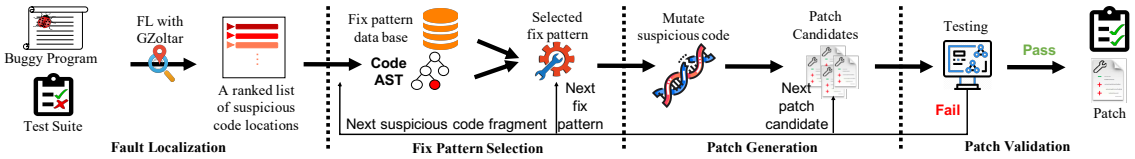


Figure 1: The overall workflow of TBar.

Table 2: Change properties of fix patterns.

Fix Pattern	Change Action	Change Graunlarity	Bug Context	Change Spread
FP1	Insert	statement	cast expression	single
FP2.1	Insert	statement	a variable or an expression returning non-primitive-type data	single
FP2.(2,3,4,5)				dual
FP3	Insert	statement	element access of array or collection variable	single
FP4.(1,2,3,4)	Insert	statement	any statement	single
FP5	Update	expression	class instance creation expression and clone method	single
FP6.1	Update	expression	conditional expression	single
FP6.2	Delete			
FP6.3	Insert			
FP7.1	Update	expression	variable declaration expression	single
FP7.2	Update	expression	cast expression	single
FP8.(1,2,3)	Update	expression	integral division expression	single
FP9.(1,2)	Update	expression	literal expression	single
FP10.1	Update	expression, or statement	method invocation, class instance creation, constructor, or super constructor	single
FP10.2				
FP10.3	Delete			
FP10.4	Insert			
FP11.1	Update	expression	assignment or infix-expression	single
FP11.2	Update	expression	arithmetic infix-expression	single
FP11.3	Update	expression	instance of expression	single
FP12	Update	expression	return statement	single
FP13.(1, 2)	Update	expression	variable expression	single
FP14	Move	statement	any statement	single or multiple
FP15.1	Delete	statement	any statement	single or multiple
FP15.2	Delete	method	any statement	multiple

Table 3: Diversity of fix patterns w.r.t change properties.

Action Type	# fix patterns	Granularity	# fix patterns	Spread	# fix patterns
Update	17	Expression	21	Single-Statement	30
Delete	4	Statement	17		
Insert	13	Method	1	Multiple-Statements	7
Move	1				

etc.) to boost automated program repair beyond the performance that a straightforward application of common fix patterns can offer. Figure 1 overviews the workflow that we have implemented in TBar. We describe in the following subsections the role and operation of each process as well as all necessary implementation details.

3.1.1 Fault Localization. Fault localization is necessary for template-based APR as it allows to identify a list of suspicious code locations (i.e., buggy statements) on which to apply the fix patterns. TBar leverages the GZoltar [9] framework to automate the execution of test cases for each buggy program. In this framework, we use the Ochiai [4] ranking metric to compute the suspiciousness scores of statements that are likely to be the faulty code locations. This ranking metric has been demonstrated in several empirical studies [56, 65, 73, 78] to be effective for localizing faults in object-oriented programs. The GZoltar framework for fault localization is also widely used in the literature of APR [18, 24, 34, 38, 39, 49, 69, 74, 76, 77], allowing for a fair assessment of TBar's performance against the state-of-the-art.

3.1.2 Fix Pattern Selection. In the execution of the repair pipeline, once the fault localization process yields a list of suspicious code locations, TBar sequentially attempts to select the encoded fix patterns from its database of fix patterns for each statement in the locations list. The selection of fix patterns is conducted in a naïve way based on the AST context information of each suspicious statement. Specifically, TBar sequentially traverses each node of the suspicious statement AST from its first child node to its last leaf node and tries to match each node against the context AST of the fix pattern. If a node can match any bug context presented in Table 2, a related fix pattern will be matched to generate patch candidates with the corresponding code change pattern. If the node is not a leaf node, TBar keeps traversing its children nodes. For example, if the first child node of a suspicious statement is a method invocation expression, it will be first matched with **FP10. Mutate Method Invocation Expression** fix pattern. If the children nodes of the method invocation start from a variable reference, it will be matched with **FP13. Mutate Variable** fix pattern as well. Other fix patterns follow the same manner. After all expression nodes of a suspicious statement are matched with fix patterns, TBar further matches fix patterns from statement and method levels respectively.

3.1.3 Patch Generation and Validation. When a matching fix pattern is found (i.e., a pattern is selected for a suspicious statement), a patch is generated by mutating the statement, then the patched program is run against the test suite. If the patched program passes all tests successfully, the patch candidate is considered as a *plausible* patch [58]. Once such a plausible patch is identified, TBar stops generating other patch candidates for this bug to fix bugs in a standard and practical program repair workflow [38, 39, 49, 76, 77], but does not generate all plausible patches for each bug, unlike PraPR [15]. Otherwise, the pattern selection and patch generation process is resumed until all AST nodes of buggy code are traversed. When several fix pattern contexts match one node, their actions are used for ordering: TBar prioritizes Update over Insert that is over Delete, which is prioritized over Move. In case of multiple donor code options for a given fix pattern, the candidate patches (each generated with a specific donor code) are ordered based on the distances between donor code node and buggy code node in the AST of the buggy code file: priority is given to smaller distances. Due to space limitation, detailed steps, illustrated in an algorithmic pseudo-code, are released in the replication package.

Considering that some buggy programs have several buggy locations, if a patch candidate can make a buggy program pass a sub-set of previously failing test cases without failing any previously passing test cases, this patch is considered as a plausible sub-patch of this buggy program. TBar will further validate other patch candidates, until either a plausible patch is generated, or all patch candidates are validated, or TBar exhausts the time limitation set (i.e., three hours) for repair attempts.

