A Survey of Test Based Automatic Program Repair

Yuzhen Liu[1, 2], Long Zhang[1, 2], Zhenyu Zhang[1*]

[1] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China.
[2] University of Chinese Academy of Sciences, Beijing, China.

* Corresponding author. Email: zhangzy@ios.ac.cn
Manuscript submitted May 25, 2018; accepted July 23, 2018.
doi: 10.17706/jsw.13.8.437-452

Abstract: Testing and debugging have always been the most time-consuming parts of the software development procedure and require large amounts of human resources. When a bug is located, manually fixing it to repair the buggy program is still a difficult and laborious task for developers. Hence automatic program repair techniques, especially the test-based approaches, have drawn great attentions in recent years. Researchers have explored and proposed various novel methods and tools, pushing the idea closer to reality. In this paper, we systematically survey the work in mainstream of test-based program repair (TBR) and discuss the properties automatically generated patches should have. We classify the state-of-the-art approaches for TBR, and evaluate their strengths and weaknesses according to their functional mechanisms. Finally, we refer to some empirical results and propose four important issues, which are supposed to be critical and constructive in this research area.

Key words: Automatic program repair, fault localization, software debugging, software testing.

1. Introduction

Bugs are exposed in software every now and then, which may be caused by developers' carelessness, changes of software functionalities, mismatches among the interfaces of different components within complex systems, or even from inaccurate specifications acquired in the requirement analysis phase [1]-[3]. Since software defects could lead to significant loss, testing and debugging take important roles in the process of software quality assurance. They help developers to find, locate, and fix bugs. However, testing and debugging can be time-consuming and of low efficiency [4]. In a large-scale software system, the time spent on testing and debugging can be even longer than the time spent on development. To reduce testing and debugging effort, researchers have studied automated testing and debugging techniques for several decades.

Program debugging consists of two tasks: locating the bug and fixing the bug to repair the program. While both tasks are challenging, in this work we talk about the latter—program repair. The concept of *Automatic Program Repair* has drawn rapidly increasing attentions within software engineering community. A variety of techniques and ideas were proposed and evaluated by researchers in recent years. Before the appearance of automatic program repair, there are research work addressing similar problems to automatically survive a program upon failure, known as fault-tolerant software or self-healing system [5], which are equipped with the ability of detecting and eliminating the fault themselves. These pioneer reseaches are considered to be a

kind of repairing in different scenarios from the context described by Weimer et al. [6], which share the same final goal.

From a global view, automatic program repair encompasses a broader range of application fields and manifestations, with several taxonomies [7]. In terms of the subjects being repaired, there are automatic repairing techniques fixing source code, binary code or data within memory. In terms of the automation level for automatic program repair, there are *fully automated* repairing methods without human interference and *semi-automated* methods to provide candidate patching alternatives to developers. In terms of repairing strategy, *state-based* repair modifies the program execution states in runtime by manipulating their registers, heaps or stacks, and so on, while *behavior-based* repair changes the program codes statically and permanently. The latter can be further divided into *specification based* and *test based* repair. With specification based repair, programs are repaired or verified through specifications given previously according to their functional logic requirements, which are often provided by programmers. Test based repair takes a faulty program and its test suite as basic inputs, as well as some additional constraints that may be required at specific circumstances. To satisfy the constraints, program synthesis is a commonly used technique in repair generation. Finally, it gives a fault-eliminated program version or a list of patch candidates as output. In this work, we demonstrate the taxonomy of program repair in Fig. 1 and give a survey of *Test Based Repair* (TBR)[1]. Even there is a more comprehensive survey given by Martin [8] that covers the total fields of program repair including behavioral and state repair, we think it is still necessary to dedicatedly and concentrately survey test based methods within the behavioral repair domain. Since the amount of novel methods and repair effectiveness both receive an explosive growth in recent years, the revisions of their correctness and feasibility are also following up. Taking existing empirical studies into account, we summarize the remaining issues and obstacles of TBR, and try to propose some further ideas and directions.

In the remainder of this article, we first describe the problem statement and definitions in section 2, and systematically introduce the test suite based automatic program repair techniques in section 3. After that, we discuss their performance. We talk about some important issues in section 4, and finally make a conclusion in Section 5.

2. Preliminaries

In this paper, we focus on TBR, such a kind of technique that uses test suite to prove the correctness of programs. It is often based on a framework of patch searching and regressive patch validation.

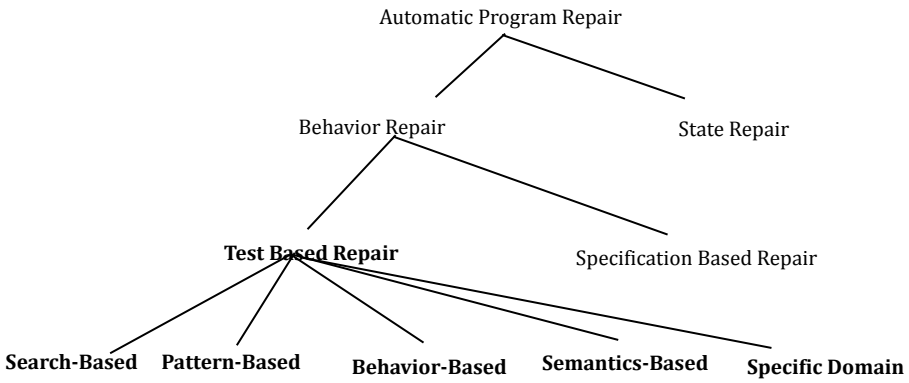


Fig. 1. Taxonomies of repair techniques.

[1] It is also the class that Weimer originally classified to [7][30].

TBR requires no additional specifications or human inspections, the classic scenario of which is a longtime multi-version software repository. A single version of program *P* is a sequence of statements. The program behaviors are described by test suite *TS*, where a test case defines the program expected output *O* corresponding to a certain input *I*. The program is run against the test suite and the set of result is *RS*, each result r_i can be passed or failed, where *i* is the index of the test. If all the test cases are passed, the program is regarded as bug free; otherwise it indicates the existence of bugs[2]. When a program is reported as buggy, repairing work is supposed to be done to locate and analyze the fault.

