SCIENCE CHINA
Information Sciences

• **RESEARCH PAPER** •

A manual inspection of Defects4J bugs and its implications for automatic program repair

Jiajun JIANG[1,2], Yingfei XIONG[1,2*] & Xin XIA[3]

[1]*Key Laboratory of High Confidence Software Technologies, MoE Institute of Software;*
[2]*Department of Computer Science and Technology, EECS, Peking University, Beijing* 100871*, China;*
[3]*The Faculty of Information Technology, Monash University, Melbourne* 3800*, Australia*

Abstract Automatic program repair techniques, which target to generate correct patches for real-world defects automatically, have gained a lot of attention in the last decade. Many different techniques and tools have been proposed and developed. However, even the most sophisticated automatic program repair techniques can only repair a small portion of defects while producing a large number of incorrect patches. A possible reason for the low performance is the test suites of real-world programs are usually too weak to guarantee the behavior of a program. To understand to what extent defects can be fixed with exiting test suites, we manually analyzed 50 real-world defects from Defects4J, where a large portion (i.e., 82%) of them were correctly fixed. This result suggests that there is much room for the current automatic program repair techniques to improve. Furthermore, we summarized seven fault localization and seven patch generation strategies that are useful in localizing and fixing these defects, and compared those strategies with current techniques. The results indicate potential directions to improve automatic program repair in the future.

Keywords Automatic defect repair, Fault localization, Manual repair, Software maintenance, Case study

Citation Jiang J J, Xiong Y F, Xia X. A manual inspection of Defects4J bugs and its implications for automatic program repair. Sci China Inf Sci, for review

1 Introduction

Automatic program repair (APR) techniques, which automatically generate patches for defects in programs and target to software automation [1], have gained a lot of attention in the last decade. A typical automatic program repair technique [2–5] takes a program and a set of tests as input, where at least one test is failed by the program, and generates a patch that will fix the defect. Different techniques and tools have been proposed. These tools generated a patch through techniques such as directed random search [2, 3], templates [6], component-based program synthesis [5, 7, 8], program transformation from examples [9–11] and machine learning [12], incorporated fault localization approaches such as spectrum-based fault localization [13,14], predicate switching [15], and angelic debugging [16], and utilized information such as testing results [2,17], existing patches [6, 9, 12], invariants [18], existing source code [19, 20], bug report text [21], and comments [4].

Despite these efforts, in practice even the most sophisticated automatic program repair techniques can only repair a small portion of defects while producing a large number of incorrect patches. For example, Prophet [12] and Angelix [5], two approaches on the C language, can only fix 14.3% and 12.2% of the defects on GenProg benchmark [22], while producing incorrect patches for other 22.8% and 22.0%

* Corresponding author (email: xiongyf@pku.edu.cn)

defects, respectively. The newest approach on Java, SimFix [20], can only fix 9.5% defects on Defects4J benchmark [23] while producing incorrect patches to other 6.2% defects.

An often attributed reason for this low performance, especially the large number of incorrect patches, is that the test suites of real world programs are usually weak. As studied by Le Goues et al. [24], and Long and Rinard [12], test suites in real world programs are often weak, and in a space of patches that pass all tests, there will be much more incorrect patches than correct ones. Moreover, Martinez et al. [25] in a large experiment on Defects4J benchmark found that the test suites even cannot guarantee the functionality completeness of the program under testing, i.e., when some functional code was deleted, the programs can still correctly pass the test suites. That is also why, by enhancing Defects4J's test suites, many incorrect patches can be successfully ruled out [26]. As a matter of fact, existing automatic program repair techniques usually rely on existing test suites, which serves as incomplete specifications of programs under repair. Therefore, it is very difficult for them to distinguish incorrect and correct patches. Since the performance of current repair techniques are still limited, it naturally raises a question: is it possible to repair a large portion of defects with existing test suites? This question is important because, if most of the defects cannot be fixed, we may change the problem settings of automatic program repair, e.g., asking the user to provide formal specifications of the programs. On the contrary, if most defects can be fixed, we can focus on improving the current techniques.

To answer this question, we manually analyzed 50 defects randomly selected from Defects4J [23], a widely-used benchmark of real-world defects in Java programs, to see how much possible these defects can be fixed. In our analysis, a defect was considered as repairable within a given time frame if and only if (1) we could identify a possible root cause[1)] of the defect, (2) we could generate a patch that tackles the root cause and passes all the tests, and (3) the patch is semantically equivalent to the developer patch.

This study could help us to understand the potential of automatic program repair and to improve current techniques. If a defect is considered as repairable in our analysis, there exists at least a manual process to obtain the patch for the defect. By decomposing and automating the manual process, we can potentially obtain an automatic method to repair the defect. Furthermore, if we found many more defects can be fixed than current state-of-the-art approaches, it indicates that current automatic program repair techniques have big potential to be improved.

During the analysis of those defects, we focus on four research questions, and obtained the corresponding results as listed below:

- **RQ1: How many of those defects can be fixed under existing test suite?** In our analysis, 41 (82.0%) of the 50 defects were correctly fixed, while 6 (12.0%) defects were incorrectly fixed because the test suite failed to provide sufficient specifications. Moreover, we failed to generate valid patches for 3 (6%) defects, which required domain-knowledge that was difficult to be obtained from the program and the tests. Though these numbers of fixed defects in the paper comes from one manual analysis and may be not generalizable, they provide insights that current APR techniques potentially have rooms for improvement.
- **RQ2: How those defects were located and what are the implications to future studies?** After decomposing the manual analysis process, we summarized seven fault localization strategies that were applied in our manual analysis, along which we compared the most related fault localization techniques with each strategy and identified the concrete points that current techniques could be improved.
- **RQ3: How those patches were generated and what are the implications to future studies?** Similarly, we summarized seven patch generation strategies from the manual analysis and compared them with related program repair techniques, and proposed implications for future study.
- **RQ4: What are the inspirations from the manual analysis?** According to the analysis, we found that though many strategies have already been explored by current techniques, they still have a lot of room to improve. Moreover, some strategies may inform new techniques.

To conclude, the main contributions of this paper are a set of fault localization and patch generation strategies learned from the manual analysis, which provide concrete directions for future research.

1) Please note that the root cause of a defect may not be the location of code to be changed, but explains the reason of a program failure.

2 Background and Related Work

2.1 Automatic Program Defect Repair

As mentioned in the introduction, in a typical defect repair setting the repair technique takes as input a program and a set of tests, where the program fails at least one test, and produces as output a patch that is expected to repair the defect when applied to the program. Since tests are used as primary tool to guarantee the correctness of the patches, we call this setting as *test-based program repair*.