If a plausible patch is generated, we further manually check the equivalence between this patch and the ground-truth patch provided by developers and available in the Defects4J benchmark. If the plausible patch is semantically equivalent to the ground-truth patch, the plausible patch is considered as *correct*. Otherwise, it is only considered as plausible. We offer a replication package with extensive details on pattern implementation within TBar. Source code is publicly available in the aforementioned GitHub repository.

3.2 Assessment Benchmark

For our empirical assessments, we selected the Defects4J [20] dataset as the evaluation benchmark of TBar. This benchmark includes test cases for buggy Java programs with the associated developer fixes. Defects4J is an ideal benchmark for the objective of this study, since it has been widely used by most recent state-of-the-art APR systems targeting Java program bugs. Table 4 provides summary statistics on the bugs and test cases available in the version 1.2.0 [1] of Defects4J which we use in this study.

Table 4: Defects4J dataset information.

Project	Chart (C)	Closure (Cl)	Lang (L)	Math (M)	Mockito (Mc)	Time (T)	Total
# bugs	26	133	65	106	38	27	395
# test cases	2,205	7,927	2,245	3,602	1,457	4,130	21,566
# fixed bugs by all APR tools (cf. [38, 39])	13	16	28	37	3	4	101

Overall, we note that, to date, **101** Defects4J bugs have been correctly fixed by at least one APR tool published in the literature. Nevertheless, we recall that SimFix [18] currently holds the record number of bugs fixed by a single tool, which is **34**.

4 ASSESSMENT

This section presents and discusses the results of repair experiments with TBar. In particular, we conduct two experiments for:

- **Experiment #1:** Assessing the effectiveness of the various fix patterns implemented in TBar. To avoid the bias that fault localization can introduce with its false positives (cf. [38]), we directly provide perfect localization information to TBar.
- **Experiment #2:** Evaluating TBar in a normal program repair scenario. We investigate in particular the tendency of fix patterns to produce more or less incorrect patches.

4.1 Repair Suitability of Fix Patterns

Our first experiment focuses on assessing the patch generation performance of fix patterns for real bugs. In particular, we investigate three research questions in Experiment #1.

Research Questions for Experiment #1

RQ1. How many real bugs from Defects4J can be correctly fixed by fix patterns from our taxonomy?
RQ2. Can each Defects4J bug be fixed by different fix patterns?
RQ3. What are the properties of fix patterns that are successfully used to fix bugs?

In a recent study, Liu et al. [38] reported how fault localization techniques substantially affect the repair performance of APR tools. Given that, in this experiment, the APR tool (namely TBar) is only used as a means to apply the fix patterns in order to assess their effectiveness, we must eliminate the fault localization bias. Therefore, we assume that the bug positions at statement level are known, and we directly provide it to the patch generation step of TBar, without running any fault localization tool (which is part of the normal APR workflow, see Figure 1). To ensure readability across our experiments, we denote this version of the APR system as $\texttt{TBar}_p$ (where p stands for *perfect localization*). Table 5 summarizes the experimental results of $\texttt{TBar}_p$.

Table 5: Number of bugs fixed by fix patterns with $\texttt{TBar}_p$.

Fixed Bugs	C	Cl	L	M	Mc	T	Total
# of Fully Fixed Bugs	12/13	20/26	13/18	23/35	3/3	3/6	74/101
# of Partially Fixed Bugs	2/4	3/6	1/4	0/4	0/0	1/1	7/20

*We provide x/y numbers: x is the number of correctly fixed bugs; y is the number of bugs fixed with plausible patches. The same notation applies to Table 7.

Among 395 bugs in the Defects4J benchmark, $\texttt{TBar}_p$ can generate plausible patches for 101 bugs. 74 of these bugs are fixed with correct patches. We also note that $\texttt{TBar}_p$ can partially fix[3] 20 bugs with plausible patches, and 8 of them are correct. In a previous study, the kPAR [38] baseline tool (i.e., a Java implementation of the PAR [23] seminal template-based APR tool) was correctly/plausibly fixing 36/55 Defects4J bugs when assuming perfect localization.

While the results of $\texttt{TBar}_p$ are promising, ~79%(=314/395) of bugs cannot be correctly fixed with the available fix patterns. We manually investigated these unfixed bugs and make the following observations as research directions for improving the fix rates:

(1) *Insufficient fix patterns.* Many bugs are not fixed by $\texttt{TBar}_p$ simply due to the absence of matching fix patterns. This suggests that the fix patterns collected in the literature are far from being representative for real-world bugs. The community must thus keep contributing with effective techniques for mining fix patterns from existing patches.
(2) *Ineffective search of fix ingredients.* Template-based APR is a kind of search-based APR [69]: some fix patterns require donor code (i.e., fix ingredients) to generate actual patches. For example, as shown in Figure 2, to apply the relevant fix pattern **FP9.2**, one needs to identify fix ingredient "`ImageMapUtilities.htmlEscape`" as the necessary in generating the patch. The current implementation of TBar limits its search space for donor code to the "***local***" file where the bug is localized. It is a limitation to find the correct donor code, but it reduces the risk of search space explosion. In addition, TBar leverages the context of buggy code to prune away irrelevant fix ingredients. Therefore, some bugs cannot be fixed by TBar although its fix pattern can match with code change actions. With more effective search strategies (e.g., larger search space such as fix ingredients from other projects as in [34]), there might be more chances to fix more bugs.