Automatic repair usually adapts the step of automatic fault localization ahead. This kind of technique produces a list of code statements ranked by the statistical probabilities of their relationship with the fault. The most suspicious statement is supposed to be ranked the highest. Within the list, repair work checks the statements one by one in order to confirm a repairable fault. TBR is mainly applied to the almost correct programs with only tiny anomalies under some specific conditions rather than large piece of code generation. When the exact location of the fault is known, which can also be called the fixing point, searching for a repair is the next task. In practice, more than one solution may be found. As a result, a set of candidate repair *R* is available. When any repair among them is applied, we get a program variant *P'* and the new result set *RS'*. A validated repair will be found once every test case in *TS* pass. In former research, the repair actually used in the benchmark is usually called the true patch. If a validated repair is equivalent to the true patch, it's considered as correct that can be trusted by developers.

Because the repair is validated by the test suite, the comprehensiveness and coverage of test suite matters a lot. Untested inputs may miss a possible bug. Weak test suite may not guarantee that the patched programs work well in practice. On one hand a number of test cases promotes the reliability of repairing; on the other hand the patch validating is also an extremely time-consuming process. At the same time, the redundant test cases can waste a lot of resources in regressive testing despite the fact that some techniques have been tried to cut candidate patch searching trials. Therefore a deliberately designed test suite, even with flexible alternatives under different test running situations, plays an important role in TBR, which we will talk in detail as an important issue in section 4.1.

3. State-of-the-art Techniques

In this section, we provide an overview of the classification of TBR techniques, and make an general discussion of the basic ideas of each category and each technique as well as their relative strength and weaknesses. The related techniques are listed but not limited to the following.

3. 1. Search-Based TBR

A faulty program usually functions well under normal conditions and only fails on special inputs, when executing a code block containing the root cause of execution failures. The bug may come from a wrong operator, constant or variable name, and so on, which can be highly vulnerable for programmers' handwriting negligence. This kind of fault is common and can be easily fixed by mutation techniques, and the attempts to repair program automatically by mutation searching are carried out early in the field of TBR.

Debroy and Wong [9] proposed the idea of mutating the mistake statements to automatically fix the faulty program. To repair the program via mutating some statements, fault localization must be conducted first. Fault localization provides a ranked list of program statements according to their faulty suspicious degree calculated by executing information. The suspicious degrees come from a statistical model, which takes a basic assumption that if a statement is executed by more failed test cases it is more probable to be faulty or

[2] In this paper, we use "bug", "fault", and "defect" interchangeably, and use "repair", "patch", and "fix" interchangeably.

more related to faults [10]. There are many approaches that automatically locate the fault in a program in recent years [1], [11]-[12][13]. However, details of them have gone out of the scope of this paper.

Given the ranked list of suspicious program statements, a programmer usually scans it to check for the root cause of failures. Correspondingly, a mutating operation will take every single statement as a candidate and try to fix it by leveraging a mutation. If all mutations are proved unable to fix the program, it takes the statement just tried as a correct statement and moves on to the next suspicious statement. A basic mutation first classifies the elements of operators such as arithmetic, relational, logical, self-increment, self-decrement, and assignment operator as well as constant, each of which also includes some operators as mutation options. Replacement on the elements within same category is usually syntactical and feasible for trial of mutation fixing. There are also some mutations like decision negation on a condition statement.

Different from random search, heuristic searching is a stochastic algorithm that finds an approximately optimal solution during the iteration of mutating and selecting according to the fitness function. The evolution of the program is gradually accumulated. The framework of heuristic searching is suitable for an undetermined question or the scenario where the final solution is hard to be calculated precisely. The repairing of a program can be seen as an undetermined task since there are a few candidate patches that can be applied to fix the same fault, and to search the perfect solution is much harder than just find an acceptable one.

Arcuri and Yao [14], [15] proposed a co-evolutionary framework to evolve the program and test suite together, and validated their method on bubble-sort program, with test cases generated initially from the basic formal specifications of sorting. They provided a flexible fitness function to calculate the current state distance from the expected. The performance of test cases can also be used to determine the test cases quality and evolve the test suite gradually. By manually inserting eight known bugs into the sort program for experiment, they found a promising fixing rate, while the efficiency remains to be improved.

Westley Weimer and his fellows used genetic programming (GP) in automatic program repair and developed GenProg [16], [17], [6]. There is a basic assumption particularly that the missing code or the code changes can be patched from another location of the program. It means that automatic repair operations do not have to compose code itself, they just search and copy a piece of existing code as a patch template. This idea is based on that programmers may make a mistake in one situation but handle it correctly in other location, such as a null pointer check, which can be used in several lines of the program but missed in one location. So the mutation in GenProg is different from other operator mutating because its fixing unit is statement, namely statement insertion, deletion and swap. The faulty program is represented as a abstract syntactical tree by CIL toolkit [12] in order to reduce complexity of the multiple statement and expression types to a small number like Instr, Return, If and Loop. GenProg based repair tools have demonstrated its ability to fix different kinds of fault in practical large scaled programs. GenProg can find patches fully automatically with no additional annotations, human interventions, or fault type limitations, though the tool is only implemented on C programs.

Arcuri and Briand [18] doubted whether the genetic programming can really bring about effectiveness since it is observed that a patch is often found among the initial random mutation before the evolving gets actually started. They implemented TrpAutoRepair [19] and RSRepair [20], which use the same mutation approaches as GenProg and run prioritized test cases to validate variants hoping a useless variant could be detected as soon as possible. Distinct from GenProg, no evolving means no use to execute all the tests to calculate the fitness. RSRepair stops patch validation regression as far as a test case fails, which reduces the time expense significantly. Experiment reports that both the patch trials and test case executions per validated fix of RSRepair are less than those of GenProg, which indicates that the benefit brought by genetic programming cannot make up the cost caused by fitness evaluations, and thus worsening the patch search

process [20]. On the contrary, Assiri and Bieman [21] provided MUT-APR to illustrate the strength of operator mutation against using the existing code to generate patch while keeping the genetic fitness evaluation in patch selection. They found that the failing rate of candidate patch generated by mutation techniques is much less than the rate of using existing code.

