Challenges of Debugging Software Errors for Automatic Program Repair

Alireza Khalilian, Ahmad Baraani-Dastjerdi, Bahman Zamani
Department of Software Engineering, Faculty of Computer Engineering, University of Isfahan, Isfahan, Iran
{khalilian, ahmadb, zamani}@eng.ui.ac.ir

ABSTRACT

This paper investigates the problem of software errors concentrating on the *newborn* research area of automatic program repair. It aims at providing a roadmap toward the research in automatic software debugging in general, and at laying bare the status quo of automatic program repair, in particular. The authors highlight the issues in software development lifecycle that give rise to introduce errors. Then, the authors elucidate which issues can be resolved and which ones not. The authors review the impediments that preclude resolving every error and releasing correct software. The authors give the strategies to eliminate software errors and the concomitant limitations to these strategies. Then, the authors shift the focus on the problem of automatic program repair which is an integral part for full-automatic software debugging. Considering the status quo, the paper elaborates on some of the significant challenges afflicting this avenue of research.

***Keywords*:** Software Quality, Software Debugging, Software Testing, Fault Localization, Automatic Program Repair, Fault, Error, Failure, Test Case, Patch

1. INTRODUCTION

Software debugging dates back to the first appearance of computers and programming languages. For decades, *every* debugging activity has been totally performed by *manual effort* [1]. The increase in complexity of the software as well as the power of the computers suggested researchers and practitioners to *automate* activities of debugging [2]. The progress that has been made to date led to automation of many of the debugging activities; many of which are even matured enough to be used in industrial practice. Among many shortcomings and research gaps, the authors turn their focus on two interrelated classes of challenges:

- The current techniques for the activities/steps of debugging are designed and optimized to work *in isolation*. A *unified* technique comprising all steps is missing. It must in effect consider the challenges of every step. What are the current challenges?
- The last step in debugging, *program repair* must be automated for *full automation* of debugging; many issues must be addressed in this *newborn* subfield. What are the current issues?

In their paper published in 2013 [3], Le Goues et al. discussed the challenges that are *unique* for automatic program repair. Since then, the research community face a proliferation of techniques along with new conclusions and insights that have been drawn from studies, despite that we are still at the beginning. Moreover, automatic program repair relies heavily on the information of the previous steps of debugging and needs customized versions of them [3]. Thus, the research community must determine the existing challenges of all activities of debugging to design successful automatic program repair techniques. To address these issues, the authors make the following contributions in this paper:

- Best-effort comprehensive *challenge-centered* review of the different steps/activities for debugging software errors targeted towards automatic program repair
- Concrete presentation of challenges concerning automatic program repair considering *recent advances in the subfield*

Figures 1 through 3 depict the graphical representation of the techniques used for making correct software. These figures are intended to serve as concise picture of the techniques that are investigated in this study as well as the categorization of them. Part of these figures has been presented by Jeffrey [4].

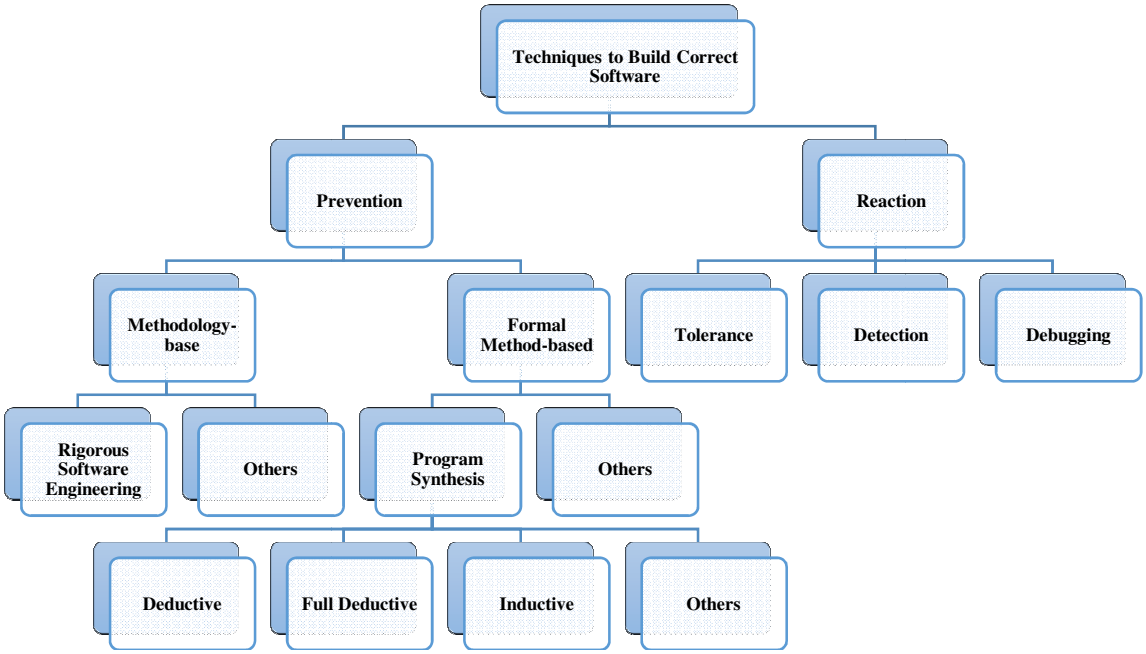

Figure 1: The categorization of the techniques used for making correct software. Prevention techniques leverage correct-by-construction approach while reaction techniques try to deal with errors. The sub-categories for detection and debugging techniques are shown in Fig. 2 and Fig. 3 respectively.