RQ1: *The collected fix patterns can be used to correctly fix 74 real bugs from the Defects4J dataset. A larger portion of the dataset remains however unfixed by $\texttt{TBar}_p$, notably due to (1) the limitations of the fix patterns set and to (2) the naïve search strategy for finding relevant fix ingredients to build concrete patches from patterns.*

Figure 3 summarizes the statistics on the number of bugs that can be fixed by one or several fix patterns. The *Y*-axis denotes the number of fix patterns (i.e., n = 1, 2, 3, 4, 5, and >5) that can generate *plausible* patches for a number of bugs (*X*-axis). The legend indicates that "**P**" represents the number of plausible patches generated by $\texttt{TBar}_p$ (i.e., those that are not found to be correct). "#k", where

[3]Partial fix: a patch makes the buggy program pass a part of previously failed test cases without causing any new failed test cases [38].

```
   public String generateToolTipFragment(String toolTipText) {
-      return " title=\"" + toolTipText
+      return " title=\"" + ImageMapUtilities.htmlEscape(toolTipText)
             + "\" alt=\"\"";
   }
```

Code Change Action:
Replace variable "toolTipText" with a method invocation expression "ImageMapUtilities.htmlEscape(toolTipText)".
Matchable fix pattern: FP9.2.

Figure 2: Patch and code change action of fixing bug C-10.

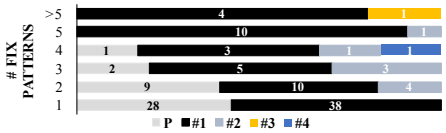


Figure 3: Number of bugs plausibly and correctly fixed by single or multiple fix patterns.

$k \in [1, 4]$, indicates that a bug can be correctly fixed by only k fix patterns (although it may be plausibly fixed by more fix patterns).

Consider for the bottom-most bar in Figure 3: 66 (=28+38) bugs can be plausibly fixed by a single pattern (*Y*-axis value is 1); it turns out that only 38 of them are correctly fixed. Note that several patterns can generate (plausible) patches for a bug, but not all patches are necessarily correct. For example, in the case of the top-most bar in Figure 3, 5 bugs are each plausibly fixed by over 5 fix patterns. However, only 1 bug is correctly fixed by 3 fix patterns.

In summary, 86% ($=\frac{38+10+5+3+10+4}{74+7}$) of correctly fixed bugs (74 fully and 7 partially fixed bugs) are exclusively fixed correctly by single patterns. In other words, generally, several fix patterns can generate patches that can pass all test cases but, in most cases, the bug is correctly fixed by only one pattern. This finding suggests that it is necessary to carefully select an appropriate fix pattern when attempting to fix a bug, in order to avoid plausible patches which may prevent the discovery of correct patches by halting the repair process (given that all tests are passing on the plausible patch).

RQ2: *Some bugs can be plausibly fixed by different fix patterns. However, in most cases, only one fix pattern is adequate for generating a correct patch. This finding suggests a need for new research on fix pattern prioritization.*

Table 6 details which bug is fixed by which fix pattern(s). We note that five fix patterns (i.e., FP3, FP4.3, FP5, FP7.2 and FP11.3) cannot be used to generate a plausible patch for any Defects4J bug. Two fix patterns (i.e., FP9.2 and FP12) lead to plausible patches for some bugs, but none of them is correct. It does not necessarily suggest that the aforementioned fix patterns are useless (or ineffective) in APR. Instead, two reasons can explain their performance:

- The search for donor code may be inefficient for finding relevant ingredients for applying these patterns
- The Defects4J dataset does not contain the types of bugs that can be addressed by these fix patterns.

In addition, twenty (20) fix patterns lead to the generation of correct patches for some bugs. Most of these fix patterns are involved in the generation of plausible patches (which turn out to be incorrect). Interestingly, we found the cases of six (6) fix patterns which can generate several[4] patch candidates, some which being correct and others being only plausible, for the same 10 bugs (as indicated in Table 6 with '◐'). This observation further highlights the importance of selecting a relevant donor code for synthesizing patches: selecting an inappropriate donor code can lead to the generation of a plausible (but incorrect) patch, which will impede the generation of correct patches in a typical repair pipeline.

Aside from fix patterns, fix ingredients collected in donor code are essential to be properly selected to avoid patches that are plausible but may yet be incorrect.

We further inspect properties of fix patterns, such as change actions, granularity, and the number of changed statements in patches. The statistics are shown in Figure 4, highlighting the number of plausible (but incorrect) and correct patches for the different property dimensions through which fix patterns can be categorized.

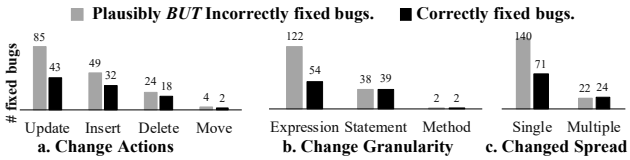


Figure 4: Qualitative statistics of bugs fixed by fix patterns.

More bugs are fixed by *Update* change actions than any by any other actions. Similarly, fix patterns targeting expressions fix more bugs correctly than patterns targeting statements and methods. However, fix patterns mutating whole statements have a higher rate of correct patches among their plausible generated patches. Finally, fix patterns changing only single statements can correctly fix more bugs than those touching multiple statements. Fix patterns targeting multi-statements have however a higher rate of correctness.

RQ3: *There are noticeable differences between successful repair among fix patterns depending on their properties related to implemented change actions, change granularity and change spread.*

4.2 Repair Performance Comparison: TBar vs State-of-the-art APR tools

Our second experiment evaluates TBar in a realistic setting for patch generation, allowing for reliable comparison against the state-of-the-art in the literature. Concretely, we investigate two research questions in Experiment #2.

Research Questions for Experiment #2

RQ4. *What performance can be achieved by* TBar *in a standard and practical repair scenario?*

RQ5. *To what extent are the different fix patterns sensitive to noise in fault localization (i.e., spotting buggy code locations)?*

In this experiment we implement a realistic scenario, using a normal fault localization (i.e., no assumption of perfect localization as for TBar_p) on Defects4J bugs. To enable a fair comparison with performance results recorded in the literature, TBar leverages a standard configuration in the literature [38] with GZoltar [9] and Ochiai [4]. Furthermore, TBar does not utilize any additional technique to improve the accuracy of fault localization, such as crashed stack trace (used by ssFix [74]), predicate switching [80] (used by ACS [76]), or test case purification [79] (used by SimFix [18]).