Tan and Rovehoudhury [22] adapted the thinking of mutation repairing into regression bugs which are generated along the multi-version software progressing. They manually inspected 73 benchmarks of the program evolution to understand the code changes in real-world regressions and divided the regression bugs into types of Local, Unmask, and Remote roughly based on the relationship of the changes codes and the direct fault location. They concluded more than one dozen operators and implement eight of them in their repairing tool relifix such as add condition, revert to previous statement, and so on. Some of them are non-parameterizable so that the patch can be easily generated from the previous code piece. They localized the fault automatically and randomly selected a contextual matched operator on it and run against test suite. They got the intuition that adds condition operator can help hide the problematic changes.

Weimer *et al*. [23] used approximate program equivalence detection between generated patches to avoid redundant regression for semantically equal candidates. Recently, Xiong et al. [24] demonstrate the strength of hints provided by variable dependency and code document when synthesis patches in ACS.

3.2. Pattern-Based TBR

Techniques of fault pattern-based repair represent the idea that one can fix a certain type of fault using its corresponding fixing operations. Like decease diagnosing, debugging procedure diagnoses the faulty program and tries to figure out the root cause of a bug via exposed symptoms. The symptoms help concluding and recognizing the fault pattern. At the same time learning from human fixing behaviors in debugging and repairing can also provide inspirations to machined automatic program repair to make the patches more readable and acceptable.

Efforts to category fault patterns can date back to a long time ago [25][26] while there was no consensus taxonomy of the most common bug root cause. Recently, Pan et al. [27] gave a new report of the understanding of fault patterns from the angle of human programming mistake and fix approaches. They found 27 detailed fault and fixing patterns manually identified from seven widely used Java projects. They discovered that the most common categories cover more than half of the bugs reported and share a similar distribution among the different single projects and individual programmers. Similarly, Martinez and Monperrus [28][29] mined the repair actions on 14 repositories of Java software to give inspirations for heuristic patch searching. Their focus is the frequency of the types and percentage is used as a hint for fix shaping. They gave an empirical evaluation of the patch searching guidance and found that their probabilistic repair actions can guide the automated fixing process in the repairing space, with a probabilistic focus on likely repair shapes first [29]. Besides, the distribution of repair actions is similar across different projects and developers, which claims a general fitness of this approach.

Based on the manually acquired fault pattern database, Kim et al. [30] provided a tool PAR to generate patches learned from human-written ones. They investigated 62,656 patches of Java programs in Eclipse JDT and analyzed the relationship between the bug features and the code AST (Abstract Syntax Tree) changes in fixing. They classified the patches as additive, subtractive, and altering patches, and studied the details of how the patches solved the buggy program. Then they confirmed a few repair patterns that are common in reality, such as altering method parameters, adding null checker, and so on. In each pattern there are some templates of fixing scripts that point out the bug features required to adapt a fix pattern and changes of the faulty code in a patterned manner. Though the work of template generating is mostly manual efforts, it is a one-time cost. Kim listed ten templates used in PAR and evaluated its performance against GenProg on 119 bugs. However, the tool has to know which statement should be modified first. There are three steps involved

in PAR, namely, AST analyzing, context checking, and template applying. The tool analyzes the variables and method calls of the suspicious faulty location, identifies the features required by a template, and decides which could be selected to fix the bug. For example, the Null Pointer Checker template can be applied where there is object reference in the statement. Similar idea is also proposed by Long et al. [31], which work SPR derived parameterized transformation schemas to customize the fault type and repairing behaviors.

Researchers also deployed machine-learning methods to utilize the latent patterns in patch examples. Jeffrey et al. [32] implemented machine-learning techniques in BugFix to automatically find the relationships between bug situation and the human fixing operations. They generated the associating rules through the training data set and adapted appropriate rules according to the certain situation when fixing a new bug. The situation is represented in three descriptions, i.e., statement structure, interesting value mapping pair pattern, and value pattern. They did not directly modify the program but provided a list of repairing suggestions ranked by the associated confidence value. Hence it is a semi-automatic technique actually. Each time the programmer selects a repair suggestion, BugFix learns from the new fix situation. Therefore, the tool is expected to become smarter when used for a period of time. Prophet [33] ranked and validated generated patches by the similarities with human written patches according to extracted program value features and modification features. Schramm et al. [34] combined the speed of Prophet's feature-based search and the reliability of SearchRepair's constraint solving. After compared several machine-learning methods to identify better patch candidates, it used random forest to finally obtain 96% accuracy. At the same time, it is also much faster than Prophet [35][33].

3.3. Behavior-Based TBR

Buggy programs do not always work as expected, since their behaviors were not properly encoded in the source. Program behavior consists of primary statements, instructions, function calls, as well as the different program structure like looping and recursion. Most operation are executed only when certain preconditions are satisfied and the execution is supposed to meet the post-conditions according to the realistic logic. Thus violations of specifications can introduce defects with specific inputs.

Dallmeier et al. [36] proposed a behavior-based repair method by distinguishing the behavioral anomalies in executions of passed and failed test cases, and implemented the tool PACHIKA. Their work aims at object-oriented programs like Java, using finite state machine to describe the states of the object and abstract the behavior models. Here, an object state is transferred via method calls and can be characterized by the properties of their attributes [37]. They instrument programs to track execution information such as method start, method end, and object initiation. The main issue in abstracting the behavior model is how to determine the abstraction level. If the abstractions are too fine-grained, there would be too much behavioral anomalies for the concrete values diverse a bit with different inputs. Otherwise, the abstractions could be too weak, and the model violations could hide behind a mass of missed information. PACHIKA mines the models for passing executions, namely the common preconditions satisfied among method invocations, which can be described by properties accessed by the method.