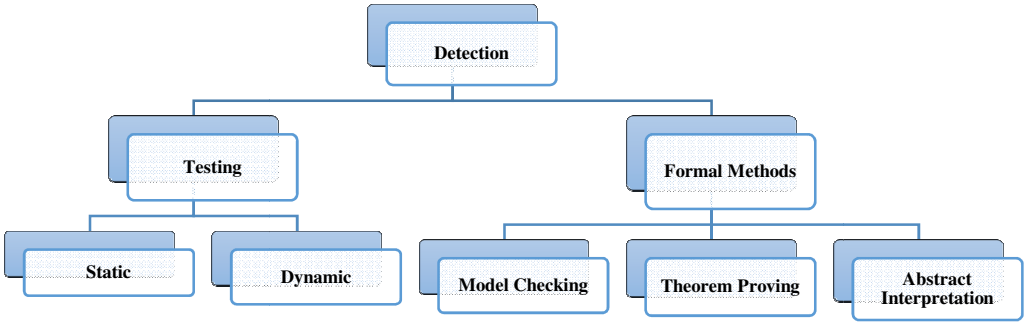

Figure 2: The sub-categories for the detection techniques.

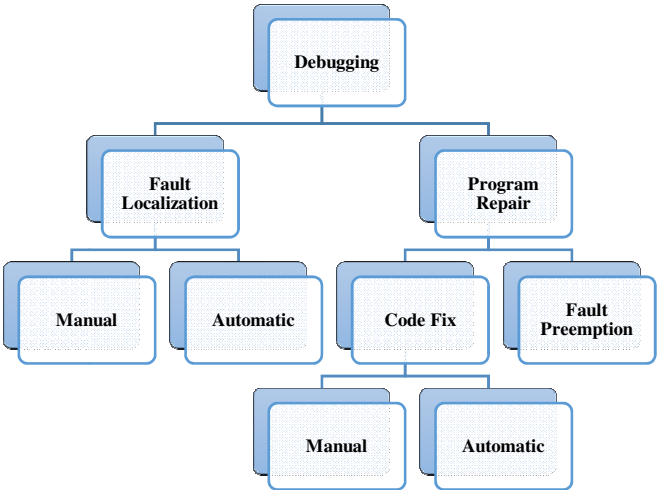

Figure 3: The sub-categories for the debugging techniques

Researchers and practitioners can benefit from this paper in the following ways:

- To provide a roadmap for researchers in designing unified and automatic program repair techniques
- To help the users of debugging techniques for better recognition of strengths and weaknesses and to help for better application in real-world situations
- To provide information for those who want to customize and tailor the current techniques for their specific usage

The rest of the paper is organized as follows: The problem of software faults is discussed in Section 2. The development of correct software is expressed in Section 3. The authors give arguments for fault prevention and reaction to faults in Section 4 and Section 5 respectively. Finally, the paper elaborates on program repair in Section 6 and concludes in Section 7.

2. THE PROBLEM OF SOFTWARE FAULTS

The application of software in all aspects of human's life is increasingly growing due to the many benefits including speed, convenience, and reliability. However, the major issue deals with the fact that the software does not always work as expected and it becomes *unreliable* [4]. Considering the widespread usage of software, the consequences of unreliability in software behavior could be catastrophic and leads to high human and monetary costs.

The major source of unreliability in software is *errors* within the source code. Software errors occur as a result of manual efforts of software development and the complexity of the software itself. The software errors are frequently introduced in the software [1, 5]; so increasingly usage and reliance to software necessitates more urgent consideration of software errors [4]. Many years of experience shows that developing error-free software is hardly attainable [6], fixing every error is nearly impossible [4], and software debugging is as twice harder as when the software is developed from scratch [7]. This is why the software is often released with *known* and *unknown* faults [8]. In addition, the following facts illustrate the problem:

- The number of faults detected by software testing is far more than those that are fixed [9].
- Manual fault detection and fixing by developers are time-consuming tasks, which make debugging high costly [10].
- Debugging resources are limited [11]. Therefore the maintenance team has to triage and prioritize faults that could be fixed within the time budget.
- The overall intrinsic complexity of software development makes the software more error-prone. There exist faults that are only exposed at certain and infrequent situations, often when the software is working at customer's site in real usage [12].

Faults in the software are responsible for unreliability.
Testing time and resources are limited.
Manual testing efforts could also result in new errors.
Software is often released with known and unknown faults.

Influenced by the mentioned factors and to preserve the commercial position, software vendors are forced to release their software product with the faults detected during testing; however, they commit their customers to eliminate the faults within a certain and short time period. As an example, Windows 2000 operating system was released with 63000 known faults [13] due to insufficient testing resources. Often, the result of fixing the faults is released through a *patch* that removes the respective fault. More about patches is given in Section 6, where we discuss about program repair.

Patches need to be released to eliminate some of remaining faults and to satisfy the vendor's commitment.

Fault detection and removal can occur at two stages: *before-deployment* and *post-release* [14]. The problem is that post-deployment debugging of faults is costly and prone to side-effects [14] such as exposition of user's sensitive and private data as well as a significant threat to the company's market share. Sometimes, the failure reproduction is difficult [15]. Currently, however, software debugging occurs at both mentioned stages even though the final end is to move as much debugging activities as possible to the first stage. The attainment to this goal needs fast and low-cost debugging activities, which could be achieved by *full automation* of every debugging task, especially fault fixing and program repair, which are far from easy. Consider some real-life examples to highlight the insufficiency and weakness of current debugging activities and the necessity for full-automation:

- There exist between 0.5 to 25 faults on average within each 1000 lines of code [16].
- The testing and debugging tasks involve about 50% to 75% of total development costs [17].
- The software system for CSA, developed by EDS, has overpaid to 9.1 millions of people and underpaid to 700 thousands of people that was total of 5.1 billions of pounds. This software contained 239 thousands of faults of which 36 thousands still remain [18].
- A recent study in 2013 at Cambridge University estimated the total cost of software debugging world-wide to be about $312 billion. This study also showed that developers are consuming roughly 50% of their time to debug the software and make it to work [19].
- An investigation [20] of manual fault fixing demonstrated that, among 2000 fixed faults, between 14% and 24% of fixes were incorrect and 43% of fixes led to security concerns, memory corruption, and software crashes or hangs. It turns out that manual fixing of faults might result in more faults [21].
- A number of well-known companies such as Google and Mozilla conducted *bug-bounty* plans, in which everyone who can find software errors is paid noticeable monetary testimonials [22].