With respect to the patch generation step, contrary to the experiment with TBar_p where all positions of multi-locations bugs were known (cf. Section 4.1), TBar adapts a "first-generated and first-selected" strategy to progressively apply fix patterns, one at a time, in various suspicious code locations: TBar generates a patch p_i,

[4]Note that, in this experiment TBar_p generates and assesses all possible patch candidates for a given pair "bug location - fix pattern" with varying ingredients.

Table 6: Defects4j bugs fixed by fix patterns.

Bug ID	FP1	FP2					FP3	FP4				FP5	FP6			FP7		FP8			FP9		FP10				FP11			FP12	FP13		FP14	FP15		
		1	2	3	4	5		1	2	3	4		1	2	3	1	2	1	2	3	1	2	1	2	3	4	1	2	3		1	2		1	2	
C-1															❍												●					❍	❍	❍		1/5
C-4		●	❍		●																															2/3
C-7											❍																				◐					1/2
C-8																															●					1/1
C-9															●							❍														1/2
C-11																															●					1/1
C-12								●																												1/1
C-14			●	●																											❍					2/3
C-18								❍							❍												❍				❍				●	1/5
C-19						●																														1/1
C-20																							●													1/1
C-24																															●					1/1
C-25			●																												❍					1/3
C-26		●	❍	●																																2/3
Cl-2		●	❍	❍																																1/3
Cl-4																							●													1/1
Cl-6														❍	❍												❍				❍	❍		●		1/6
Cl-10																							●													1/1
Cl-11													❍		❍												❍				❍				●	1/5
Cl-13																																	●			1/1
Cl-18														●													❍									1/2
Cl-21													❍		❍												❍				❍				●	1/5
Cl-22													❍		❍												❍				❍				●	1/5
Cl-31														●													❍									1/2
Cl-38													❍	❍													●									1/3
Cl-40																					●															1/1
Cl-46																																			●	1/1
Cl-62													❍	❍	❍												◐								❍	1/5
Cl-63													❍	❍	❍												◐								❍	1/5
Cl-70																					●															1/1
Cl-73																											●									1/1
Cl-85																																			●	1/1
Cl-86																					●															1/1
Cl-102								●																									●			2/2
Cl-106																																❍			●	1/2
Cl-115													❍		❍																❍		❍		●	1/5
Cl-126			❍	❍									❍		❍																❍				●	1/6
L-6																															●					1/1
L-7													❍		❍																❍				●	1/4
L-10															◐																				●	2/2
L-15													❍	●	❍												❍								❍	1/5
L-22														❍	❍												◐				❍				❍	1/5
L-24															●																					1/1
L-26																										●										1/1
L-33			●																																	1/1
L-39			●								❍																					❍				1/3
L-47				●																																1/1
L-51									●																											1/1
L-57		❍	❍																											❍	●	●				2/5
L-59																															●					1/1
L-63															❍									❍			❍				❍	❍	❍	●		1/7
M-4			●																																	1/1
M-5																															●	❍				1/2
M-11																		●	●	●	●															4/4
M-15															◐																					1/1
M-22																					●	❍														1/2
M-30																◐																				1/1
M-33															❍							❍									●					1/3
M-34																																●				1/1
M-35								●																												1/1
M-50											❍		❍		❍							❍					❍	❍			❍	❍		●		1/9
M-57																●																				1/1
M-58																									●											1/1
M-59																															●	❍				1/2
M-65																											●									1/1
M-70																										●										1/1
M-75																							●													1/1
M-77											❍																●				❍				●	2/4
M-79																◐																				1/1
M-80																						❍						●			❍	❍				1/4
M-82											❍				❍							❍					●				❍					1/5
M-85											❍		◐		◐							❍					●				❍	❍		❍		3/8
M-89	●																																			1/1
M-98																															●					1/1
Mc-26																					●															1/1
Mc-29		●		●																																2/2
Mc-38				●		●																														2/2
T-3											◐																									1/1
T-7																															●	❍				1/2
T-19															❍												●									1/2
T-26																										●										1/1
# 1	1	6	5	4	1	1	0	3	1	0	1	0	1	3	5	3	0	1	1	1	6	0	3	1	1	3	11	1	0	0	12	2	2	13	2	
# 2	1	7	10	6	1	1	0	4	1	0	14	0	15	12	32	3	0	1	1	1	6	7	4	2	2	3	24	2	0	1	43	19	6	25	4	

∗ ● indicates that the bug is correctly fixed and ❍ indicates that the generated patch is plausible but not correct. ◐means that the fix pattern can generate both correct patch and plausible patch for a bug. ● and ❍ denote that the bug can be partially fixed by the corresponding fix pattern. In the last column, we provide *x*/*y* numbers: *x* is the number of fix patterns that can generate correct patches for a bug, and *y* is the number of fix patterns that can generate plausible patches for a bug. ***Note that***, the bugs that can be plausible but incorrectly fixed by fix patterns are not shown in this table. **# 1**: number of bugs correctly fixed by a fix pattern. **# 2**: number of bugs plausible fixed by a fix pattern.

using a fix pattern that matches a given bug. If p_i passes a subset of previously-failing test cases without failing any previously-passing test case, TBar selects p_i as a plausible patch for the bug. Then, TBar continues to validate another patch p_{i+1} (which can be generated by the same fix pattern on the same code entity with other ingredients, or on another code location). When p_{i+1} passes a subset of test cases as p_i, if p_{i+1} is generated for the same buggy code entity as p_i, p_{i+1} will be abandoned; otherwise, TBar takes p_{i+1} as another plausible patch as well. Through this process, TBar creates a patch set $P = \{p_i, p_{i+1}, ...\}$ of plausible patches. Here, as soon as any patch can pass all the given test cases for a given bug, TBar takes it as a plausible patch for the given bug, which is regarded as a *fully-fixed*

Table 7: Comparing TBar against the state-of-the-art APR tools.