A key issue to evaluate the performance of repair tools is how to determine the correctness of the generated patches. In the early studies [2, 6] of automatic repair, a patch is usually considered correct if the patched program passes all the tests. In recent studies [3–5, 12, 19, 20, 27, 28], a patch is usually considered as correct if it is semantically identical to the patch produced by human. Note that neither approach can produce an ideal measurement of correctness: the former may overstate the number of correct patches (because the test suites may be too weak to guarantee correctness) while the latter may understate the number of correct patches (because a defect may be repaired in different ways). However, as studied by Qi et al. [24], the former approach is very imprecise for real world programs because the test suites are usually weak. Similarly, Smith et al. [29] studied that inadequate test suite would lead to over fitting patches and suggested that repair techniques must go beyond testing to characterize functional correctness of patches. As a result, in this paper we take the latter approach, determining the correctness by the equivalence with human patches previously generated by the developers of the programs.

Many defect repair approaches follow a "generate-and-validate" approach, i.e., these approaches first try to locate a likely patch in a large patch space, and then validate the patch using all the tests. There are two main challenges in the repair process. The first is to ensure the correctness of the generated patches. As mentioned above, the tests in the real world programs are often not enough to guarantee the correctness of the generated patches. The second is on the generation of correct patches to a large number of defects. Since the patches need to be validated against all tests, the number of generated patches cannot be large. In order to locate a small number of likely patches from the patch space, current approaches cannot support a large patch space. As studied by Long and Rinard [30] and Zhong and Su [31], most defects cannot be fixed by the patch space considered in current approaches.

For example, to reduce the search space, some techniques are proposed to follow predefined templates for patch generation, which are similar to the strategies proposed in this paper. Kim et al. [6] and Tan et al. [32] defined a set of repair patterns or anti-patterns respectively to guide patch generation. Similarly, Long et al. [3] and Saha et al. [28] proposed a set of program transformation schemas to constrain the search space of patch generation. However, compared with the strategies derived from our analysis, the templates used in these approaches are mainly syntactic templates derived from the changes, while our strategies try to reason why the program failed from a developer point of view and connect more on the process of how the patches can be deduced.

There are also defect repair approaches that use a different problem setting. For example, some approaches assume that there exists a full specification of the program [33, 34], and some approaches consider a concrete class of defects such as memory leaks [35], deadlocks [36] and build failures [37]. These different problem settings are not the focus of our paper.

2.2 Empirical Studies on Defect Repair

There exist several empirical studies on defect repair. Zhong and Su [31] studied real bug fixes through analyzing commits of five open source projects. They analyzed distributions of fault locations and modified files. To investigate the complexity of fixing bugs, they analyzed data dependence among faulty lines. More concretely, they analyzed operations of bug fixes and frequency of them related to APIs. As another study, Martinez and Monperrus [38] studied the distribution of real bug fixes by analyzing a large number of bug fix transactions in software repositories. To better understand the nature of bug fixes, they classified those bug fixes with different classification models. Besides, Soto et al. [39] analyzed a great deal of bug-fixing commits in Java projects aiming to provide a guidance for future APR approaches. In

contrast to our study, their studies focus on the distribution of characteristics about defects and patches but not how these defects were fixed, it is difficult to derive conclusions on the repairability of the defects.

In addition, Yang et al. [40] proposed to filter out overfitting patches by enhancing existing test cases, which cannot tell how much is the possibility to repair existing bugs under given test cases. On the contrary, in our empirical study, we not only analyze the possibility of defects to be repaired, but also identify several concrete directions to improve existing techniques, which are orthogonal to their work. Similarly, several previous studies [41, 42] also revealed that better test suite is helpful to more accurate fault localization results. Yang et al. [43] studied the difference of repair results under two statement selecting strategies for statement modification, i.e., suspiciousness-first algorithm (SFA) based on the suspiciousness of statements and rank-first algorithm (RFA) relying on the rank of statements. Their study is similar to ours that proposes implications to guide future studies but from different perspectives.

There exist other human involved studies. Tao et al. [44] conducted a study that repair real defects manually under the help of APR techniques. It is different from ours because they focus on how the generated patches help the developers rather than how patches can be derived. Several researchers studied the debugging process of human developers. Lawrance et al. [45] studied how human developers navigate through the debugging process and created a model for predicting the navigation process. LaToza and Mayers [46] studied the questions developers asked during the debugging. Murphy-Hill et al. [47] studied the factors developers considered during the debugging. Different from these studies, our study focus on analyzing the repairability of defects rather than understanding how human developers behave.

3 Dataset and Environment

We conducted our case study on Defects4J [23] (v1.0), which consists of 357 defects from five open source projects, JFree**Chart**, **Closure** compiler, Apache commons-**Lang**, Apache commons-**Math** and Joda-**Time**. Since the whole Defects4J is too large for manual analysis, we randomly selected ten defects from each project, and thus have a dataset of 50 defects.

To understand how many defects can be repaired, we analyzed each defect in the dataset to determine whether we can locate a correct patch for the defect. Our manual analysis is performed under the following three environment settings like many existing automatic program repair techniques [2, 4, 5, 8, 20, 48].

- We do not have prior knowledge of the programs under analysis. In other words, we do not know the complete specifications of the programs except the test suites.
- We only rely on the source code of the program to generate the patch, including both implementation code, and testing code. In particular, we have no access to the patch of the defect provided by the developer in the benchmark.
- During the analysis, we can access the Javadoc and comments in the source code but no extra documents were provided. Besides, we can access the Internet, but cannot search the bug directly.

In this way, we put ourselves into the same environment setting as most test-based program repair techniques, which mainly depend on the source code and test suite. If we obtain the correct patch for a defect under this setting, it indicates potential to fix the defect automatically by decomposing and automating the manual repair process.

More concretely, our analysis would classify the defects into *repairable* and *difficult to repair*, and the classification is based on the following steps for each defect. The first author of the paper, who is a Ph.D. student with four-year's experience in Java programming, performed the manual analysis.

- Under the above manual analysis settings, we try to locate a possible root cause of the defect.
- We generate a candidate patch for the defect, and run all tests to validate the patch. If the patch does not pass all the tests, we restart from the first step.
- If the patch passes all tests, we further compare it with the developer's patch. If the two patches are semantically equivalent, we regard the patch as correct and the defect as *repairable*, otherwise we regard the patch as incorrect and the defect as *difficult to repair*. More concretely, We determine semantic equivalence by considering all possible system states when entering the patched method and checking if

the system state is equivalent when the method returns. For all possible system states we mean the states that can be reached through any public method with any input allowed by the method signature.

• If we cannot obtain a patch that passes all tests within 5 hours, we stop and consider the defect *difficult to repair*.

During the analysis, the details are recorded and summarized as strategies. Then, the other two authors further check the validity of the strategies and refine them based on the analysis records until reaching an agreement. The detailed analysis is available at https://sites.google.com/site/d4jinpection.

4 Methodology for Manual Analysis

Following the usual design of automatic repair, we view a repair process as two phases: fault localization and patch generation. The former is to identify the root cause of the failure, based on which the latter is to generate a patch that can fix the failure. We will decompose the repair process from the two phases.