These facts and numbers motivated the research community to introduce automatic debugging and program repair techniques. Program repair, which has received recent attention, is harder than fault detection and localization [23].

Post-deployment treatment of faults could be harmful and costly; we need full automation of debugging before-deployment.

3. THE DEVELOPMENT OF CORRECT SOFTWARE

In order to manage software faults and to develop correct software, two strategies can be considered [4]: *prevention* (*a priori development*) and *reaction* (*a posteriori development*). The former employs techniques to develop software whose quality is guaranteed during the development process. The latter strategy is applied when the faults are introduced within the software. Prevention techniques are further divided into two categories, neither of which can be used for existing or legacy software:

- The techniques that are applied to the software development *methodology* such as *rigorous software engineering* [24].
- The techniques that develop error-free software using *formal methods* such as proof-guided construction and design-by-contracts [14].

Contrary to prevention techniques, reaction ones are applied to the software code or other respective artifacts such as specifications. They attempt to *detect* and *fix* faults, or else to avoid or *tolerate* the consequences and side-effects of software faults [4]. Detection of software faults could be accomplished via *software testing*, *model checking*, *abstract interpretation*, *correctness proof*, and *formal verification*. Debugging is then used for program repair or fault preemption.

Despite the need to expertise and maturity, prevention from introducing faults during software development would be a better solution, but is not applicable for existing or legacy software.

4. FAULT PREVENTION

Prevention involves applying techniques during software development that the likelihood of introducing faults is immensely reduced. To construct the correct software, program *synthesis* could be a choice [25]. This method takes as input a set of constraints that reflect the user's requirements and intentions. Then, program synthesis attempts to find or construct a *provably-correct* program that conforms to the specified constraints. Deductive and inductive syntheses are instances of such techniques.

Full deductive synthesis [26] uses well-defined rules to construct the respective program. The issue concerning this kind of synthesis is the need to high programming expertise. Consequently, full automation is encumbered with full deductive synthesis. This impediment restricts its application to specific areas such as safety-critical software systems [27]. Inductive synthesis extends some sample programs to construct a program that satisfies the specifications. In fact, the complete desired program is constructed from faulty incomplete ones [28]. In summary, the current synthesis-based techniques suffer from *scalability* issues and hold promise only for specific-purpose cases [14, 25].

Scalability concerns and high programming expertise are negative sides of synthesis-based techniques.

5. REACTION TO FAULTS

Considering that the software is still developed by manual effort, at least in near future, a priori techniques do not completely prevent from introducing faults in the software. Therefore, a family of techniques is required to react against the faults. These techniques must be capable to handle the faults both in legacy software and the software that will be developed by human at near future.

Typically, the first step to resolve the faults is to record them in a *bug-tracking database*, which is very common in large software projects to assist in handling and fixing of faults [29]. Such database records in special *bug-report* formats, the information pertinent to the exposed faults and failures. The aim is to provide capabilities for developers to facilitate the process of *identification*, *triage*, *reproduction*, *understanding*, and *fixing* of faults. Unfortunately, however, bug-reports do not always lead to fix the faults; the bug-report might be *redundant* considering the study [30, 31] that showed that between 25% and 35% of bug-reports were redundant. Even in some cases, the developer in charge of the reported bug might fail to reproduce the bug. This is a situation where the bug-report is closed with no action.

In order to resolve the reported bug, the developer starts debugging to *find* and *fix* the fault [32, 33]. Fault finding consists of three steps: *failure detection* (something failed), *fault diagnosis* (why did it fail), and *fault localization* (where is the root cause). To fix the fault, we need to *infer the repair* (the appropriate action to resolve the fault) [34]. These four steps are sharing some commonalities [34]. In addition, each step requires to the information of the previous step.

Debugging is traditionally carried out *manually* and the research community is continuously working to automate it. The reason lies in the fact that debugging steps are immensely *time-consuming* and the increasingly growing the *size* and *complexity* of software even aggravate the situation.

To realize fully-automated debugging, several impediments must be addressed. Currently, the existing techniques for debugging have been designed based on the assumptions (e.g., *perfect fault understanding* [35]) that do not hold in practice, and have been optimized to work *in isolation*. As a consequence, a certain step may fail to benefit from the information provided by an effective technique in the previous step [3]. The evidence to this claim is the results [36] that imply to the ineffectiveness of existing automatic fault localization techniques for automatic program repair, since they have been designed to be used by human developer. Note that automation of the program repair itself also remains a major challenge. Note also that, typically, part of an effective fault localization or program repair process, is borrowed from the techniques in the software testing context. This argument and the evidences strengthen the conjecture that for a perfect full-automatic debugging technique to be highly effective and practical, the designer needs to incorporate every

step together. As a result, it is reasonable to hope to obtain an effective integrated technique with consistent components. We acknowledge that this argument, while significant, is not sufficient to draw such conclusion; however, we hope to encourage conducting studies for further investigation.

Failure detection and fault diagnosis can be realized by *software testing* (and fault localization) or *verification*, which are discussed in the Section 5.1 and Section 5.2 respectively. Then, in Section 6, program repair is discussed in detail.

Manual debugging is no longer cost-effective for current large and complex software.
Failure detection, fault diagnosis, fault localization, and program repair are interdependent steps. Much benefit is expected to be achieved by integrating them in a single technique.

5.1 Testing and Fault Localization

Software testing techniques have been matured and are widely used in industrial practices [37-39]. Software is tested either *statically* or *dynamically*. Static software testing can be accomplished by *software review* techniques [40] such as *formal* or *informal code inspection*, *code review*, *code walkthrough*, *pair programming*, or *technical review* [41]. Due to manual effort employed in such techniques, the rate of false positives would be low and more complex logical faults could be resolved. Thus, they are appropriate choices for safety-critical application. However, manual inspection makes them much *time-consuming* and less *scalable*. Dynamic testing runs the software against some test cases and compares the output to *test oracles* [42]. However, some significant challenges preclude dynamic testing to be fully automated and highly effective. In general, dynamic testing does not guarantee the correctness of the software since *exhaustive testing* is computationally infeasible. In particular, generating appropriate *coverage-adequate* [43, 44] and *bug-inducing* test cases still remain a research challenge [45], especially for concurrent applications. The problem of test oracle is another significant challenge [42]. Nevertheless, software testing is the dominant method of checking the correctness and reliability of the software [46] both in academia and industrial scale.