Later studies of AutoFix by Wei et al. [38][39][40] used contracts to reflect the program behavior specifications, implemented in Eiffel. These contracts include preconditions (require), post-conditions (ensure), intermediate assertions (check), and class invariants. They characterized the object states via the Boolean queries of the class and implications to formulate the fault profiles. Then they compared the differences between the passing and failing runs to find the state invariants that hold in all passed test cases but not in the failed. After further narrowing down of causing predicates, they gave a modification of the predicates as fix that reverses the failed runs pass. Wei et al. [41][39][40] also integrated the special handles to the linearly constrained assertions to the framework. AutoFix uses four modification schemas to generate

the candidate fix and validate in regression as well. It provides the dual repair idea of fixing implementations assuming that the contracts are correct, implemented as SpecFix [41].

Recent advances of the behavior-based repair are the NOPOL reported by DeMarco et al. [42]. NOPOL used Satisfiability Modulo Theories (SMT) [43][44] to fix the buggy IF conditions and synthesize additional preconditions. They adapted the concept of angelic value that is defined as a brilliant mutation during the execution that turns the failed runs into passed runs. And the searching space of angelic value here is narrowed to the "if" conditions. They forced the output of the suspicious boolean "if" statement to be true or false with certain execution and checked whether it is an angelic value. If an "if" condition cannot be fixed, they searched for the missing preconditions, which should be evaluated as false to skip the fault causing method in failed runs and true for normal execution. The expected output is synthesized by the primitive elements, and the synthesis is handled by SMT solver.

3.4. Semantics-Based TBR

Behavior-based repair fixes the program with specifications mined during the test suite execution, while semantics-based method relies on the specifications directly encoded from the test cases. Along with the improvement in symbolic execution and constraint solving, programs are easily transferred to a form of conjunctions of expressions and solved by some solvers like Z3. The result of solving is a precise guidance of program synthesizing, which reduces the time cost significantly when generating patches.

Symbolic execution takes symbolic values instead of exact input values as the inputs to statically run the program [45][46]. Given the suspicious statement, He et al. [47] carried out experiments with tiny programs combining formal analysis and software testing to locate and repaired the fault by finding the path-based weakest precondition. SemFix [48] generates the constraints of the program using symbolic execution or other program verification tools. The constraint are represented in a formalized format, such as STM-LIB2 [49], which will be solved by SMT solver like Z3. Here, a solver is used to verify whether the constraint is satisfiable and find the solution. The result given by constraint solver is abstract and thus we should synthesize a repair to implement the function semantics. The solver reduces the complexity of the constraints so that the patch is simplified.

DirecFix [50] combines the locating and repairing to a single step using partial MaxSMT solver to find the maximum subset of constraint clauses, which makes unsatisfiable constraints satisfiable. The clauses are divided into soft and hard clause, and soft clauses can be removed to reduce the restriction.

Experiments on semantics-based approaches show faster speed and more accuracy in repairing, compared to searching based techniques, in general. The constraint figures out the modifications of the faulty code more precisely, reducing the patch searching space. However, because the semantic analysis takes much more time when program scale grows up, the symbolic execution may encounter path explosion. Mechtaev et al. [51] promoted a novel semantics-based method called Angelix, which is more scalable and can handle multiple bugs at the same time. As for some situations such as large bound of cycle or recursion where the constraint generation performs weak, there are newly supporting trials on recursive program repair by Kneuss et al. [52]. Le et al. [53]compared and explored the strengths and weakness of different synthesis engines and found that forging the results from them can increase the performance of repair by generally 50%.

3.5. Specific Domain TBR

Besides the generalized techniques of automatic program repair, some technique concentrate on the specific type of fault in a certain programming situation. By adding a few restrictions to the bug, the features of fault can be explicitly characterized and it becomes much easier to recognize the bug situation so that the fixing operations can be targeted and parameterized partly. The limitations help researchers find better repair methods to increase the repairing success rate and speed up the patch searching process owing to the

reduced state space. With the help of domain knowledge, the way that the test suite contributes to patch search differs a lot.

Ocariza et al. [54] worked on javascript program fixing. The widely used browser-side language has flexible properties. Further investigation illustrates four major fix patterns, namely parameter modification, DOM element validation, method/ property modification, and major refactoring. Eventually they proposed three repair methods in their tool Vejovis. They are parameter string replacement, index replacement, and null/undefined checks. The evaluation results suggest a much higher success rate than the generalized techniques mentioned above.

Some techniques aim at invalid HTML bugs generated by PHP. In most modern web browsers, the invalid HTML can be fixed quietly, however there are still some severe bugs that have a significant impact on the demonstrating effect [55]. Further study found that the HTML bugs usually come from the PHP print codes, especially the constant string printing, which is a common programming manner at server side. Because the constant string printing always behaves the same under a number of different test cases, the correct value could be figured out by solving the string constraint for the input and output mappings. This narrows down the bug checking space and performs well in HTML generation bugs in PHP.

Specific program repair consists of a number of techniques to automatically repair buggy program in various specific circumstances. They could heavily rely on the programming environment that reduces the scalability on the trading off with practical usability. As a result of the broad scope of their concentration, the techniques used vary a bit. General fixing also encountered the issue that expected program behavior is hard to implement in detailed code for lack of hints. While specific program repair is sometimes more effective for the repairing goals. For example, we can insert a break statement on an appropriate location to terminate useless looping earlier.

4. Research Issues

Despite a burst in research of automatic program repair in recent years, people are still facing a number of challenges to make the repair techniques practical. In this section we highlight four important research issues that we consider fundamental and talk about some empirical findings by other researchers, alongside we'll give our advises to move forwards during the journey to automatic program repair in industrial software.