In an abstract view, both the phases can be seen as locating a solution in a (possibly finite or infinite) space of solutions. In fault localization, the space is the power set of all statements and we try to locate one statement or a few statements that is the root cause of the defect. In patch generation, the space is all possible patches and we try to generate a patch that can fix the current defect.

To understand how the defects can be repaired, we need to decompose the fault localization and the patch generation process used in our analysis. To provide useful guidance for automatic repair techniques, we assume a model with strategies and try to derive strategies from our analysis. Concretely, both the fault localization and patch generation processes can be viewed as search and rank procedures. In a more fine-grained level, the analyzing process is a series of attempts to apply different strategies to the current problem. A strategy, when applied, either adds (filters out) or increases (decreases) the rank of some solutions in the search space. A strategy is usually associated with a precondition, which must be satisfied before applying the solution. During the analysis, we always need to simultaneously consider a large set of strategies and determine which of them can be applied. For example, a simple strategy of fault localization is to exclude all statements that are not executed during the failed test execution. This is equivalent to filter out solutions containing these statements. This strategy can be applied only when there is an executable test that is failed by the program (this precondition is always satisfied under the setting of automatic defect repair). As another example strategy, if we observe a rare statement that breaks usual programming practice, such as `if(a=1)` rather than `if(a==1)`, we can increase the ranking of this statement among all candidate statements during fault localization.

Under this view, to understand how defects could be fixed under the given test suite, we try to decompose the repair processes into a set of strategies. Totally, we identified seven strategies for fault localization and seven strategies for patch generation. A further observation on the strategies is that the distinction between fault localization and patch generation is not always clear. A strategy can contribute to both two phases. For example, the aforementioned strategy on programming practice not only provide guidance on fault localization, but also gives us a solution in patch generation, i.e., change `a=1` to `a==1`. Therefore, if a strategy contributes to both sub-processes, we classify it based on its main contribution.

5 Results

In this section, we first will present the overall result of the analysis and compare it with existing automatic program repair techniques. Then, we will introduce those strategies used in the analysis for fault localization and patch generation, respectively, along which we will compare the most related techniques with each strategy in our study and identify concrete points to improve current techniques.

5.1 RQ1: Manual Analysis Result of Defects

Among the 50 defects we analyzed, we correctly repaired 41 (82%) defects, which are regarded as *repairable*, while failed to repair the other 9 (18%) defects that are regarded as *difficult to repair*. Table 1

shows the detailed data per project as well as the comparison with a set of existing program repair approaches. As we can see from the table, the performance of existing program repair approaches can only repair a very small portion of repairable defects, indicating a large room for improvement.

Table 1 Compare our analysis result with existing automatic repair techniques on our dataset.

	jGenProg	jKali	Nopol	ACS	HDR	ssFix	ELIXIR	JAID	CapGen	SimFix	Munal
Chart	0/4	0/2	1/1	0/0	2/-	1/3	3/1	0/2	2/0	3/0	7/3
Closure	-/-	-/-	-/-	-/-	1/-	0/1	-/-	0/1	-/-	0/0	8/1
Lang	0/0	0/0	0/0	1/0	2/-	1/1	1/0	0/0	1/0	0/1	10/0
Math	2/1	1/1	0/0	3/0	1/-	0/4	1/1	1/0	1/0	1/3	7/2
Time	0/1	0/1	0/0	0/0	0/-	0/1	1/0	0/0	0/0	1/0	9/0
Total	2/6	1/4	1/1	4/0	6/-	2/10	6/2	1/3	4/0	5/4	41/6

The results of the first three approaches come from [25] and the results of others come from the corresponding research papers: ACS [4], HDR [49], ssFix [50], ELIXIR [28], JAID [27], CapGen [19], SimFix [20]. In the table, X/Y denotes that X defects are correctly repaired while Y defects are wrongly repaired. "-" denotes the missing data.

Finding 1. *A large portion of (i.e., 82%) defects were correctly fixed in our manual analysis, indicating that most of the defects have a great potential to be fixed under existing test suite.*

Among the 9 defects regarded as *difficult to repair*, we generated incorrect patches for 6 defects while failed to generate valid patches that can pass all the tests for 3 defects. We further investigated the 6 incorrect patches, and found out that in all those cases, the tests in the program do not provide enough information to reveal the full scope of the defect. Without knowing the precise specifications of the programs, we would generate incomplete patches based on only the test suite. For example, a defect from *Chart-10* is related to `String` transformation. According to the failing test, character "\"" in the input should be replaced with "`&quot;`". We generated a patch to handle this and it successfully passed all the tests. However, in fact there are many other characters should be replaced, which are not covered by existing test suite. As a result, we generated an overfitting patch to this defect.

We further investigated why we could not generate a patch for the three defects in our manual analysis. The reason is similar: these defects require domain knowledge either specific to the project or specific to a particular domain, where a developer may not be familiar with. Among the three defects, *Math-2* is a defect about floating-point precision, where the standard patch changes an inaccurate expression into a mathematically equivalent but more accurate expression. Fixing the defect requires the knowledge of accurate arithmetic. *Closure-4* and *Time-6* are related to the uses of the methods and classes in the project, where the buggy code does not correctly interpret the semantics of called methods or the preconditions of called methods are not properly satisfied. Fixing the defect requires the knowledge of the project, especially the preconditions and semantics of each method. Lacking the domain knowledge, it is difficult for an average developer to locate the root cause of the three defects.

5.2 RQ2: Fault Localization Strategies and Implications

In this section, we present the strategies used for fault localization in the manual analysis process, along which we compare the related existing techniques with each strategy and propose implications to inform future studies. The details of the strategies are listed in Table 2. The first column lists the strategy names, the second column briefly describes how these strategies work, and the last column lists the defects to which each strategy was applied in our manual analysis.

Strategy 1: *Excluding unexecuted statements.* This strategy is very simple: when a statement is not executed in the failed execution, it cannot be the root cause. This strategy is implicitly applied when we try to find the root cause of a defect. Actually, this strategy can be applied to almost all defects.

Related: This strategy is almost adopted in any fault localization approaches. Some approaches [51,52] can further exclude statements not related to the failure even executed in failed executions.

Strategy 2: *Excluding unlikely candidates.* Given a list of possible candidates of root causes, we could

Table 2 Strategies applied to locate faulty method in our analysis.