Software testing suffers from many issues; however it is common and ubiquitous. Any small improvement ameliorates the situation.

Software testing is used to detect faults. After that we must find the *location* at which the fault occurs. This is typically performed using *fault localization* techniques [47]. The problem with such techniques is that there exist situations where the *root cause* of the fault is elsewhere in the program and thus is different from the point at which the fault was exposed. In light of this issue, the process of fault localization would become *time-consuming* and might report *incorrect location*. Concurrent programs and nondeterministic faults [48] also exacerbate the situation. Figure 4 demonstrates some of the most common and important techniques for fault localization.

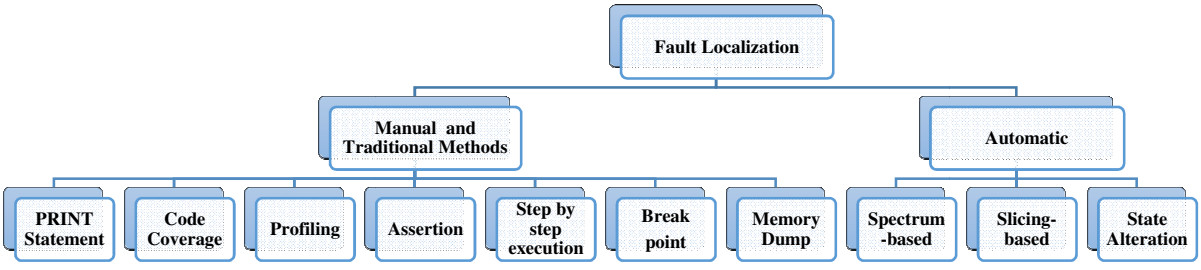

Figure 4: Most common and important fault localization techniques

In summary, every technique or tool that is intended to find faults from the source code of programs needs three essential components [14]: the definition of *correct behavior*, given by for example, explicit test cases or formal specifications, or by inferred specification [49-51]; a method to predict or observe *program's behavior* given by, for example, static or dynamic techniques; a method to *collect deviation* from correct behavior given by, for example, bug reports. Static techniques use program analysis on the source code and may lead to *false positives* due to *overestimation* of program's behavior. On the other hand, dynamic techniques run programs to gain information of program's behavior and may lead to *false negatives* due to *underestimation* of program's behavior.

Difference of the locations corresponding for root cause and exposition of a fault introduces challenges for localization. Concurrency and nondeterminism are other concerns.

5.2 Verification

The process in which the output of a software development stage is checked to satisfy the specified requirements at the previous stage is called verification [2]. It is also used for proving that the program lacks defects with respect to its specifications [1]. Verification provides *stronger guarantees* of program's correctness than software testing. It employs *formal methods* such as *theorem proving* and *model checking* to check and prove the correctness of the given program with respect to the given formal specifications. Verification is more technical than software testing and requires the knowledge of software artifacts, requirements, and specifications [52]. Currently, formal specifications are not common in industrial practices because *high expertise* is warranted for developers. The need to the tools such as theorem prover [52] or proof assistant [53] also incurs *heavy computational costs* and brings *scalability concerns* [54]. Therefore, the present application of verification is roughly restricted to safety-critical software systems.

High expertise, heavy computations, and scalability concerns are major challenges of verification.

6. PROGRAM REPAIR

Program repair is the last step in debugging and of utmost importance, which requires the knowledge of the program's *logic*. This requirement makes the automation of this step a difficult problem [55, 56], possibly with complex solutions. The faulty program often has characteristics that help developers to fix the faulty program using *small modifications*. One is that developers do not program at random [57]. As a consequence, the faulty program is structurally very analogous to the correct one. Another is the studies that corroborate most of program repairs are simple and short [58, 59]. These observations suggest that, by enforcing certain constraints, generating most of the possible short repairs would become computationally *tractable* [33]. However, in general, the state space of possible modifications to the faulty program is *infinite* [60]. To address this problem, program repair techniques explore for the ways to control the exploration and make it more *targeted* and effective. *Search-Based Software Engineering* (SBSE) [61] could help achieve this goal. Figure 5 depicts different program repair scenarios, which are in turn described in the following subsections.

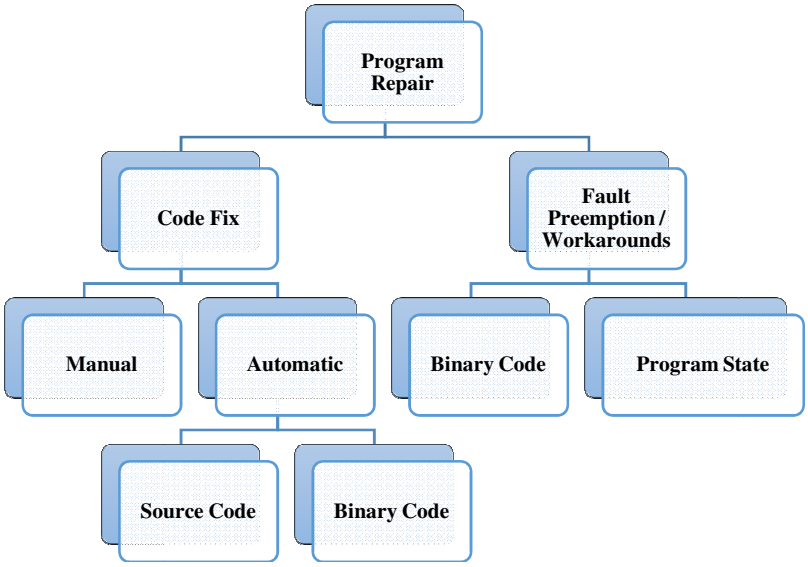

Figure 5: Different program repair scenarios

The result of a program repair technique is often given as a *patch*. A patch is the code without the fault under debugging and fixes that fault. The faulty code is often replaced by the patch. In some cases a jump statement is inserted within the code that transfers the control flow from the faulty code to the fixed code. Some techniques consider patch as a set of modifications that must be sequentially enforced on the faulty code so that the fault would be eliminated. In fact, this is dependent upon the output of the underlying technique. The *correctness* of a patch might be evaluated using some regression test cases [62]. If the patch passes the whole existing regression test cases, it is considered *plausible*; if the patch perfectly removes the fault, it would be *correct* [63]. The problem of determining whether a certain patch has fixed a certain fault is far from easy; even in some cases it is *not well-defined* [63].