Project	jGenProg	jKali	jMutRepair	HDRepair	Nopol	ACS	ELIXIR	JAID	ssFix	CapGen	SketchFix	FixMiner	LSRepair	SimFix	kPAR	AVATAR	TBar	
																	Fully fixed	Partially fixed
Chart	0/7	0/6	1/4	0/2	1/6	2/2	4/7	2/4	3/7	4/4	6/8	5/8	3/8	4/8	3/10	5/12	**9/14**	0/4
Closure	0/0	0/0	0/0	0/7	0/0	0/0	0/0	5/11	2/11	0/0	3/5	5/5	0/0	6/8	5/9	8/12	**8/12**	1/5
Lang	0/0	0/0	0/1	2/6	3/7	3/4	8/12	1/8	5/12	5/5	3/4	2/3	8/14	**9**/13	1/8	5/11	5/**14**	0/3
Math	5/18	1/14	2/11	4/7	1/21	12/16	12/19	1/8	10/26	12/16	7/8	12/14	7/14	14/26	7/18	6/13	**19/36**	0/4
Mockito	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/0	1/1	0/0	1/2	**2/2**	1/2	0/0
Time	0/2	0/2	0/1	0/1	0/1	1/1	**2**/3	0/0	0/**4**	0/0	0/1	1/1	0/0	1/1	1/2	1/3	1/3	1/2
Total	5/27	1/22	3/17	6/23	5/35	18/23	26/41	9/31	20/60	21/25	19/26	25/31	19/37	34/56	18/49	27/53	**43/81**	2/18
P(%)	18.5	4.5	17.6	26.1	14.3	78.3	63.4	29.0	33.3	**84.0**	73.1	80.6	51.4	60.7	36.7	50.9	53.1	11.1

*"**P**" is the probability of generated plausible patches to be correct. The data of other APR tools are excerpted from the corresponding work. **kPAR** [38] is an open-source implementation of PAR [23].

Table 8: Per-pattern repair performance.

	FP1	FP2					FP3	FP4				FP5	FP6			FP7		FP8			FP9		FP10				FP11			FP12	FP13		FP14	FP15	
		1	2	3	4	5		1	2	3	4		1	2	3	1	2	1	2	3	1	2	1	2	3	4	1	2	3		1	2		1	2
Correct	1	4	2	1	0	1	0	1	0	0	0	0	0	0	3	3	0	0	0	1	2	0	1	1	1	1	7	1	0	0	9	1	0	2	2
Avg position*	(1)	(16)	(1)	(5)	-	(5)	-	(5)	-	-	-	-	-	-	(23)	(16)	-	-	-	(9)	(1)	-	(2)	(62)	(6)	(1)	(12)	(18)	-	-	(5)	(1)	-	(2)	(1)
Plausible (all)	1	7	4	1	0	1	0	3	0	0	0	0	1	0	11	4	0	0	0	1	4	0	2	2	1	1	12	1	0	0	25	4	1	7	5
Avg position*	(1)	(12)†	(191)	(5)	-	(5)	-	(20)	-	-	-	-	(8)	-	(27)†	(15)	-	-	-	(9)	(18)	-	(4)	(49)	(6)	(1)	(15)†	(18)	-	-	(8)†	(20)	(15)	(26)	(16)

*Average position of the exact buggy position in the list of suspicious statements yield by fault localization tool. † The exact buggy positions of some bugs cannot be yield by fault localizaiton tool.

bug, and all $p_i \in P$ will be abandoned. Otherwise, our tool yields P, a set of plausible patches that can each partially fix the given bug.

We run the TBar APR system against the buggy programs of the Defects4J dataset. Table 7 presents the performance of TBar in comparison with recent state-of-the-art APR tools from the literature. TBar can fix 81 bugs with plausible patches, 43 of which are correctly fixed. No other APR tool had reached this number of fixed bugs. Nevertheless, its precision (ratio of correct vs. plausible patches) is lower than some recent tools such as CapGen and SimFix which employs sophisticated techniques to select fix ingredients. Nonetheless, it is noteworthy that, despite using fix patterns catalogued in the literature, we can fix three bugs (namely Cl-86,L-47,M-11) which had never been fixed by any APR system: M-11 is fixed by a pattern found by a standalone fix pattern mining tool [35] but which was not encoded by any APR system yet. Cl-86 and L-47 are fixed by patterns that were not applied to Defects4J.

RQ4: TBar *outperforms all recent state-of-the-art APR tools that were evaluated on the Defects4J dataset. It correctly fixes 43 bugs, while the runner-up (SimFix) is reported to correctly fix 34 bugs.*

It is noteworthy that TBar performs significantly less than TBar_p (43 vs. 74 correctly fixed bugs). This result is in line with a recent study [38], which demonstrated that fault localization imprecision is detrimental to APR repair performance. Table 6 summarizes information about the number of bugs each fix pattern contributed to fixing with TBar_p. While only 4 fix patterns did not lead to the generation of any plausible patch when assuming perfect localization. With TBar, it is the case for 13 fix patterns (see Table 8). This observation further confirms the impact of fault localization noise.

We propose to examine the locations where TBar applied fix patterns to generate plausible but incorrect patches. As shown in Figure 5, TBar has made changes on incorrect positions (i.e., non-buggy locations) for 24 out of the 38 fully-fixed and 15 out of the 16 partially-fixed bugs.