Strategy	Description	Defects*
Excluding unexecuted statements	Exclude those statements not executed by failing test.	All defects.
Excluding unlikely candidates	Filter all non-related candidates based on their functionalities and complexities.	*L-1,2,4,7,9; M-5,10; Ch-2; Cl-9; T-1,4,10*
Stack trace analysis	Locate faulty locations based on the stack trace information thrown by failing test cases.	*L-1,5,6; M-3,4,8; Ch-4,9; Cl-2; T-2,5,7,8,10*
Locating undesirable value changes	Locate those statements that change the input values to the final faulty values of failing test cases.	*L-8; Cl-1,3,5,7,8,10; T-3,9*
Checking programming practice	Identify those code that obviously violate some programming principles based on previous programming experience.	*L-6,8; Ch-1,7,8*
Predicate switching	Inverse condition statements to get expected output, the inversed condition statement is the error location.	*L-3; Ch-1,9; Cl-10*
Program understanding	Understand the logic of faulty program and the functionalities of relevant objects and methods.	*L-10; M-6,9; Ch-3; Cl-9; T-3,9*

*L, M, Ch, Cl and T denote Lang, Math, Chart, Closure and Time project, respectively.

examine them one by one, and exclude those that are unlikely to contain defects. Though technically this strategy can be applied to different granularities, applying it on the method level was effective in our analysis. That is, given a list of methods invoked during the failed test execution, we will examine them one by one and exclude unlikely ones. We found that the following two criteria are effective.

- When a method is a library function or the test itself, it is unlikely to contain defect.
- In Java, because the lack of default parameter or the use of polymorphism, it is often the case that a method is just a wrapper of another, and the purpose is only to pass a default argument or to adapt to an interface. When a method is a simple wrapper method, this method is unlikely to contain defect.

Note that technically the methods excluded by this strategy still have the possibility to contain defects, but their probability is significantly smaller than others.

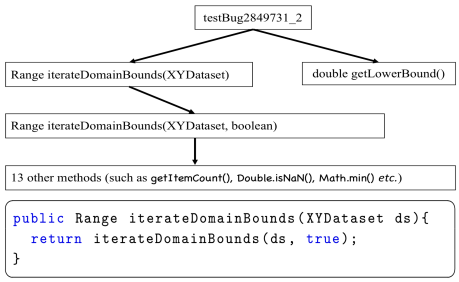

Figure 1 The call graph of *Chart-2*

This was a very effective strategy in our analysis, as we could locate the faulty method using only this strategy and strategy 1. For example, Figure 1 shows the call graph of defect *Chart-2*. After excluding those library methods (e.g., `Double.isNaN`) and simple wrapper methods (e.g., `iterateDomainBounds(XYDataset)`), the only remaining method is `iterateDomainBounds(XYDataset,boolean)`, which turns out to be the faulty method. Apparently, this process need not know the specifications about the program and even need not understand the full functionality of the relevant methods.

Related: The most related approach is fault prediction [53, 54], which predicts the probabilities of different software components to contain defects based on features of the software components.

Improve: *Incorporate richer dynamic information of test failure.* Current fault prediction techniques judge whether a given method is correct or not mostly based on the characteristics of the program itself but do not consider the features of failing test cases. Take *Chart-2* as an example, there are only several methods in the execution of failing test case, which greatly helps to eliminate some candidate locations.

Strategy 3: *Stack trace analysis.* When an uncaught exception is triggered, the program crashes and the stack trace information is printed. A stack trace lists a sequence of locations in the program where a method is called but is not returned before the point of crash. Usually the root cause of the fault is close to the locations listed in the stack trace. That is, the ranks of statements near the locations in the stack trace will be increased. In our manual analysis, 15 defects were located with the contribution of this strategy. By further combining with other strategies, we can often locate the root cause of the defect.

For example, Figure 2 is a stack trace screenshot of *Lang-1*, which throws a `NumberFormatException`. The stack trace lists seven candidate faulty locations. Then we can filter the locations based on strategy 2. Among them, all the first five locations (APIs and wrapper) and the last location (test method) are filtered out. The only possible location is the sixth.

Related: It has been adopted by many fault localization approaches. For example, Wu et al. [55] propose a fault localization approach mainly based on stack trace information. Wong et al. [56] propose to combine stack trace analysis with bug reports to enhance the accuracy of fault localization, while Zhong and Mei [57] proposed MiMo that mines exception-related fix patterns from open-source projects to repair new defects, which improved the performance of program repair.

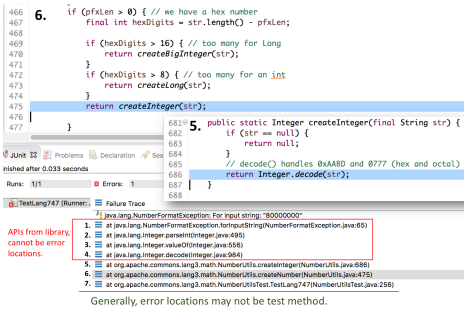

Figure 2 Stack trace of defect *Lang-1*. Lines 469 and 472 are real faulty conditions.

Strategy 4: *Locating undesirable value changes.* A failed test execution produces an output that is different from the desired output. However, sometimes the desired output has already been constructed during the test execution, but the execution of some statements, S, turn it into an undesirable one. In such cases, S or the statements that S control dependent on are likely to be faulty. Note that the latter should be included because they are the reason why S is executed.

In our analysis, those cases frequently occur when testing the optimization component in the Closure project. In a typical such defect, the optimizer changes the input program string into another one that is not semantically-equivalent to the original program. In such cases, the statements that make such an undesirable change will be considered with a high ranking. For an instance, the method call `removeChild` in *Closure-1* wrongly deleted the argument `a` in `window.f=function(a){}`, so either this method or the statements leading to the call of this method might be faulty.

Related: Within our knowledge, this strategy is not directly adopted by existing fault localization approaches. A loosely related approach, delta debugging proposed by Cleve and Zeller [58], locates the transitions that cause the fault. However, delta debugging requires (1) a mechanism to determine the test result and (2) a comparable passed test, which do not apply to the bugs solved by this strategy in our analysis. Another related approach mines program invariants to assist fault localization [59]. This is different with ours as it depends on multiple successful executions while we do not.

Improve: *Correctly identify undesirable value changes.* To overcome the problem, we need to introduce a new technique that could identify undesirable changes in a test execution. A possible way is to define a partial order between states to measure how close to the desirable state a state is, where a standard test execution should only make the state more close to the desirable state rather than make it further.

Strategy 5: *Checking programming practice.* Though in principle the language construct can be combined in any way to form a program, in practice people would only use a small subset of combinations. Basically, a programming practice defines a constraint on the combination, and a piece of code violating the constraint is likely to be faulty. A typical practice, as mentioned before, is that assignment is unlikely to be used in an "if" condition, and thus statement like `if(a=0)` is likely to be faulty[2)].

As another example, the right piece of code comes from *Lang-6*. In this piece of code, the `for` loop iteratively accesses the elements in the sequence with an unbounded variable `pos`. This piece of code violates common programming practice and is likely to be faulty. We found that the violation of programming practice is usually an indication of fault and is useful in fault localization.

```
1 for(int pt=0;pt<consumed;pt++){
2  pos+=Character.charCount(
3      Character.codePointAt(input,pos));
4 }
```

Related: Static bug detection, such as FindBugs [60], checks bad programming practice in the code to decide potential bugs. However, the templates defined in FindBugs mostly do not depend on runtime information. For example, it is hard to determine the faulty code of *Lang-6* based on the static features of the program, the exception caused by variable `pos` also helps greatly.