Program repair needs effective and efficient techniques to search within the possible-modification space and to evaluate the resulting patches.

6.1 Manual Program Repair

Manual fixing of faults have been traditionally used by developers and is at the moment the *dominant* way of program repair. To accomplish this, the developer typically utilizes assistant tools [64, 65] for step-wise inspection of the faulty program. These tools help trace the faulty program and reproduce the fault to obtain some information about its nature, the root cause, and the relative location. Next, the developer tries to change the code and tests the resulting program. Often, after several trials the developer finds an appropriate repair and finalizes the modifications. In situations where the developer fails to find a patch, he or she can learn from trials and obtains information from the nature of the fault. A *good repair* must ideally satisfy *five* properties [14]: fixes the faulty behavior; does not introduce new faults; preserves the current correct functionality; remains consistent with the overall goals of the system's design; maintainability of the repaired program is possible in future.

Manual fixing of faults is currently widely-used in practice, even though it is time-consuming and costly.

6.2 Automatic Program Repair

In this section, we elaborate on the newborn research area of automatic program repair and give further details. Automatic program repair can be thought of a *recommender system* for software engineering [34] that adapts the procedure of manual bug fixing by human. Automatic support for program repair has been investigated by two approaches: *debugging assistance* and *code modifications*. The first approach involves techniques that produce recommendations and comments for developers to facilitate the manual bug fixing [66, 67]. The second approach changes the code to repair it [9]. This latter approach comprises two classes of techniques namely as *correct-by-construction* (CC) and *generate-and-validate* (GV) [68].

By *formal formulation* of the problem and the constraints, the first class builds *sound* patches [69]. The constraints include contracts or the specifications given explicitly or inferred implicitly [70]. The resulting patches are *provably-correct* [68]. Often, a single patch or few patches are constructed per fault. The efficiency of this class of techniques is highly commensurate with the power of the proof system [69]. Safety-critical systems can take significant advantages of this class of techniques.

The second class leverages *heuristic techniques* [68] to explore within the space of candidate patches [70]. These techniques typically produce many candidate patches for a certain fault that need to be evaluated [69], with *testing* to be the common method of patch evaluation. This way of patch generation and validation adapts the ideas behind SBSE [61] and mutation-based testing [71] or exploit predefined templates [72]. Some of the GV techniques [9, 73] have been designed based on the *Competent Programmer Hypothesis* (CPH) [74]. *Repair* strategy and *test* strategy are the major influencing factors to the cost of any GV technique [73]. Test case-based patch evaluation for GV techniques is highly effective for *legacy software* for which no specifications exist [69]. To find an appropriate patch, GV techniques must overcome several impediments [73]: fault space, fix space, minimizing the bloated code, and evaluating the candidate patch. For the case of fix space, some techniques [75] take the repair code from elsewhere at the faulty program. This design decision has its roots from experimental observations [76] showing *internal repetition* within the code and is better understood by drawing an analogy between program repair and the plastic surgery.

Currently, two classes of repair scenarios exist [34]: *online* and *offline*. The first class comprises temporary run-time repairs and is sometimes called *workarounds*. This class works on either binary code or program's state such as registers or data structures. The second class is permanent repair for software maintenance and is carried out on either source code or binary code.

6.3 Challenges of Automatic Program Repair

In recent years, the automatic program repair received increasingly much attention by the research community. However, there is a host of issues that are at the moment challenging. Thus, this section is devoted to a concrete discussion of some major challenges considering the recent advances in the subfield.

Automation. The current techniques require many information and configuration that must be manually carried out. Thus, the existing techniques are in fact *semi-automated*. Parameter setting, providing suitable inputs, initialization, test oracles, formal specifications and the like are among the necessary operations for a certain technique [3, 9]. Even if the test case generation, fault localization, and the overall repair process are fully automated, the developer is still in charge for evaluating the resulting patches and for controlling to the deployment [32].

Efficacy. Currently, any near-automated program repair technique is limited to a certain *language* and *paradigm*, needs particularities, and repairs a certain *class of faults*. This is expected considering that the automatic debugging problem would be simpler and more *tractable* on *limited models of computation* [32]. The efficacy of the current repair techniques can be achieved when the technique is designed for a certain class of defects [34, 63].

Finding Good Repair. A major issue concerning the current program repair techniques is the correctness and quality of the repaired programs. The results of experiments [63, 70, 74] on most of the existing repair techniques show that they produce incorrect repairs for many of the faulty input programs. Moreover, even many of the correct repaired programs are hardly understandable by human developers. The reason is that they may be generated according to some random modifications on the code. Thus, it's likely to have programs that pass all of the available test cases, yet the logic and purpose behind some statements of the repaired program is unclear and unknown. Low understandability of the repaired code makes serious problems for future maintenance activities. In summary, finding *correct* and *high-quality* repair as well as measuring the *understandability* of the code is highly an open research problem [14].