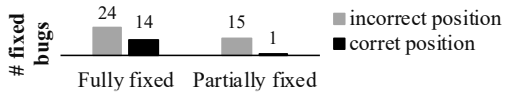
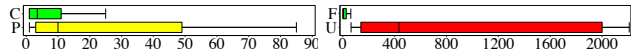


Figure 5: The mutated code positions of plausibly but incorrectly fixed bugs.

Even when TBar applies a fix pattern to the precise buggy location, the generated patch may be incorrect. As shown in Figure 5, 14 patches that fully fix Defects4J bugs mutate the correct locations: in 3 cases, the fix patterns were inappropriate; in 2 other cases, TBar failed to locate relevant donor code; for the remaining, TBar does not support the required fix patterns.

Finally, Figure 6 illustrates the impact of fault localization performance: unfixed bugs (but correctly fixed by TBar_p) are generally more poorly localized than correctly fixed bugs. Similarly, we note that many plausible but incorrect patches are generated for bugs which are not well localized (i.e., several false positive buggy locations are mutated leading to plausible but incorrect patches).



*** X-axis: Bug positions in suspicious list reported by fault localization.**

Figure 6: Distribution of the positions of buggy code locations in fault localization list of suspicious statements. *C* and *P* denote Correctly- and Plausibly- (but incorrectly) fixed bugs, respectively. *F* and *U* denote Fixed and Unfixed bugs.

Average positions bugs (in fault localization suspicious list) are also provided in Table 8. It appears that some fix patterns (e.g., FP2.1, FP6.3, FP10.2) can correctly fix bugs that are poorly localized, showing less sensitivity to fault localization noise than others.

RQ5: *Fault localization noise has a significant impact on the performance of* TBar. *Fix patterns are diversely sensitive to the false positive locations that are recommended as buggy positions.*

5 DISCUSSION

Overall, our investigations reveal that a large catalogue of fix patterns can help improve APR performance. However, at the same time, there are other challenges that must be dealt with: more accurate fault localization, effective search of relevant donor code, fix pattern prioritization. While we will work on some of these research directions in future work, we discuss in this section some threats to validity of the study and practical limitations of TBar.

5.1 Threats to Validity

Threats to external validity include the target language of this study, i.e., Java. Fix patterns studied in this paper only cover the fix patterns targeting at Java program bugs released by the state-of-the-art pattern-based APR systems. However, we believe that most fix patterns presented in this study could be applied to other languages since fix patterns are illustrated as abstract syntax tree level. Another threat to external validity could be the fix pattern diversity. Our study may not consider all available fix patterns so far in the literature. To reduce this threat, we systematically reviewed the

research on pattern-based program repair in the literature. Nevertheless, we acknowledge that integrating more fix patterns may not necessarily lead to increased number of bugs that are correctly fixed. With too many fix patterns, the search space of fix patterns and patch candidates will explode. Eventually, the APR tool will produce a huge number of plausible patches, many of which might be validated before the correct ones [69]. A future research direction could be on the construction and curation of fix patterns database for APR.

Our strategy of fix pattern selection can be a threat to internal validity: it naïvely matches patterns based on the AST context around buggy locations. More advanced strategies would give a higher probability to select appropriate patterns to fix more bugs. Our approach to searching for donor code also carries some threats to validity: `TBar` focuses on the local buggy file, while previous works have shown that the adequate donor code, for some bugs, is available in other files [18, 69]. In future work, we will investigate the search of donor code beyond local files, while using heuristics to cope with the potential search space explosion. Finally, the selected benchmark for evaluation constitutes another threat to external validity for assessment. The performance achieved by `TBar` on Defects4J may not be reached on a bigger, more diverse and more representative dataset. To address this threat, new benchmarks such as Bugs.jar [61] and Bears [46] should be investigated.

5.2 Limitations

`TBar` selects fix patterns in a naïve way, it thus would be necessary to design a sophisticated strategy (such as bug symptom, bug type, or other information from bug reports) for fix pattern selection to reduce the noise from inappropriate fix patterns. Searching donor code for synthesis patches is another limitation of `TBar`, as the correct donor code for fixing some bugs is located in the code files that do not contain the bug [18, 69]. If `TBar` extends the donor code searching to other non-buggy code files, it will cause the search space explosion.

6 RELATED WORK

Fault Localization. In general, most APR pipelines start with fault localization (FL), as shown in Figure 1. Once the buggy position is localized, ARP tools can mutate the buggy code entity to generate patches. To identify defect locations in a program, several automated FL techniques have been proposed [72]: slice-based [47, 71], spectrum-based [6, 57], statistics-based [32, 33], etc.

Spectrum-based FL is widely adopted in APR systems since they identify bug position at the statement level. It relies on the ranking metrics (e.g., Trantula [19], Ochiai [5]) to calculate the suspiciousness of each statement. GZoltar [9] and Ochiai have been widely integrated into APR systems since their effectiveness has been demonstrated in several empirical studies [56, 65, 73, 78]. As reported by Liu et al. [38] and studied in this paper, this FL configuration still has a limitation on localizing bug positions. Therefore, researchers tried to enhance FL techniques with new techniques, such as predicate switching [76, 80] and test case purification [18, 79].

Patch Generation. Another key process of APR pipelines is searching for another shape of a program (i.e., a patch) in the space of all possible programs [30, 43]. If the search space is too small, it might not include the correct patches. [69]. To reduce this threat, a straightforward strategy is to expand the search space, however, which could lead to other two problems: (1) at worst, there still is no correct patch in it; and (2) the expanded search space includes more plausible patches that enlarge the possibility of generating plausible patches before correct ones [34, 69].