Improve: *Incorporate dynamic information of test failure.* A typical static bug detection approach

2) Please note that this code convention is useful for C but not on Java, as `if(a=0)` will cause type error in Java. We cite this example just for illustration, and this is not a convention we discovered.

simply considers the static information, which is not sufficient. For instance, as explained above, without the `IndexOutOfBoundException` caused by variable `pos`, it is hard to determine the faulty code. On the contrary, since `pt` is restricted by the length of the input string, if we replace variable `pos` with `pt`, the exception can be avoided. Therefore, correctly checking such programming practice not only needs to know the common practice patterns but also needs to combine the failure information.

Strategy 6: *Predicate switching.* This strategy is very similar to the automatic fault localization technique with the same name [15]. In some cases, if we inverse the result of an "if" condition and force the execution to switch to the other branch, the failed test could pass, where we may consider the "if" condition may be faulty and increase its rank in candidate locations.

```
1 try{
2   Float f=createFloat(str);
3   if(...)  return f;
4 }catch(NumberFormatException e) {}
5   try {
6     Double d = createDouble(str);
7   if(...)  return d;
8 }catch(NumberFormatException e) {}
```

For example, the failed test in *Lang-3* expects a `Double` number when the input is `3.40282354e+38`, whereas a `Float` number was returned. Assuming the value of the condition at line 3 to be `false`, the desired `Double` number will be possibly returned at line 7. Therefore, we can rank the first `if` condition higher.

Related: As discussed before, this strategy is very similar to the predicate switching approach proposed by Zhang et al. [15]. In fact, automatic predicate switching is even more powerful than that used in our analysis because of computer's superb computation ability, and has been employed by many automatic program repair techniques [3–5, 17].

Strategy 7: *Program understanding.* The strategies we have seen so far can be applied without a full understanding and specifications of the program, and many faults can be located by only using these strategies. However, not all faults can be found only relying on them and a certain amount of program understanding is required.

Program understanding is a complex process and here we try to describe it in terms of the general logic reasoning process. Given a faulty program, we try to infer likely constraints on program behavior from different sources, and check consistency between them. If constraints inferred from different sources are inconsistent, the related source code is likely to be faulty. Otherwise, the related source code is unlikely to be faulty. Typical sources include the following.

- Implementation Code. By interpreting the semantics of the source code, we can infer constraints on how the source transforms one state into another state.
- Test Executions. Basically, each test gives a constraint on the desired output for each test input.
- Identifier Names. We often try to infer likely constraint from the names of the identifiers. For example, a method named "remove" should reduce the number of items in some container. A variable named "max" should contain the maximum element in some container.
- Comments. Sometimes the comments describe the intended behavior of a piece of code, and constraints could be inferred from the comments.

To understand how this strategy works, let us consider the defect *Closure-1* which we have been introduced in strategy 4. Using strategy 4, we can isolate the defect to method `removeChild` and its callers, and we know the removal is undesired. However, from the name of `removeChild`, we can infer a constraint that this method should remove an item. Since this semantics is consistent with its implementation code, we know the removal within this method is desired. Therefore, the fault should be in the methods calling `removeChild`. In other words, `removeChild` should not be called.

5.3 RQ3 : Patch Generation Strategies and Implications

In this section, we present the strategies used for patch generation in the manual analysis. Table 3 shows the seven strategies we summarized on patch generation. Similarly to Table 2, the first column is the identification for each strategy, the second column briefly describes the strategy, and the last column lists the defects to which the strategy was applied.

Strategy 8: *Add NullPointer checker.* This strategy was usually used in our manual analysis when a test failed because of `NullPointerException`. A typical way to fix such a defect is to surround the

Table 3 Strategies used to generate patches in our analysis.

Strategy	Description	Defects
Add NullPointer checker	Add null pointer checker before using the object to avoid `NullPointerException`	*M-4; Ch-4; Cl-2*
Return expected output	Return the expected value according to the assertions.	*L-2,7,9; M-3,5,10; T-1,3*
Replace an identifier with a similar one	Replace an identifier with another one that has the similar name and same type in the scope.	*L-6,8; Ch-7,8*
Compare test executions.	Generate patches by comparing the failed tests with those passed tests with similar test inputs.	*L-2,5*
Interpret comments	Generate patches by directly interpreting comments written in natural language.	*M-9; Cl-1,5,7,9; T-8,9*
Imitate similar code element	Imitate the code that is near the error location and has similar structures.	*L-4,5; M-6,8; Ch-1,2,7,9; Cl-3,8,10; T-5,7,10*
Fix by program understanding	Generate patches by understanding the functionality of program.	*L-1,3,9,10; M-6,9; Ch-2,3; Cl-3,8; T-1,2,4,10*

statement and all following dependent statements with a guarded condition `x!=null`, where `x` is the variable causing this exception.

The code on the right side is the patch for *Chart-4*. In this patch, an exception is thrown at line 5. The patch adds an "if" statement to surround line 5 and all following statements that depend on it. Though null pointer checker is often added to avoid `NullPointerException`, the strategy alone usually cannot decide a patch. In this case, we may also change the method `getRendererForDataset` so as not to return `null`. We come to this patch by further considering two facts: (1) applying this patch to make all tests pass, (2) there is also a checker for variable `r` at line 3, indicating that returning `null` is a valid behavior of `getRendererForDataset`. We use strategy 14 to summarize these consideration, which will be explained later.

```
1   r=getRendererForDataset(d);
2   if(isDomainAxis){
3     if(r!=null){...} ... }
4+  if(r!=null){
5     Collection c=r.getAnnotations();
6     Iterator i=c.iterator(); ...
7+  }
```

Related: This strategy is similar to a template used in the repair approach PAR [44] and ELIXIR [28], which apply a set of templates to the located statement to generate patches.

Improve: *Correctly identify the location of the null pointer checker.* As discussed before, there are often more than one place to add the null pointer checker, and identifying the correct location is the key for avoiding incorrect patches. In our manual analysis process, different strategies are combined together to decide the correct location. Similarly, ELIXIR depends on a machine-learned model to determine which template to use while PAR simply try different templates one-by-one. However, neither of them consider the runtime information of failing test cases, such as in the example of *Chart-4*, the exception thrown by the failed test case almost decides the desired template.

Strategy 9: *Return expected output.* When programming, we often encounter boundary cases that should be considered separately from the main programming logic, and such boundary cases are easily neglected by developers. A boundary case is typically handled by a statement `if(c) return v;`, where `v` is the expected result and `c` is a condition to capture the boundary case.