4.1. Impact of Test Suite Quality

In test based repair, the validity of patch is only guaranteed by test suite. That is, the repair work is unable to really check the semantics accuracy of a patch so that its functional impact is actually unknown when adapted into the buggy program, which corrodes the basis of test based repair deeply. Martinez et al. [56] and Qi et al. [57] respectively found that the correctness rate of automatically generated patches is not satisfying, here the correctness rate is the ratio of correct repaired faults and total faults. Martinez et al. conducted experiments on a large dataset with real-world Java bugs called Defects4J, and found that 47 out of 224 bugs being fixed, but only 11 of 84 generated patches were confirmed as correct.

Within the domain of TBR, the correctness of repair relies on the test suite quality mostly, especially the distribution of testing coverage. Ideally, if the passed test suite covers all the possible circumstances, the patch can always meet the functional requirements. Hence the first task is to provide more adequate test suite, but that task seems to be equally hard compared with human debugging because the test case construction still relies heavily on human labor. We believe the ways to represent the program requirements formally and rapidly can help a lot, also automatic test case generation that explores the exhaustive branch paths grows to be a promising direction [58][59], Jiang et al. [60] integrated metamorphic testing into automatic program repair to alleviate the oracle problem. Future repair methods

are supposed to demonstrate their fixing rate against strong test suites that are convincing enough.

But test suite can never be perfect. The second thing is that we strongly advice researchers evaluate the correctness rate of their automatic repair as well as fixing rate, no matter by human inspection or ground truth comparison in benchmark. Existing repair systems may fail to generate true patch due to test suite overfitting [61], [62], which is a concept in statistics or machine learning. Here overfitting means that the repair helps program perform well within test suite while fail in real usage. Yang et al. [63] introduced fuzz testing to help detecting over-fitted patches more accurately. Liu *et al.* [64] promoted an approach to heuristically determine the correctness of generated patches. They listed two basic observations: (1) PATCH-SIM: after a patch is applied, a passed test usually behave similarly as before, while a failed test usually behaves differently, (2) TEST-SIM: when two tests have similar executions, they are likely to have the same test results. They managed to prevent 56.3% of the incorrect patches without blocking any correct one.

Also, in the common framework of "generate and validate" the amount of test cases is roughly linear with the total repair time consuming. Regression testing in patch validating is a critical huge time consumer, which limits the test suite scale in reality. For this reason, test case redundancy elimination, and some strategies such as test cases prioritization can help terminate the regressive testing as soon as possible. Qi et al. [20] illustrated the strength of most-failing-cases-executed-first in performance. We believe that the properly designed validating strategy can harmonize the contradictions between coverage and performance to a considerable degree.

4.2. Impact of Fault Localization Accuracy

To automatically repair the fault, programmers have to know the exact fault location or area. In most repair techniques fault detection and localization is overwhelmingly carried out as the first step [9][6]. A fault localization technique outputs a list of suspicious program statement, then it's fed to the repairing step. Since the repair work picks a suspicious statement in the list to search for the patch candidate and then picks the next repeatedly, the effectiveness of localization is calculated as the percentage of code that have to be examined before the real buggy statement is reached in the list. Thus the localization effectiveness has a significant influence on repairing performance. Intuitively the total time cost is linear with the localization effectiveness. In GenProg, Le Goues et al. [65] used a probabilistic valuation of fault location and found that adapting the state of art fault localization techniques improved its performance significantly.

Through decades of research, fault localization becomes increasingly complicated, while the efficiency and effectiveness are still unsatisfying[66][67]. Especially there remain some intractable bug situations such as multiple bugs, missing code bugs, and complex structural bugs. How do the multiple bugs influence each other in a single program is uncleared, and how to detect the code omission as well as rank its location into the list is not solved smoothly. Unfortunately the assumed one point bug is too much restricted and idealized while the special situation comes much more commonly. That is exactly the obstacle in both sections of fault localization and repairing that decreases the success rate of most repair methods. Newly arisen method must cover these situations as much as possible to be useful.

Existing repair techniques divide the two individual steps of locating and repairing separately, the interface between two them is the identified suspicious program point, i.e., the location of the root cause or the location where patch should be applied. However, the procedure of localization analyzes the program structure, scans the execution traces, and so on. Separated architecture can lose critical information in patch searching step, causing redundant work load. The repeated trial according to the rank list also requires redundant patch searching. However, mergence of fault localization and repairing is a considerable alternative. The idea of Mutation based localization [68] can provide some hints for that. It takes a statement as suspicious when a mutation on it switches some failed runs to pass, meanwhile the mutation can be a

candidate patch in test based repair. Mechtaev et al. [50] fixed bugs by sketching the program combined with test cases and partial SMT solving. More innovations in the working mechanism of fault locating for repairing are also worthwhile.

At the same time, fault localization only evaluates the buggy code point. Recently, it is proposed that the bug inducing code may not be actually the perfect fixing point by Fry [69], which may mislead both the programmer and automatic repair in debugging. This might be a new research direction in fault localization.

4.3. Concerns in Patch Generation

Given the fault location and a test suite, the most critical question in TBR is how to generate the true patch. Program code varies significantly due to its various functional requirements. No matter a bug comes from human carelessness in implementation or the confines rooted inherently in the design logic, it is always an open challenge to reconstruct a program following the programmer's original intention.

There are techniques generating patches by using existing code in the program, applying fixing template, and synthesizing expressions in single statement. They actually do not know what the patch should be like or what features the patch should maintain. The exponentially augmented searching space remains the pivotal obstacle in search-based repair. which we find inevitable if repairing merely relies on test suite. Le Goues *et al.* [71] recommended that test suite and specifications should be combined to enhance repair validity. Specifications like assertions stipulate the object behaviors or method innovations that can help reduce the searching space, and in certain situations specifications describe the program requirements better [72], [73], [74]. Our wild guess is that approaches like automatic programming and program sketching [75] may be helpful, and any innovations on the patch formulation deserve sustained attentions.