To improve repair performance, many APR systems have been explored to address the search space problem. Synthesis-based APR systems [42, 76, 77] explored to limit the search space on conditional bug fixes by synthesizing new conditional expressions with variables identified from the buggy code. Pattern-based APR tools [13, 17, 18, 23, 25, 29, 39–41, 62] are designed to purify the search space by following fix patterns to mutate buggy code entities with retrieved donor code. Other APR pipelines focus on specific search methods for donor code or patch synthesizing strategies, to address the search space problem, such as contract-based [10, 66], symbolic execution based [53], learning based [7, 16, 44, 59, 64, 70], and donor code searching [21, 51] APR tools. Various existing APR tools have achieved promising results on fixing real bugs, but there is still an opportunity to improve the performance; for example, mining more fix patterns, improving pattern selection and donor code retrieving strategy, exploring a new strategy for patch generation, and prioritizing bug positions.

Patch Correctness. The ultimate goal of APR systems is to automatically generate a correct patch that can resolve the program defects. In the beginning, patch correctness is evaluated by passing all test cases [23, 29, 67]. However, these patches could be overfitting [27, 58] and even worse than the bug [63]. Since then, APR systems are evaluated with the precision of generating correct patches [18, 39, 69, 76]. Recently, researchers start to explore automated frameworks that can identify patch correctness for APR systems automatically [28, 75].

7 CONCLUSION

Fix patterns have been studied in various scenarios to understand bug fixes in the wild. They are further implemented in different APR pipelines to generate patches automatically. Although template-based APR tools have achieved promising results, no extensive investigation on the effectiveness fix patterns was conducted. We fill this gap in this work by revisiting the repair performance of fix patterns via a systematic study assessing the effectiveness of a variety of fix patterns summarized from the literature. In particular, we build a straightforward template-based APR tool, `TBar`, which we evaluate on the Defects4J benchmark. On the one hand, assuming a perfect fault localization, `TBar` fixes 74/101 bugs correctly/plausibly. On the other hand, in a normal/practical APR pipeline, `TBar` correctly fixes 43 bugs despite the noise of fault localization false positives. This constitutes a record performance in the literature on Java program repair. We expect `TBar` to be established as the new baseline APR system, leading researchers to propose better techniques for substantial improvement of the state-of-the-art.

ACKNOWLEDGMENTS

This work is supported by the Fonds National de la Recherche (FNR), Luxembourg, through RECOMMEND 15/IS/10449467 and FIXPATTERN C15/IS/9964569.

REFERENCES

[1] Last Accessed: May. 2019. Defecst4J. https://github.com/rjust/defects4j/releases/tag/v1.2.0.

[2] Last Accessed: May. 2019. PAR Fix Templates. https://sites.google.com/site/autofixhkust/home/fix-templates.

[3] Last Accessed: May. 2019. Program Repair. http://program-repair.org.

[4] Rui Abreu, Arjan JC Van Gemund, and Peter Zoeteweij. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*. IEEE, 89–98.

[5] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. 2009. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software* 82, 11 (2009), 1780–1792.

[6] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2009. Spectrum-based multiple fault localization. In *Proceedings of the 24th International Conference on Automated Software Engineering*. IEEE, 88–99.

[7] Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. 2018. Neuro-symbolic program corrector for introductory programming assignments. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 60–70.

[8] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. 2013. Reversible debugging software. *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep* (2013).

[9] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. 2012. Gzoltar: an eclipse plug-in for testing and debugging. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE/ACM, 378–381.

[10] Liushan Chen, Yu Pei, and Carlo A Furia. 2017. Contract-based program repair without the contracts. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 637–647.

[11] Zack Coker and Munawar Hafiz. 2013. Program transformations to fix C integers. In *Proceedings of the 35th IEEE/ACM International Conference on Software Engineering*. IEEE/ACM, 792–801.

[12] Kinga Dobolyi and Westley Weimer. 2008. Changing java's semantics for handling null pointer exceptions. In *Proceedings of the 19th International Symposium on Software Reliability Engineering*. IEEE, 47–56.

[13] Thomas Durieux, Benoit Cornu, Lionel Seinturier, and Martin Monperrus. 2017. Dynamic patch generation for null pointer exceptions using metaprogramming. In *Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 349–358.

[14] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ACM, 313–324.

[15] Ali Ghanbari and Lingming Zhang. 2018. Practical program repair via bytecode mutation. *arXiv preprint arXiv:1807.03512* (2018).

[16] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the 31st AAAI Conference on Artificial Intelligence*. AAAI Press, 1345–1351.

[17] 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*. ACM, 12–23.

[18] 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*. ACM, 298–309.

[19] James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM, 273–282.

[20] René 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 23rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 437–440.

[21] Yalin Ke, Kathryn T Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing programs with semantic code search (t). In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 295–306.

[22] Stephen W Kent. 2008. Dynamic error remediation: A case study with null pointer exceptions. *University of Texas Master's Thesis* (2008).

[23] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *Proceedings of the 35th International Conference on Software Engineering*. IEEE, 802–811.

[24] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2018. FixMiner: Mining Relevant Fix Patterns for Automated Program Repair. *arXiv preprint arXiv:1810.01791* (2018).

[25] 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 11th Joint Meeting on Foundations of Software Engineering*. ACM, 593–604.

[26] Xuan-Bach D Le, Quang Loc Le, David Lo, and Claire Le Goues. 2016. Enhancing automated program repair with deductive verification. In *Proceedings of the 32nd International Conference on Software Maintenance and Evolution*. IEEE, 428–432.

[27] Xuan Bach D Le, Ferdian Thung, David Lo, and Claire Le Goues. 2018. Overfitting in semantics-based automated program repair. *Empirical Software Engineering* (2018), 1–27.

[28] Xuan-Bach D. Le, Lingfeng Bao, David Lo, Xin Xia, and Shanping Li. 2019. On Reliability of Patch Correctness Assessment. In *Proceedings of the 41th International Conference on Software Engineering*.

[29] Xuan-Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering*, Vol. 1. IEEE, 213–224.

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

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

[32] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. 2005. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*. ACM, 15–26.

[33] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and Samuel P Midkiff. 2006. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on Software Engineering* 32, 10 (2006), 831–848.