Repair Quality by Test Cases. Measuring the quality of the repair is an open research problem. The correctness of a program is measured with respect to its *specification* [33], which could also be used for repair evaluation. At the moment, repairs are evaluated by two means [70]: *test cases* and *specifications*. The unreliability of the repairs resulted from test case-based techniques is inevitable since running the repaired program against every test case cannot be realized. Any repair technique that works based on test cases, fails to check anything that is not observable by test cases. As a consequence, the architectural aspects, non-functional properties and high-level goals of the system are missed. In addition, the quality of the repaired programs would be dependent upon the quality of the applied test cases. The repaired program is as good as the available test cases [32], which are of course *incomplete* and *non-exhaustive* [77]. Finding an appropriate suite of test cases to evaluate a program is also a search problem [78]. The number of programs that satisfy a partial specification such as some test cases is *infinite* and the repair technique generates one of them. As a result, it is very likely that the generated program does not adhere to the unavailable full desired specification [79]. Moreover, the likelihood of introducing other faults increases [63]. The final outcome is *poor-quality* patches.

Repair Quality by Formal Specifications. Evaluating of the patches can also be accomplished via formal specifications. The problem is the difficulties that arise when they are manually produced. The rate of *false positives* is very high even for automated techniques that mine specifications from the context of programs [51].

Incomplete or Incorrect Specifications. An equally serious problem that occurs for the techniques utilizing formal specifications as oracles for patch evaluation is the assumption that such specifications are complete and correct. Unfortunately, this assumption does not always hold [80] and leads to *poor-quality* or *incorrect* patches. Accordingly, specification repair is necessitated before program repair [80].

Fitness Function. Designing a suitable fitness function is fundamental to any *evolutionary* algorithm. Hence, it becomes a major challenge influencing the success of the repair techniques that benefit from such computational model [78].

Timing Costs. Program repair is naturally a *trial-and-error* process in which many patches are generated and evaluated. Therefore, test case-based techniques must overcome two issues to be cost-effective: the *high number* of test cases and *long-running* test cases. In fact, these issues provide a serious bottleneck for patch evaluation emphasizing that we will have to trade *quality* for *cost-effectiveness* for any single test case that is removed from consideration [81].

Fault Class. Among many classifications for faults, for the specific domain of automatic program repair, faults can be *fixable* (approachable) or *hard-to-reproduce* [32], and *deep* or *shallow* [82]. Both of these classifications are very coarse-grained. The current automatic program repair techniques work to repair only on fixable and shallow faults. Fixable faults are those that can be reappeared somehow, while reappearance of hard-to-reproduce faults is far from easy; they might be nondeterministic or occur at special situations. In the second classification, shallow faults refer to functional defects that are observed at code level. Deep faults, however, involve design issues, non-functional defects, liveness, and the like.

Concurrency Faults. The repair of concurrency faults remains a major challenge and need special treatment, despite the fact that sophisticated techniques exist for detection of such faults. *Atomicity violation*,

data races, *order violation*, and *deadlock* are well-know instances of concurrency faults. Automatic repair techniques that claim to work on general defects fail to repair concurrent faults. Sometimes, the released patches for concurrency faults also introduce new faults such as data race or deadlocks [83], and often lead to serialization and interleaving bottlenecks.

Fix Space. Any search-based automatic program repair technique needs to search within the space that is highly likely to contain the desired repair. Moreover, due to extremely huge fix space, more efficient and intelligent exploration methods must be designed [63].

7. CONCLUSIONS

Reliance of the current daily life on software necessitates effective and efficient techniques to ensure the reliability and robustness of software. Many challenges arise from the current software testing, verification, and debugging techniques that need to be addressed. This paper provided a concise but roughly comprehensive review of the field to highlight the research advancements and gaps to put light on the road ahead. A separate section is also dedicated to the subfield of automatic program repair, which has gained recent much attention. Automatic generation of test oracles and dealing with scalability are some of the major challenges, among others.

REFERENCES

[1] Zeller, A. (2009). *Why programs fail: a guide to systematic debugging*. Elsevier.

[2] Ammann, P., & Offutt, J. (2016). *Introduction to software testing*. Cambridge University Press.

[3] Le Goues, C., Forrest, S., & Weimer, W. (2013). Current challenges in automatic software repair. *Software Quality Journal*, *21*(3), 421-443.

[4] Jeffrey, D. B. (2009). *Dynamic state alteration techniques for automatically locating software errors* (Doctoral dissertation, UNIVERSITY OF CALIFORNIA RIVERSIDE).

[5] Sink, E. (2006). Why we all sell code with bugs. *The Guardian*. 25 May 2006. Retrieved January 30, 2017, from https://www.theguardian.com/technology/2006/may/25/insideit.guardianweeklytechnologysection.

[6] Computer Security Resource Center. (2009). *National Vulnerability Database*. Retrieved January 30, 2017, from https://nvd.nist.gov/view/vuln/statistics.

[7] Kernighan, B. W., & Plauger, P. J. (1978). The elements of programming style. *The elements of programming style, by Kernighan, Brian W.; Plauger, PJ New York: McGraw-Hill, c1978.*

[8] Weimer, W., Nguyen, T., Le Goues, C., & Forrest, S. (2009, May). Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering* (pp. 364-374). IEEE Computer Society.

[9] Le Goues, C., Nguyen, T., Forrest, S., & Weimer, W. (2012). Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, *38*(1), 54-72.

[10] Weiss, C., Premraj, R., Zimmermann, T., & Zeller, A. (2007, May). How long will it take to fix this bug?. In *Proceedings of the Fourth International Workshop on Mining Software Repositories* (p. 1). IEEE Computer Society.

[11] Naone, E. (2010). So Many Bugs, So Little Time. *MIT Technology Review*. Retrieved January 30, 2017, from https://www.technologyreview.com/s/419975/so-many-bugs-so-little-time/.

[12] Muller, T., & Friedenberg, D. (2011). Certified tester foundation level syllabus.*Journal of International Software Testing Qualifications Board*.

[13] Lickridge, R. (2000). Will bugs scare off users of new Windows 2000. *CNN*. Retrieved January 30, 2017 from http://www.edition.cnn.com/2000/TECH/computing/02/17/windows.2000/index.html.

[14] Le Goues, C. (2013). *Automatic program repair using genetic programming*(Doctoral dissertation, University of Virginia).