At the same time, one fault can be fixed in several ways which make differences in performance, readability, and robustness. So candidate selection can play an important role. For instance, semantic equivalence checking can help avoid redundant candidate patch trial if one patch is equal to a known failed one. In the opposite way if two patches are proved to be correct we can reserve the one with higher quality while abandon the other. Tao et al. found that when provided as repairing hints, patches with higher quality can help programmers fix the program more quickly and correctly, raising the correctness rate from 33% to 71% on their experiment benckmark [76]. Patch code simplification is also considerable if the so called simpleness can be prudently defined, which has already been applied in GenProg [71] and demonstrated helpful.

Among the aspects of patch quality measurement, researchers pay most attentions to the readability at present, which has been recognized as an important factor of software maintenance decades ago[77]. In the research field of TBR, programs are mostly repaired automatically while maintained by human in a long term. Programmers can only trust the fix patched by machine when they can understand the fixing code. Existing methods evaluate the generated patches by the number of code lines or complexity of abstract syntactic tree. Some people explored the maintainability model of software [78][77] and carried out empirical study of relationships between maintainability and code features such as the ratio of variable used in assignments [69]. Recently developed approach is expected to adapt some precise measurements to manipulate the patching code more wisely.

4.4. Evaluation Metrics

As we mentioned that a fault can be fixed in more than one way, in most conditions finding an approximately optimal choice is important and matters a lot in practice. So we are interested in how to evaluate automatic repair via investigating their generated patch codes and the theoretical mechanism, concluding several criteria that are commonly used in the existing techniques.

The capability of an automatic repair method denotes the ability to fix different types of fault in general, regardless of the program languages, application scenarios, data structures, and fixing operations. Measurement of a repair is of great importance, which may be evaluated using success repair rate and correct repair rate. Some existing work provide their approaches and tools by restricting the research field such as infinite loop detection [79]. This restriction can help produce patches more accurately. Researchers tried to promote their generalization by conducting their experiments on large real programs [54] with real bugs reported and compared the machine generated patches with human made ones [30]. Comparisons among existing repair systems rely on same benchmarks, Defects4J [56], ManyBugs [80]and Codeflaws [1] could be considerable alternatives.

Machine generated patches are usually less understandable than human written ones, and this feature plays an important role in the maintainability of software system [81]. Techniques that repair programs syntactically, semantically, and minimally are generally preferred as more understandable, but the understandability is hard to quantitively measured in practice. All factors such as lines of code changes, complexity of patching code structure, the number of variables, and number of method usage may be taken into account.

5. Conclusion

In this paper, we surveyed the newly arising topic of test based automatic program repair, which has drawn a great number of research attentions. We represent the basic techniques from different angles to solve the problem, analyze their functional principles, and compare their properties. Then we list some important issues to be addressed in this research domain.

Although there are some promising techniques like genetic programming and SMT solving that both receive in-depth research efforts, the fully automatic repair is still far away from practical application. And there is not a systematic theory in this area yet, which is essential to sustain the prosperity of a research trend in a long term. We believe that there will be more follow-up work and novel ideas coming out.

Acknowledgment

This work was supported by a grant from the National Natural Science Foundation of China (project no. 61379045), a grant from the China Scholarship Council (project no. 201604910232), an open project from the State Key Laboratory of Computer Science (project no. SYSKF1608).

References

[1] Anvik, J., Hiew, L., & Murphy, G. C. (2005). Coping with an open bug repository. *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*.

[2] Hailpern, B., & Santhanam, P. (2002). Software debugging, testing, and verification. *IBM Systems Journal 41.1* (4-12).

[3] Zeller, A. (2005). *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc.

[4] Kim, H., *et al*. (2005). The art of testing less without sacrificing quality. *IEEE/ACM, IEEE International Conference on Software Engineering IEEE*.

[5] Koopman, P. (2003). Elements of the self-healing system problem space. *Proceedings of International Conference on Software Engineering.*

[6] Weimer, W., Nguyen, T., Goues, C. L., & Forrest, S. (2009). Automatically finding patches using genetic programming. *Proceedings of the 31st International Conference on Software Engineering (ICSE). IEEE Computer Society*.

[7] Monperrus. M. (2014). A critical review of automatic patch generation learned from human-written patches: essay on the problem statement and the evaluation of automatic software repair. *Proceedings of the 36th International Conference on Software Engineering (ICSE). ACM*.

[8] Monperrus, M. (2015). Automatic software repair: A bibliography. *Acm Computing Surveys*, *51*.

[9] Debroy, V., & Wong, W. E. (2010). Using mutation to automatically suggest fixes for faulty programs. *International Conference on Software Testing, Verification and Validation*.

[10] Naish, L., Lee, H. J., & Ramamohanarao, K. (2009). Spectral debugging with weights and incremental ranking. *Software Engineering Conference*.

[11] Jones, J. A., & Harrold, M. J. (2005). Empirical evaluation of the tarantula automatic fault-localization technique. *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*.

[12] Necula, G. C., McPeak, S., Rahul, S. P., & Weimer, W. (2002). Cil: An infrastructure for C program analysis and transformation. *International Conference on Compiler Construction*.

[13] Xie, X., Chen, T. Y., Kuo, F.-C., & Xu, B. (2013). A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software Engineering and Methodology*.

[14] Arcuri, A., & Yao, X. (2007). Coevolving programs and unit tests from their specification. *IEEE International Conference on Automated Software Engineering*.

[15] Arcuri, A., & Yao, X. (2008). A novel co-evolutionary approach to automatic software bug fixing. *IEEE World Congress on Computational Intelligence.*

[16] Forrest, S., Nguyen, T., Weimer, W., & Goues, C. l. (2009). A genetic programming approach to automated software repair. *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*.

[17] Nguyen, T., Weimer, W., Goues, C. L., & Forrest, S. (2009). Using execution paths to evolve software patches. *International Conference on Software Testing, Verification and Validation Workshops.*

[18] Arcuri, A., & Briand, L. (2011). A practical guide for using statistical tests to assess randomized algorithms in software engineering. *International Conference on Software Engineering*.