As a result, if the failed test execution is a boundary case, we may consider patches using the above form. For example, the following code snippet is a failing test from *Math-3*. If we can identify that an array of length one is a boundary case, we can come to the fix as inserting statement `if(len==1){return a[0]*b[0];}` into the method `linearCombination`. Here variable `len` represents the length of input arrays, while `a[0]*b[0]` is just the expected result. However, this strategy heavily depends on the developer's experience to decide boundary cases. Otherwise the generated patch may overfit to the current test suite.

```
1 void testLinearCombination(){
2  double[] a={1.23456789};
3  double[] b={98765432.1};
4  Assert.assertEquals(a[0]*b[0],
5   MathArrays.linearCombination(a,b),0d);
6 }
```

Related: This is similar to a template in ACS [4].

Improve: *Correctly identify boundary cases.* ACS can only tackle simple boundary cases, such as comparing with constants. Since this strategy is usually used along with boundary identification, therefore, when complex boundaries cannot be correctly identified by the approach, the repair will fail as well, such as the boundary case that will be introduced in Strategy 11

(*Lang-2*), which is hard for ACS. As a result, to better utilize this strategy, a powerful boundary identification mechanism is needed.

Strategy 10: *Replace an identifier with a similar one.* When the names of two identifiers are similar, developers may confuse the two identifiers. As a result, a possible patch is to replace an identifier with another one whose name is similar. Of course, this strategy alone can hardly decide a patch, but this strategy can be used together with other strategies for patch ranking.

For example, in defect *Lang-6* introduced in strategy 5, we can observe that two variables, `pos` and `pt`, have very similar names. In fact, if we replace the last occurrence of `pos` with `pt`, the piece of code no longer violates the programming practice. Furthermore, rerunning all tests could reveal that this patch passes all the tests. Putting all together, the correct patch will be preferably selected.

Related: Some existing approaches consider replacing variables [3] or methods [3, 6]. Most recent automatic repair techniques further identify the similarity between variables [19, 20, 50] with respect to variable names, types, and the like, which improved the state-of-art.

Strategy 11: *Compare test executions.* It is common that more than one test case exists to test a specific method, and only one of them fails. By comparing the passed tests and the failed tests, we can often obtain useful information on patch generation.

For defect *Lang-2*, the test inputs of all passed tests do not contain the character "`#`", whereas both the two failed test cases contain it, which suggests that containing "`#`" is probably a boundary case. Therefore, together with its expected output, `IllegalArgumentException`, the desired patch can be generated.

Related: The related approaches are mining invariants from executions as patch ingredients [18,27,34].

Improve: *Generalize boundary cases from executions.* In our analysis, we usually compare one or several successful executions with the failing one to identify the boundary cases that are related to the current failure. Therefore, they usually can be used directly to generate patches. On the contrary, existing approaches depend on a large set of successful executions to mine a set of invariants as fix ingredients, most of which are non-related. Furthermore, to mine complex boundary cases, such as the example introduced in this strategy, is also hard for existing techniques. Therefore, learning boundary cases from a small set of examples can potentially improve the precision of existing techniques.

Strategy 12: *Interpret comments.* Program source code may contain comments explaining properties of the program, such as functionality, precondition, and the like. In particular, Java programs often come with Javadoc annotations explaining the method, the parameters, the return value, and exceptions that might be thrown. These comments often provide important information to guide patch generation.

For example, the following method was used to create a `DateTimeZone` object based on the given `hours` and `minutes` (*Time-9*). The failed test expects an `IllegalArgumentException` to be thrown at the input of 24 and 0. Again, this is a boundary case where strategy 9 can be applied. However, we still do not know what condition should be used to capture this boundary case. By reading the Javadoc, we can know that `hours` should be in the range of $-23 \sim 23$, and the following patch is straightforward.

```
1 // the offset in hours from UTC, from -23 to +23
2 public DateTimeZone forOffsetHM(int hours,int minutes) throws IllegalArgumentException{
3+  if(hours<-23||hours>23) throw new IllegalArgumentException();
```

Related: Some approaches have adopted natural language processing techniques to analyze comments and other documents in a natural language. For example, ACS [4] analyzes the Javadoc to exclude unlikely variables in an "if" condition, and R2Fix [21] generates patches by analyzing the bug reports in a natural language.

Improve: *Incorporate dynamic information of test failure.* The depth of automatic analysis still cannot match that in our analysis. For example, the following patch is generated to fix *Closure-9* based on the comments. For current automatic techniques, it is impossible to interpret this comment to the corresponding source code. Even though they can parse the natural language, it may be confused about which character should be replaced. Therefore, we need to associate the comments with the runtime information. In this example, only the failed test cases contain character "\", which is the very character to be replaced. As a result, more robust natural language understanding is imperative. Besides, incorporating the dynamic information with the natural language understanding is needed as well.

```
  //The DOS command shell will normalize "/" to "\", so we have to wrestle it back.
+ filename = filename.replace("\\", "/");
```

Strategy 13: *Imitate similar code element.* In general, programs with similar functions often have similar structures. When similar code pieces exist near the buggy code, we can generate patch by imitating the similar code. This strategy is often useful when we found the program fails to handle some cases, but we do not know how to handle these cases without the full specification. However, if we can find code pieces handling similar cases, we can imitate these code pieces.

For example, the patch on the right side comes from *Chart-9*. According to the failing test, when the `startIndex` is greater than `endIndex`, no exception should be thrown, which can lead us to generate the condition statement `if(startIndex>endIndex)`. However, we still do not know the `if` body. By reading the code nearby, we find that the `if` at line 1 is used to handle a similar case, so we can generate the desired patch by imitating the first `if` condition.

```
1   if(endIndex<0) emptyRange=true;
2 + if(startIndex>endIndex)
3 +     emptyRange=true;
4   if(emptyRange) { ... }
```

Related: A related strategy adopted by several automatic program repair approaches is to mine fix ingredients from existing source code for patch generation [2, 4, 19, 20, 50, 61].

Improve: *More flexible code adaptation.* Related approaches either do not perform any code adaptation [2, 4, 61] or only perform elementary variable replacing [19, 20, 50], which is not sufficient to tackle complicated cases. For example, to fix *Chart-2*, a method call of `intervalXYData.getXValue(series,item)` should be inserted, which does not exist in the similar code. However, another, `icd.getValue(row,column)`, was referred to generate the patch in our manual analysis. We can see that besides the variables, we need to further transform the method call and sometimes the cases can be even more complicated. Therefore, to improve current techniques, more powerful code adaptation should be developed.

Strategy 14: *Fix by program understanding.* Similar to fault localization, this strategy is placed to capture the case where we generate the patch by understanding the functionality of the program. The process is similar to the fault localization case, but the potential patches become another source for generating constraints. If we found the constraints generated from a patch are consistent with all other constraints, we would rank the corresponding patch higher.

Similar to fault localization, we still lack a full understanding of the program understanding process, and future work is needed to further understand this process.

5.4 RQ4: Inspiration from Analysis

Based on the previous analysis and comparison, we can find that although many of the strategies have already been considered in existing techniques, still some of them (e.g. *Locating undesirable value changes*) have not been considered by any approaches, and some of them (e.g. *Imitate similar code element*) are not applied in the same way or in the same depth as we do, especially the combination of static and dynamic information. The result indicates that existing techniques still have room for improvement.