[34] Kui Liu, Koyuncu Anil, Kisub Kim, Dongsun Kim, and Tegawendé F. Bissyandé. 2018. LSRepair: Live Search of Fix Ingredients for Automated Program Repair. In *Proceedings of the 25th Asia-Pacific Software Engineering Conference*. 658–662.

[35] Kui Liu, Dongsun Kim, Tegawendé F Bissyandé, Shin Yoo, and Yves Le Traon. 2018. Mining fix patterns for findbugs violations. *IEEE Transactions on Software Engineering* (2018).

[36] Kui Liu, Dongsun Kim, Tegawendé François D Assise Bissyande, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. 2019. Learning to Sport and Refactor Inconsistent Method Names. In *Proceedings of the 41st ACM/IEEE International Conference on Software Engineering*. IEEE.

[37] Kui Liu, Dongsun Kim, Anil Koyuncu, Li Li, Tegawendé F Bissyandé, and Yves Le Traon. 2018. A closer look at real-world patches. In *Proceedings of the 34th International Conference on Software Maintenance and Evolution*. IEEE, 275–286.

[38] Kui Liu, Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems. In *Proceedings of the 12th IEEE International Conference on Software Testing, Verification and Validation*. IEEE.

[39] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. AVATAR : Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE.

[40] Xuliang Liu and Hao Zhong. 2018. Mining stackoverflow for program repair. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 118–129.

[41] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic inference of code transforms for patch generation. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, 727–739.

[42] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, 166–178.

[43] Fan Long and Martin Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 702–713.

[44] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 298–312.

[45] Fan Long, Stelios Sidiroglou-Douskos, and Martin Rinard. 2014. Automatic runtime error repair and containment via recovery shepherding. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, Vol. 49. ACM, 227–238.

[46] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. 2019. Bears: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 468–478.

[47] Xiaoguang Mao, Yan Lei, Ziying Dai, Yuhua Qi, and Chengsong Wang. 2014. Slice-based statistical fault localization. *Journal of Systems and Software* 89 (2014), 51–62.

[48] Matias Martinez and Martin Monperrus. 2015. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering* 20, 1 (2015), 176–205.

[49] Matias Martinez and Martin Monperrus. 2016. Astor: A program repair library for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 441–444.

[50] Matias Martinez and Martin Monperrus. 2018. Ultra-Large Repair Search Space with Automatically Mined Templates: The Cardumen Mode of Astor. In *Proceedings of the International Symposium on Search Based Software Engineering*. Springer, 65–86.

[51] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 448–458.

[52] Martin Monperrus. 2018. Automatic software repair: a bibliography. *Comput. Surveys* 51, 1 (2018), 17:1–17:24.

[53] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: program repair via semantic analysis. In *Proceedings of the 35th International Conference on Software Engineering*. IEEE, 772–781.

[54] NIST. Last Accessed: Jan. 2019.. Software Errors Cost U.S. Economy $59.5 Billion Annually. http://www.abeacha.com/NIST_press_release_bugs_cost.htm.

[55] Kai Pan, Sunghun Kim, and E James Whitehead. 2009. Toward an understanding of bug fix patterns. *Empirical Software Engineering* 14, 3 (2009), 286–315.

[56] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE/ACM, 609–620.

[57] Alexandre Perez, Rui Abreu, and Arie van Deursen. 2017. A test-suite diagnosability metric for spectrum-based fault localization approaches. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE/ACM, 654–664.

[58] 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 24th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 24–36.

[59] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering*. IEEE/ACM, 404–415.

[60] Reudismam Rolim, Gustavo Soares, Rohit Gheyi, and Loris D'Antoni. 2018. Learning Quick Fixes from Code Repositories. *arXiv preprint arXiv:1803.03806* (2018).

[61] Ripon Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul Prasad. 2018. Bugs.jar: a large-scale, diverse dataset of real-world java bugs. In *Proceedings of the IEEE/ACM 15th International Conference on Mining Software Repositories*. IEEE, 10–13.

[62] 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*. IEEE, 648–659.

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

[64] Mauricio Soto and Claire Le Goues. 2018. Using a probabilistic model to predict bug fixes. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 221–231.

[65] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. 2013. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 314–324.

[66] Yi Wei, Yu Pei, Carlo A Furia, Lucas S Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. 2010. Automated fixing of programs with contracts. In *Proceedings of the 19th international symposium on Software testing and analysis*. ACM, 61–72.

[67] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE, 364–374.

[68] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2017. An empirical analysis of the influence of fault space on search-based automated program repair. *arXiv preprint arXiv:1707.05172* (2017).

[69] 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 IEEE/ACM International Conference on Software Engineering*. IEEE/ACM, 1–11.

[70] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. 2019. Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE.

[71] W Eric Wong, Vidroha Debroy, and Byoungju Choi. 2010. A family of code coverage-based heuristics for effective fault localization. *Journal of Systems and Software* 83, 2 (2010), 188–208.

[72] 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.

[73] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. 2013. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software Engineering and Methodology* 22, 4 (2013), 31:1–31:40.

[74] Qi Xin and Steven P Reiss. 2017. Leveraging syntax-related code for automated program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE/ACM, 660–670.

[75] 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*. ACM, 789–799.

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

[77] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering* 43, 1 (2017), 34–55.

[78] Jifeng Xuan and Martin Monperrus. 2014. Learning to combine multiple ranking metrics for fault localization. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution*. IEEE, 191–200.

[79] Jifeng Xuan and Martin Monperrus. 2014. Test case purification for improving fault localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 52–63.

[80] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Locating faults through automated predicate switching. In *Proceedings of the 28th International Conference on Software Engineering*. ACM, 272–281.

[81] Hao Zhong and Zhendong Su. 2015. An empirical study on real bug fixes. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering-Volume 1*. IEEE/ACM, 913–923.