[15] Jin, W., & Orso, A. (2012, June). BugRedux: reproducing field failures for in-house debugging. In *Software Engineering (ICSE), 2012 34th International Conference on* (pp. 474-484). IEEE.
[16] McConnell, S. (2004). *Code complete*. Pearson Education.
[17] Hailpern, B., & Santhanam, P. (2002). Software debugging, testing, and verification. *IBM Systems Journal*, *41*(1), 4-12.
[18] Barker, C. (2007). The top 10 IT disasters of all time. *ZDNet*. Retrieved January 30, 2017, from http://www.zdnet.com/article/the-top-10-it-disasters-of-all-time/.
[19] Britton, T., Jeng, L., Carver, G., Cheak, P., & Katzenellenbogen, T. (2013). Reversible debugging software. *University of Cambridge-Judge Business School, Tech. Rep*.
[20] Yin, Z., Yuan, D., Zhou, Y., Pasupathy, S., & Bairavasundaram, L. (2011, September). How do fixes become bugs?. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering* (pp. 26-36). ACM.
[21] Malik, M. Z., Ghori, K., Elkarablieh, B., & Khurshid, S. (2009, November). A case for automated debugging using data structure repair. In *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on* (pp. 620-624). IEEE.
[22] Le Goues, C., Dewey-Vogt, M., Forrest, S., & Weimer, W. (2012, June). A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Software Engineering (ICSE), 2012 34th International Conference on* (pp. 3-13). IEEE.
[23] Harman, M. (2010). Automated patching techniques: the fix is in: technical perspective. *Communications of the ACM*, *53*(5), 108-108.
[24] Almeida, J. B., Frade, M. J., Pinto, J. S., & de Sousa, S. M. (2011). *Rigorous software development: an introduction to program verification*. Springer Science & Business Media.
[25] Gulwani, S. (2010, July). Dimensions in program synthesis. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming* (pp. 13-24). ACM.
[26] Smith, D. R. (1990). KIDS: A semiautomatic program development system.*IEEE transactions on software engineering*, *16*(9), 1024-1043.
[27] Emerson, T., & Burstein, M. H. (1999, October). Development of a constraint-based airlift scheduler by program synthesis from formal specifications. In*Automated Software Engineering, 1999. 14th IEEE International Conference on.* (pp. 267-270). IEEE.
[28] Solar-Lezama, A. (2009, December). The sketching approach to program synthesis. In *Asian Symposium on Programming Languages and Systems*(pp. 4-13). Springer Berlin Heidelberg.
[29] Reis, C. R., & de Mattos Fortes, R. P. (2002, February). An overview of the software engineering process and tools in the Mozilla project. In *Proceedings of the Open Source Software Development Workshop* (pp. 155-175).
[30] Anvik, J., Hiew, L., & Murphy, G. C. (2006, May). Who should fix this bug?. In *Proceedings of the 28th international conference on Software engineering*(pp. 361-370). ACM.
[31] Jalbert, N., & Weimer, W. (2008, June). Automated duplicate detection for bug tracking systems. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on* (pp. 52-61). IEEE.
[32] Pei, Y., Furia, C. A., Nordio, M., Wei, Y., Meyer, B., & Zeller, A. (2014). Automated fixing of programs with contracts. *Ieee transactions on software engineering*, *40*(5), 427-449.
[33] Pei, Y., Wei, Y., Furia, C. A., Nordio, M., & Meyer, B. (2011, November). Code-based automated program fixing. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering* (pp. 392-395). IEEE Computer Society.
[34] Monperrus, M. (2014, May). A critical review of automatic patch generation learned from human-written patches: essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering* (pp. 234-242). ACM.

[35] Parnin, C., & Orso, A. (2011, July). Are automated debugging techniques actually helping programmers?. In *Proceedings of the 2011 international symposium on software testing and analysis* (pp. 199-209). ACM.

[36] Qi, Y., Mao, X., Lei, Y., & Wang, C. (2013, July). Using automated program repair for evaluating the effectiveness of fault localization techniques. In*Proceedings of the 2013 International Symposium on Software Testing and Analysis* (pp. 191-201). ACM.

[37] Penix, J. (2012, June). Large-scale test automation in the cloud (invited industrial talk). In *Software Engineering (ICSE), 2012 34th International Conference on* (pp. 1122-1122). IEEE.

[38] Godefroid, P., Levin, M. Y., & Molnar, D. (2012). SAGE: whitebox fuzzing for security testing. *Queue*, *10*(1), 20.

[39] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., ... & Engler, D. (2010). A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, *53*(2), 66-75.

[40] Wiegers, K. E. (2002). *Peer reviews in software: A practical guide*. Boston: Addison-Wesley.

[41] Myers, G. J., Sandler, C., & Badgett, T. (2011). *The art of software testing*. John Wiley & Sons.

[42] Barr, E. T., Harman, M., McMinn, P., Shahbaz, M., & Yoo, S. (2015). The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, *41*(5), 507-525.

[43] Parsa, S., & Khalilian, A. (2009, December). A bi-objective model inspired greedy algorithm for test suite minimization. In *International Conference on Future Generation Information Technology* (pp. 208-215). Springer Berlin Heidelberg.

[44] Khalilian, A., & Parsa, S. (2009, October). Bi-criteria test suite reduction by cluster analysis of execution profiles. In *IFIP Central and East European Conference on Software Engineering Techniques* (pp. 243-256). Springer Berlin Heidelberg.

[45] Anand, S., Burke, E. K., Chen, T. Y., Clark, J., Cohen, M. B., Grieskamp, W., ... & Mcminn, P. (2013). An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*,*86*(8), 1978-2001.

[46] Orso, A., & Rothermel, G. (2014, May). Software testing: a research travelogue (2000–2014). In *Proceedings of the on Future of Software Engineering* (pp. 117-132). ACM.

[47] 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 (TOSEM)*, *22*(4), 31.

[48] Jeffrey, D., Wang, Y., Tian, C., & Gupta, R. (2011). Isolating bugs in multithreaded programs using execution suppression. *Software: Practice and Experience*, *41*(11), 1259-1288.