Finding 2. *While existing techniques have already explored strategies similar to some of the strategies we identified, they have potential to be further improved based on the identified strategies.*

By further observation, we can find that many strategies are simple heuristic rules that do not require deep semantic analysis or full understanding of the program, indicating a high possibility to automate them. Many strategies perform only mechanical operations and can be easily automated. For example, *Stack trace analysis* and *Locating undesirable value changes* performs only mechanical operations as introduced previously. Some strategies require human experience, but such experience has a high potential to be summarized as heuristic rules. For example, *Excluding unlikely candidates* relies on a few heuristics rules to determine whether a method may be faulty. Some simple strategies, such as excluding library functions, used in our analysis have been listed in strategy 2, which can be easily expanded by developers. In fact, only the last strategy in each category, strategy 7 and strategy 14, require full program understanding.

Additionally, as our results show, no single strategy can be effective on a large portion of the defects. Furthermore, most of the defects require multiple strategies to locate and to repair. For instance, to

correctly locate the faulty code of *Lang-1*, we not only use the *Stack trace analysis* but also *Excluding unlikely candidates* strategy. Furthermore, we can notice that both of the defects explained in strategies 11 and 12 applied strategy *Return expected output* besides the strategy explained for each. This observation calls for the studies on combining different fault localization and patch generation approaches.

Finding 3. *Many strategies are simple heuristic rules, such as Excluding unlikely candidates, Locating undesirable value changes, Add NullPointer checker, Compare test execution , and Imitate similar code element, etc., that do not require deep analysis nor full understanding of the defects, indicating possibility of automating these strategies to improve current automatic repair techniques*

Finding 4. *No strategy can handle all defects. Combinations of strategies are needed to repair a large portion of defects.*

6 Threats to Validity

First, we discuss the generalizability of our results. Since the case study only involves 50 defects and 5 projects, they may not be representative for a wide range of defects in different types of projects. As a result, our results on the effectiveness of the strategies may not be generalizable to a wider range of projects. However, Defects4J [23] is a widely-used defect benchmark, and so far no generalizability issue on it is reported. Furthermore, we evenly sampled the defects among the 5 projects and the effectiveness of those strategies has been evaluated on them. These facts give us a reasonable degree of confidence on the generalizability of our results.

Second, even though we have no prior knowledge about those defects to be analyzed, some basic insights about those projects can be implicitly obtained along with the analysis going on, which may cause training effects to the subsequent analysis. As a result, when summarizing the defects requiring the two program understanding strategies, we may accidentally miss some defects as the program understanding happened unintentionally. To avoid this problem, we have carefully reviewed the analysis record to ensure that the rest of the defects can be fixed without program understanding. Please also note that the validity of the main findings, including the strategies and improvements suggested to existing techniques, are not affected by the threat.

Third, as also mentioned in the introduction, our results should not be interpreted as an upper bound of the performance of automatic program repair techniques since they may be superior to human developers on some aspects as well, e.g., by utilizing its computation power. In other words, our results show what automatic techniques can potentially do, but not what they cannot do.

Fourth, our study should not be interpreted as an understanding of how human debugs. Our manual analysis settings is different from general human debugging and a single analysis session is not enough to answer such a question. In Section 2, we have summarized some related work on that problem. Besides, though a bug can get repaired with different ways and we only depend on the standard patches in Defects4J to determine patches correctness, the final result are not affected since those correct patches are proved correct while those overfitting patches are obvious and definitely incorrect ones after examination.

7 Conclusion and Future Work

In this paper, we analyzed 50 real world defects to identify to what extent they can be fixed under existing test suites, based on which we summarized the fault localization and patch generation strategies used in our analysis, and discussed the potential of them to be automated to improve existing techniques. Our findings suggest that most of these defects can be fixed in our analysis even though without complete specifications and there is potentially a lot of room for current techniques to improve, and the strategies we identified could potentially be automated and combined to improve the performance of automatic program repair, which calls future work on the automation of those strategies and their combinations.

Acknowledgements This work was supported by the National Key Research and Development Program of China (Grant No. 2017YFB1001803) and National Natural Science Foundation of China (Grant No. 61672045).