[19] Qi, Y., Mao, X., & Lei, Y. (2013). Efficient automated program repair through fault-recorded testing prioritization. *Proceedings of the 36th International Conference on Software Engineering*.

[20] Qi, Y., Mao, X., Lei, Y., Dai, Z., & Wang, C. (2014). The strength of random search on automated program repair. *Proceedings of the 36th International Conference on Software Engineering.*

[21] Assiri, F. Y., & Bieman, J. M. (2014). An assessment of the quality of automated program operator repair. *Brian P Robinson,* 273-282.

[22] Tan, S. H., & Roychoudhury, A. (2015). Relifix: Automated repair of software regressions. *Proceedings of the International Conference on Software Engineering*.

[23] Weimer, W., Fry, Z. P., & Forrest, S. (2013). Leveraging program equivalence for adaptive program repair: Models and first results. *Proceedings of the 28th International Conference on Automated Software Engineering*.

[24] Xiong, Y., Wang, J., Yan, R., Zhang, J., Han, S., & Huang, G., *et al.* (2017). Precise condition synthesis for program repair. *Proceedings of International Conference on Software Engineering*.

[25] Basili, V. R., & Perricone, B. T. (1984). Software errors and complexity: An empirical investigation. *Communications of the ACM*.

[26] Endres, A. (1975). An analysis of errors and their causes in system programs. *ACM Sigplan Notices.*

[27] Pan, K., Kim, S., & Whitehead, J. E. J. (2009). Toward an understanding of bug fix patterns. *Empirical Software Engineering*.

[28] Martinez, M., & Monperrus, M. (2013). Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*.

[29] Martinez, M., & M. Monperrus. (2012). Mining repair actions for guiding automated program fixing. *Diss. Inria*.

[30] Kim, D., Nam, J., Song, J., & Kim, S. (2013). Automatic patch generation learned from human-written patches. *Proceedings of the International Conference on Software Engineering.*

[31] Long, F., & Rinard, M. (2015). Staged program repair with condition synthesis. *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*.

[32] Jeffrey, D., Feng, M., Gupta, N., & Gupta, R. (2009). BugFix: A learning-based tool to assist developers in fixing bugs. *Proceedings of the IEEE 17th International Conference on Program Comprehension.*

[33] Long, F., & Rinard, M. (2016). Automatic patch generation by learning correct code. *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

[34] Schramm, L. (2017). Improving performance of automatic program repair using learned heuristics. *Joint Meeting*.

[35] Ke, Y., Stolee, K., Goues, C. L., & Brun, Y. (2015). Repairing programs with semantic code search. *Proceedings of the 30th IEEE/ACM International Conference the Automated Software Engineering.*

[36] Dallmeier, V., Zeller, A., & Meyer, B. (2009). Generating fixes from object behavior anomalies. *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[37] Dallmeier, V., Lindig, C., Wasylkowski, A., & Zeller, A. (2006). Mining object behavior with ADABU. *Proceedings of the 2006 international workshop on Dynamic systems analysis.*

[38] Pei, Y., Carlo, A. F., Martin, N., & Bertrand, M. (2015). Automated program repair in an integrated development environment. *Proceedings of the 37th International Conference on Software Engineering.*

[39] Wei, Y., Pei, Y., Furia, C. A., Silva, L. S., Buchholz, S., Meyer, B., & Zeller, A. (2010). Automated fixing of programs with contracts. *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA)*.

[40] Wei, Y., Pei, Y., Furia, C. A., Silva, L. S., Buchholz, S., Meyer, B., & Zeller, A. (2010). Automated fixing of programs with contracts. *International Symposium on Software Testing and Analysis*.

[41] Pei, Y., Furia, C. A., Nordio, M., & Meyer, B. (2014). Automatic program repair by fixing contracts. *Fundamental Approaches to Software Engineering. Springer Berlin Heidelberg*.

[42] DeMarco, F., Xuan, J., Berre, D. L., & Monperrus, M. (2014). Automatic repair of buggy if conditions and missing preconditions with smt. *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*.

[43] Moura, L. D., & Bjørner, N. (2011). Satisfiability modulo theories: introduction and applications. *Communications of the ACM*.

[44] Moura, L. D., Dutertre, B., & Shankar, N. (2007). A tutorial on satisfiability modulo theories. *Computer Aided Verification*.

[45] Cadar, C., Goderfroid, P., Khurshid, S., Păsăreanu, C. S., Sen, K., Tillmann, N., & Visser, W. (2011). Symbolic execution for software testing in practice: preliminary assessment. *Proceedings of International Conference on Software Engineering*.

[46] Păsăreanu, C. S., & Visser, W. (2009). A survey of new trends in symbolic execution for software testing and analysis. *International Journal on Software Tools for Technology Transfer*.

[47] He, H., & Gupta, N. (2004). Automated debugging using path-based weakest preconditions. *Fundamental Approaches to Software Engineering, International Conference*.

[48] Nguyen, H. D. T., Qi, D., Roychoudhury, A., & Chandra, S. (2013). SemFix: Program repair via semantic

analysis. *Proceedings of International Conference on Software Engineering*.

[49] Barrett, C., Stump, A., & Tinelli, C. (2010). The smt-lib standard: Version 2.6. *Pediatric Dentistry*.

[50] Mechtaev, S., Jooyong, Y., & Roychoudhury, A. (2015). Directfix: Looking for simple program repairs. *Proceedings of the 37th International Conference on Software Engineering*.

[51] Sergey, M., Yi, J., & Roychoudhury, A. (2016). Angelix: Scalable multiline program patch synthesis via symbolic analysis. *IEEE/ACM, International Conference on Software Engineering*.

[52] Kneuss, E., Koukoutos, M., & Kuncak, V. (2015). On deductive program repair in leon. *Computer Aided Verification. Springer International Publishing*.

[53] Xuan, L., Bach, D., Lo, D. & Goues, C. L. (2017). Empirical study on synthesis engines for semantics-based program repair. *IEEE International Conference on Software Maintenance and Evolution IEEE*.