[49] Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., & Xiao, C. (2007). The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, *69*(1), 35-45.

[50] Gabel, M., & Su, Z. (2012, November). Testing mined specifications. In*Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (p. 4). ACM.

[51] Le Goues, C., & Weimer, W. (2012). Measuring code quality to improve specification mining. *IEEE Transactions on Software Engineering*, *38*(1), 175-190.

[52] De Moura, L., & Bjørner, N. (2008, March). Z3: An efficient SMT solver. In*International conference on Tools and Algorithms for the Construction and Analysis of Systems* (pp. 337-340). Springer Berlin Heidelberg.

[53] Owre, S., Rushby, J. M., & Shankar, N. (1992, June). PVS: A prototype verification system. In *International Conference on Automated Deduction* (pp. 748-752). Springer Berlin Heidelberg.

[54] Flanagan, C., & Leino, K. R. M. (2001, March). Houdini, an annotation assistant for ESC/Java. In *International Symposium of Formal Methods Europe* (pp. 500-517). Springer Berlin Heidelberg.

[55] Zojaji, Z., Ladani, B. T., & Khalilian, A. (2016). Automated program repair using genetic programming and model checking. *Applied Intelligence*, *45*(4), 1066-1088.

[56] Khalilian, A., Baraani-Dastjerdi, A., & Zamani, B. (2016, May). On the evaluation of automatic program repair techniques and tools. In *Electrical Engineering (ICEE), 2016 24th Iranian Conference on* (pp. 61-66). IEEE.

[57] DeMillo, R. A., Lipton, R. J., & Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *Computer*, *11*(4), 34-41.

[58] Martinez, M., & Monperrus, M. (2015). Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, *20*(1), 176-205.

[59] Dallmeier, V., & Zimmermann, T. (2007, November). Extraction of bug localization benchmarks from history. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering* (pp. 433-436). ACM.

[60] Arcuri, A. (2011). Evolutionary repair of faulty software. *Applied Soft Computing*, *11*(4), 3494-3514.

[61] Harman, M., Mansouri, S. A., & Zhang, Y. (2012). Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)*, *45*(1), 11.

[62] Khalilian, A., Azgomi, M. A., & Fazlalizadeh, Y. (2012). An improved method for test case prioritization by incorporating historical test case data. *Science of Computer Programming*, *78*(1), 93-116.

[63] Qi, Z., Long, F., Achour, S., & Rinard, M. (2015, July). 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* (pp. 24-36). ACM.

[64] Stallman, R., Pesch, R., & Shebs, S. (2002). Debugging with GDB. *Free Software Foundation*, *51*, 02110-1301.

[65] Nethercote, N., & Seward, J. (2007, June). Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices* (Vol. 42, No. 6, pp. 89-100). ACM.

[66] Jeffrey, D., Feng, M., Gupta, N., & Gupta, R. (2009, May). BugFix: A learning-based tool to assist developers in fixing bugs. In *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on* (pp. 70-79). IEEE.

[67] Kaleeswaran, S., Tulsian, V., Kanade, A., & Orso, A. (2014, May). Minthint: Automated synthesis of repair hints. In *Proceedings of the 36th International Conference on Software Engineering* (pp. 266-276). ACM.

[68] Le Goues, C., Holtschulte, N., Smith, E. K., Brun, Y., Devanbu, P., Forrest, S., & Weimer, W. (2015). The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Transactions on Software Engineering*,*41*(12), 1236-1256.

[69] Weimer, W. (2013, August). Advances in automated program repair and a call to arms. In *International Symposium on Search Based Software Engineering* (pp. 1-3). Springer Berlin Heidelberg.

[70] Smith, E. K., Barr, E. T., Le Goues, C., & Brun, Y. (2015, August). 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* (pp. 532-543). ACM.

[71] Jia, Y., & Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, *37*(5), 649-678.

[72] Kim, D., Nam, J., Song, J., & Kim, S. (2013, May). Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering* (pp. 802-811). IEEE Press.

[73] Weimer, W., Fry, Z. P., & Forrest, S. (2013, November). Leveraging program equivalence for adaptive program repair: Models and first results. In*Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on* (pp. 356-366). IEEE.

[74] Fry, Z. P., Landau, B., & Weimer, W. (2012, July). A human study of patch maintainability. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (pp. 177-187). ACM.

[75] Alkhalaf, M., Aydin, A., & Bultan, T. (2014, July). Semantic differential repair for input validation and sanitization. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (pp. 225-236). ACM.

[76] Barr, E. T., Brun, Y., Devanbu, P., Harman, M., & Sarro, F. (2014, November). The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*(pp. 306-317). ACM.

[77] Tassey, G. (2002). The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*,*7007*(011).

[78] Wilkerson, J. L., & Tauritz, D. (2010, July). Coevolutionary automated software correction. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation* (pp. 1391-1392). ACM.

[79] Ke, Y., Stolee, K. T., Le Goues, C., & Brun, Y. (2015, November). Repairing programs with semantic code search. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on* (pp. 295-306). IEEE.

[80] Pei, Y., Furia, C. A., Nordio, M., & Meyer, B. (2014, April). Automatic program repair by fixing contracts. In *International Conference on Fundamental Approaches to Software Engineering* (pp. 246-260). Springer Berlin Heidelberg.

[81] Qi, Y., Mao, X., & Lei, Y. (2013, September). Efficient automated program repair through fault-recorded testing prioritization. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on* (pp. 180-189). IEEE.

[82] Wei, Y., Pei, Y., Furia, C. A., Silva, L. S., Buchholz, S., Meyer, B., & Zeller, A. (2010, July). Automated fixing of programs with contracts. In *Proceedings of the 19th international symposium on Software testing and analysis* (pp. 61-72). ACM.

[83] Jin, G., Song, L., Zhang, W., Lu, S., & Liblit, B. (2011, June). Automated atomicity-violation fixing. In *ACM SIGPLAN Notices* (Vol. 46, No. 6, pp. 389-400). ACM.