References

1 Mei H, Zhang L. Can big data bring a breakthrough for software automation? Sci China Inf Sci, 2018, 61(5):056101
2 Le Goues C, Nguyen T, Forrest S, et al. Genprog: A generic method for automatic software repair. IEEE Trans Softw Eng, 2012. 38:54–72
3 Long F, Rinard M. Staged program repair with condition synthesis. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. ACM, 2015. 166–178
4 Xiong Y F, Wang J, Yan R F, et al. Precise condition synthesis for program repair. In: Proceedings of the 39th International Conference on Software Engineering. IEEE, 2017. 416-426
5 Mechtaev S, Yi J, Roychoudhury A. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In: Proceedings of the 38th International Conference on Software Engineering. ACM, 2016. 691–701
6 Kim D, Nam J, Song J, et al. Automatic patch generation learned from human-written patches. In: Proceedings of the 2013 International Conference on Software Engineering. IEEE, 2013. 802–811
7 Thien Nguyen H D, Qi D, Roychoudhury A, et al. Semfix: program repair via semantic analysis. In: Proceedings of the 2013 International Conference on Software Engineering. IEEE, 2013. 772–781
8 Mechtaev S, Yi J, Roychoudhury A. Directfix: Looking for simple program repairs. In: Proceedings of the 37th International Conference on Software Engineering. IEEE, 2015. 448–458
9 Gao Q, Zhang H S, Wang J, et al. Fixing recurring crash bugs via analyzing q&a sites (t). In: Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering. 2015. 307–318
10 Long F, Amidon P, Rinard M. Automatic inference of code transforms for patch generation. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ACM, 2017. 727–739
11 Rolim R, Soares G, Dantoni L, et al. Learning syntactic program transformations from examples. In: Proceedings of the 39th International Conference on Software Engineering. IEEE, 2017. 404-415
12 Long F, Rinard M. Automatic patch generation by learning correct code. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM, 2016. 298–312
13 Abreu R, Zoeteweij P, Van Gemund A J. On the accuracy of spectrum-based fault localization. In: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques. 2017. 89–98
14 Abreu R, Zoeteweij P, Van Gemund A J. Spectrum-based multiple fault localization. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering. 2009. 88–99
15 Zhang X Y, Gupta N, Gupta R. Locating faults through automated predicate switching. In: Proceedings of the 28th international conference on Software engineering. ACM, 2006. 272–281
16 Chandra S, Torlak E, Barman S, et al. Angelic debugging. In: Proceedings of the 33rd International Conference on Software Engineering. ACM 2011. 121–130
17 Marcote S L, Durieux T, Le Berre D. Nopol: Automatic repair of conditional statement bugs in java programs. IEEE Trans Softw Eng, 2016. 43:34–55
18 Perkins J H, Kim S, Larsen S, et al. Automatically patching errors in deployed software. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. ACM, 2009. 87–102
19 Wen M, Chen J J, Wu R X, et al. Context-aware patch generation for better automated program repair. In: Proceedings of the 40th International Conference on Software Engineering. ACM, 2018. 1–11
20 Jiang J J, Xiong Y F, Zhang H Y, et al. 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, 2018. 298-309
21 Liu C, Yang J Q, Tan L, et al. R2fix: Automatically generating bug fixes from bug reports. In: Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation. 2013. 282–291
22 Le Goues C, Dewey-Vogt M, Forrest S, et al. 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, 2012. 3–13
23 Just R, Jalali D, Ernst M D. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In: Proceedings of International Symposium on Software Testing and Analysis. ACM, 2014. 437–440
24 Qi Z, Long F, Achour S, et al. 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. ACM, 2015. 24–36
25 Martinez M, Durieux T, Sommerard R, et al. Automatic repair of real bugs in java: a large-scale experiment on the Defects4J dataset. Empir Softw Eng. 2016. 1–29
26 Xiong Y F, Liu X Y, Zeng M H, et al. Identifying patch correctness in test-based program repair. In: Proceedings of the 40th International Conference on Software Engineering. ACM, 2018. 789–799
27 Chen L S, Pei Y, Furia C A. Contract-based program repair without the contracts. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. IEEE, 2017. 637-647
28 Saha R K, Lyu Y J, Yoshida H, et al. Elixir: Effective object oriented program repair. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. IEEE, 2017. 648-659
29 Smith E K, Barr E T, Le Goues C, et al. 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. ACM, 2015. 532–543
30 Long F, Rinard M. An analysis of the search spaces for generate and validate patch generation systems. In: Proceedings of the 38th International Conference on Software Engineering. ACM, 2016. 702–713
31 Zhong H, Su Z D. An empirical study on real bug fixes. In: Proceedings of the 37th International Conference on Software Engineering. IEEE, 2015. 913–923
32 Tan S H, Yoshida H, Prasad M R, et al. Anti-patterns in search-based program repair. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 2016. 727–738
33 DAntoni L, Samanta R, Singh R. Qlose: Program repair with quantiative objectives. In: Proceedings of International Conference on Computer Aided Verification. Springer, 2016. 383–401
34 Wei Y, Pei Y, Furia C A, et al. Automated fixing of programs with contracts. In: Proceedings of the 19th international symposium on Software testing and analysis. ACM, 2010. 61–72
35 Gao Q, Xiong Y F, Mi Y Q, et al. Safe memory-leak fixing for c programs. In: Proceedings of IEEE/ACM 37th IEEE International Conference on Software Engineering. 2015. 459–470
36 Cai Y, Cao L W. Fixing deadlocks via lock pre-acquisitions. In: Proceedings of the 38th International Conference on Software Engineering. ACM, 2016. 1109-1120
37 Hassan F, Wang X Y. Hirebuild: An automatic approach to history-driven repair of build scripts. In: Proceedings of the 40th International Conference on Software Engineering. ACM, 2018. 1078–1089
38 Martinez M, Monperrus M. Mining software repair models for reasoning on the search space of automated program fixing. Empir Softw Eng, 2015. 20:176–205
39 Soto M, Thung F, Wong CP, et al. A deeper look into bug fixes: patterns, replacements, deletions, and additions. In: Proceedings of the 13th International Workshop on Mining Software Repositories. 2016. 512–515
40 Yang J Q, Zhikhartsev A, Liu Y F, et al. Better test cases for better automated program repair. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ACM, 2017. 831–841
41 Baudry B, Fleurey F, Le Traon Y. Improving test suites for efficient fault localization. In: Proceedings of the 28th international conference on Software engineering. ACM, 2006. 82–91
42 Artzi S, Dolby J, Tip F, et al. Directed test generation for effective fault localization. In: Proceedings of the 19th international symposium on Software testing and analysis. ACM, 2010. 49–60
43 Yang D H, Qi Y H, Mao X G. Evaluating the strategies of statement selection in automated program repair. In: Proceedings of International Conference on Software Analysis, Testing, and Evolution. Springer, 2018. 33–48
44 Tao Y D, Kim J, Kim S, et al. Automatically generated patches as debugging aids: a human study. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 2014. 64–74
45 Lawrance J, Bogart C, Burnett M, et al. How programmers debug, revisited: An information foraging theory perspective. IEEE Trans Softw Eng. 2013. 39:197–215
46 LaToza T D, Myers B A. Hard-to-answer questions about code. In: Proceedings of Evaluation and Usability of Programming Languages and Tools. ACM, 2010. 8:1–8:6
47 Murphy-Hill E, Zimmermann T, Bird C, et al. The design space of bug fixes and how developers navigate it. IEEE Trans Softw Eng. 2015. 41:65–81
48 Qi Y H, Mao X G, Lei Y, et al. The strength of random search on automated program repair. In: Proceedings of the 36th International Conference on Software Engineering. ACM, 2014. 254–265
49 Le XB D, Lo D, Le Goues C. History driven program repair. In: Proceedings of IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, IEEE, 2016. 213-224
50 Xin Q, Reiss S P. Leveraging syntax-related code for automated program repair. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. IEEE, 2017. 660-670
51 Agrawal H, Horgan J R. Dynamic program slicing. In: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation. ACM, 1990. 246–256
52 Zhang X Y, Gupta N, Gupta R. Pruning dynamic slices with confidence. In: Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, 2006. 169–180
53 Rathore S S, Kumar S. Predicting number of faults in software system using genetic programming. In: Proceedings of International Conference on Soft Computing and Software Engineering. 2015. 62:303–311
54 Tahir A, MacDonell S G. A systematic mapping study on dynamic metrics and software quality. In: Proceedings of the 2012 IEEE International Conference on Software Maintenance. 2012. 326–335
55 Wu R X, Zhang H Y, Cheung SC, et al. Crashlocator: Locating crashing faults based on crash stacks. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. ACM, 2014. 204–214
56 Wong CP, Xiong Y F, Zhang H Y, et al. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In: Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution. 2014. 181–190
57 Zhong H, Mei H. Mining repair model for exception-related bug. J Syst Softw. 2018. 141:16 – 31
58 Cleve H, Zeller A. Locating causes of program failures. In: Proceedings of the 27th international conference on Software engineering. ACM, 2005. 342–351
59 B Le TD, Lo D, Le Goues C, et al. A learning-to-rank based fault localization approach using likely invariants. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. ACM, 2016. 177–188
60 Ayewah N, Hovemeyer D, Morgenthaler J D, et al. Using static analysis to find bugs. IEEE Softw. 2008. 25(5):22–29
61 Weimer W, Fry Z P, Forrest S. Leveraging program equivalence for adaptive program repair: Models and first results. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering. 2013. 356–366