[54] Jr, F. S. O., Pattabiraman, K., & Mesbah, A. (2014). Vejovis: Suggesting fixes for JavaScript faults. *Proceedings of the 36th International Conference on Software Engineering*.

[55] Samimi, H., Schäfer, M., Artzi, S., Millstein, T., Tip, F., & Hendren, L. (2012). Automated repair of HTML generation errors in PHP applications using string constraint solving. *Proceedings of the 34th International Conference on Software Engineering*.

[56] Martinez, M., Durieux, T., Xuan, J., Sommerard, R., & Monperrus, M. (2015). Automatic repair of real bugs: An experience report on the defects4j dataset. *arXiv preprint arXiv:1505.07002*.

[57] Qi, Z., Long, F., Achour, S., & Rinard, M. (2015). Efficient automatic patch generation and defect identification in kali. *Proceedings of the International Symposium on Software Testing and Analysis.*

[58] Goues, C. L., Forrest, S., & Weimer, W. (2013). Current challenges in automatic software repair. *Software Quality Journal.*

[59] Sen, K. (2007). Concolic testing. *In Automated Software Engineering*.

[60] Jiang, M., Chen, T., Kuo, F., Ding, Z., *et al.* (2017). A revisit of the integration of metamorphic testing and test suite based automated program repair. *International Workshop on Metamorphic Testing IEEE Press*.

[61] Qi, Z., Long, F., Achour, S., & Rinard, M. (2015). An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*.

[62] Smith, E. K., Barr, E. T., Goues, C. L., & Brun, Y. (2015). Is the cure worse than the disease? Overfitting in automated program repair. *Joint Meeting on Foundations of Software Engineering ACM*.

[63] Yang, J., Zhikhartsev, A., Liu, Y., & Tan, L. (2017). Better test cases for better automated program repair. *Joint Meeting on Foundations of Software Engineering ACM*.

[64] Liu, X., *et al*. (2017). Identifying patch correctness in test-based automatic program repair.

[65] Goues, C. L., Dewey-Vogt, M., Forrest, S., & Weimer, W. (2012). A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. *Proceedings of the 34th International Conference on Software Engineering*.

[66] Abreu, R., Zoeteweij, P., & Gemund, A. J. C. V. (2007). On the accuracy of spectrum-based fault localization. *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION. Taicpart-Mutation*.

[67] Ostrand, T. J., Weyuker, E. J., & Bell, R. M. (2005). Predicting the location and number of faults in large software systems. *IEEE Transactions on Software Engineering*.

[68] Moon, S., Kim, Y., & Kim, M. (2014). Ask the mutants: Mutating faulty programs for fault localization. *Proceedings of the 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*.

[69] Fry, Z. P., Landau, B., & Weimer, W. (2012). A human study of patch maintainability. *Proceedings of the*

International Symposium on Software Testing and Analysis ACM.

[70] R. Abhik, *et al*. (2017). Codeflaws: A programming competition benchmark for evaluating automated program repair tools. *Proceedings of the International Conference on Software Engineering Companion IEEE Press*.

[71] Goues, C. L., Nguyen, T., Forrest, S., & Weimer, W. (2012). GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*.

[72] Könighofer, R., & Roderick, B. (2011). Automated error localization and correction for imperative programs. *Formal Methods in Computer-Aided Design*.

[73] Liu, P., & Zhang, C. (2012). Axis: Automatically fixing atomicity violations through solving control constraints. *Proceedings of the 34th International Conference on Software Engineering (ICSE)*.

[74] Perkins, J. H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., Wong, W.-F., Zibin, Y., Ernst, M. D., & Rinard, M. (2009). Automatically patching errors in deployed software. *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*.

[75] Solar-Lezama, A. (2013). Program sketching. *International Journal on Software Tools for Technology Transfer*.

[76] Tao, Y., Kim, J., Kim, S., & Xu, C. (2014). Automatically generated patches as debugging aids: a human study. *ACM Sigsoft International Symposium on Foundations of Software Engineering*.

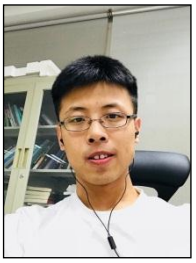
[77] Vessey, I., & Weber, R. (1983). Some factors affecting program repair maintenance: an empirical study. *Communications of the ACM*.

[78] Heitlager, I., Kuipers, T., & Visser, J. (2007). A practical model for measuring maintainability. 6th International *Conference on the Quality of Information and Communications Technology,. QUATIC*. *IEEE*.

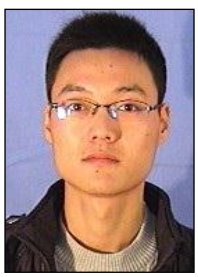
[79] Nistor, A., Chang, P.-C., Radoi, C., & Lu, S. (2015). CARAMEL: Detecting and fixing performance problems that have non-intrusive fixes. *Proceedings of the IEEE International Conference on Software Engineering IEEE*.

[80] Goues, C. L., Holtschulte, N., Smith, E., *et al*. (2015). The manybugs and introclass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering, 41(12)*,1236-1256.

[81] N. Hiroki, *et al.* (2017). Toward developer-like automated program repair — Modification comparisons between genprog and developers. *Software Engineering Conference IEEE*, 241-248.



Yuzhen Liu is an MSc student at the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences in Beijing. He obtained his bachelor degree in software engineering from North China Electric Power University in 2014. His research interests are software testing and automatic program repair.



Long Zhang is a PhD candidate at the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences in Beijing. He obtained his bachelor degree in computer science and technology from Hefei University of Technology. His research interests include program debugging and program verification.

Zhenyu Zhang is an associate professor at the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences. He obtained the PhD degree from The University of Hong Kong, and master and bachelor degrees from Tsinghua University. His research interests are debugging and testing for software and systems as well as the reliability issues of web-based services and cloud-based systems. He has published research results in venues such as TSE, TSC, TRel, Computer, ICSE, FSE, ASE, and WWW.